Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques

YANGDONG DENG, YUFEI NI, and ZONGHUI LI, Tsinghua University
SHUAI MU, Qualcomm
WENJUN ZHANG, Tsinghua University

Ray tracing has long been considered as the next-generation technology for graphics rendering. Recently, there has been strong momentum to adopt ray tracing–based rendering techniques on consumer-level platforms due to the inability of further enhancing user experience by increasing display resolution. On the other hand, the computing workload of ray tracing is still overwhelming. A 10-fold performance gap has to be narrowed for real-time applications, even on the latest graphics processing units (GPUs). As a result, hardware acceleration techniques are critical to delivering a satisfying level performance while at the same time meeting an acceptable power budget. A large body of research on ray-tracing hardware has been proposed over the past decade. This article is aimed at providing a timely survey on hardware techniques to accelerate the ray-tracing algorithm. First, a quantitative profiling on the ray-tracing workload is presented. We then review hardware techniques for the main functional blocks in a ray-tracing pipeline. On such a basis, the ray-tracing microarchitectures for both ASIC and processors are surveyed by following a systematic taxonomy.

Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Three-Dimensional Graphics and Realism-Display Algorithms; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism-Raytracing

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Ray tracing, GPU, spatial data structure, construction, traversal

ACM Reference format:
Yangdong Deng, Yufei Ni, Zonghui Li, Shuai Mu, and Wenjun Zhang. 2017. Toward Real-Time Ray Tracing: A Survey on Hardware Acceleration and Microarchitecture Techniques. *ACM Comput. Surv.* 50, 4, Article 58 (August 2017), 41 pages.
https://doi.org/10.1145/3104067

1 INTRODUCTION

As a central topic of computer graphics, rendering is the process of synthesizing an image from a three-dimensional scene. It has been the most popular means of human–computer interface and will likely continue to be so in the foreseeable future. Among many different approaches of rendering, physically realistic rendering plays a central role in modern graphics applications, exemplified by high-end gaming, movie special effects, and computer-aided design and manufacturing.

This work was partially supported by the China National Science Foundation under grant no. 61272085 and grant no. 61527812.
Authors' addresses: Y. Deng and Z. Li, School of Software Engineering, Tsinghua University, Beijing, China; emails: dengyd@mail.tsinghua.edu.cn, cqlzh1218@163.com; Y. Ni, S. Mu, and W. Zhang, Institute of Microelectronics, Tsinghua University, Beijing, China; emails: niyufei@aliyun.com, mus04ster@gmail.com, zwj@tsinghua.edu.cn.

ACM 0360-0300/2017/08-ART58 $15.00
https://doi.org/10.1145/3104067

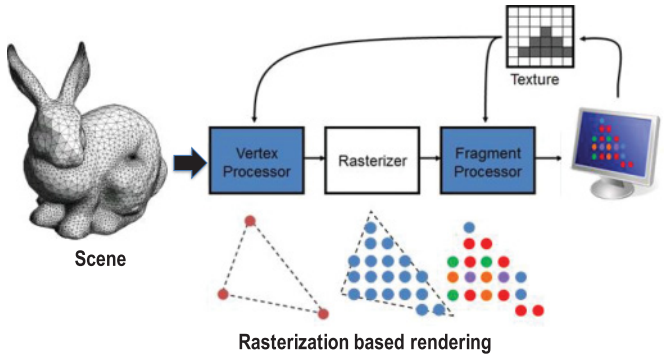


Fig. 1. Rasterization-based rendering.

Until now, the dominant rendering technology is rasterization (Akenine-Moller et al. 2008). Figure 1 illustrates its overall flow. An object in a scene to be displayed is divided into a mesh of basic geometric shapes (typically triangles), which are often designated as primitives. After the scene is created, the primitives are sent to a graphics pipeline for rendering processing. The first step is vertex processing; typical operations at this step include coordinate transformation as well as lighting and movement. Next, the primitives are fed to a rasterizer for spatial sampling. Each primitive is decomposed into a group of fragments. Then, the fragment-processing stage involves various operations to derive the color of every fragment. A Z-buffer algorithm is performed to derive an ordering of the fragments. The shading result of every pixel is sent to the frame buffer for output. Rasterization-based rendering is designed as a highly parallel process in which every step consists of many independent operations on different parts of a scene. This divide-and-conquer strategy significantly lowers the complexity of the underlying hardware by allowing modern graphics processing units (GPUs) to employ a large number of relatively simple cores and delivers performance in a more scalable manner.

Despite the effectiveness of the rasterization algorithm, its major problem is that it takes a "deconstructionism" approach and handles primitives separately (Smith 2014). However, most optical rendering effects—for example, shadow, reflection, and refraction—can only be generated by considering multiple primitives simultaneously (Dutre et al. 2006; Pharr and Humphreys 2010). It is also challenging for the rasterization technique to handle indirect illumination. In practice, complex techniques have to be devised to approximate these effects using rasterization. There are other techniques, nevertheless, which are more innately suited to delivering realistic images. Among these, ray tracing is an alternative mechanism for photorealistic rendering by offering a natural way to synthesize optical effects and indirect illumination (Glassner 1989; Pharr and Humphreys 2010). The ray-tracing algorithm is based on the emulation of the basic principle of vision. An observer—for example, a human being or a camera—sees an object because it reflects rays emitted from a light source into the eyes. The ray-tracing algorithm performs this process from the reverse direction so that rays not entering the eyes do not need to be considered. As illustrated in Figure 2, the idea is to emulate the process of shooting rays from the eye toward pixels on the display. If a ray intersects an object in the three-dimensional scene to be displayed, the shading of corresponding pixel is determined by the lighting and material at the given intersection point.

Ray tracing–based rendering engines have been widely used in high-end graphics applications, such as movie special effects (e.g., Christensen et al. (2006)) and advanced computer-aided manufacturing (e.g., Dietrich et al. (2007)). However, it is challenging to attain real-time ray tracing

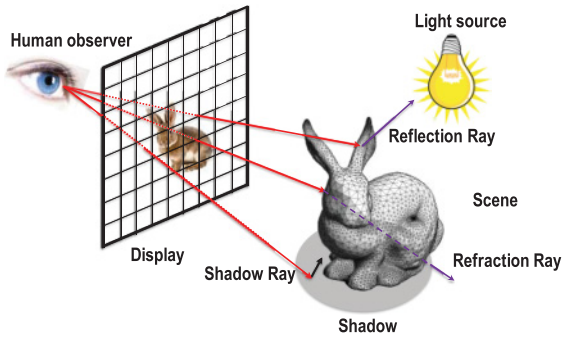


Fig. 2. Ray tracing–based rendering.

on consumer-level hardware platforms. Although the momentum toward the democratization of ray tracing has stirred many expectations in the past 20 years, it has yet to happen. A quick analysis reveals that at a typical display resolution, a throughput of three billion rays per second, is required for computing the intersection of rays and primitives in a scene (the details will be discussed in Section 2). On the other hand, today, even the latest GPU can only sustain a throughput of ~300 million rays per second on complex scenes (i.e., over one million primitives) due to the inefficiency in handling irregular memory traffic of ray tracing. Imagination recently announced a mobile GPU with dedicated hardware for ray traversal (McCombe 2014), but it is mainly used to enhance the rasterization effects. As a result, the only viable path to breaking the impasse and continuing the momentum of offering ever-growing visual effects on consumer-level platforms seems to be resorting to dedicated ray-tracing hardware. In the past decade, a whole spectrum of hardware and microarchitecture techniques has been proposed to enable real-time ray tracing. We envision that a wave of research efforts on ray-tracing hardware will blossom in a few years. This article is aimed at offering a timely review of the existing techniques and then identifying potential research directions for future work.

The remainder of this article is organized as follows. The ray-tracing algorithm is reviewed in Section 2. Section 3 briefly introduces current GPU microarchitectures, which are designed for rasterization-based rendering but can also perform ray tracing. In Section 4, we survey hardware techniques for major tasks in the ray-tracing algorithm. Section 5 covers the overall microarchitectures for ray tracing. We conclude this article and propose future research directions in Section 6.

2 RAY TRACING ALGORITHM AND DATA STRUCTURES

While the basic idea can be traced back to Euclid (315–250 BCE) (Reif et al. 1994), the ray-tracing algorithm was first proposed by Appel (1968). The earliest version was designated as the ray-casting algorithm because it only shoots rays from the eye and then checks the intersection. Clearly, it only allowed a roughly approximated image of the scene to be displayed. Whitted (1980) introduced a ray-tracing algorithm to consider more complex illumination effects. Upon hitting a primitive in the scene, a ray may generate so-called secondary rays, including reflection, refraction, and shadow rays. Such rays can then be processed in a recursive manner. The Whitted algorithm is the major ray-tracing algorithm for real-time applications. A closely related concept is path tracing, which is a family of techniques for more indirect illumination effects. Path tracing can be realized by shooting rays in random directions to capture the global illumination effects. The number of rays has to be sufficiently large so that a good stochastic approximation can be derived. In this work,

we focus on ray tracing, but all hardware techniques reviewed in this work can also be applied to path tracing.

2.1 Ray Tracing Algorithm

Algorithm 1 lists the pseudo code of the Whitted ray-tracing algorithm. Given a scene to be displayed, the objects in the scene are already subdivided into a mesh of geometric primitives by the scene-generation procedure. The Whitted ray-tracing algorithm takes the set of primitives as input. The major body of the algorithm is a loop through all pixels. One ray is emitted from the eye (or camera) toward each pixel. In practice, we actually cast several rays to random positions inside a pixel for antialiasing. The rays coming from the eye point are designated as primary rays. In Lines 6 to 14, the algorithm loops through all primitives and identifies the nearest one seen from the observer. If a ray does not hit anything, then it contributes nothing to the pixel's shading. When the closest primitive can be located, secondary rays—such as shadow rays, reflection rays, and refraction rays—have to be created and ray traced. A straightforward implementation of the pseudo code has a complexity of $O(NM)$, where N is the number of rays and M is the number of primitives. Given a complicated scene with millions of primitives, the complexity is too overwhelming. The acceleration structure, therefore, is another key concept under the framework of ray tracing and will be elaborated in the next section.

2.2 Acceleration Structure

The ray-tracing algorithm needs to identify the nearest intersected primitive of each ray. A brute force approach would require checking every primitive for each ray. Acceleration structures are designed to lower the complexity of the search process. The basic idea is to organize the primitives into a hierarchical spatial structure by adapting to the distribution of the primitives. With such a structure, the closest intersection of primitives by a ray can be efficiently identified by fast culling the irrelevant space. The most straightforward acceleration structure is a grid regularly partitioning the space (Fujimoto et al. 1986). Such a simple approach, however, cannot guarantee efficiency for scenes with irregular distribution of primitives because a huge number of primitives can be allocated into a single cell. As a result, acceleration structures should be able to adapt to the distribution of primitives. The most commonly used of these structures include the kd-tree (Havran 2000; Hapala and Havran 2011) and bounding volume hierarchy (BVH) (Clark 1976). The remainder of this section briefly reviews both structures.

2.2.1 Kd-tree. The kd-tree, or k-dimensional tree, was originally proposed as a space-partitioning data structure to indexing points in k-dimensional space (Bentley 1975). The kd-tree is a special case of binary space partitioning tree (Fuches 1980). It is built by recursively partitioning the space into two half-spaces along a given axis. The cut position is selected such that the two halves have relatively equal numbers of objects. When used for ray tracing, it is generally not the best strategy to assign an equal number of primitives to two half-spaces during cutting a space. A better heuristic is to consider both the number of primitives and the probability of a random ray hitting either half-space. This is actually the basic idea of surface area heuristic (SAH) (Goldsmith and Salmon 1987; MacDonald and Booth 1989). It is relatively costly to accurately compute SAH; thus, various approximation heuristics have been proposed (e.g., Wald and Havran (2006), Wald (2007), and Zhou et al (2008)). Figure 3 is an example distribution of triangular primitives in a two-dimensional plane and its respective kd-tree.

Recently, a fully parallel construction algorithm was proposed by Li et al. (2017). In these works, a maximum level of parallelism is enabled by starting from a Morton code–based ordering of

ALGORITHM 1: Whitted Ray Tracing Algorithm

```
1: function TraceImage
2: Input: A scene S consisting a set of primitives
3: Output: Rendering of S
4: begin
5:          for each pixel P on the display
6:                    Create ray R from eyepoint passing through P
7:                    Initialize NearestT to INFINITY and NearestPrimitive to NULL
8:                    Color C = TraceRay(R,S NearestT)
9:                    Shade the current pixel with C
10:         end for
11: end function
12:
13: function TraceRay
14: Input: A scene S consisting a set of primitives, Intersection with the nearest primitive NearestT
15: Output: Shading result for the current ray
16: begin
17:         for every primitive in scene
18:                   if ray intersects this primitive and if intersection point t is less than NearestT then
19:                             Set NearestT to t of the intersection
20:                             Set NearestPrimitive to this primitive
21:                   end if
22:         end for
23:
24:         if NearestPrimitive is NULL then
25:                   Return background color
26:         else
27:                   C = Black;
27:                   Shoot a ray to each light source to check if in shadow
28:                   if surface is reflective, generate a reflection ray R1
29:                             C+ = TraceRay(R1, S NearestT)
30:                   if surface is transparent, generate a refraction ray R2
31:                             C+ = TraceRay(R2, S NearestT)
32:                   C+ = Shading(NearestPrimitive, NearestT) //calling shading function
33:                   return C;
34:         end if
35: end function
```

primitives (Morton 1966) and simultaneously working on all leaf nodes or internal nodes to form a hierarchy in a bottom-up manner.

With graphics primitives organized into a kd-tree, a ray-tracing program performs a traversal operation for each ray. Starting from the root node, we recursively check how the input ray intersects with the respective space. Since the spaces associated with both child nodes can be intersected, a stack is maintained to record the related nodes. When a leaf node is arrived at, we iterate through all primitives inside the node to determine the closest one intersected by the current ray. This stage is often termed an intersection test. The traversal algorithm does not well match the single-instruction, multiple-data (SIMD) hardware of modern GPUs (the architecture of GPU will be discussed in Section 3) since different rays may follow varying execution paths (Aila and Laine 2009; Santos et al. 2012). Thus, novel traversal algorithms—such as stack-less traversal and ray

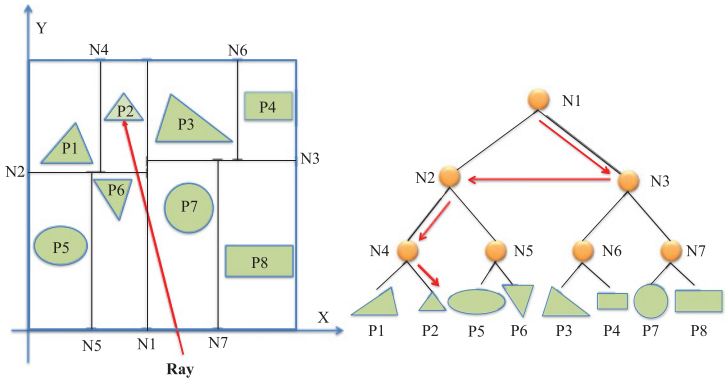


Fig. 3. An example kd-tree.

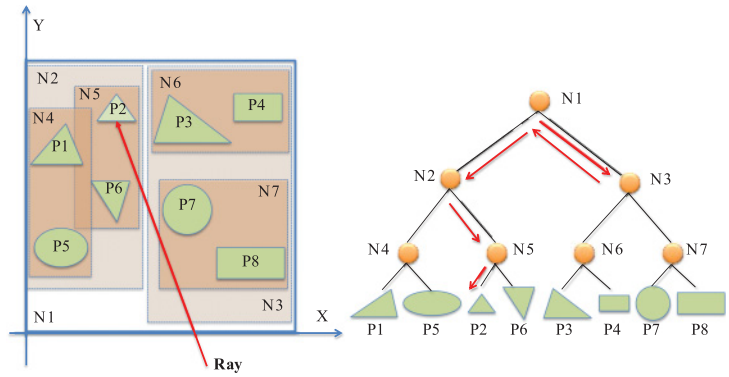


Fig. 4. Illustration of an example scene and its respective BVH.

reordering—were introduced to improve parallel execution efficiency (Popov et al. 2007; Hapala and Havran 2011). Refer to related literature (Aila and Laine 2009; Hapala and Havran 2011; Santos et al. 2012) for a more detailed treatment of traversal efficiency.

2.2.2 BVH. BVH also has a tree structure with each node associated with a bounding box, designated as axis-aligned bounding box (AABB). A key difference of a BVH from a kd-tree is that nodes at the same level of a BVH may have the respective AABBs overlapped. Early BVH construction algorithms followed a simple bottom-up approach. Starting from primitives, a first-level clustering generates a group of leaf nodes. Then, the nodes are clustered upward level by level to form a hierarchy.

The simple clustering approach leads to a fast construction process. Researchers proposed various approaches to improve the quality of BVH. Today, most BVH constructors are based on the bin-based algorithm in which the Morton code is used as a basis for clustering. A typical BVH construction procedure is listed in Algorithm 2 (Lauterbach et al. 2009; Garanzha et al. 2011). It takes a top-down processing flow. At each level, primitives belonging to a node are partitioned into a given number of bins along every axis and pick up the best direction and position for a cut. Again, there are many metrics to evaluate the effectiveness of a cut. Among these metrics, the SAH is usually considered as the best one, but a few recent works managed to derive good partition quality to avoid the relatively expensive heuristic and purely depend on the Morton code (Pantaleoni and

ALGORITHM 2: Bin-Based Top-down BVH Construction Algorithm

```
1: Input: Root node root, S = {p1, p2,..., pn} is the set of n primitives in k-dimension.
2: Output: A BVH starting at root node
3: begin
4:          buildBVH(root, S)
5: end
6:
7: Function buildBVH(node curNode, primitive set curS)
8: begin
9:          if curS is small enough
10:                 mark curNode as a leaf node and assign curS to n
11:                 compute the bounding box of curNode
12:         return
13:         bestCut ← ∞
14:         Cut curS into a given number groups evenly along x-, y- and z-axes
15:         Update bestCut by all possible cut in the K groups
16:         Split curS into S+ and S- according to bestCut
17:         create a new node left and call buildBVH(left, S+)
18:         create a new node right and call buildBVH(right, S)
19: end
```

Luebke 2010; Karras 2012; Li et al. 2017). Pantaleoni and Luebke (2010) systematically improved the performance of BVH construction by introducing efficient procedures to handle the Morton code and take advantage of the inherent coherence among primitives. Karras and Aila (2013) proposed the idea of reconstructing an existing low-quality BVH by concurrently working on local tree structures. The idea is amenable to massively parallel processing and has been demonstrated to be highly efficient.

Algorithm 3 shows a BVH traversal algorithm, which follows a straightforward recursive processing flow that is similar to its kd-tree equivalent. A large body of research has been dedicated to improving the efficiency of BVH traversal. Refer to Aila and Laine (2009) and Aila et al. (2013) for the implementation and evaluation of state-of-the-art traversal techniques. It is still an open problem whether the kd-tree or BVH enables a higher level of traversal efficiency. An inherent problem of BVH is that two nodes at the same level may overlap in space and thus a large number of nodes are visited before locating a hit. This is not a problem for the kd-tree for point data, but extra processing steps are needed when handling primitives. Similar to the case of the kd-tree, the BVH-based traversal algorithms also have an intersection test stage.

2.3 Ray-Tracing Pipeline and Computing Requirements

Today's graphics rendering process is organized into a graphics pipeline (Khronos 2013a; Microsoft 2013; Sugerman et al. 2009). Similarly, we can generalize the ray-tracing processing flow based on the discussion in the previous two sections as a pipeline. Figure 5 is an illustration of the major steps in the ray-tracing pipeline.

The first stage of a ray-tracing pipeline is the construction the spatial structure, which is typically stored in the global memory of the GPU. For static scenes, the acceleration structures can be reused for many frames. Given dynamic scenes, however, the acceleration structures have to be built or updated for each frame. The second stage of the ray-tracing pipeline is the traversal stage, which can be further decomposed into several steps. The first step is to generate primary rays that are emitted from the eye point and cast toward every pixel. In the second step, each ray

ALGORITHM 3: BVH Traversal Algorithm

```
1: Input: Root node root, Input ray
2: Output: Intersection results
3: begin
4:              traverse (ray, root)
5: end
6:
7: Function traverse (ray curRay, node curNode)
8: begin
9:              if curRay does not intersect AABB then
10:                             return no object intersected
11:             end
12:             if curNode is a leaf and not empty then
13:                             list all the objects in curNode ------> LST Operation
14:                             intersect curRay with each object ------> INT Operation
15:                             if any intersection exists inside the leaf then
16:                                             return closest object to the ray origin
17 :                            end
18 :            else
19 :                            if curRay hits curNode.left and does not hit curNode.righ
20 :                                            traverse (curRay, curNode.left)
21 :                            else if hits curNode.right and does not hit curNode.left
22 :                                            traverse (curRay, curNode.right)
23 :                            else                                                        } TRV Operation
24 :                                            traverse (curRay, curNode.left)
25 :                                            traverse (curRay, curNode.right)
26 :                            end
27 :            end
28 : end
```

has to traverse the acceleration stage in a level-by-level fashion. The parallelism of this stage is abundant because every ray is independent of others. In fact, ray tracing is believed to be a classical example of embarrassingly parallel applications (Bienia et al. 2008). However, it is challenging to fully unleash the inherent parallelism due to the complexity of the ray-tracing workload. On the one hand, the SIMD hardware in which a group of parallel tasks always executes the same instruction proved to offer superior performance for rasterization-based rendering, but the advantage shrinks when handling control divergence (i.e., parallel tasks in a SIMD group follow diverse execution paths) and memory divergence (i.e., parallel tasks in a SIMD group access different addresses). On the other hand, theoretically, multiple-instructions multiple-data (MIMD) is arguably the best architecture for ray tracing by completely removing the overhead introduced by SIMD execution. Nevertheless, MIMD hardware is more costly than its SIMD counterparts in terms of silicon real estate per core. Hence, given the same silicon budget, MIMD hardware will have a lower peak throughput. The third step in the traversal stage is the intersection test. This is a compute-intensive stage because a large number of primitives have to be processed and each one needs around 10s of floating-number computations. Upon a hit, this step also needs to generate secondary rays. The final step computes the shading results of pixels.

The graphics pipeline on computers and mobile platforms is responsible for real-time rendering. Today's videos and games typically require a frame rate of 60 frames per second (FPS) or higher

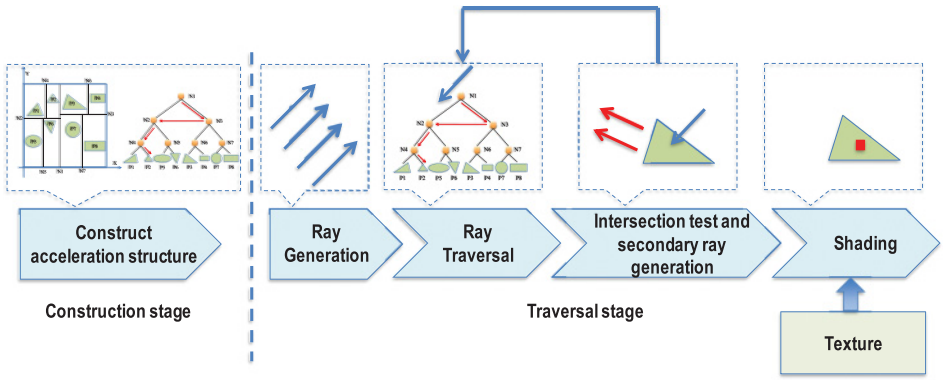
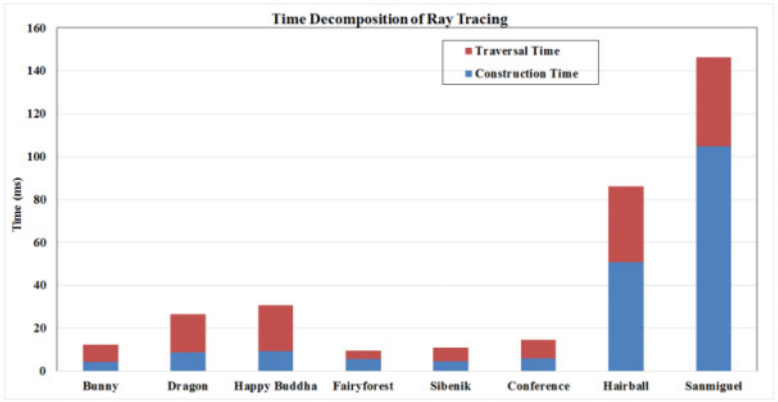


Fig. 5. Ray-tracing pipeline.

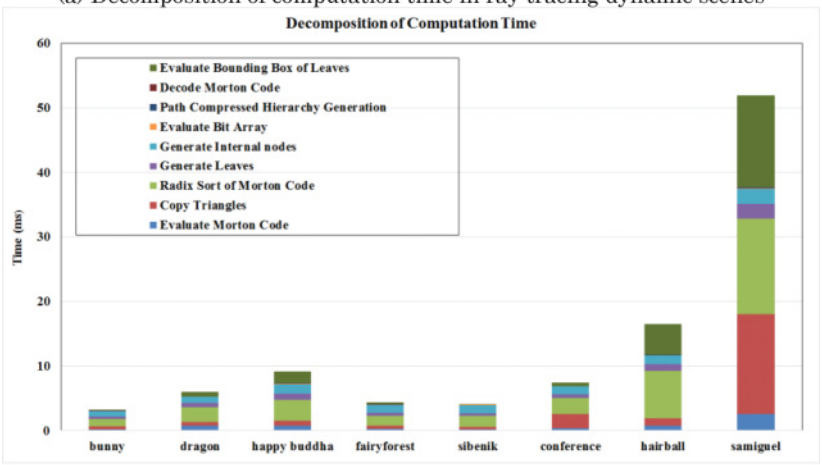
(Hoffmann et al. 2006), but an FPS of 30 is a minimum requirement. Meanwhile, a high-end display has a resolution of 2560 × 1440, or around 3.7M pixels. For each pixel, we shoot 10 rays to minimize aliasing. Then, totally, we have 37M primary rays. When a ray hits a primitive, multiple reflection and/or refraction rays will be generated. Such second rays will incur newer rays and usually we set a limit on the depth of recursion. For a reasonable rendering quality, a conservative estimation is that a primary generates 10 secondary rays, on average. Now, the ray count is 370M. For a frame rate of 30, we need a processing throughput of 10G rays per second. The above analysis is just for the ray traversal stage, while the construction stage is also compute intensive. For instance, the construction process needs to sort the primitives according to their coordinates, while a typical scene in today's graphics applications has over 1 million primitives.

To better understand the ray-tracing workload, we performed profiling by using a state-of-the-art ray tracer (Li et al. 2017). The ray tracer proposed by Li et al. (2017) integrates a highly efficient, fully parallel construction algorithm that applies to both the kd-tree and BVH and a leading-edge ray-traversal procedure. We used six commonly used scenes, Bunny (69K triangles), Sibenik (80K triangles), Fairy forest (169K triangles), Conference (283K triangles), Dragon (871K triangles), Happy Buddha (1.08M triangles), Hairball (2.9M triangles) and Samiguel (10.5M triangles). The kd-tree was chosen as the acceleration structure. Figure 6(a) shows the decomposition of construction and traversal time[1]. A further decomposition of construction time among various computing kernels is shown in Figure 6(b). The construction algorithm consists of seven main steps. The sorting step is the most time consuming. Finally, a breakdown of traversal time is depicted in Figure 6(c). Given a ray, the traversal time consists of both the time for traversing the kd-tree and the time for testing if the ray intersects a triangle. An effective practice of GPU-based ray tracers is to merge the traversing and intersection test codes into a single kernel (Aila and Laine 2009) and launch a large number of concurrent threads. As a result, the kd-tree traversing and ray–triangle test are actually two overlapping processes when considering all threads. In order to understand the computation demand, we artificially divide the two steps into two kernels that are activated sequentially and report the respective kernel times in Figure 6(c). All results are collected on a NVIDIA TITAN graphics card featuring 2880 CUDA cores (organized in 15 streaming multiprocessors) and 6GB GDDR5 memory.

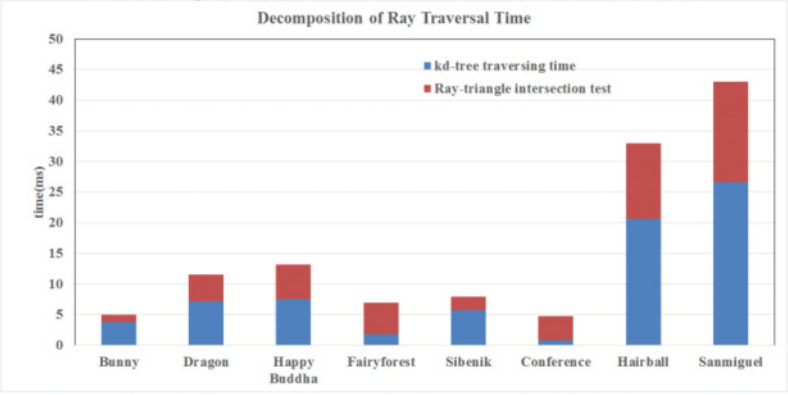
[1]The time decomposition was measured with a state-of-the-art ray tracer (Li et al. 2017), which uses a kd-tree as the acceleration structure. It is based on a fully parallel kd-tree construction procedure; thus, the construction time is relatively short. Only primary rays are used during traversal.



(a) Decomposition of computation time in ray tracing dynamic scenes



(b) Decomposition of kd-tree construction time w.r.t. scenes



(c) Decomposition of kd-tree traversal time w.r.t. scenes

Fig. 6. Profiling of ray-tracing workload.

In addition to computing, ray tracing also poses a unique challenge to memory system design. During construction, the major consumer of memory bandwidth is the sorting (typically, radix sort) process featuring an irregular memory access pattern. In the traversal stage, it is the loading of the acceleration structures that dominates memory usage. Although the total memory traffic is relatively moderate (Aila et al. 2013), the access pattern is highly random. It is especially serious for modern GPUs because they are designed to quickly access a large chunk of continuous memory addresses. Accordingly, many techniques have been developed to improve the coherence of parallel ray-tracing tasks (Wang and Deng 2013). The key idea is to group rays with (potentially) similar traversal behaviors and then issue the memory requests in parallel. For an intensively optimized traversal procedure, it is claimed that the available memory bandwidth on a high-end GPU is no longer a bottleneck. Besides the irregular access pattern, the memory behavior of the ray traversal process is unique in that the acceleration structure is read-only and the rendering results are both write-only and write-once. As a result, the read efficiency is critical, while the write process can be performed with a write-around policy.

3 COMMERCIAL PLATFORMS FOR RAY TRACING

Today's computers, mobile platforms, and game consoles perform rendering with a GPU, which has become the most powerful single-chip computing machine in order to meet the ever-growing demand for a better visual experience. With the introduction of general purpose GPU programming models, such as CUDA (NVIDIA 2013) and OpenCL (Khronos 2013b), GPUs have evolved into highly programmable platforms and received dramatic success in accelerating scientific and engineering computations (Blythe 2008; Nickolls and Dally 2010; Deng and Mu 2013). Ray-tracing applications significantly benefit from the increasing computing power and programmability of GPUs. For instance, the Optix ray-tracing framework allows a ray-traversal throughput of up to 300M rays per second (Parker et al. 2010; McAllister et al. 2014).

GPUs can be classified into two categories, general purpose GPUs and mobile GPUs[2]. These two categories of GPUs are designed under different sets of constraints. The former category allows a higher hardware budget to pursue high performance, while the latter is constrained by a much tighter power budget. Currently, NVIDIA and AMD are two major manufacturers of high-end general-purpose GPUs and Intel has a considerable market share in middle-end and low-end GPUs. The leading players in the mobile GPU market include Imagination Technologies (i.e., PowerVR), ARM (i.e., Mali), and Qualcomm (i.e., SnapDragon). In this section, we review the GPU microarchitectures and their general-purpose programming models. We choose NVIDIA's Pascal (NVIDIA 2016), Imagination's PowerVR Rogue (Imagination 2014), and Intel's MIC processor as the representatives of high-performance GPUs, mobile GPUs, and computing accelerators.

3.1 NVIDIA Pascal GPU

The Pascal GPU is NVIDIA's latest generation of GPU architecture at the time of this writing. It is able to offer over 5 and 10 TFLOPS of double- and single-precision throughputs, respectively. The overall microarchitecture is illustrated in Figure 7(a). Like previous generations of NVIDIA GPUs, Pascal consists of a number[3] of streaming multiprocessors (SMs), which are the basic blocks of program execution. All SMs share a unified level-2 (L2) cache. The on-chip L2 cache has a capacity of up to 4MB and is connected to the off-chip global memory through eight memory controllers. Thanks to the GDDR5 memory (JEDEC 2009) as well as a wide memory bus (384b), Pascal supports

[2]Laptop GPUs are positioned in between and more oriented to general purpose GPUs.
[3]The respective number varies with GPU models.

Fig. 7. Pascal GPU microarchitecture.

an off-chip memory bandwidth as high as 720GB/s. A GPU communicates with the CPU via a PCI Express 3.0 (PCIe) bus (PCI-SIG 2002).

The computing resource of a Pascal GPU consists of twenty streaming multiprocessors (SMs). The microarchitecture of the SM is shown in Figure 7(b). The internal logic of the SM is allocated into four processing blocks, as illustrated in Figure 7(c). Each processing block has 32 CUDA cores for floating point and integer computations, 8 load/store (LD/ST) units for memory operations, 32 special functional units (SFUs) for complex mathematical functions, and 8 texture units for texture data access. These elements share a common instruction fetch and decoding logic, but are scheduled as three independent groups. In each instruction cycle, two groups receive decoded instructions from two instruction dispatch units. The SM follows the Single Instruction Stream, Multiple Threads execution model, which can be considered as a combination of SIMD and multithreading. A GPU program typically launches a large number of threads for high-throughput computing. Threads assigned to an SM are organized into warps of 32 threads. A warp of 32 threads is simultaneously processed by the SIMD hardware of an SM. Different warps share the SM in a multithreaded fashion. The SM is equipped with a 16K, 32b register, a 96KB shared memory (i.e., scratchpad memory) and an L1 instruction cache. Thread mapped to an SM has its own register files; thus, the context switching of warps can be done in one single-instruction cycle. The SM also supports instruction level parallelism.

3.2 Imagination Power Rogue GPU

Today's mobile GPUs are becoming powerful computing platforms. Figure 8 illustrates the architecture of Imagination's latest PowerVR series 6 GPU (codenamed Rogue) (Imagination 2014). It is equipped with a vertex data master, a pixel data master, and a computer master, which are front ends for vertex, fragment, and computing tasks, respectively. The former two take OpenGL

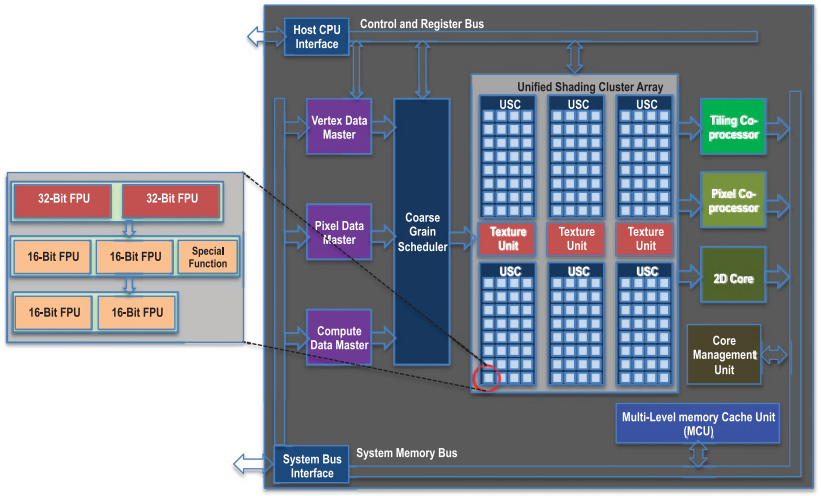


Fig. 8. Imagination Power series 6 (Rogue) GPU microarchitecture (adapted from Imagination (2014)).

programs as input, while the latter executes OpenCL programs. A global scheduler then dispatches the decoded instructions to a unified shading cluster (USC) array consisting of up to 6 USCs. The data computed by USCs are fed to a few dedicated hardware modules for final rendering. The GPU has two buses, one for control and configuration and the other is a memory bus connected to the system bus of the application processor in which the GPU is integrated. The USC is often considered as the equivalent of a multiprocessor in an NVIDIA GPU, but it lacks the capability of fetching and issuing instructions. It is composed of 32 cores; thus, the PowerVR series 6 GPU is often regarded as having 192 cores. The 32 cores execute instructions in a SIMD manner. Different from the NVIDIA GPU, whose core has a single stage consisting of an integer unit and a floating-point unit (FPU), a core of USC has several stages of FPUs. The first stage has two 32b FPUs, while the other two stages each have two 16b FPUs. These stages can be independently scheduled for better taking advantage of instruction-level parallelism.

3.3 Intel Many Integrated Core Architecture (MIC)

Intel's MIC many-core processor has its origin rooted from the Larrabee architecture (Seiler et al. 2008). Larrabee was designed as a hybrid graphics and high-performance computing (HPC) platform with a new level of ray-tracing performance. Later, the architecture was repositioned as a pure computing platform designated as Xeon Phi Many Integrated Core (MIC) Architecture. At this writing, the latest product of the Xeon Phi series is Knight's Landing, with a die size of ~683mm^2 fabricated in a 14nm process (Reinders 2014). The MIC features a group of interconnected CPU cores. The CPU core in the MIC is developed on the basis of an in-order Intel Pentium Core (Intel 2013) with both scalar and vector computing facility. It is able to issue two (noninterdependent) instructions in one cycle to two separate pipelines. Either pipeline can use a single unit from the following computing logics: two integer scalar ALUs, a floating processing unit, and a 16-wide floating point (32b) SIMD unit. The 512b-wide SIMD unit is able to execute 16 single-precision operations or 8 double-precision operations in a single instruction. Each core has its

own private L2 cache and cache coherence is maintained by dedicated hardware[4]. With 72 cores organized as 36 tiles, it achieves a computing throughput of over 1 TFLOP when executing the LINPACK benchmark. Wald et al. (2014) proposed a high-performance ray-tracing solution based on the MIC processor.

4 HARDWARE DESIGN FOR RAY-TRACING BUILDING BLOCKS

This section surveys the literature on designing hardware for each stage in the pipeline illustrated in Figure 5. Beginning from the numerical system, we will introduce dedicated hardware design for constructing acceleration structure, ray generation, traversing acceleration structure, and testing ray–triangle intersection. Another essential part of ray-tracing hardware is the design of memory system, which is highly dependent on the overall microarchitecture design. Therefore, we will review the memory system design along with the microarchitecture integrating these building blocks in Section 5.

4.1 Numerical System for Ray Tracing

The ray-tracing algorithm poses considerable challenges to the target hardware. During the construction process, a sorting process involving comparing floating-point numbers is usually the bottleneck. During the traversal process, the algorithm needs a few computations to determine the next node to visit. After hitting a leaf node, tens of floating-number operations have to be performed on each primitive. Such computations are usually performed on single-precision floating numbers (Wald et al. 2007). FPGA-based ray-tracing hardware (e.g., Purcell et al. (2005)) often exploits floating-point IPs following the IEEE 754 standard (IEEE 2008). Meanwhile, a few works use fix-point numbers or a reduced precision (e.g., 24b instead of 32b) to save hardware cost (Carr et al. 2002). On the other hand, it should be noted that the numerical values that can be represented by valid 32b codes defined by IEEE 754 are not evenly distributed (IEEE 2008). Given normalized values between 0 and 1, there are fewer and fewer values that can be represented when getting closer to 1. In graphics applications, it is a common practice to normalize the coordinates of primitives to the range of [0, 1]. As a result, the corresponding resolution of numerical values gets worse when moving off the origin of the scene (Heinlya et al. 2009). From this point of view, the fix-point number system has the advantage of being able to represent numerical values in a uniform manner. In addition, the fixed-point numbers enable simpler hardware designs and computing procedures (Hanika and Keller 2007; Heinlya et al. 2009). Keely (2014), Vaidyanathan et al. (2016), and Liktor and Vaidyanathan (2016) performed a systematic study on the precision requirement of ray tracing and proposed a reduced precision solution that can be integrated into current GPUs.

4.2 Hardware for Acceleration Structure Construction

For static and offline rendering applications, the construction of acceleration structures is typically performed by an instruction processor instead of ray-tracing acceleration hardware (e.g., Schmittler et al. (2002), Schmittler (2006), and Nah et al. (2011)). However, the growing applications of rendering dynamic scenes generate a strong demand for online construction and updating of the acceleration structure. Recent breakthroughs on fully data-parallel construction algorithms (Karras (2012) and Li et al. (2017)) also make it feasible to design efficient construction hardware.

4.2.1 Kd-tree Construction. To the authors' knowledge, Woop et al. (2005) proposed the first solution toward dynamically updating kd-trees with a customized processor. The so-called update

[4]Currently, heterogeneous processors, such as the AMD Accelerated Processing Unit (APU), support coherence between CPU and GPU cores, but cannot maintain coherence among multiple cores of a GPU.

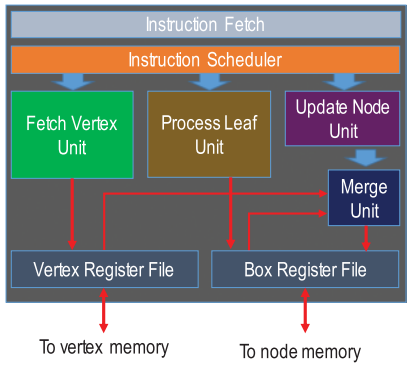
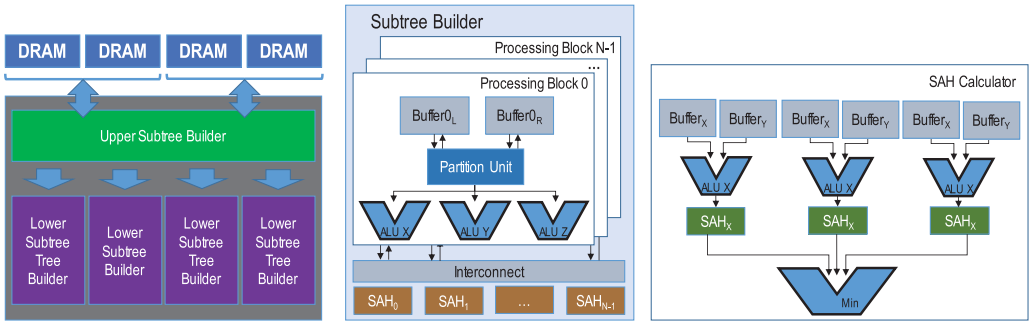


Fig. 9. Kd-tree update processor (adapted from Woop (2006)).

processor has dedicated units for various tasks in a construction flow but still follows the overall structure of an instruction processor as illustrated in Figure 9. It is responsible for revising kd-trees when a respective frame involves only small changes. The update processor supports three special instructions—"update node," "process leaf," and "load vertex"—which cover the basic operations of construction. An instruction fetch unit reads instructions of a construction program. An instruction scheduling unit checks if the arguments of fetched instructions are already loaded into the registers and then issues the ready instructions. Upon the "load vertex" instruction, a fetch vertex unit loads the coordinates of triangle vertices and puts them into a vertex register. The "process leaf" instruction loads the bounding box of a leaf node into a box register. The "update node" instruction activates a merge unit to recompute the bounding box of the current node by considering the changed vertex coordinates and writes the results back to memory. The update processor was implemented on FPGA and tested on a set of small scenes. The major bottleneck is the external DRAM. In fact, the update processor has a DRAM usage ratio of about 99%, which means that the DRAM is being accessed in almost every cycle. On the other hand, DRAM efficiency, which is the ratio of the number of cycles for actual data transfer over the total number of cycles with memory activity, is only 20%. This certainly suggests an opportunity for optimization by improving the memory scheduling mechanism and/or adopting a multibank DRAM organization to better hide the overhead of DRAM operations.

Given dynamic rendering applications with frequent changing scenes, the updating hardware cannot sustain a satisfying level of performance. Therefore, researchers recently proposed dedicated construction hardware to pursue higher performance in tracing dynamic scenes. Nah et al. (2014) recently proposed a Tree Building Unit (TBU) by adopting a binning-based construction algorithm (e.g., Shevtsov et al. (2007b) and Djeu et al. (2011)). The algorithm allocates primitives to a regular grid and generates partitions only at the boundary of the cells in the grid. The TBU applies the binning-based algorithm to construct upper layers where parallelism is limited. A more accurate and expensive algorithm is used to build lower layers of the kd-tree. As upper layers offer limited parallelism, reducing the computing effort for them leads to considerable savings in overall construction time. The building unit has a binning pipeline that clusters triangles into groups with regard to vertex coordinates. The sorting pipelines compute partitioning results with the SAH heuristic. The TBU stores the data for tree construction in the on-chip SRAM to reduce off-chip memory traffic. The TBU consumes 1.6mm^2 of silicon estate when fabricated in a 28nm process as an ASIC. It outperforms the results derived on an earlier-version NVIDIA GPU (Hou et al. 2011; Wu et al. 2011). However, the construction speed does not have an advantage when compared with the latest fully parallel GPU-based algorithms (Li et al. 2017).



(a) Overall architecture (b) Subtree builder architecture (c) SAH calculator

Fig. 10. SAH optimized BVH construction hardware unit (adapted from Doyle et al. (2013)).

The binning-based construction algorithm incurs a minor loss in the quality of resultant kd-trees (Nah et al. 2014). To address the problem and enable an even higher construction throughput, Liu et al. (2015) proposed a hardware accelerator, FastTree. The FastTree hardware adopts the fully parallel construction algorithm proposed by Li et al. (2017). The construction algorithm starts from a conceptually complete binary tree, which corresponds to a regular division of the space into grid cells and then concurrently works on all leaf nodes in a bottom-up manner. An advantage of the algorithm is that the major operations can be mapped to a group of prefix-sum and radix sort units, which is efficiently implemented into hardware by Liu et al. (2015). FastTree was validated by an FPGA prototype and evaluated as an ASIC implementation. Results of experiments show that it outperforms the aforementioned TBU (Nah et al. 2014) by a factor of nearly 4X at a similar area and power budget.

4.2.2 BVH Construction. Early works on BVH construction typically follow a bottom-up approach by hierarchically clustering triangles (Arvo and Kirk 1989). Recent works (e.g., Wald (2007), Garanzha et al. (2011), Wald (2012), and Aila et al. (2013)) resort to the binning approach proposed by Wald (2007) to improve the traversal speed. The key idea is to allocate primitives to regularly placed bins and then identify a best cut at the boundary of bins. Here, the best cut is evaluated in terms of SAH cost. The SAH-driven BVH construction algorithm of Wald (2007) consists of both a vertically parallel stage and a horizontally parallel stage. The former stage deploys one thread to work on the topmost subtree starting from the root, while the latter stage uses multiple threads to concurrently handle subtrees descendent from the top subtree. This arrangement maximizes the available parallelism.

Doyle et al. (2013) proposed a hardware unit for BVH construction by following the algorithm introduced by Wald (2007). As illustrated in Figure 10(a), it is composed of an upper subtree builder unit and multiple lower subtree builder units, corresponding to the two stages of the abovementioned algorithm. Except for memory interfaces, both builders have similar structures by following the binning-based construction flow. Figure 10(b) shows the basic structure of the subtree builder. It features one or more processing blocks, with each built around a partition unit. The partition unit in the upper subtree builder reads primitives from a pair of off-chip DRAMs, which serves as input and output storage alternatively. A partitioning unit is connected to three binning units corresponding to the three axes: x, y, and z. A binning unit assigns primitives into bins along the respective axis according to the center of the axis-aligned bounding box. After binning, an SAH calculate unit, with its structure shown in Figure 10(c), derives the cost for every possible partition and returns the minimum. The construction unit has multiple lower subtree builders. As soon as

the upper subtree builder has a lower subtree to construct, an idle lower subtree builder is activated. It reads primitives from internal buffers. The partition results are computed by a group of SAH units in a multithreaded manner, which means that multiple subtrees can be concurrently processed in the subtree builder. When implemented in a 65nm process, a subtree builder will have a die area of 31.88mm^2 and an upper builder takes 5mm^2. When clocked at 500MHz, the BVH construction hardware outperforms a respective CPU-based software implementation by a factor of around 10. However, it is 2 to 4 times slower than a state-of-the-art GPU implementation of a construction algorithm (Garanzha et al. 2011). In addition, such a dedicated hardware BVH construction unit can only support a single-construction algorithm. An overhaul is required to take advantage of recently proposed fully parallel construction algorithms, such as Karras (2012) and Li et al. (2017). In addition, the dedicated hardware cannot be used for other GPU computations. Such a usage of silicon estate needs further justification for commercial GPU products.

4.3 Ray Generation

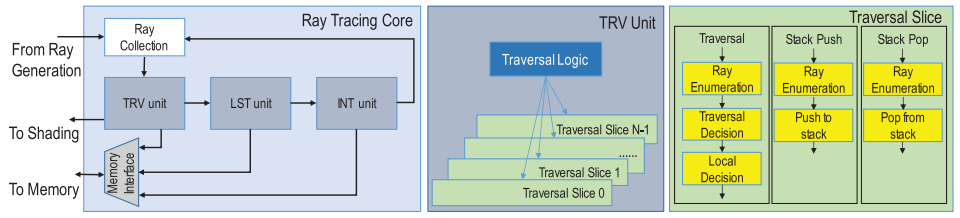
The ray-generation stage is perhaps the simplest one in the ray-tracing pipeline. It is responsible for generating primary and secondary rays and then issuing the rays to the traversal stage. Since secondary rays have to be created according to the specific intersection scenario, the respective generation process is often performed as a by-product of the ray-intersection stage. No matter where the rays are generated, they all need to be buffered by the ray-generation unit so that they can be dispatched to ray-traversal hardware in a batched manner.

Nah et al. (2014) introduced dedicated hardware for ray generation. It generates primary rays ordered by Morton code, which are calculated by a counter and a group of shift registers. The generation unit has a 16-entry to store reflection and refraction rays. Shadow rays are created by the shading hardware and transferred to the generation unit.

The ray-generation unit is often equipped with a ray-accumulation unit or a ray buffer, which stores rays that have pending memory requests (e.g., Nah et al. (2011) and Lee et al. (2013a)). Such a mechanism can be associated with a multithreading paradigm to hide memory latency.

4.4 Hardware for Ray Traversal

The ray-traversal process, which has to be performed on both dynamic and static scenes, is another performance bottleneck of ray tracing–based rendering applications. A careful checking of Algorithm 3 reveals that every ray-traversal algorithm is built around three basic operations, that is, *traversal* (TRV), *list* (LST), and *intersection* (INT). The TRV operation checks how a ray goes through the space corresponding to a node in an acceleration structure and then determines the next node to visit. The remainder of this section focuses on TRV. The LST operation is activated when the current ray hits a leaf node to enumerate all primitives allocated to the node. It can be straightforwardly implemented as a state machine issuing a sequence of memory requests. In many ray-tracing architectures, LST is integrated with the INT unit. The INT operation is performed to determine how the current ray intersects with a primitive and its design will be elaborated in Section 4.5. A large body of research on traversal hardware architectures has been proposed to design efficient architecture and configuration of the above basic operations. To the authors' knowledge, pioneering studies on ray-tracing hardware design include Tigershark (Humphreys and Scott 1996), 3DCGiRAM (Kobayashi et al. 2001), AR250/AR350 (Hall 2001), reconfigurable ray tracer (Todman and Luk 2001; Fender and Rose 2003) and SaarCor (Schmittler et al. 2004; Schmittler 2006). With only a few exceptions (e.g., Steffen and Zambreno (2009) and Steffen and Zambreno (2010)), the existing works typically use kd-tree and BVH as acceleration structures. In this section, we review the ray-traversal hardware for both kd-tree and BVH traversal.



(a) Overall architecture (b) TRV architecture (c) Traversal slice

Fig. 11. SaarCor's architecture of ray traversal (adapted from Schmittler (2006)).

4.4.1 Kd-Tree Traversal. Researchers at Saarland University conducted extensive research on exploring the solution space of ray traversal on kd-trees (Schmittler et al. 2004; Woop et al. 2005; Schmittler 2006; Woop et al. 2006; Davidovic et al. 2011). Their designs were integrated into SaarCOR and RPU ray-tracing processors.

As depicted in Figure 11(a), the overall microarchitecture of SaarCor is built around three basic units, TRV, LST, and INT. A ray-collection unit is responsible for collecting rays from the ray-generation unit and secondary rays generated upon a hit. A memory interface provides memory accesses for the three basic units. The Ray Tracing Core takes the packet-based ray tracing mechanism in which multiple rays are organized into a packet (Wald et al. 2001; Dmitriev et al. 2004; Benthin 2006; Wald 2006). Accordingly, the TRV unit follows an SIMD processing pattern by assigning rays in a packet to multiple traversal slices. Each slice handles rays in a multithreaded fashion to hide memory latency. The processing logic is organized into three separate pipelines to handle instructions for ray traversal, push to stack, and pop from stack, respectively. On top of the traversal slices, the traversal logic takes the traversal results of the current step and determines what the next operations should be. When hitting to a leaf node, the traversal unit calls the list unit to load triangles for an intersection test. Upon a hit, secondary rays may be generated and sent to the ray-collection unit for further processing. The traversal unit is validated with FPGA. With four traversal units, each equipped with 4 traversal slices, a SaarCOR processor sustains a throughput of 3M to 12M rays per second when running at 90MHz.

The traversal unit of SaarCor was later redesigned in the RPU ray-tracing processor (Woop et al. 2005; Woop et al. 2006) for enhanced programmability. The architecture is illustrated in Figure 12. It incorporates for traversal slices in an SIMD processing style and always concurrently processes 4 rays in a packet. The major difference from SaarCor is that the RPU organizes the hardware resources into a more integrated and pipelined manner. A stack control unit actually takes the responsibility of scheduling packets of rays. All rays have to be pushed into the stack before they can be traced. A 32-entry stack suffices for complex scenes consisting of millions of triangles. When the stack control unit determines which node to process, a memory access unit fetches the respective node data (both current node and child nodes) from memory via a cache. The node is sent to all four traversal slices for parallel processing. The 4 decision units are relatively simple and use a few comparison operations to determine the next node to be visited for the respective ray. Finally, a packet-level decision has to be made by merging all local decisions. At a resolution of 512×386, a rendering rate of 11.6 frames per second[5] can be achieved on the conference scene (282K triangles). Note that the performance is lower than that of SaarCor.

Nah et al. (2011) proposed the Traversal and Intersection (T&I) engine to accelerate the ray-traversal process (including traversing the acceleration structure and intersection test). The overall

[5]No throughput in terms of million rays per second is disclosed.

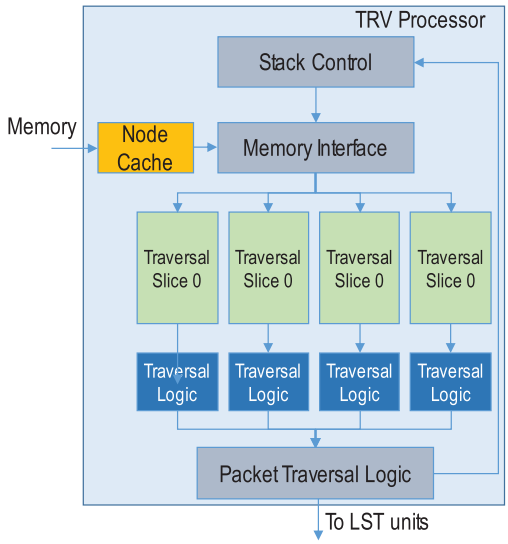


Fig. 12. Traversal processor of RPU (adapted from Woop et al. (2006)).

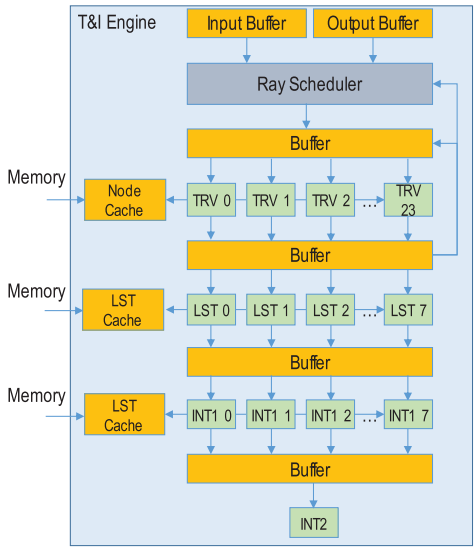


Fig. 13. T&I architecture and pipeline (redrawn by adapting Nah et al. (2011) and Nah et al. (2014)).

architecture is depicted in Figure 13(a). The T&I engine is highly customized for ray traversal. It has a scheduler to assign active rays in the input buffer to execution units. These units are organized into a four-level pipeline in which the intersection is partitioned into two stages (i.e., INT1 and INT2). Within each pipeline stage, the T&I engine still follows the SIMD processing pattern and allocates a different number of traversal units (TRVs), list units (LSTs), and intersection units (INT1 and INT2). Targeting a 500MHz clock and a 65nm process, the proposed architecture achieves a performance level of 186MRPS (million rays per second). Later, the T&I engine was redesigned as a unified pipeline in which the three tasks are executed by a common set of hardware as three

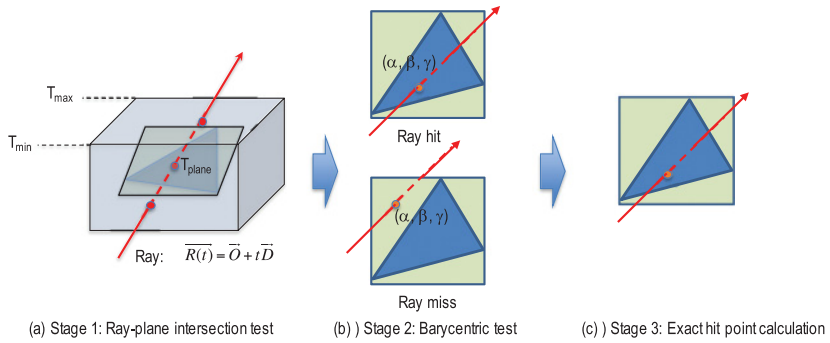


Fig. 14. Three stages of ray-triangle intersection test.

different modes (Nah et al. 2014). The purpose of such unification is to simplify the control and interface logic, although it does incur hardware overhead.

4.4.2 BVH Traversal. The traversal hardware for the kd-tree and BVH are rather similar. Integer-based BVH traversal hardware was proposed in early works (Mahovsky 2005; Mahovsky and Wyvill 2006). The major consideration was to minimize hardware cost by removing more expensive floating-point units. Mahovsky and Wyvill (2006) proposed a regular comparator array for BVH traversal. Different relative placements of ray and node boundary can be captured as eight unique patterns that can be executed by the comparator array in a similar fashion. Chang et al. (2008) developed a relatively simple 4-stage pipeline for BVH traversal that achieves a throughput of 100 million rays per second when fabricated in a 0.13μm process.

The state-of-the-art BVH traversal hardware was proposed by Lee et al. (2013a). The major innovation over its predecessor is the replacement of the original single pipeline with three parallel pipelines. The traversal process at one level of BVH consists of three stages: checking if a node is an inner one or a leaf, ray-box intersection test, and stack operations. Now, the three stages can work in parallel and the results can be exchanged through two crossbars. With four traversal units, the total traversal throughputs for primary rays, ambient occlusion rays, and diffusion rays can be up to 46, 121, and 44 million-ray per second, respectively.

4.5 Hardware for Ray–Triangle Intersection Test

After traversing to a leaf node, a ray checks if it hits any triangle inside the node. If there are one or more hits, the closest intersection point has to be identified. Early works handled this ray–triangle intersection as a whole process (Schmittler et al. 2004; Woop et al. 2006). A recent trend is to organize the operations for a given triangle as three steps (Wald 2004). First, the intersection point of the ray and the plane defined by the triangle is located. If the point is outside inside the interval associated with the current node, it means that there is no need to further test, that is, an early termination is feasible. Otherwise, the second step checks if the ray hit the triangle. Many efficient solutions have been introduced by using the Barycentric coordinate tests (Wald 2004; Shevtsov et al. 2007a). The idea is to derive the Barycentric coordinates of the hit point, that is, $\vec{H} = \alpha\vec{A} + \beta\vec{B} + \gamma\vec{C}$, where $\vec{A}$, $\vec{B}$, and $\vec{C}$ are the three vertices of the triangle. If $0 \leq \alpha,\ \beta,\ \gamma \leq 1$, the ray hits the triangle. Otherwise, the intersection test terminates right away. The third step computes the exact coordinates of the hit point. Figure 14 depicts the 3-step intersection test.

The abovementioned multistep process can be readily mapped by a pipelined architecture, with each stage implemented by a set of respective floating-point operators. One advantage is that it

Table 1. Taxonomy of Ray-Tracing Microarchitectures

		ASIC or ASIP Oriented	General-Purpose Processor Oriented
SIMD	SIMD	SaarCor [Schmittler et al. 2004; Schmittler 2006] RPU [Woop et al. 2005; Woop et al. 2006] D-RPU [Woop 2006]	
	SIMD with Streaming	StreamRay [Ramani et al. 2009]	
	SIMT	MRTP [Kim et al. 2012; Kim et al. 2013]	NVIDIA Pascal GPU [NVIDIA 2016] AMD RADEON™ HD 7990 GPU [AMD 2014], PowerVR™ Series 5 & 6 [Imagination 2014]
MIMD	Coarse-Grained MIMD	Ray tracing multiprocessor SoC [Nery et al. 2013]	Copernicus [Govindaraj et al. 2008] Intel MIC [Seiler et al. 2008]
	Fine-Grained MIMD	T&I [Nah et al. 2011] SGRT [Lee et al. 2013a; Lee et al. 2013b; Shin et al. 2013] RayCore [Nah et al. 2014] HART [Nah et al. 2015]	TRAX [Spjut et al. 2009] MIMD [Kopta et al. 2010]

allows early termination. In fact, typically, more rays will not intersect any triangle. This observation suggests that only a smaller number of processing elements is necessary for the last one or two steps. Accordingly, the multistep intersection test algorithm illustrated in Figure 14 is adopted by many recent hardware designs (Nah et al. 2011; Lee et al. 2013a; Nah et al. 2014). In the T&I engine (Nah et al. 2011), the three steps are assigned to two hardware units INT1 and INT2. The former can be configured into two modes corresponding to the ray–plane test and Barycentric test, respectively, due to the similarity in the underlying computing patterns. The T&I engine utilizes eight INT1 units in a single traversal and intersection pipeline. The latter unit is dedicated to the hit point calculation that involves a long-latency reciprocal computation. Since most rays are already filtered away, only one INT2 unit is required for the eight INT1 units. Both the eight INT1 units and the INT2 unit require 56 adders (all for IST1), 59 multipliers (56 for INT1 and 3 for INT2), 24 comparators (all for IST1), and 1 reciprocal unit (for IST2). All these are floating-number arithmetic units. In the RayCore engine, the intersection computation is performed by a unified pipeline that is also responsible for the whole ray-traversal process from tree traversal to intersection test (Nah et al. 2014). The details are already covered in Section 4.4.

5 RAY TRACING MICROARCHITECTURE

The previous section reviewed the hardware design for major building blocks of a ray-tracing pipeline. In this section, we survey hardware architectures integrating such building blocks and the respective the memory organization. The ray-tracing workload features sufficient ray-level parallelism but limited instruction-level parallelism (Spjut et al. 2009). As a result, all ray-tracing microarchitectures exploit parallel threads to handle rays in parallel. This section is organized into four separate sections. The first section classifies existing works using a two-dimensional taxonomy, the next two sections review SIMD and MIMD parallel architectures for ray tracing, respectively, and the final section discusses the lessons learned from the surveyed works.

5.1 A Taxonomy for Ray-Tracing Microarchitectures

This work classifies the large body of research on ray-tracing architectures according to the taxonomy listed in Table 1. Existing designs are categorized with regard to two dimensions. The first dimension concerns design style. Current ray-tracing hardware can be classified as Application

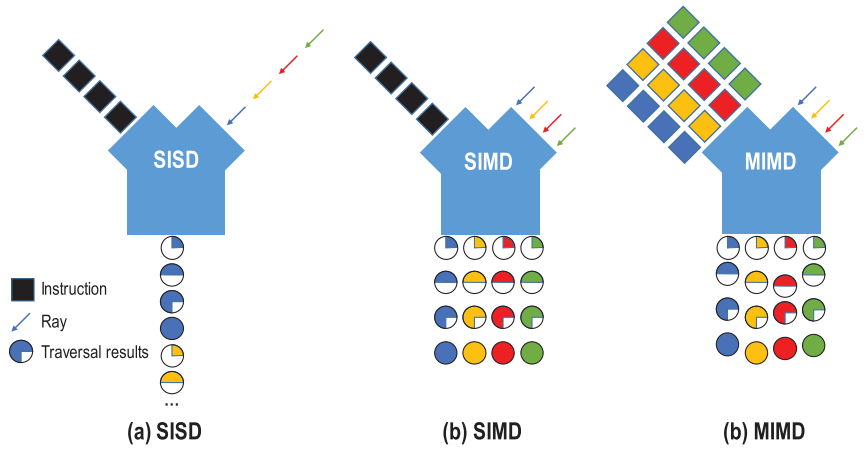


Fig. 15. Parallel mechanisms for ray traversal.

Specific Integrated Circuit (ASIC) or Application Specific Integrated Processor (ASIP) oriented and general-purpose processor oriented. ASIC-oriented designs focus on delivering the best performance and the lowest power consumption under a given area of silicon real estate. To handle complex graphics applications, ASIC designs are often enhanced as ASIPs to be able to execute customized programs. Meanwhile, modern processors—including both GPUs and CPUs—have become powerful enough to support ray tracing with a limited level of performance. As such processors enable maximized flexibility in graphics processing, there also exists a considerable body of work focusing on developing hardware and software frameworks for ray tracing.

The second dimension concerns how to exploit parallelism in the ray-tracing process, which is perhaps the most typical example of an embarrassingly parallel workload because all rays can be handled indecently. Due to the nature of the ray-tracing computing, SIMD streams and MIMD streams, as defined by Flynn's classical taxonomy of parallel machines (Flynn 1972), have been extensively explored in exiting ray microarchitectures. Figure 15 compares the SIMD and MIMD processing patterns with the sequential Single Instruction Single Data (SISD) stream architecture under the context of ray tracing.

In an SIMD platform, all rays are handled by the same instruction in an instruction cycle or macro-step. Modern GPUs relax the enforcement of the same instruction to all data by allowing multiple groups of threads to follow varying paths of instruction execution in a multithreaded manner for enhanced efficiency. Such a paradigm is designated as Single Instruction stream, Multiple Threads (SIMT) or Single Program, Multiple Data (SPMD) in the literature. It is adopted by many ray-tracing architectures for more efficient usage of underlying hardware. Another enhancement to the SIMD concept is to adopt the streaming processing pattern (Dally et al. 2004). One such example is StreamRay (Ramani et al. 2009), in which rays are handled by multiple hardware modules processing different levels of the acceleration structure.

By allowing each ray to be processed at different rhythms, MIMD architectures offer the best execution flexibility. Multicore CPU-based ray-tracing architectures fall into the category of MIMD machines, but they typically adopt a coarse-grained multithreading approach. Typically, each core has its own ray buffers and independently handles rays. On the other hand, researchers proposed fine-grained MIMD machines consisting of many full-fledged but relatively simple cores. These cores fetch rays from a shared buffer and perform the traversal process in a single-ray or packet manner.

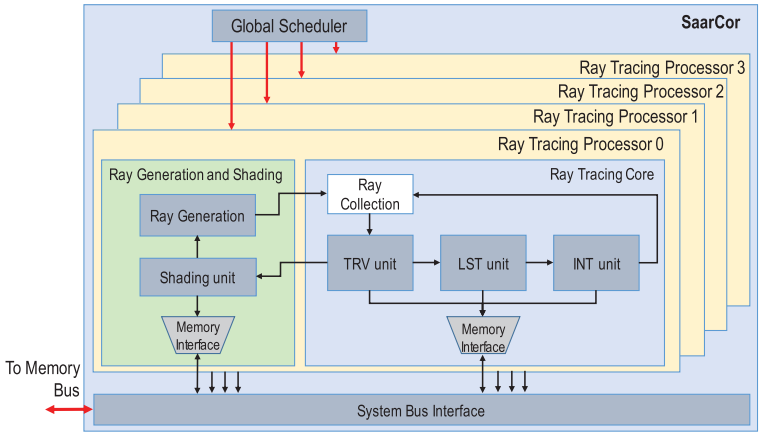


Fig. 16. The SaarCor architecture (adapted from Schmittler (2006)).

A single-ray handling module (usually in the form of a full-fledged processor) on a coarse-grained MIMD platform has its own ray buffers and independently handles a bunch of rays. On a fine-grained MIMD platform, each module fetches a ray from a shared ray buffer and performs the traversal process as a unit job.

This discussion for parallel machines is for the ray-traversal process. The construction part of ray tracing, which is essential for dynamic scenes, has been less investigated in terms of parallel processing, though. In fact, only recently have efficient fully parallel construction algorithms been proposed. The idea is to use data-parallel (i.e., SIMD) operators, for example, prefix sum and radix sort, to assemble a completely parallel flow. Since the construction process targets a single tree-like structure, MIMD and MISD machines are unlikely to excel.

5.2 SIMD/SIMT

It is rather straightforward to associate SIMD with ray traversal because all rays are handled by the same traversal program. As a result, a significant portion of the existing works resort to SIMD for efficient ray tracing. However, a closer check of the ray-traversal flow suggests that a pure SIMD mechanism cannot offer sufficient flexibility. First, rays go through different paths in the underlying kd-tree or BVH. From a program execution perspective, different paths correspond to varying actually executed instruction streams. Second, a ray hitting a tree node may need to traverse both child nodes. A stack is thus needed to keep track of branching nodes so as not to miss a potential intersection. Such stack operations introduce another level of irregularity in the instruction flow. Third, the memory access pattern of ray tracing is highly irregular. In fact, the parallel works in an SIMD group may visit memory locations distributed in a wide range. The various sources of inherence pose a significant challenge to a pure SIMD platform. As a result, early ray-tracing architectures adopted the SIMD paradigm, but later works gradually migrated to the SIMT mechanism. SIMT offers powerful support for ray tracing in two aspects: (1) more flexible support for branch instructions within jobs executed by a group of SIMD hardware and (2) integration with multithreading to hide memory latency by time-multiplex usage of SIMD hardware.

To the authors' knowledge, SaarCor (Schmittler et al. 2004; Schmittler 2006) is the first dedicated hardware accelerator for ray tracing. SaarCor only focuses on the traversal part of ray tracing and the construction of acceleration structure is constructed on a CPU. As shown in Figure 16, the overall structure of SaarCor consists of a global scheduler (i.e., a ray-generation controller by

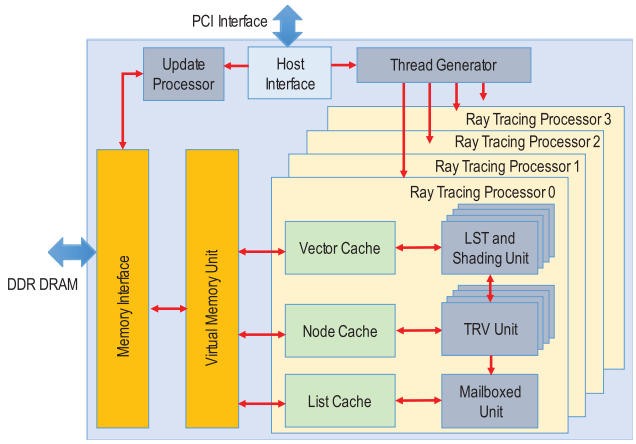


Fig. 17. The RPU architecture (adapted from Woop et al. (2005)).

Schmittler et al. (2004)), four ray-tracing cores (RTC), and a memory bus interface. The scheduler assigns workloads to RTCs for concurrent execution (actually in an MIMD manner). Each ray-tracing processor consists of a ray-generation and shading (RGS) unit as well as a ray-tracing core (RTC). The RTC core is basically an SIMD platform elaborated in Section 4.4.1. The RGS has a master to schedule rays and dispatch them to the computing resources in the RTC. All memory requests of RTCs are sent to a memory bus through three caches, which target the acceleration structure, the triangle lists on leaf nodes, and intersection computation, respectively. SaarCor boosted ray-tracing performance to a whole new level: it allows a rendering speed of 47MRPS at a clock rate of 533MHz and a memory bandwidth of 1GB/s at a resolution of 1024×768.

SaarCor was succeeded by the Ray-Processing Unit (RPU) and Dynamic Ray Processing Unit (DRPU) (Woop et al. 2005; Woop et al. 2006). The improvements lie in several aspects. First, the RPU obtained better programmability through an SIMD instruction set customized for ray tracing. Second, the RPU adopted the SIMT mechanism for better usage of SIMD hardware. Third, it began to support dynamic scenes by allowing updates of the acceleration structure with hardware. As illustrated in Figure 17, a DRPU chip has a number of ray-tracing processors, each equipped with a set of LST/shading units (i.e., shader processing units in the terminology of Woop et al. (2006)), a group of TRV units, and a mailboxed unit. The LST/shading unit is designed as an instruction processor for intersection and shading computation. The TRV unit is still from the tree traversal processing. A given number of LST/shading units and TRVs are integrated as a chunk, which serves as the basic unit of instruction execution. Each chunk is able to run N threads in a multithreaded fashion. The mailboxed unit is provided a buffered mechanism for LST operations. It is activated by the TRV when a ray hits a nonempty leaf node. The lists of primitives to be loaded can be merged and buffered in the mailboxed unit. This mechanism helps improve memory efficiency. The ray-tracing processor has separate caches for shading operations, nodes in the acceleration structure, and graphics primitives. Compared with SaarCor, the RPU delivers a better level of flexibility at the cost of performance. The DRPU was evaluated as an ASIC implementation (Woop et al. 2006). For the reason that both the MRPS and number of rays shot per pixel are not given, we calculate the MRPS by assuming that 1 ray is cast for each pixel. For the 282K-traiangle "conference" scene with a single rendering unit, the DRPU performs a ray processing throughput of 60.9MRPS when equipped with 8 rendering units working at 400MHz clock rate and 25.6 GB/s memory bandwidth.

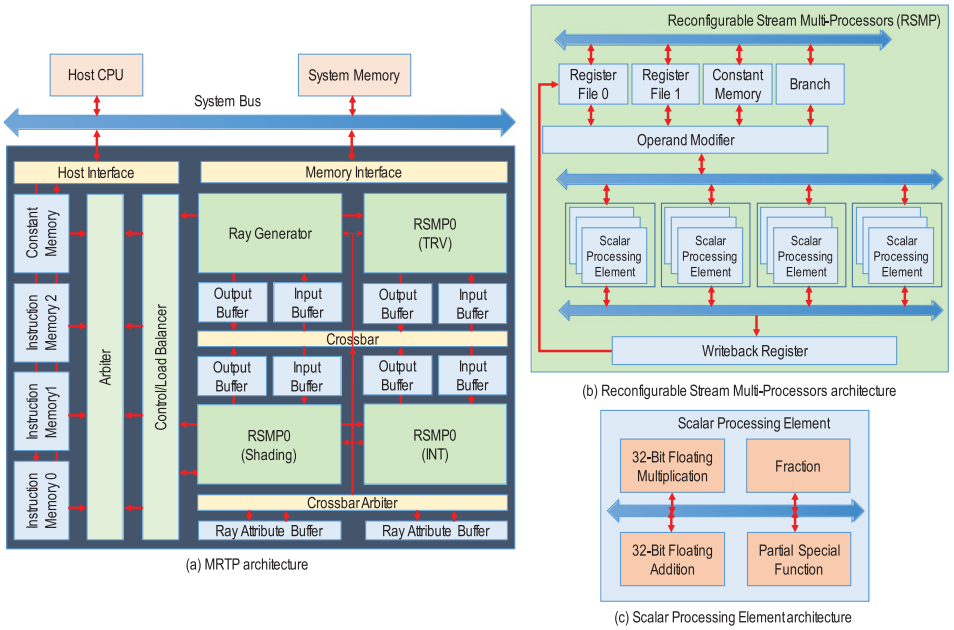


Fig. 18. MRTP microarchitecture (adapted from Kim et al. (2012)).

The ray-tracing workload is relatively irregular when compared with the traditional rasterization flow. As a result, running efficiency may suffer on SIMD/SIMT hardware because not all hardware units can have jobs to execute. One potential solution to this problem is to adapt the underlying hardware to the available parallelism. The Mobile Ray-Tracing Processor (MRTP) (Kim et al. 2012; Kim et al. 2013) is a SIMT architecture featuring reconfigurable stream multiprocessors (RSMPs). The overall architecture of MRTP is illustrated in Figure 18(a). MRTP receives data from a host-embedded CPU from a system bus. The acceleration structure and instructions of the ray tracing kernels are stored in on-chip memory. The computing capability of MRTP comes from three RSMPs, which accept rays from a ray generator. As shown in Figure 18(b), an RSMP has 12 scalar processing elements (SPEs), which can execute instructions in an SIMT fashion (i.e., 24 threads are concurrently running on the 12 SPEs). Each SPE has floating point units for 32b multiplication, addition, fraction, and conversion operations. The reconfigurability of RSMP comes from the capability of adapting the vector width with workload. RSMP allows the SIMD width to be configured as either a 12-wide computing channel or three 4-wide computing channels. When the inherent parallelism of a given instruction is not sufficient for 12-wide execution, potentially three instructions can be dispatched to SPEs. Three identical RSMPs are deployed on an MRTP. They are configured to perform TRV, INT (including LST), and shading operations, respectively. The MRTP was designed for mobile applications and only consumes a power of 156mW. However, its performance is lower than that of the DRPU.

Modern GPUs are highly optimized SIMT engines. When associated with high-performance ray-tracing algorithms, interactive ray tracing is already feasible (Parker et al. 2010). The microarchitecture of commercial GPUs is already covered in Section 3.1. It is true that the area and power efficiencies of GPUs are lower than those of previous dedicated architectures, but the powerful programming support of GPUs make them irreplaceable for ray-tracing applications. A major problem of ray tracing on GPUs is that the irregular access patterns to the acceleration

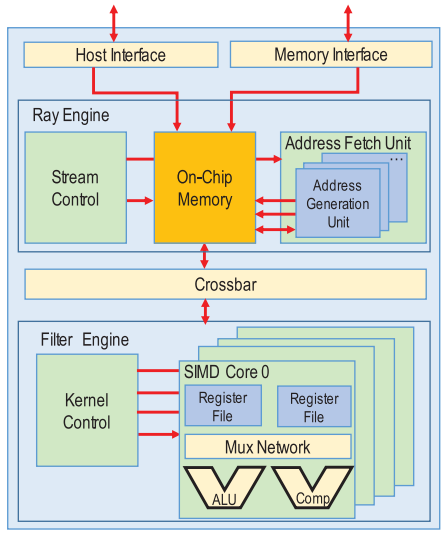


Fig. 19. StreamRay (adapted from Ramani et al. (2009)).

structure, which is too big to be completely stored in a GPU's on-chip scratchpad memory (i.e., shared memory in NVIDIA's terminology), leads to poor cache behavior and redundant loading. Aila and Karras (2013) sketched a revised GPU microarchitecture that works with treelets, which are subtrees of the original acceleration structure and small enough to fit into on-chip cache or scratchpad memory. The enhanced GPU regroups rays according to the hit treelet and schedules rays for traversal. The combined effect of hardware and software optimization leads to a total reduction of 80% to 85% of memory traffic. Although a detailed hardware evaluation is not available yet, the architecture proposed by Aila and Karras (2013) proved the feasibility of dynamically scheduling rays with regard to their memory accessing pattern as well as preloading subtrees in an on-demand manner.

Another path to enhance the SIMD paradigm under the ray-tracing context is to adopt the streaming processing pattern (Dally et al. 2004). Under such a paradigm, data items in a stream sequentially go through a series of processing stages, while each stage performs a specific computation and/or communication task. These stages can be executed in parallel. StreamRay (Ramani et al. 2009) is a representative of such streaming SIMD architecture. Rays to be traversed are treated as a stream of items, which are filtered to identify an active subset. These active rays are further handled by a series of cores in a streaming manner. An active set contains rays with good coherence. Each SIMD core executes customized micro-code for a given stage, for example, TRV, LST, INT, and shading, in the traversal flow. It has a register set and respective execution logics and comparators. SIMD cores in the neighboring stage are interconnected with a network for fast data exchange. As a result, these SIMD cores enable "vertical" parallelism. Meanwhile, each SIMD core allows multiple rays concurrently. This is actually "horizontal" parallelism. StreamRay has an address unit to compute addresses to different banks of the memory system. The overall structure is drawn in Figure 19. StreamRay achieved a throughput of 20.32MRPS (assuming 1 ray per pixel) for the 282K triangles "conference" scene at a resolution of 1024×1024 and a clock rate of 500MHz. The importance of StreamRay lies in that it validates the feasibility of active identification of coherent rays with custom-designed hardware in a dynamic manner. The coherent rays tend to follow similar traversal paths. Such an aggressive approach of coherence extraction is essential for

future ray-tracing hardware approaches for which more hardware units can be deployed. However, the efficiency of StreamRay for more incoherent secondary rays is still an open problem worth investigating.

5.3 MIMD

MIMD architectures offer a higher level of freedom to organize parallel works but also incur a higher overhead in terms of hardware cost. In fact, an SIMD engine requires only a single set of instruction decoding and dispatch logic, while an MIMD platform has to maintain multiple sets of such instruction handling logic. In this work, we further divide MIMD engines into coarse-grained and fine-grained classifications. The former is more oriented to multicore microprocessors, in which each core is a full-fledged CPU. The latter is more customized to directly support the ray-tracing algorithmic flow.

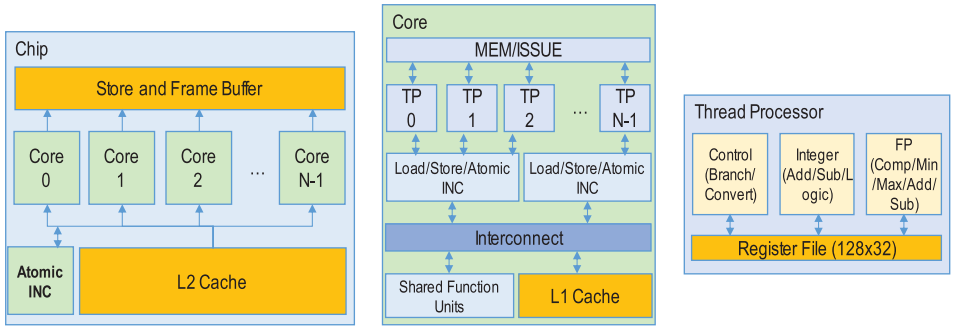
5.3.1 Coarse-Grained MIMD. Coarse-grained MIMD architectures naturally match the concept of the tile-based[6] rendering paradigm (Molnar et al. 1994; Sommefeldt 2015). The basic idea is to partition the scene or display plane into multiple parts and use a processing unit to handle the rays fallen into one part. It must be pointed out that the ray-tracing workload is not totally amenable with tile-based rendering because a single ray may traverse the acceleration structures fallen into more than one tile. As a result, a ray that fails to intersection with triangles in one tile may have to be revisited in another tile.

Copernicus (Govindaraj et al. 2008) is a multicore architecture for real-time ray tracing. It has sixteen tiles, each consisting of eight processor cores and a cache. The processor core is equipped with a 4-wide SIMD unit and supports up to eight threads. The tiles are backed up by a large shared L2 cache and an accelerator. The accelerator is a pool of commonly used computation units, such as square root, trigonometric equations, and light attenuation. A scene to be traversed is partitioned to 16 blocks and a block is processed by a tile. Copernicus is potentially the most flexible ray-tracing architecture that inherits the programmability of general-purpose processors. However, it is relatively expensive in terms of silicon estate. A single tile in Copernicus is able to process 10 million rays per second and employs 16mm^2 of die area in a 22nm technology. Working at 4GHz and supported with 15GB/s memory bandwidth, Copernicus attains a throughput of 10MRPS without dynamic scene update.

5.3.2 Fine-Grained MIMD. Fine-grained MIMD architectures actually have their root originated from SIMD machines. The traversal of incoherent rays may incur significantly varying execution paths in the instructions of a ray-traversal program. Therefore, researchers enhanced SIMD/SIMT machines with multiple sets of instruction fetch, decode, and dispatch logics for better utilization of the underlying computation hardware. These architectures already fall into the category of MIMD machines. In the current literature, Trax (Spjut et al. 2009) and several architectures based on T&I (Nah et al. 2011) are two representatives of fine-grained MIMD ray-tracing architectures. The former is more oriented to being a general-purpose computer, while the latter is devised to be more application specific. Both were continuously improved into a series of derivative architectures.

5.3.2.1 TRaX and Its Derivative. The TRaX microarchitecture was first proposed by Spjut et al. (2009) and then continuously enhanced in a series of works (Kopta et al. 2010; Spjut et al. 2012; Kopta et al. 2013). The design philosophy of TRaX is to maximize the efficiency of thread, which is

[6]Note that here "tile" means a subregion of the display plane and is different from the "tile" defined as an independent processing unit in Copernicus architecture.



(a) Overall architecture (b) Core architecture (c) Thread processor architecture

Fig. 20. TRaX microarchitecture (adapted from Spjut et al. (2009)).

the finest unit of work in ray traversal. TRaX adopts a multicore paradigm as shown in Figure 20(a). These cores work concurrently but share an L2 cache and atomic operation logic.

Each core has the internal organization depicted in Figure 20(b). The core is built around a multithreaded issuing logic, which delivers threads to 32 thread processors (TPs). It is equipped with L1 instruction and data caches to be shared by all TPs. The core also has an interconnection network that can feed data to and fetch results from a pool of shared function units. These units are fully pipelined and perform frequently used complex functions such as floating-point multiplication, integer multiplication, square root, and reciprocal. A TP is the basic unit of thread code execution and its architecture is illustrated in Figure 20(c). Each TP is a simple in-order pipelined processor that has a private program counter and a small instruction cache. It independently decodes instructions in a thread and dispatches them to its functional units, including a simple integer unit and a floating-point unit as well as a branching module. A local register file serves as the high-speed memory for instruction operands. Given instructions involving complex operations, a TP can send the operands to the shared functional units inside the respective core. At a clock rate of 500MHz, TRaX allows a ray throughput of 177MRPS of rendering speed. The memory system of TRaX sustains a bandwidth of 15GB/s memory.

TRaX improves the efficiency and flexibility of thread execution by leveraging the MIMD architecture. A throughput of 387MRPS is achieved for a 282K-triangle scene when the design is implemented with a 1GHz clock and 10GB/s memory bandwidth. The respective silicon estate is at a die area of 200 mm^2 (65nm). The MIMD architecture incurs overhead in terms of multiple sets of instruction handling logic. In addition, the flexibility of such MIMD architectures for non-ray traversal problem is still an open problem. So far, TRaX and its derivatives do not support a dynamic scene and have to depend on a CPU to build or update the acceleration structure.

5.3.2.2 T&I Derivatives. The T&I engine (Nah et al. 2011) is a hardware accelerator for ray tracing–based rendering. The traversal and intersection pipeline of T&I is already covered in Section 4.4.2. Several ray-tracing architectures have been proposed by using T&I as the basic building block.

The Samsung reconfigurable GPU based on Ray Tracing (SGRT) (Lee et al. 2013a; Lee et al. 2013b; Shin et al. 2013) was evolved from the T&I engine. As shown in Figure 21, SGRT consists of a T&I module and a Samsung Reconfigurable Processor (SRP). During the rendering process, the TRV, LST, and INT operations are processed on the T&I module, while ray generation and shading are performed by the SRP. The SRP integrates both a very long instruction word (VLIW) processor and a coarse-grained reconfigurable array (CGRA). The former aims at general-purpose

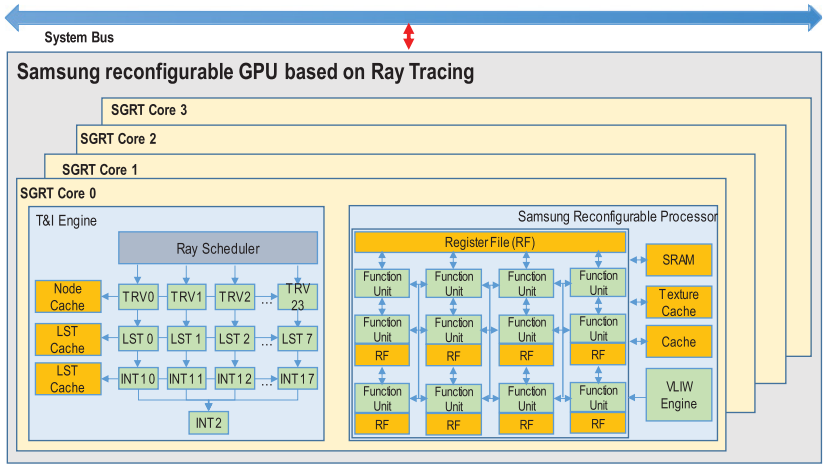


Fig. 21. The architecture of the Samsung reconfigurable GPU based on ray tracing (SGRT; adapted from Lee et al. (2013a)).

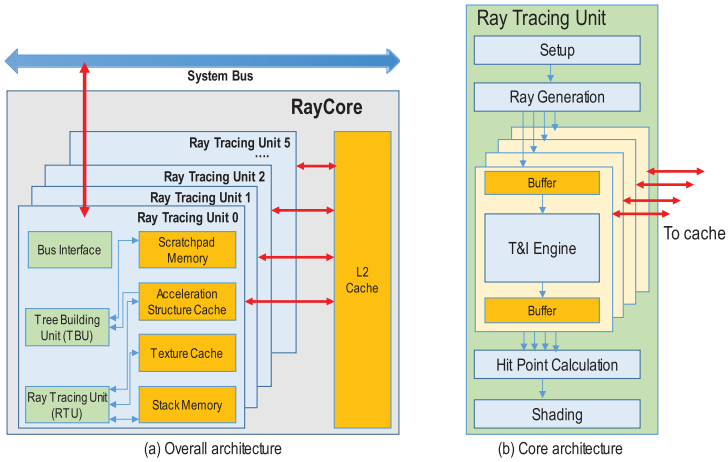


Fig. 22. RayCore architecture (adapted from Nah et al. (2014)).

computations and the latter focuses on computation-intensive tasks by dynamically adapting to the processing patterns of the target application. Leveraging the power of MIMD and reconfigurable hardware, SGRT delivers rather impressive performance. When working at 500MHz, it attains a ray throughput of 175MRPS for a scene with 179K triangles at a resolution of 1920 $\times$ 1080 working at 500MHz. SGRT requires a memory bandwidth of 12.8 GB/s and consumes a relatively small die area of 25 mm^2, which make it ideal for mobile platforms.

RayCore (Nah et al. 2014) was proposed to support the complete ray-tracing pipeline, including tree construction, tree traversal, intersection test, and shading. The overall architecture is shown in Figure 22(a). RayCore integrates both a TBU and a ray-traversal unit (RTU). To guarantee the power and area efficiency posed by mobile platforms, RayCore is equipped with fixed-function circuits for compute-intensive tasks. The TBU is responsible for kd-tree construction. The TBU has two pipelines with different underlying algorithms. One is the binning-based tree building

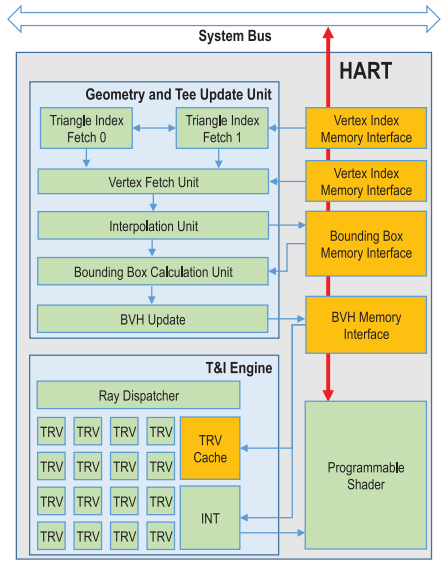


Fig. 23. HART microarchitecture.

pipeline (TBP), which enables a faster construction speed at the cost of poorer tree quality. The other is a sorting-based TBP that is slower but allows better quality. The RTU is built around the T&I engine elaborated in Section 4.4.1. The RTU is supported by a carefully designed memory hierarchy, as shown in Figure 22(a). The central design strategy of the memory system is a "looping for the next chance" scheme in which a cache miss is treated as a prefetch to the next level memory. Such a scheme is efficiently implemented with two FIFOs deployed between two neighboring levels of memories (i.e., L1 cache and L2 cache as well as L2 cache and main memory). RayCore is implemented and validated with four Xilinx Virtex-6 LX550 FPGAs at 500MHz with a 1.1 GB/s bandwidth memory. An ASIC evaluation using a TSMC 28nm library proves that the die area of a single RTU can be as small as $3mm^2$ and that of a single TBU is $1.6mm^2$. RayCore attains a rendering speed of 193MRPS.

A further extension on the T&I engine with an emphasis on dynamic scenes is the HART architecture (Nah et al. 2015). Its architecture is illustrated in Figure 29. The major improvement over previous generations is a heterogeneous Geometry and Tree Update (GTU) unit that uses both programmable and dedicated hardware for fast tree updating, while the tree construction is conducted by a CPU. The underlying consideration is that the tree construction is expensive but does not need to be done for every frame. Figure 23 shows the pipelined architecture of the GTU. The tree updating process is organized into five pipeline stages, index fetch (retrieving the index of triangles), vertex fetch (reading triangle vertices), interpolation (interpolating vertices by comparing two key frames), AABB/triAccel calculation (computing bounding box), and BV refitting (updating acceleration structure). Among these, four stages are directly backed up by respective DRAM interfaces for memory efficiency. Another improvement is a shallow tree structure using a primitive's axis-aligned bounding boxes. The traversal and intersection are executed on an enhanced version of the T&I engine shown in Figure 23. The HART architecture is able to attain real-time performance (383MRPS) for Whitted ray tracing of scenes at a 500MHz clock rate and a resolution of 1920×1200.

5.4 Discussion

The ray-tracing microarchitectures reviewed in Sections 5.2 through 5.4 and other works in the literature are listed in Table 2 for an overall comparison. We evaluate these architectures in the following six aspects.

- Dynamic scene support. The support for dynamic scenes is becoming increasingly important as the ray-tracing rendering is gradually adopted in mobile and consumer-level applications. Such applications tend to be highly interactive and thus different from traditional applications, such as movie rendering. Objects can be moved, inserted, and removed in dynamic scenes. The efficiency of reconstructing and/or updating the acceleration structure is thus growing equally important as that of the traversal process. Unfortunately, the construction/updating process is rather expensive and has a significantly different computing pattern from the traversal process. Research practices prove that the support for dynamic scenes calls for customized hardware for construction and updating.
- Acceleration structure. In Table 2, we list the acceleration structures used in various works. All existing ray-tracing microarchitectures adopt either kd-tree or BVH, while general-purpose computing devices such as GPUs and MIC support both. The advantages and disadvantages of the two structures are intensively discussed in Wald et al. (2007). Today, it seems that the two structures can be highly efficient at the cost of careful performance tuning.
- Programmability. Complex graphics effects often require the underlying hardware to offer high programmability. Meanwhile, GPU-based general-purpose computing has also gained a large application domain in both desktop and mobile domains. Accordingly, application-specific ray-tracing processors without sufficient programmability are unlikely to harvest commercial success even by offering impressive performance. On the other hand, it must be noted that programmability is not totally conflicting with performance. The Kepler GPU, although not customized for ray tracing, can still sustain a striking level of performance because its flexible programming support allows the implementation of highly complex algorithms and intensively tuned coding.
- Performance. It is extremely challenging to compare the performance of the proposed ray-tracing hardware, which span a time period of over ten 10 years. The reported performance figures in the literature were collected on a large number of benchmark scenes with widely varying set up in terms of scene complexity, resolution, and number of rays per pixel. In the surveyed literature, the performance of ray tracing is measured in FPS and/or MRPS). Although the FPS metric directly reflects the real-time visual experience, an apple-to-apple comparison of FPS among different microarchitectures is tricky, as the complexity of ray tracing a frame is dependent on the resolution (i.e., every pixel requires a given number of primary rays), the characteristics (e.g., the spatial distribution and materials of primitives) of a given scene, and the types of rays (e.g., primary rays and secondary rays, such as diffuse rays and ambient occlusion rays). The number of primary rays is directly related to the display resolution, but the specific number of rays for each pixel may vary among different ray tracers. The number of secondary rays is a complex function of the scene to be rendered. As a result, the FPS comparison only makes sense when performance evaluations are derived under exactly the same experimental setup. On the other hand, we believe that MRPS is a more proper metric that is directly related to the raw processing capability of a ray-tracing platform. Although it is still dependent on the scene complexity, the impacts of resolution and number of rays per pixel can be normalized. Accordingly, we choose MRPS as the performance metric in Table 2 to evaluate the ray-tracing microarchitectures survey in this work. Note that MRPS is purely for traversal performance. Therefore, we explicitly

Table 2. Comparison of Ray-Tracing Architectures

	RayTracing Architecture		Key Hardware Design Parameters	Hardware for Dynamic Scene Support	Acceleration Structure	Programming	Million rays per second (MRPS)	Hardware cost
SIMD/ SIMT	SIMD	SaarCor [Schmittler et al. 2004; Schmittler 2006]	533MHz clock rate 1 GB/s memory bandwidth 1094KB on-chip memory	No	Kd-tree	Fixed function	47 (a scene with 3.6M triangles, 1024 x 768 resolution, no dynamic update)	192 floatingpoint units. 822KB register files, 272KB cache, 1094KB on-chip memory
	Streaming SIMD	StreamRay [Ramani et al. 2009]	500MHz clock rate	No	N/A	C++	20.32MRPS (assuming 1 ray per pixel, the "conference" scene with 282K triangles, 1024 x 1024 resolution, no dynamic update, secondary rays involved)	N/A (targeting 90nm)
	SIMT	RPU [Woop et al. 2005; Woop et al. 2006]	66MHz clock rate. 352MB/s memory bandwidth.	Partial	Kd-tree	Custom SIMD instruction set	1.03MRPS (assuming 1 ray per pixel, the "conference" scene with 282K triangles, 512 x 384 resolution, no dynamic update)	99% of logic slices, 88% of on-chip block memories and 13% of block multipliers on Xilinx Virtex-II 6000 FPGA
		DRPU [Woop 2006]	400MHz clock rate. 25.6GB/s memory bandwidth.	Partial	B-kd Tree	Custom SIMD instruction set	60.9MRPS (assuming 1 ray per pixel, the "conference" scene with 282K triangles, 1024 x 768 resolution, no dynamic update)	186.6mm^2 (130nm)
		MRTP [Kim et al. 2012; Kim et al. 2013]	100MHz clock rate. 14.5KB SRAM.	No	N/A	C++, HLSL [NVIDIA 2008]	0.67 (no dynamic updating)	1.2M gates (156mW), 16mm^2 (130nm)

(Continued)

Table 2. Continued

	RayTracing Architecture		Key Hardware Design Parameters	Hardware for Dynamic Scene Support	Acceleration Structure	Programming	Million rays per second (MRPS)	Hardware cost
		NVIDIA Kepler GPU	1GHz clock rate. 192GB/s memory bandwidth. 64KB L1 cache.	Yes	Kd-tree, BVH, and others	OpenGL CUDA, OpenCL, OptiX [Parker et al. 2010]	432.6 (a scene with 1.6M triangles, 1920 ×768 resolution, no dynamic update) [Aila 2012]	294mm^2 (28nm)
		Enhanced GPU [Aila and Karras. 2010]	48KB L1 cache 768KB L2 cache.	Yes	Kd-tree, BVH, and others	OpenGL, CUDA, OpenCL	N/A	N/A
MIMD	Coarse-Grained MIMD	Copernicus [Govindaraj et al. 2008]	4GHz clock rate. 15GB/s memory bandwidth. 32KB L1 cache	Yes	Kd-tree	Razor [Djeu et al. 2011]	10 (no dynamic update)	240mm^2 (22nm)
		Ray tracing multiprocessor SoC [Nery et al. 2013]	125MHz clock rate.	No	Kd-tree	No	N/A	5 MicroBlaze cores on Vertex-5 FPGA
		Intel MIC [Seiler et al. 2008, Wald et al. 2014]	3.5GHz clock rate.	Yes	Kd-tree, BVH, and others	OpenGL, OpenCL	101.07MRPS (assuming 1 ray per pixel, the "conference" scene with 282K triangles, 1920 ×1200 resolution, no dynamic update)	~650mm^2 (45nm)
	Fine-Grained MIMD	TRAX [Spjut et al. 2009]	500MHz clock rate. 6GB/s memory bandwidth. 64KB L1 cache.	No	BVH	C	177 (no dynamic update)	200mm^2 (65nm)
		MIMD [Kopta et al. 2010]	1GHz clock rate. 10GB/s memory bandwidth. 64KB L1 cache.	No	BVH	C++	387 (no dynamic update)	175mm^2 (65nm)

(Continued)

Table 2. Continued

	RayTracing Architecture		Key Hardware Design Parameters	Hardware for Dynamic Scene Support	Acceleration Structure	Programming	Million rays per second (MRPS)	Hardware cost
		T&I [Nah et al. 2011]	500MHz clock rate. 40GB/s memory bandwidth. 28KB L1 cache	No	Kd-tree	No	186 (no dynamic update)	12.12mm^2 (65nm)
		SGRT [Lee et al. 2013a; Lee et al. 2013b; Shin et al. 2013]	500MHz clock rate. 12.8GB/s memory bandwidth. 64KB for TRV and 128KB for IST L1 cache.	No	BVH	Similar to OpenGL and CUDA	175 (no dynamic update)	25mm^2 (65nm)
		RayCore [Nah et al. 2014]	500MHz clock rate. 1.1GB/s memory bandwidth. 124.38KB L1 list cache.	Yes	Kd-tree	No	193 (without dynamic update)	3mm^2 for an RTU 1.6 mm^2 for a TBU (28nm)
		HART [Nah et al. 2015]	500MHz clock rate. 384KB L1 cache.	Yes	Shallow BVH	Similar to OpenGL and CUDA	383 (with dynamic update)	1.07mm^2 for construction, 9.51mm^2 for traversal (65nm)

Note: "Partial" dynamic scene support means that the acceleration structure can be updated but cannot be reconstructed. The factor K reflects the impact of multiple rays per second. Typical values of K are in the range of 1 to 10.

indicate if the metric is derived with or without dynamic update in Table 2. For those works (e.g., Woop et al. 2005; Woop 2006; Seiler et al 2008; Ramani et al. (2009), and Wald et al. (2014)) in which only FPS results were reported, we translate the FPS values into MRPS by considering the respective resolution. If the numbers of rays per pixel is also missing, we compute the MRPS values by assuming one ray per pixel and explicitly use a factor K to reflect the impact of multiple rays per second. Typical values of K are in the range of 1 to 10.

- Hardware cost. It is also hard to compare different microarchitectures because they are implemented and/or evaluated with different hardware platforms (e.g., FPGA and ASIC) in varying technology nodes (e.g., 130nm, 65nm, 45nm, and 28nm). In Table 2, we choose to report the original value reported in the original literature without a direct scaling because the actual scaling among respective technology nodes is highly complicated.

From the metrics reported in Table 2 through the years, it can be seen that dramatic progress has been made in pursuing real-time ray tracing. Currently, we are already at a historically unique point to witness the democratization of interactive ray tracing to consumer-level

platforms. By surveying the representative ray-tracing microarchitectures in the literature, we have reached the following observations.

- GPUs have become powerful ray-tracing platforms but still need enhancement to get fit into mobile platforms.

At the present time, commercial high-end desktop GPUs offer the highest ray-processing throughput, which is enabled by the architecture innovation, the large silicon estate (at an advanced process technology node) dedicated to computing and the highly flexible programming support for intensive algorithm and coding optimization. As a result, high-end desktop GPUs will continue to be the workhorse for industry-level rendering solutions. However, the area footprint (a few square centimeters) and power budget (~100W) limit the usage of such architectures in mobile platforms. NVIDIA [2014 already released the Tegra series of mobile GPUs based on the same microarchitecture as their high-end equivalents, but their significantly smaller number of cores cannot sustain real-time ray-tracing applications (Wang et al. 2014). Accordingly, we believe that architectural innovations are still expected to improve the performance per unit area. From this perspective, the reduced precision architecture proposed by Keely (2014) provides a viable solution to adapt high-end GPU architectures for mobile applications[7]. Another important direction is to introduce hardware that explicitly reshapes memory traffic to unleash an increasing amount of parallelism from the ray-tracing workload. One such example is proposed by Aila and Karras (2010).

- Ray-tracing microarchitectures enhanced with dedicated hardware are fast maturing for real-time performance.

Ray-tracing microarchitectures with dedicated hardware support are already feasible to perform ray tracing–based rendering on scenes consisting of a few hundred or even over one million primitives at an interactive level. Among these, SGRT (Lee et al. 2013a; Lee et al. 2013b; Shin et al. 2013) and HART (Nah et al. 2015) are powerful enough to process dynamic scenes of medium complexity (i.e., ~100K primitives) in real time. Consuming a die area of one or two ARM Cortex A8 CPUs, ray-tracing hardware has become viable to be integrated into application processors of mobile platforms.

- Fine-grained MIMD machines excel.

In the ray-tracing microarchitectures surveyed in this work, fine-grained MIMD machines offer the highest performance. In fact, the SIMT scheme has been successful in enhancing the SIMD scheme with a multithreading mechanism, which allows more flexibility on hiding memory latency (Burtscher et al. 2012). The fine-grained MIMD offers even more flexibility by enabling more "personalized" treatment of rays and is less likely to suffer from control divergence, which can be a serious problem for SIMD/SIMT machines. Such an approach is especially feasible for ray-tracing microarchitectures because all rays are processed by one or more relatively compact kernels. As a result, the MIMD hardware can be built by hierarchically organizing highly simplified cores with simple instruction decoding logic, as those proposed in the TRaX microarchitecture (Kopta et al. 2010; Spjut et al. 2012; Kopta et al. 2013). In other words, the overhead of MIMD processing can be kept at a low level given the ray-tracing workloads. We believe that the fine-grained MIMD microarchitectures can be integrated with dynamic memory traffic reshaping techniques (e.g., Aila and Karras (2010)) to enable an even higher level of overall performance. Meanwhile, it is also

[7]The reduced precision may not be feasible for desktop GPUs that have to support rasterization and general-purpose computing workloads.

appealing to investigate how to incorporate instruction logics to concurrently handle multiple instruction streams in the multiprocessors of modern GPUs.

- How can we make the best of the ray-tracing hardware?

Dedicated ray-tracing hardware is promising in terms of rendering performance at the cost of extra silicon real estate. The extra silicon overhead, if it cannot be used by other applications, can be a stumbling block because the emergent ray-tracing applications have to coexist with rasterization applications. On the other hand, it is appealing to explore the potential of making dedicated ray-tracing hardware both accessible and useful for general-purpose applications. Many computing patterns—such as sorting, prefix-sum, and tree traversal, which are essential for ray tracing—actually are also performance critical for many other applications (e.g., Deng and Mu (2013), Zhou et al. (2014), and Ren et al. (2017)). Therefore, the respective ray-tracing modules can be readily made accessible to other applications. As a matter of fact, the modern GPU has become a universal computing platform with its powerful support for compute-intensive workloads with regular computing and memory accessing patterns. Considering the increasing importance of irregular applications (Pingali et al. 2011), the reuse of ray-tracing hardware to accelerate workloads with irregular computing and memory patterns can open a new path for general-purpose GPU computing.

6 CONCLUSION

Ray tracing–based physically realistic rendering has long been considered to be a promising approach to enabling a new level of visual experience, but its daunting computation intensity limits its application to offline rendering in the past. This survey suggests that we are already at the turning point of the democratization of interactive ray tracing to consumer platforms. With the introduction of novel construction hardware for the acceleration structure, it is already feasible to support dynamic scenes in real time. Meanwhile, fine-grained MIMD machines with dedicated hardware support are almost ready to be integrated into mobile application processors for real-time ray tracing on mobile platforms. In the future, a major concern is how dedicated ray-tracing hardware can be fused with existing graphics hardware. We believe that a potentially important research direction is a "refactoring" of the SIMD/SIMT-based GPU microarchitecture to (partially) support the fine-grained MIMD scheme. It is also worth investigating if the ray-traversal hardware can be generalized into an architecture that benefits other irregular applications.

REFERENCES

Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*. 113–122.

Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics*. 145–149.

Timo Aila, Tero Karras, and Samuli Laine. 2013. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*. 101–107.

Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. 2008. *Real-Time Rendering*. CRC Press, Boca Raton, FL.

AMD. 2014. AMD Radeon™ HD 7990 Graphics Card. Retrieved July 25, 2017 from http://www.amd.com/en-us/products/graphics/desktop/7000/7990.

Arthur Appel. 1968. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. 37–45.

James Arvo and David Kirk. 1989. A survey of ray tracing acceleration techniques. *An Introduction to Ray Tracing*. 201–262.

Carsten Benthin. 2006. Realtime ray tracing on current CPU architectures. PhD thesis. Saarland University, Saarbrücken, Saarland, Germany.

Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM*. 18, 9, 509–517.

Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. 72–81.

David Blythe. 2008. Rise of the graphics processor. In *Proceeding of IEEE*. 761–778.

Martin Burtscher, Rupesh Nasre, and Keshav Pingali. 2012. A quantitative study of irregular programs on GPUs. In *IEEE International Symposium on Workload Characterization (IISWC'12)*. 141–151.

Nathan A. Carr, Jesse D. Hall, and John C. Hart. 2002. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. 37–46.

Chen-Haur Chang, Chuan-Yiu Lee, and Shao-Yi Chien. 2008. Hardware architecture design and implementation of ray-triangle intersection with bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 179–179.

Per H. Christensen, Julian Fong, David M. Laur, and Dana Batali. 2006. Ray tracing for the movie 'Cars. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 1–6.

James H. Clark. 1976. Hierarchical geometric models for visible surface algorithms. In *Communications of the ACM* 19, 10, 547–554.

William J. Dally, Ujval J. Kapasi, Brucek Khailany, Jung Ho Ahn, and Abhishek Das. 2004. Stream processors: Programmability with efficiency. *ACM Queue* 2, 1, 52–62.

Tomas Davidovic, Lukas Marsalek, and Philipp Slusallek. 2011. Performance considerations when using a dedicated ray traversal engine. In *Proceedings of the WSCG*.

Y. Deng and S. Mu. 2013. A survey on GPU based electronic design automation computing. *Foundation and Trends in Electronics Design Automation*.

Andreas Dietrich, Abe Stephens, and Ingo Wald. 2007. Exploring a Boeing 777: Ray tracing large-scale CAD Data. *IEEE Computer Graphics and Applications*. 27, 6, 36–46.

Peter Djeu, Warren A. Hunt, Rui Wang, Ikrima Elhassan, Gordon Stoll, and William R. Mark. 2011. Razor: An architecture for dynamic multiresolution ray tracing. *ACM Transactions on Graphics*. 30, 5, Article 115, 115:1–115:26.

Kirill Dmitriev, Vlastimil Havran, and Hans-Peter Seidel. 2004. Faster ray tracing with SIMD shaft culling. *Research Report MPI-I-2004-4-006*. Max-Planck-Institut für Informatik, Saarbrücken, Germany.

Michael J. Doyle, Colin Fowler, and Michael Manzke. 2013. A hardware unit for fast SAH-optimised BVH construction. *ACM Transactions on Graphics* 32, 4, 139.

Philip Dutre, Philippe Bekaert, and Kavita Bala. 2006. *Advanced Global Illumination* (2nd ed). A. K. Peters/CRC Press, Boca Raton, FL.

Joshua Fender and Jonathan Rose. 2003. A high-speed ray tracing engine built on a field-programmable system. In *Proceedings of Field-Programmable Technology (FPT'03)*.

Michael J. Flynn. 1972. Some computer organizations and their effectiveness. *IEEE Transactions on Computers* 100, 9, 948–960.

Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. 1980. On visible surface generation by a priori tree structures. In *Proceedings of the ACM SIGGRAPH Computer Graphic Conference* 14, 3, 124–133.

Akira Fujimoto, Tanaka Takayuki, and Iwata Kansei. 1986. Arts: Accelerated ray-tracing system. In *IEEE Computer Graphics and Applications* 6, 4, 16–26.

Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of the High Performance Graphics*. 59–64.

Andrew S. Glassner (ed.). 1989. *An Introduction to Ray Tracing*. Elsevier, New York, NY.

Jeffrey Goldsmith and John Salmon. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications* 7, 5, 14–20.

Venkatraman Govindaraju, Peter Djeu, Karthikeyan Sankaralingam, Mary Vernon, and William R. Mark. 2008. Toward a multicore architecture for real-time ray-tracing. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*. 176–187.

Daniel Hall. 2001. The AR350: Today's ray trace rendering processor. In *Proceedings of the EUROGRAPHICS/SIGGRAPH Workshop on Graphics Hardware - Hot 3D Session*. 1–2.

Johannes Hanika and Alexander Keller. 2007. Towards hardware ray tracing using fixed-point arithmetic. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 119–128.

Michal Hapala and Vlastimil Havran. 2011. Review: Kd-tree traversal algorithms for ray tracing. *Computer Graphics Forum*. 30, 1, 199–213.

Vlastimil Havran. 2000. Heuristic ray shooting algorithms. Ph.D. Dissertation, Faculty of Electrical Engineering, Czech Technical University, Prague.

Jared Heinlya, Shawn Reckera, Kevin Bensemaa, Jesse Porch, and Christiaan Gribble. 2009. Integer ray tracing. *Journal of Graphics, GPU, and Game Tools* 14, 4, 31–56.

Hans Hoffmann, Takebumi Itagaki, David Wood, and Alois Bock. 2006. Studies on the bit rate requirements for a HDTV format with 1920 × 1080 pixel resolution, progressive scanning at 50 Hz frame rate targeting large flat panel displays. *IEEE Transactions on Broadcasting* 52, 4, 420–434.

Qiming Hou, Xin Sun, Kun Zhou, Christian Lauterbach, and Dinesh Manocha. 2011. Memory-scalable GPU spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics* 17, 3, 466–474.

Greg Humphreys and C. Scott Ananian. 1996. Tigershark: A hardware accelerated ray-tracing engine. Senior Independent Work, Princeton University, Princeton, NJ.

IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754^{TM}-2008.

Imagination. 2014. PowerVR Ray Tracing. Retrieved July 25, 2017 from http://www.imgtec.com/powervr/raytracing.asp.

Intel. 2013. Intel® Xeon PhiTMCore Micro-architecture. Retrieved from July 25, 2017 https://software.intel.com/en-us/articles/intel-xeon-phi-core-micro-architecture.

JEDEC. 2009. JEDEC Standard: GDDR5 SGRAM. Retrieved from July 25, 2017 http://www.jedec.org/standards-documents/docs/jesd212.

Tero Karras. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the 4th ACM SIGGRAPH/EUROGRAPHICS Conference on High-Performance Graphics*. 33–37.

Tero Karras and Timo Aila. 2013. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*. 89–99.

Sean Keely. 2014. Reduced precision for hardware ray tracing in GPUs. In *Proceedings of High-Performance Graphics*. 29–40.

Khronos. 2013a. OpenGL Shading Language 4.40 Specification.

Khronos. 2013b. OpenCL: The open standard for parallel programming of heterogeneous systems. Retrieved July 25, 2017 from http://www.khronos.org/opencl/.

Hong-Yun Kim, Young-Jun Kim, and Lee-Sup Kim. 2012. MRTP: Mobile ray tracing processor with reconfigurable stream multi-processors for high datapath utilization. *IEEE Journal of Solid State Circuits* 47, 2. 518–535.

Hong-Yun Kim, Young-Jun Kim, Jie-Hwan Oh, and Lee-Sup Kim. 2013. A reconfigurable SIMT processor for mobile ray tracing with contention reduction in shared memory. *IEEE Transactions on Circuits and Systems-I*. 938–950.

Hiroaki Kobayashi, Ken-ichi Suzuki, Kentaro Sano, Yoshiyuki Kaeriyama, Yasumasa Saida, Nobuyuki Oba, and Tadao Nakamura. 2001. 3dcgiram: An intelligent memory architecture for photo-realistic image synthesis. In *ICCD 2001* (462–467).

Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2013. An energy and bandwidth efficient ray tracing architecture. In *Proceedings of the 5th High-Performance Graphics Conference*. 121–128.

Daniel Kopta, Josef Spjut, Erik Brunvand, and Al Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *Proceedings of the International Conference on Computer Design*. 9–16.

Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*. 28, 2, 375–384.

Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013a. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*. 109–119.

Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013b. Real-time ray tracing on future mobile computing platform. In *Proceedings of the SIGGRAPH Asia Symposium on Mobile Graphics and Interactive Applications*. 56.

Zonghui Li, Yangdong Deng, and Ming Gu. 2017. Path Compression kd-trees with multi-layer parallel construction: A case study on ray tracing. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. Article 16.

G. Liktor and K. Vaidyanathan. 2016. Bandwidth-efficient BVH layout for incremental hardware traversal. In *Proceedings of High Performance Graphics* (51–61), *Eurographics Association*.

Xingyu Liu, Yangdong Deng, Yufei Ni, and Zonghui Li. 2015. FastTree: A hardware KD-tree construction acceleration engine for real-time ray tracing. In *Design, Automation & Test in Europe Conference & Exhibition (DATE'15)*. 1595–1598

J. David MacDonald and Kellogg S. Booth. 1989. Heuristics for ray tracing using space subdivision. In *Graphics Interface Proceedings*.152–163.

Jeffrey A. Mahovsky. 2005. Ray Tracing with Reduced-precision Bounding Volume Hierarchies, Ph.D. thesis, University of Calgary, Calgary, Alberta, Canada.

Jeffrey Mahovsky and Brian Wyvill. 2004. Fast ray-axis aligned bounding box overlap tests with Plücker coordinates. *Journal of Graphics Tools* 9, 1, 35–46.

David McAllister, Jan Tománek, and James Bigler. 2014. Accelerating ray tracing using OptiX. *GPU Technology Conference*.

James A. McCombe. 2014. Introduction to PowerVR Ray Tracing. In *Game Development Conference*.

Microsoft. 2013. Programming Guide for Direct3D 11. Retrieved July 25, 2017 from https://msdn.microsoft.com/zh-cn/library/windows/desktop/ff476345(v=vs.85).aspx.

Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications* 14, 4, 23–32.

G. M. Morton. 1966. A computer oriented geodetic data base and a new technique in file sequencing. Technical Report, IBM Ltd. Ottawa, Canada,

Jae-Ho Nah, Hyuck-Joo Kwon, Dong-Seok Kim, Cheol-Ho Jeong, Jinhong Park, Tack-Don Han, Dinesh Manocha, and Woo-Chan Park. 2014. RayCore: A ray-tracing hardware architecture for mobile devices. *ACM Transactions on Graphics* 30, 6, 162.

Jae-Ho Nah, Jin-Woo Kim, Junho Park, Won-Jong Lee, Jeong-Soo Park, Seok-Yoon Jung, Woo-Chan Park, D. Manocha, and Tack-Don Han. 2015. HART: A hybrid architecture for ray tracing animated scenes. *IEEE Transactions on Visualization and Computer Graphics* 21, 3, 389–401.

Jae-Ho Nah, Jeong-Soo Park, Chanmin Park, Jin-Woo Kim, Yun-Hye Jung, Woo-Chan Park, and Tack-Don Han. 2011. T&I Engine: Traversal and intersection engine for hardware accelerated ray tracing. In *ACM Transactions on Graphics* 30, 6, 160.

Alexandre S. Nery, Nadia Nedjah, Felipe M. G. França, Lech Jozwiak, and Henk Corporaal. 2013. A reconfigurable ray-tracing multi-processor SoC with hardware replication-aware instruction set extension. *Algorithms and Architectures for Parallel Processing Lecture Notes in Computer Science.*

John Nickolls and William J. Dally. 2010. The GPU Computing Era. *IEEE Micro* 30 2, 56–69.

NVIDIA. 2008. NVIDIA Shader Library — HLSL. Retrieved July 25, 2017 from http://developer.download.nvidia.com/shaderlibrary/webpages/hlsl_shaders.html.

NVIDIA. 2016. NVIDIA GeForce GTX 1080. Retrieved July 25, 2017 from http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf.

NVIDIA. 2013. CUDA C Programming Guide. Retrieved July 25, 2017 from http://docs.nvidia.com/cuda/cuda-c-programming-guide/#abstract.

NVIDIA. 2014. Tegra K1: A New Era in Mobile Computing. Retrieved July 25, 2017 from http://www.nvidia.com/content/PDF/tegra_white_papers/Tegra-K1-whitepaper-v1.0.pdf.

J. Pantaleoni and D. Luebke. 2010. HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High-Performance Graphics. Eurographics Association.* 87–95.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics* 29, 4, 66.

PCI-SIG. 2002. PCI Express Base Specification. Revision 1.0. Retrieved July 25, 2017 from http://home.mit.bme.hu/~feher/MSC_RA/External_Bus/pci_express_10.pdf.

Matt Pharr and Greg Humphreys. 2010. *Physically Based Rendering: From Theory to Implementations* (2nd ed.). Elsevier, New York, NY.

Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The tao of parallelism in algorithms. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 12–25.

Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. 2007. Stackless KD-tree traversal for high performance GPU ray tracing. In *Computer Graphics Forum.*

Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. 2005. Ray tracing on programmable graphics hardware. In *ACM Transactions on Graphics* 21, 3, 703–712.

Karthik Ramani, Christiaan P. Gribble, and Al Davis. 2009. StreamRay: A stream filtering architecture for coherent ray tracing. In *ACM Sigplan Notices* 44, 3, 325–336.

John H. Reif, Doug Tygar, and Akitoshi Yoshida. 1994. The computability and complexity of ray tracing. In *Discrete and Computational Geometry.* 265–287.

James Reinders. 2014. Knights Corner: Your path to Knights Landing. Retrieved July 25, 2017 from https://software.intel.com/sites/default/files/managed/e9/b5/Knights-Corner-is-your-path-to-Knights-Landing.pdf.

Yiyi Ren, Xiang Xie, Guolin Li, and Zhihua Wang. 2017. A scan-line forest growing based hand segmentation framework with multi-priority vertex stereo matching for wearable devices. *IEEE Transactions on Cybernetics.* Online First.

Artur Santos, João Marcelo Teixeira, Thiago Farias, Veronica Teichrieb, and Judith Kelner. 2012. Understanding the efficiency of KD-tree ray-traversal techniques over a GPGPU architecture. *International Journal of Parallel Programming* 40, 3, 331–352.

Jörg Schmittler. 2006. SaarCOR: A hardware-architecture for realtime ray tracing. Ph.D. Thesis. Saarland University, Saarbrücken, Saarland, Germany.

Jörg Schmittler, Ingo Wald, and Philipp Slusallek. 2002. SaarCOR: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. 27–36.

Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. 2004. Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. 95–106.

Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*. 27, 3, 1–15.

Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007a. Raytriangle intersection algorithm for modern CPU architecture. In *Proceedings of GraphiCon*. 33–39.

Maxim Shevtsov, Alexei Soupikov, and Alexander Kapustin. 2007b. Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. In *Computer Graphics Forum* 26, 3, 395–404.

Youngsam Shin, Won-Jong Lee, Jaedon Lee, Shi-Hwa Lee, Soojung Ryu, and Jeongwook Kim. 2013. Energy efficient data transmission for ray tracing on mobile computing platform. In *Proceedings of the SIGGRAPH Asia Symposium on Mobile Graphics and Interactive Applications*. 64.

Ryan Smith. 2014. Imagination announces PowerVR wizard GPU family: Rogue learns ray tracing. Retrieved July 25, 2017 from http://www.anandtech.com/comments/7870/imagination-announces-powervr-wizard-gpu-family-rogue-learns-ray-tracing/382257.

Rys Sommefeldt. 2015. A look at the PowerVR graphics architecture: Tile-based rendering. Retrieved from http://blog.imgtec.com/powervr/a-look-at-the-powervr-graphics-architecture-tile-based-rendering.

Josef Spjut, Andrew Kensler, Daniel Kopta, and Erik Brunvand. 2009. TRaX: A multicore hardware architecture for real-time ray tracing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 28, 12. 1802–1815.

Josef Spjut, Daniel Kopta, Erik Brunvand, and Al Davis. 2012. A mobile accelerator architecture for ray tracing. In *Proceedings of 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW'12)*.

Michael Steffen and Joseph Zambreno. 2009. Design and evaluation of a hardware accelerated ray tracing data structure. In *TPCG*.

Michael Steffen and Joseph Zambreno. 2010. A hardware pipeline for accelerating ray traversal algorithms on streaming processors. In *The IEEE 8th Symposium on the Application Specific Processors (SASP'10)*. IEEE, 2010.

Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. 2009. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics* 28, 1, 4.

Tim Todman and Wayne Luk. 2001. Reconfigurable designs for ray tracing. In *the 9th Annual IEEE Symposium on Proceeding of Field-Programmable Custom Computing Machines (FCCM'01)*. IEEE, 2001.

K. Vaidyanathan, T. Akenine-Möller, and M. Salvi. 2016. Watertight ray traversal with reduced precision. In *Proceeding of High-Performance Graph*.

Ingo Wald. 2004. Realtime ray tracing and interactive global illumination. Ph.D. Thesis. Saarland University, Saarbrücken, Saarland, Germany.

Ingo Wald. 2007. On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*. 33–40

Ingo Wald. 2012. Fast construction of SAH BVHs on the Intel many integrated core (MIC) architecture. *IEEE Transactions on Visualization and Computer Graphics* 18, 1, 47–57.

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum* 20, 3, 153–164.

Ingo Wald and Vlastimil Havran. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*. 61–69.

Ingo Wald, William R. Mark, Johannes Guenther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. 2007. State of the art in ray tracing animated scenes. In *Computer Graphics Forum* 28, 6, 1691–1722.

Ingo Wald, Sven Woop, Carsten Benthin, Greg S. Johnson, and Manfred Ernst. 2014. Embree — a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics* 33, 4, 143:1–143:8.

Tong Wang and Yangdong Deng. 2013. Mining effective parallelism from hidden coherence for GPU based path tracing. In *SIGGRAPH Asia 2013 Technical Briefs*.

Yunbo Wang, Chunfeng Liu, and Yangdong Deng. 2014. A feasibility study of ray tracing on mobile GPUs. *SIGGRAPH Asia 2014 Mobile Graphics and Interactive Applications*.

Turner Whitted. 1980. An improved illumination model for shaded display. In *Proceedings of the ACM SIGGRAPH Computer Graphics* 13, 2, 14.

Sven Woop. 2006. DRPU: A programmable hardware architecture for real-time ray tracing of coherent dynamic scenes. Ph.D. Thesis. Computer Graphics Lab, Saarland University, Saarbrücken, Saarland, Germany.

Sven Woop, Erik Bruvand, and Philipp Slusallek. 2006. Estimating performance of a ray-tracing ASIC design. In *Proceedings of the 2006 IEEE/EG Symposium on Interactive Ray Tracing*. 7–14.

Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: A programmable ray processing unit for realtime ray tracing. In *Proceedings of the ACM SIGGRAPH*. 434–444.

Zhefeng Wu, Fukai Zhao, and Xinguo Liu. 2011. SAH KD-tree construction on GPU. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. 71–78.

Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics* 27, 5, 1–11.

Ying Zhou, Dan Wang, Xiang Xie, Yiyi Ren, Guolin Li, Yangdong Deng, and Zhihua Wang. 2014. A fast accurate segmentation method for ordered LiDAR point cloud of large scale scenes. *IEEE Geoscience and Remote Sensing Letters* 11, 11, 1981–1985.

Received March 2016; revised March 2017; accepted May 2017