Intersection Prediction for Accelerated GPU Ray Tracing

Lufei Liu
liulufei@student.ubc.ca
University of British Columbia
Canada

Wesley Chang
wchang22@student.ubc.ca
University of British Columbia
Canada

Francois Demoullin
fdemoullin@ece.ubc.ca
Qualcomm
Canada

Yuan Hsi Chou
yuanhsi@ece.ubc.ca
University of British Columbia
Canada

Mohammadreza Saed
mrsaed@ece.ubc.ca
University of British Columbia
Canada

David Pankratz
pankratz@ualberta.ca
University of Alberta
Canada

Tyler Nowicki
tyler.bryce.nowicki@huawei.com
Huawei
Canada

Tor M. Aamodt
aamodt@ece.ubc.ca
University of British Columbia
Canada

ABSTRACT

Ray tracing has been used for years in motion picture to generate photorealistic images while faster raster-based shading techniques have been preferred for video games to meet real-time requirements. However, recent Graphics Processing Units (GPUs) incorporate hardware accelerator units designed for ray tracing. These accelerator units target the process of traversing hierarchical tree data structures used to test for ray-object intersections. Distinct rays following similar paths through these structures execute many redundant ray-box intersection tests. We propose a ray intersection predictor that speculatively elides redundant operations during this process and proceeds directly to test primitives that the ray is likely to intersect. A key aspect of our predictor strategy involves identifying hash functions that preserve enough spatial information to identify redundant traversals. We explore how to integrate our ray prediction strategy into existing GPU pipelines along with improving the predictor effectiveness by predicting nodes higher in the tree as well as regrouping and scheduling traversal operations in a low cost, judicious manner. On a mobile class GPU with a ray tracing accelerator unit, we find the addition of a 5.5KB predictor per streaming multiprocessor improves performance for ambient occlusion workloads by a geometric mean of 26%.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**; **Graphics processors**; **Modeling and simulation**.

KEYWORDS

ray tracing, graphics, hardware accelerator, GPU

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480097

ACM Reference Format:
Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M. Aamodt. 2021. Intersection Prediction for Accelerated GPU Ray Tracing. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3466752.3480097

1 INTRODUCTION

Real-time ray tracing is poised to change the landscape of video games. GPUs with hardware accelerated ray tracing enable developers to fit more convincing visual effects in their computational budget [9]. Ray tracing produces realistic images through physically accurate rendering algorithms. The common Whitted-style ray tracing [59] models light transport as rays that originate from the camera and interact with the environment. These rays traverse the scene and accumulate color as they intersect with objects and lights. As a result, ray tracing can render visual effects that are not possible in raster-based graphics. For example, rasterization culls objects not visible from the camera and thus cannot track indirect illumination originating from the culled objects.

Ray tracing features abundant parallelism and appears well suited to Single Instruction Multiple Data (SIMD) execution. Modern ray tracing makes use of acceleration structures (AS), such as the Bounding Volume Hierarchy (BVH), which organize scene data to improve the efficiency of searching for ray intersections. AS traversal must be performed for each ray, and rays randomly scattering through a scene will become incoherent[1]—intersecting objects that are potentially far apart. A naive implementation of ray tracing would therefore be inefficient in both power and memory utilization. For example, incoherent rays lead to computational overhead and memory divergence, especially on SIMD architectures. These incoherent rays also place a strain on the memory hierarchy by competing for memory bandwidth and thrashing the cache [55]. However, incoherent rays produce the most impressive graphical effects, creating contention between visual quality and computational footprint [23].

[1] In the graphics community the term "incoherent" refers to a lack of locality.

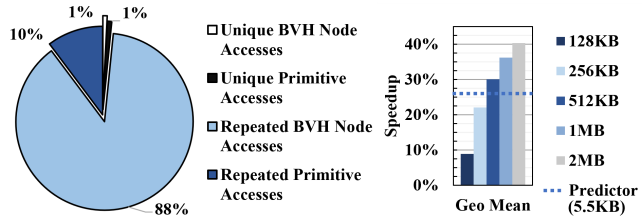


Figure 1: Distribution of memory accesses for ambient occlusion workloads averaged over seven scenes with 1024×1024 pixels and 4 samples per pixel (left). Speedups of varying L1 cache sizes without the predictor, relative to 64KB baseline (right).

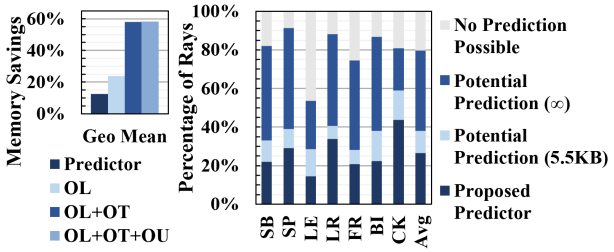


Figure 2: Limit study of proposed predictor, highlighting potential memory savings (left) through improved verified rates (right).

Ambient Occlusion (AO) is one of such effects that benefits from ray tracing acceleration. AO rays are known as *occlusion rays* and test for any object intersection, without requiring the closest intersection to be found. Occlusion rays are common and performance critical as evidenced by the support to skip closest-hit shader execution in modern ray tracing APIs such as Vulkan [57], DXR [36], and OptiX [42]. Basic rendering can be performed efficiently with rasterization then augmented with ray tracing to maintain real-time performance [6]. Many commercial games extend their existing rasterization approach with ray-traced effects [8, 49]; the hardware should mimic this composition.

The AO workload has high computational requirements because each intersection point must be sampled with many rays. Furthermore, rays for AO are typically short and exhibit significant redundancy in the AS nodes that they visit. Figure 1 illustrates the distribution of memory access types for each unique ray in an AO workload, averaged over seven scenes. *Repeated BVH Node Accesses* in Figure 1 (left) form around 88% of memory accesses. Since they are not part of the final ray intersection computation, there is an opportunity to improve performance by predicting the traversal and skipping them. Simply using a cache would require a prohibitively large structure to achieve similar speedups seen in Figure 1 (right) due to the large working set size.

Prior works attempt to accelerate ray tracing with dedicated hardware to speed up AS traversal and ray intersection tests [26, 28, 38, 39, 50, 54, 61]. However, these designs are implemented as individual accelerators in isolation from the GPU. Another area of research explores efficient use of GPU hardware by reducing ray incoherence [1, 2, 7, 20, 31, 34, 44, 58] or creating optimized acceleration structures that better fit the SIMD nature of a GPU [30, 62]. However, current ray tracing performance is still insufficient for real-time rendering on "AAA games" [56].

We propose a ray intersection predictor to exploit redundancy in occlusion ray traversal and improve memory hierarchy utilization. Unlike previous inter-frame memoization works in raster graphics [3, 4], our predictor focuses on redundancies in the current frame. We leverage the property of occlusion rays that they only test for any intersection rather than the closest. The predictor forms its predictions using results from previous AS traversals of similar rays—as determined by a hashing scheme. A successful prediction may take a ray directly to a leaf node and elide the entire traversal. Mispredictions can recover by restarting from the root node. After the prediction, our proposed predictor performs repacking between warps currently traversing the AS to improve SIMT efficiency.

Figure 2 plots results from a limit study evaluating the potential of ray prediction (details in Section 6.3). The results indicate 38% of rays could be predicted using a predictor with a capacity of 5.5KB, idealized only with the ability to perform oracle lookups that can always identify the correct entry in the table that the ray intersects if one exists (*Potential Prediction (5.5KB)*). In contrast, the implementable 5.5KB ray predictor we propose (Section 4) achieves a 26% speedup while identifying intersections for 27% of rays (*Proposed Predictor*).

To evaluate our predictor, we model a baseline ray tracing unit that accelerates ray tracing within the context of a modern GPU, similar to the NVIDIA RT Core [9]. We demonstrate that the cycle-level simulation results of our ray tracing unit in isolation correlate with the performance of an NVIDIA RT Core. Modeling a GPU with a ray tracing unit supports realistic workloads that implement hybrid rasterization and ray tracing. Our simulator is available at https://github.com/ubc-aamodt-group/ray-intersection-predictor on Github.

The contributions of this paper are as follows:

- We propose and evaluate a ray intersection predictor module that speculatively skips ray-box intersection tests in a BVH tree traversal and proceeds directly to nodes deep in the AS.
- We introduce a warp repacking extension in the predictor that provides similar work distribution amongst threads in a warp and creates memory access coalescing opportunities.
- We model a detailed baseline ray tracing unit in the cycle-level GPU simulator GPGPU-Sim [24] correlated with the NVIDIA RTX 2080Ti.

Section 2 introduces relevant background information on ray tracing. Section 3 and 4 describe our proposed ray intersection predictor. Section 5 details our methodology with a description of our baseline ray tracing unit, followed by results in Section 6 and related works in Section 7.

2 BACKGROUND & MOTIVATION

This section gives an overview of GPU architecture and ray tracing. We motivate our predictor by describing the challenges of ray-traced ambient occlusion.

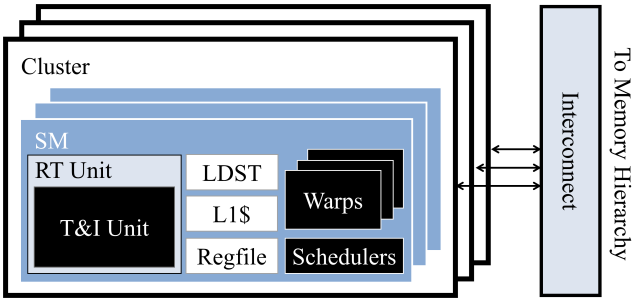


Figure 3: GPU architecture with RT unit

2.1 GPU Architecture

GPUs are massively parallel processors consisting of many streaming multiprocessors (SMs). Figure 3 shows that multiple warps can execute on a single SM. Each warp consists of 32 threads that execute in lockstep, or SIMT fashion. Warp schedulers issue warps in the SM and memory related units such as the load/store unit (LDST), L1 cache/shared memory, and register file. Multiple SMs form a cluster, and each cluster connects to the memory hierarchy through the interconnect.

RT Cores are specialized units that accelerate ray tracing in NVIDIA GPUs, but NVIDIA provides few details of their implementation. Therefore, we model our own version of a ray tracing accelerator, the RT unit, detailed in Section 5.1.

2.2 Ray Tracing

Ray tracing simulates global lighting effects by tracing rays of light in a scene. The number of rays traced per pixel and the number of pixels in the image determine the quality. For a 1024×1024 image with four ray samples per pixel (SPP), more than four million rays need to be traced. A naive implementation would require over one trillion intersection tests to render a scene such as Crytek Sponza with 262K triangles. Therefore, primitives are often organized into an acceleration structure (AS), further detailed in Section 2.4. Rays traverse this AS to optimize intersection testing. An efficient hardware implementation of this AS traversal is critical for real-time ray tracing applications.

Rays are mathematically characterized as a semi-infinite line of $o + t \cdot d$ with an origin, direction, and length. Even though rays traverse through the AS individually, rays with similar origins (o) and directions (d) take similar paths through the AS. For these rays, there is an opportunity to memoize traversal and optimize future rays in the frame. This memoization benefits occlusion rays, such as AO and shadow rays, because the search terminates upon finding the first intersection, allowing the entire traversal to potentially be skipped.

2.3 Ambient Occlusion

Ambient lighting is the base level of illumination in a scene, often approximated by the artist as a constant value. Ambient occlusion is a visual effect where crevices appear darker because less ambient light can reach them. Ray tracing produces AO by tracing many short rays covering a hemisphere that originate from the point of interest to determine the amount of occlusion. Unlike other rays, occlusion rays do not require the closest-hit point. Any ray-object intersection in AO indicates that the point is shadowed from some ambient light. The proportion of rays that intersect objects represents the amount of blocked ambient light.

This form of global AO cannot be accurately implemented in raster-based graphics because it requires global information. Alternatively, screen-space AO is available to raster-based graphics but produces poor results due to fundamental problems such as sampling behind objects or outside of the screen [23]. In ray tracing, global AO is costly to compute due to the large number of rays required to achieve a high quality result.

Algorithm 1: BVH Traversal for Occlusion Rays

Input: *ray*, *rootNode* of the BVH
Output: *hit*: whether or not the ray hit a triangle

```
1  hit ← false
2  node ← rootNode
3  tStack ← ∅
4  while node ≠ ∅ do
5      while node is not leaf do
6          foreach child in node do
7              hitNode ← RayBoxTest(ray,
                child.boundingBox)
8              if hitNode then tStack.push(child)
9          node ← tStack.pop()
10     while node is leaf do
11         foreach triangle in node do
12             hit ← RayTriTest(ray, triangle)
13             if hit then break
14         node ← tStack.pop()
15         if hit then node ← ∅
```

2.4 Ray Traversal Algorithm

Acceleration structures reduce the number of ray-primitive test computations required to find an intersection, and the Bounding Volume Hierarchy (BVH) is the current AS standard for ray tracing [35]. BVH trees enclose primitives in leaf nodes and recursively bounds lower level axis-aligned bounding boxes (AABBs) with larger AABBs. If a ray misses an interior node of the tree, it will also miss the enclosed child nodes. This property reduces the time complexity of intersection testing from linear to logarithmic, making the BVH critical to the performance of real-time ray tracing.

The BVH traversal algorithm can be implemented in software as a while-while loop [2], described in Algorithm 1 for occlusion rays. The outer while loop iterates until the ray has completed its traversal. The inner while loops perform a depth-first traversal through the BVH. Depth-first traversal often requires a per-thread traversal stack (*tStack*) or potentially a bit trail for binary trees [27]. If the first inner while loop reaches a leaf node, then a second inner while loop tests for primitive intersections. Traversal continues until all ray-primitive intersections are identified, or just any ray-primitive intersection for occlusion rays. If no ray-primitive intersections

occur, the ray is considered to have *missed* the scene. We refer the reader to [52] for more background on ray tracing.

For efficiency during traversal, a ray visits the child node closer to the ray origin first. The first intersection may not be the closest in the case that children nodes overlap. Consequently, each ray-primitive intersection is only a candidate, and all candidates must be identified to determine the actual closest-hit. Fortunately, BVH trees can be implemented using balanced nodes and have predictable memory usage. We choose to use BVH trees for this reason and because they are commonly used in practice, as evident in OptiX [42].

The NVIDIA RT Core follows the while-while algorithm closely. It features a traversal unit with *Box Intersection Evaluators* and an intersection unit with *Triangle Intersection Evaluators* [9]. The RT Core begins with a *Ray Query* from the streaming multiprocessor (SM) and triggers the traversal process. The RT Core fetches and decodes BVH nodes from memory and performs the appropriate intersection tests repeatedly until a closest-hit is found or a miss is confirmed. This result is returned to the SM for processing and shading.

2.5 Accelerating Ray Tracing

Reflection rays are often incoherent because they bounce in random directions. They access different nodes and parts of the AS, causing memory divergence and low SIMT efficiency on GPUs. Prior work attempts to organize these rays into more coherent packets [1, 2, 16, 44]; however, they do not address the concern of high memory and compute latencies noted in [17] as a major source of inefficiency. Our solution approaches the problem differently by skipping AS traversal for similar rays, which reduces the overall latency, and can be combined with these techniques to tackle divergence issues as well.

Our predictor is comparable to virtual address translation and page-table walking [45, 46]. Rays traverse AS nodes in the same manner that address translations walk down the page table. The predictor table, which stores previous ray traversal results, is comparable to the TLB that stores prior virtual-to-physical address translations. A pointer cache [10] similarly stores pointer dereferences. However, unlike the TLB and pointer cache, we do not encounter the exact same ray again, so we cannot reuse stored results except to make predictions. Each of the millions of rays are likely to be entirely unique and our hashing scheme attempts to create collisions between similar rays much like constructive aliasing. When an intersection for a similar ray is correctly predicted, we skip the rest of the traversal for that ray, resulting in fewer memory accesses and less computation.

3 RAY INTERSECTION PREDICTOR ALGORITHM

In this section, we describe an abstract overview of the ray intersection predictor algorithm and illustrate how it can skip traversing a significant portion of BVH interior nodes.

As shown in Figure 4, before a ray begins traversing the BVH, we first compute its hash and perform a lookup in the predictor table. If an entry corresponding to the hash exists, a prediction with one or more predicted nodes is returned to be evaluated. The ray traverses

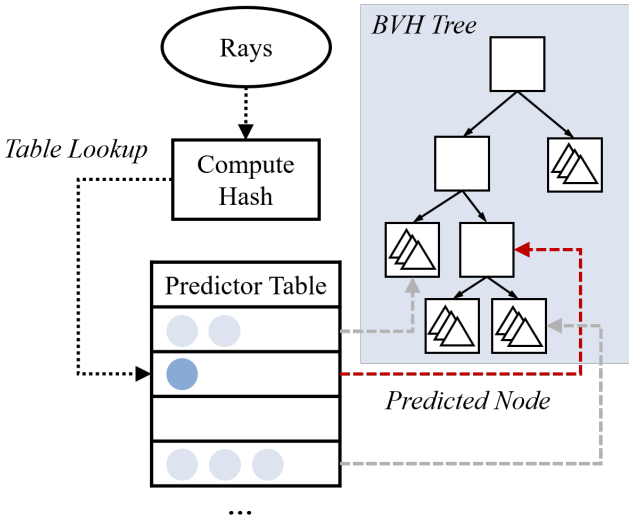


Figure 4: Using the predictor to skip interior nodes. If an entry in the predictor table is found for a ray hash, the ray begins traversing from the predicted nodes pointed by the red arrow.

the BVH from the predicted nodes until it finds an intersection. If found, traversal is complete, and interior nodes are successfully skipped. Otherwise, it is unknown whether the ray missed the scene or if it simply failed to find an intersection using the prediction. In this case, the ray must perform the entire traversal from the root node as it would have without the predictor. If an intersection is found, the predictor table is updated with the corresponding node. We can predict both interior and leaf nodes, which we discuss further in Section 4.3.

First, we define terms for the predictor. A ray *hits* if it intersects the scene, regardless of whether the predictor is used. Only rays that *hit* can skip BVH nodes. Rays are *predicted* if a prediction is found in the predictor table, *verified* if the ray finds an intersection using the prediction, and *mispredicted* if the ray is predicted but not verified. A good predictor should verify many rays, but the percentage of verified rays is limited by the percentage of predicted and hit rays.

We can estimate the number of BVH nodes skipped by a predictor as follows. Let p and v be the percentage of all rays that are predicted and verified for a given render, respectively. Also, a ray must, on average, fetch n nodes on a full traversal, evaluate k predictions from the predictor entry, and fetch m nodes when traversing from each of the predictions. We divide this into three cases:

(1) $(1-p)\%$ of rays are not predicted and traverse n nodes.
(2) $v\%$ of rays are verified and traverse only km nodes.
(3) $(p-v)\%$ of rays are mispredicted and must traverse the predicted nodes as well as the full traversal, for a total of $km+n$ nodes.

Combining these, the number of nodes N that an average ray traverses can be estimated as:

$$\begin{aligned} N &= (1-p)n + vkm + (p-v)(km+n) \\ &= n - pn + vkm + pkm + pn - vkm - vn \\ &= n + pkm - vn \end{aligned}$$

As a result, the number of nodes skipped $n - N$ is:

$$n - N = vn - pkm \tag{1}$$

Equation 1 shows the intuitive result that the overall number of nodes skipped is equal to nodes skipped from verified rays minus the overhead of evaluating the predicted nodes. As a result, the number of nodes skipped is increased by having a high verified rate v and decreased by overpredicting (increasing p) or traversing more nodes on predictions (increasing k or m). We explore these tradeoffs in the next section.

4 PROPOSED ARCHITECTURE

This section outlines the challenges in predicting ray intersections and how our proposed hardware predictor overcomes them. We describe the predictor table architecture and hashing schemes, then discuss support for predicting interior nodes with the Go Up Level. Because predictor mispredictions may contribute to divergence of the traversal algorithm, we also discuss how to mitigate this with warp repacking.

4.1 Predictor Table Architecture

Node predictions are stored in a predictor table per SM with the structure in Figure 5. Each row in a set-associative *way* is a *predictor entry* consisting of a valid bit, a ray hash tag, and one or more slots to store predicted nodes (an offset into the BVH tree buffer). The predictor contains entries for previously encountered rays within the same frame. Given that typical ray generation shaders select ray directions using pseudo-random numbers, no two rays in a single frame are likely to be identical. To identify an entry inserted by a prior ray that a new ray is likely to intersect, we employ hashing. When accessing the predictor, the ray parameters are hashed and the resulting hash pattern is employed for predictor lookup. The hash pattern is used to index the table and compared against tags in all ways for the selected row. If a tag match is found, the ray is considered predicted and the corresponding node addresses are returned for verification. As noted earlier, verifying a ray prediction involves traversing the BVH tree starting from the predicted node. During this verification step, the full precision ray parameters are employed while performing intersection tests.

Depending on the hash function, the number of bits n used for the hash may be larger than a table indexed with m bits. In this case, we fold the n-bit hash into m-bits by splitting the hash into $\lceil n/m \rceil$ components and combining them with bitwise-xor, similar to the gshare branch predictor [32].

Although the structure of the ray intersection predictor in Figure 5 may seem, superficially, similar to a cache, the predictor table stores addresses, not node data. Unlike a BTB or more sophisticated branch target predictors (e.g., ITTAGE [51]), the addresses stored by the ray intersection predictor entries identify BVH tree nodes rather than instructions. Unlike a TLB, a ray lookup in the intersection predictor is not guaranteed to find a matching entry even if an entry containing a node that the ray intersects is present in the table.

We compare different predictor table parameters in Section 6 and find that using a 4-way set-associative predictor table, with 1024 total entries, one node per entry, and 15 bits for the tag, performs the best. As shown in Equation 1, while increasing the number of nodes per entry can increase the number of verified rays, each ray must evaluate more nodes.

When using more than one node per entry, a node replacement policy is required to evict predictions from entries. We use 27 bits for each node index, which adequately manages BVH trees with up to 2^{27} = 134 million nodes, or at least 67 million triangles [2]. This choice is ample for modern commercial games, such as Modern Warfare Initial Intel, which uses 24 million triangles per frame [21].

Since there are 32 rays in each warp, there could potentially be up to 32 threads attempting a predictor table lookup or update in each cycle. It is unrealistic to maintain 32 access ports to our predictor table, so we implement a FIFO queue for predictor lookups and a separate queue for predictor updates. We empirically find that using four access ports is ideal. The memory system is not overwhelmed by bursts of prediction verifications, and the hardware area is constrained to a reasonable size. Even if including 32 access ports was feasible, it would not improve performance. Requests would bottleneck in the memory hierarchy, and the predictor unit would stay idle until the next warp is ready.

4.2 Hashing

To identify rays that should share a predictor entry, we explore strategies that quantify ray similarity. Intuitively, rays are similar if they traverse similar parts of the BVH tree. Prior work on offline ray tracing has explored how to improve cache hit rates by sorting rays [44]. As it is unclear which nodes a ray intersects prior to traversal, most ray sorting techniques combine the three dimensional ray origin and the two dimensional ray direction to obtain a sorting key [34]. We similarly encode the ray origin and direction into a single value, but rather than using it to sort rays, we use it to index the predictor table. The goal is to maximize predictor table collisions between similar rays while minimizing collisions between different rays. We evaluate two different hash functions.

4.2.1 Grid Spherical. Figure 6a illustrates the *Grid Spherical* hash function. This hash function combines the quantized ray origin and direction in cartesian coordinates and spherical coordinates, respectively. The ray origin components x, y, z are each mapped to the integer range $[0, 2^n)$ using the scene bounding box, which is represented by the two extreme corners. These are then concatenated to a single value. We refer to this as the Grid Hash block. Likewise, the ray direction components $\theta \in [0, 180), \phi \in [0, 360)$ are quantized by discretizing to an integer and extracting the most significant m bits from integer θ, and $m + 1$ bits from integer ϕ, before being concatenated. The two values are combined using bitwise-xor. The choice of n, the number of bits used for the origin, and m, the number of bits used for the direction, control how *tight*

[2] This assumes the worst case of a perfect binary tree with one triangle per leaf node.

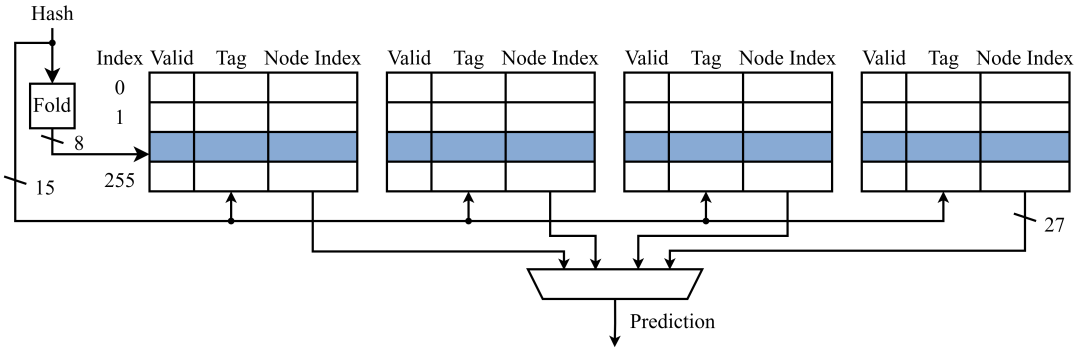
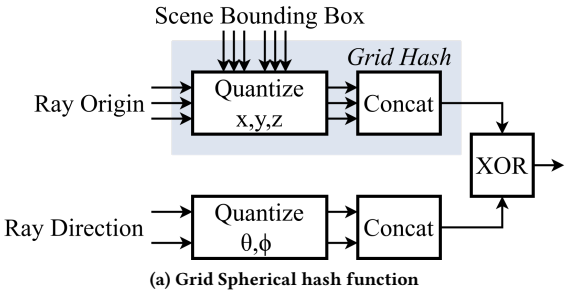
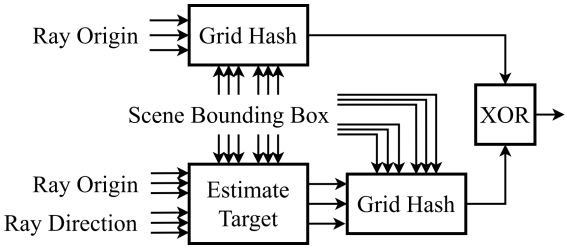


Figure 5: Predictor Table Structure



(a) Grid Spherical hash function



(b) Two Point hash function. See Figure 6a for the Grid Hash block.

Figure 6: Hash functions

the hash function is. That is, using more bits will cause fewer rays to map to the same entry.

4.2.2 Two Point. Figure 6b illustrates the *Two Point* hash function. This hash function is based on the Two Point sorting method [34], which combines the ray origin and an estimated target intersection point. The ray origin is hashed with the same method as in Grid Spherical. The estimated target intersection point is computed as $t = o + r \cdot l \cdot d$, where t is the target point, o is the origin, d is the normalized direction, l is the length of the maximum extent of the scene bounding box, and r is a fixed estimated length ratio to be chosen. This estimated target point is then passed through the Grid Hash block and xor-ed with the hashed origin.

We find that using the Grid Spherical hash function produces slightly better results than using Two Point. We use five origin bits and three direction bits, resulting in a ray hash of 15 bits.

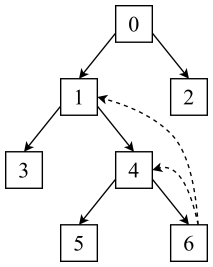


Figure 7: Go Up Level: A ray intersects node 6. For a similar ray, node 6 is predicted with Go Up Level 0, node 4 with level 1, and node 1 with level 2.

The above two hash functions were selected empirically and may not be optimized. We leave the discovery of other hash functions, along with more sophisticated hashing techniques such as combining multiple hash functions or adaptively selecting the number of bits to use, to future work.

4.3 Go Up Level

Even very similar rays can intersect distinct primitives. We observed that such rays will often intersect different leaf nodes, but similar ancestor nodes of the leaves. Motivated by this, we introduce the *Go Up Level*: an optimization in which instead of predicting leaf nodes, we predict interior nodes covering a larger region of space. We define the Go Up Level as the BVH level of the prediction relative to the leaf node with the intersected primitive. In other words, with a Go Up Level of k, the k-th ancestor of the leaf node is stored in the predictor table to be used as a prediction for similar future rays. With a Go Up Level of 0, the leaf node itself is predicted, while with a Go Up Level of 1 and 2, the parent and grandparent are predicted, respectively, as seen in Figure 7.

With a higher Go Up Level, the probability of verifying a ray increases since slightly different leaf nodes can share an ancestor; however, for each prediction, the ray must then traverse a larger portion of the BVH tree to verify the prediction and find an intersection. This tradeoff is presented in Equation 1; increasing the Go Up Level increases v, but also m. We find a Go Up Level of around 3 works best. A more detailed evaluation is included in Section 6.2.1.

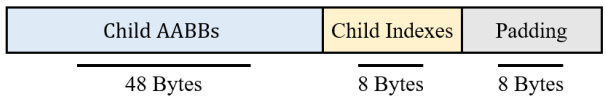


Figure 8: BVH node structure.

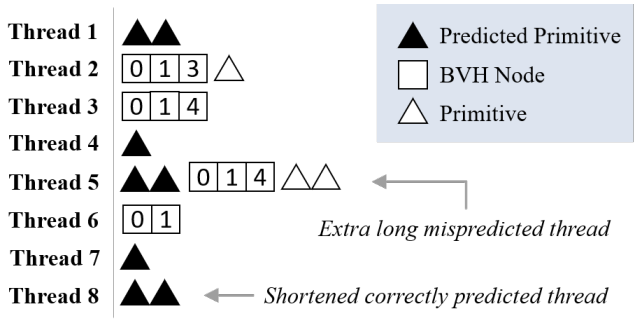


Figure 9: Example ray traversal for a warp of eight threads using the predictor.

To implement Go Up Level, leaf nodes need to track ancestors, but most BVH nodes do not store parent pointers. Alternatively, we can use additional stack memory to keep track of nodes during traversal down the tree. Since we use the Aila and Laine BVH tree [2], we can retrieve an ancestor without using additional stack memory or incurring additional memory accesses by precomputing the k-th ancestor of each node while building the BVH and storing it in the empty padded space. Figure 8 shows the structure of the Aila and Laine node. The ancestor node index is then fetched together with the child indexes on node access.

4.4 Warp Repacking

Naively implementing the predictor leads to incoherence in SIMD execution. Therefore, we propose to restructure warps after prediction to prevent mispredictions from delaying the progress of the entire warp. In contrast to prior warp reformation approaches [15], this restructuring takes place within the accelerator unit during the execution of a single complex instruction per warp, which helps avoid impacting other hardware structures such as the register file.

A warp takes as long to execute as its slowest thread. Given a warp of 32 threads (32 rays), it is likely that there is at least one misprediction where the ray performs a full traversal in addition to evaluating predictions. Figure 9 shows an example warp with eight rays traversing through the BVH tree from Figure 7 after a predictor table lookup. The prediction can comprise of leaf nodes with one or more primitives. Thread 5 mispredicts and executes intersection tests with the predicted primitives before performing regular BVH traversal. Mispredicted rays access more nodes than without the predictor, introducing a "long tail" problem and prolonging warp execution. All other threads must wait for Thread 5 to finish before the warp can complete. SIMT efficiency is also significantly reduced towards the end of the traversal since only mispredicted rays (Thread 5) are still executing.

There are three possible outcomes: an intersection is predicted correctly (Threads 1, 4, 7, 8), mispredicted (Thread 5), or not predicted at all (Threads 2, 3, 6). Ideally, we should keep all correctly predicted rays together so that slow threads do not delay the warp. Rays that were not predicted should also be grouped so that they can continue to benefit from memory coalescing within the warp. Lastly, mispredicted rays should be maintained separately to avoid prolonging traversal for the warp.

Even though we cannot distinguish between correct predictions and mispredictions until after testing the predicted primitives, we find that simply isolating *predicted rays* from *not predicted* rays is sufficient to correct the problematic behavior. For the example in Figure 9, Threads 1, 4, 5, 7, and 8 are separated from Threads 2, 3, and 6. The original warp is no longer full, but the remaining threads benefit from intra-warp memory coalescing and generally complete their traversal in a similar timeframe. This step is necessary for the predictor to provide performance benefits. Results and a comparison to a predictor without repacking are included in Section 6.2.2.

4.4.1 Implementation. We perform repacking in the predictor unit. After threads have completed their predictor table lookup, the *predicted* rays are removed from the warp and *not predicted* rays remain. Predicted rays are queued in the partial warp collector illustrated in Figure 10. This structure only stores the ray IDs of *predicted* rays until it fills up with 32 rays or reaches a timeout, at which point they are queued for traversal. We only track ray IDs because the remainder of the data associated with each ray is stored in the ray buffer, indexed by the ray ID. The collector totals only 0.2% of the register file size, storing up to 64 ray IDs and maintaining a 5-bit timeout counter. We impose a short timeout to ensure timely processing when there are insufficient rays. We find timeout values ranging from 5-30 cycles show insignificant differences. We allow for up to 64 ray IDs in order to store overflowing rays. For example, if the collector had 30 rays in it before the timeout and another warp adds 15 predicted rays, then there would be 45 rays in the collector for one cycle until 32 rays are moved out and queued for traversal.

During repacking, ray IDs are moved between buffers used to track the progress of warps within the RT unit. Since each thread traces a single ray, most of the associated data is already stored in the ray buffer described in Section 5.1.1. In the context of the baseline RT unit, this repacking step only updates the thread index into the ray buffer and bypasses the complex constraints of aligning to the register file. Other warp reformation methods such as Dynamic Warp Formation [15], Thread Block Compaction [14] and Large Warps [40] require threads to remain in their respective lane after shuffling to different warps, but this is not necessary for the predictor repacking proposed here. The predictor repacking persists only within the RT unit, so no architecturally visible register values need to move for subsequent instructions. We assume that threads are arbitrarily grouped into warps for the traversal only, and mixing these groupings does not impact future shading instructions as all results are stored based on ray ID.

4.4.2 Additional Warps. Since warps become under-utilized after the repacking step, there should be enough resources to sustain additional warps in the ray tracing unit. We can increase the limit on the number of simultaneously executing warps. To ensure the

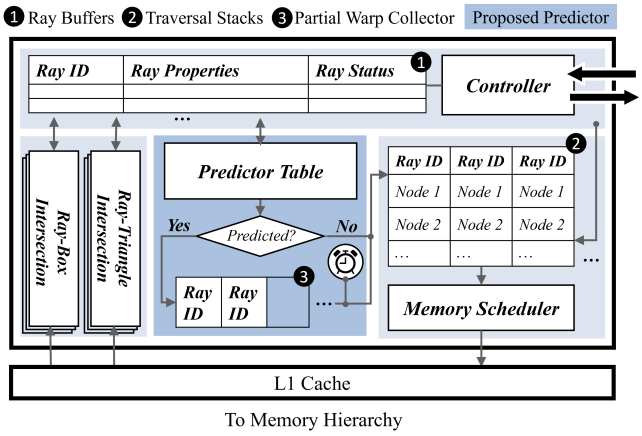


Figure 10: Diagram of the baseline RT unit enhanced with the ray intersection predictor and warp repacking logic.

RT unit is not overwhelmed, only newly created warps can still be dispatched when the RT unit is "full". A new repacked warp can be dispatched as long as more than 32 ray buffer slots are still available. Although this is not as crucial as implementing the basic warp repacking, the benefits of including additional warps are compared in Section 6.2.2.

4.5 Hardware Implementation

Figure 10 summarizes the baseline RT unit (detailed in Section 5.1), augmented with the predictor. The *Ray Buffer* stores *Ray ID*, *Ray Properties*, and *Ray Status*—the current stage of traversal for this ray—for rays in the RT unit. A *Traversal Stack* is allocated for each ray to track their next accesses. Any new rays will be scheduled to perform a predictor table lookup using a hash computed based on the *Ray Properties*. If the lookup is successful and a prediction is made, the predicted nodes are pushed to the top of the ray's *Traversal Stack*. Otherwise, ray traversal proceeds as normal.

5 METHODOLOGY

This section describes the simulator and simulation setup for the ray intersection predictor. We extended GPGPU-Sim 4.0.1 [24] to model an RT unit similar to NVIDIA's RT Core for our baseline, described in Section 5.1. We do not simulate the full graphics pipeline because our predictor is only involved in the ray traced portion of the workload. In hybrid rendering, ray traced AO would typically run after traditional raster graphics passes and is combined using blending [8]. Section 5.2 describes the ray tracing benchmarks, the ray generation process, and the simulator configuration.

For energy analysis, we use GPUWattch [29] to estimate energy changes in the GPU, including cache and DRAM accesses. We model the remaining major components using CACTI7 [5] with a 45nm process. The predictor table, traversal stacks, ray buffer, and partial warp collector from Figure 10 are all modeled as SRAMs with overheads conservatively calculated based on access energy. Warp repacking energy covers the partial warp collector and additional accesses to the ray buffer. We conservatively estimate energy for the intersection units [39] as adders and multipliers [19], and assume all other control elements have relatively negligible energy overheads.

5.1 Ray Tracing Unit

"Magic" instructions that perform hundreds of operations and interact with custom storage units have been established as the solution to energy efficient computers [18]. In the context of ray tracing, the entire traversal and intersection process can be treated as a single `__traceray()` instruction like the NVIDIA RT Core *Ray Query*. For our baseline architecture, we model a ray tracing unit consisting of a traversal block and a ray-triangle intersection block similar to the NVIDIA RT Core. The RT unit is accessed using the special `__traceray()` instruction that passes in all necessary ray data and the root node of the AS. The instruction is intercepted from normal GPU execution, redirected to execute in the RT unit, and the results are written back.

The RT unit is set up as a variable latency function unit, similar to the LDST unit in GPUs. For each iteration of Algorithm 1, the RT unit fetches the relevant bounding volume or primitive data. The RT unit is multiplexed with the LDST unit to access the L1 cache and the remainder of the memory hierarchy.

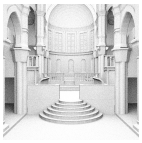
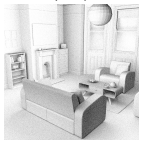
5.1.1 Interface. Our CUDA ray tracing kernel traces one ray per thread. The ray data is retrieved using the thread ID and includes origin, direction, and t-parameters that define the ray length. This data, along with the root node of the BVH tree, is passed into the RT unit for each thread. The RT unit stores the data in the ray buffer, indexed by the ray ID, and tracks warp completion with counters in the controller. The RT unit executes up to eight warps at once, so the ray buffer stores a maximum of $32 \times 8 = 256$ rays.

5.1.2 Traversal Unit. Every new `__traceray()` command received triggers the controller to initialize a traversal stack. We configure the traversal stack to accomodate eight entries, which is sufficient for many traversals, and occasionally overflows to thread-local memory as described in [2]. The traversal begins with the root node, then proceeds depth-first through the BVH structure, pushing unvisited nodes to the traversal stack.

Multiple warps may reside in the RT unit at once and thus the memory scheduler must select which one to prioritize when scheduling memory requests. Motivated by greedy-then-oldest scheduling [48], we prioritize a given warp until all the threads in the warp are waiting for memory requests before proceeding to the next warp. The memory scheduler first selects a warp, then selects the next node using a FIFO queue, merging any identical nodes from different rays of the same warp into a single memory request in an MSHR-like fashion. This is especially important early in the traversal process, where most rays access the same nodes (such as the root node). The next node is then popped from the stack in each iteration, requested from the L1 cache in thread order, and tracked in the ray buffer with its appropriate ray.

When the data returns, the node is broadcasted to the ray buffer. Any matching entries trigger an intersection test, and the ray and node data are forwarded to the intersection unit. The controller updates the ray status and warp-tracking counters based on the intersection test results. When the counters indicate a completed

Table 1: Summary of benchmark scenes

Scene	Sibenik (SB)	Crytek Sponza (SP)	Lost Empire (LE)	Living Room (LR)	Fireplace Room (FR)	Bistro (Interior) (BI)	Country Kitchen (CK)
							
Triangles	75K	262K	225K	581K	143K	1M	1.4M
BVH Tree Depth	23	23	22	23	23	25	27
AO Rays Traced	4.2M	4.2M	3.9M	4.2M	4.2M	4.2M	4.0M

warp, the controller frees the traversal stacks and signals warp completion.

5.1.3 Intersection Unit. The intersection unit accelerates two types of intersection tests, the ray-box test while traversing through BVH nodes and the ray-triangle test once a leaf node is reached. We model this unit after the traversal and intersection units of the T&I engine [39], using a two stage pipelined ray-triangle test and a simple set of adders, multipliers, and comparators for ray-box tests [28]. There are 32 pipelined hardware units for each test type, such that all 32 threads in a warp can compute their ray intersection in parallel. Since the memory scheduler only requests data for a single warp at once, an input queue is unnecessary. At each cycle, the ray and node data are forwarded and the appropriate intersection units are signaled by the controller to perform computation based on the data type. Once the pipelined intersection test is complete, the output is returned through the ray buffer and the controller updates the ray status accordingly. The next node is pushed onto the traversal stack if necessary.

5.1.4 Memory Interface. Memory requests from the RT unit are sent to the L1 cache, which then connects to the remainder of the memory hierarchy. Since there are usually no other competing instructions, the L1 cache can sufficiently manage ray tracing memory requests. We use a 64KB L1 cache and explore other possible configurations, such as a dedicated RT cache, in Section 6.2.3.

5.1.5 Latency. Each element in the RT unit contributes to the overall traversal latency. A minimum traversal would require one cycle to queue, two cycles to access the predictor table, one cycle to access the L1 cache, and two cycles to perform an intersection test. This latency would vary depending on predictor results, the length of the traversal, and memory access latencies. A latency sensitivity study is included in Section 6.2.4.

5.1.6 Correlation. Although the RT unit has a traversal unit and an intersection unit just as the NVIDIA RT Core, we cannot confirm similarities beyond this. Key factors such as how multiple rays co-exist within NVIDIA's RT Core, how memory requests are scheduled, and how ray data is maintained are not disclosed by NVIDIA. Therefore, we do not attempt to model the RT Core exactly, but rather a similar, generalized RT unit.

We simulate primary rays and reflection rays on a few common benchmark scenes listed in Table 1 on our RT unit. We compare this to a basic Vulkan implementation [60] of primary rays and reflection rays on an NVIDIA RTX 2080Ti GPU. The same benchmark scenes with a similar number of rays are compared. Figure 11 shows the rays/s correlation between the two.

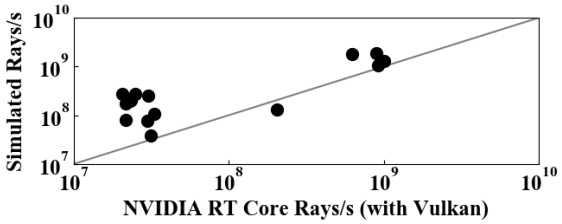


Figure 11: Correlation between simulated ray tracing unit and the NVIDIA RT Core

Table 2: GPGPU-Sim Configuration

# Compute Clusters	2
# SM / Compute Cluster	1
# Streaming Multiprocessors (SM)	2
Max Warps / SM	64
Warp Size	32
Number of Threads / SM	2048
Baseline Scheduler	GTO
Number of Warp Schedulers / SM	4
Number of Registers / SM	32768
Constant Cache Size / SM	64KB
Instruction Cache	128KB, 128B line, 16-way assoc.
L1 Data Cache + Shared Memory	64KB, 128B line, Fully assoc. LRU
L2 Unified Cache	1MB, 128B line, 16-way assoc. LRU
Compute Core Clock	1365 MHz
Interconnect Clock	1365 MHz
L2 Clock	1365 MHz
Memory Clock	3500 MHz
DRAM request queue capacity	64
Interconnect Flit Size	40
Interconnect Input Buffer Size	512
Cluster Ejection Buffer Size	32
# RT units	2
# Predictors	2

We achieve a high correlation coefficient of 0.9, but the test sample is small and the two approaches are not identical. We correlate our baseline to a high-end NVIDIA GPU for validation as it is currently the only consumer GPU with accelerated ray tracing. However, we target our design at mobile GPUs, aiming to speed up ray tracing in the more demanding mobile context.

Table 3: Predictor simulation configurations

Number of Entries	1024
Number of Nodes per Entry	1
Hash Function	Grid Spherical: 5 origin bits, 3 direction bits
Predictor Table Placement Policy	4-way set-associative
Predictor Table Replacement Policy	LRU
Node Replacement Policy	LRU
Access Bandwidth	4 accesses per cycle
Access Latency	1 cycle
Go Up Level	3
Repacking Enabled	Yes

Table 4: Energy analysis breakdown in nJ/ray

Component	Baseline RT unit	Change from Predictor
Base GPU	293.43	-20.01
Predictor table	–	+0.02
Warp repacking	–	+0.05
Traversal stack	0.24	-0.03
Ray buffer	1.71	-0.19
Ray intersections	0.88	-0.11
Total	**296 nJ/ray**	**-20 nJ/ray**

5.2 Simulation

From this baseline RT unit, we then model the proposed ray intersection predictor and assess its effectiveness by comparing results to the baseline. We choose seven common scenes [33] as our benchmarks listed in Table 1. Around four million AO rays are generated for each scene by first computing the primary ray hit point for each pixel in a 1024×1024 viewport, and then creating four AO rays per hit point by random cosine sampling the upper hemisphere surrounding the point. These rays have a length matching 25-40% of the scene bounding box diagonal length to represent relevant areas near the point that could potentially block ambient light. We compare the effect of ray ordering by also sorting the rays with the Aila and Laine Morton order quicksort algorithm [2]. We simulate these rays in GPGPU-Sim, testing for any-hit intersections using the RT unit and predictor.

Table 2 outlines the GPU configurations for GPGPU-Sim and Table 3 lists the predictor settings we simulate.

6 RESULTS

Figure 12 plots the speedup and Figure 13 plots memory accesses for our proposed predictor. These results show that the predictor reduces memory accesses by 13% and overall execution time by a geometric average of 26%. Sorted rays benefit less from the predictor because similar rays are traced close together and do not have an opportunity to train the predictor. Ray sorting is generally advantageous because it reduces divergence. However, recent work has shown that the overhead of ray sorting is larger than the gains it provides when using RTX acceleration [34]. We include sorted ray results to show that our predictor exploits behavior orthogonal to sorting.

The predictor generates an additional 9% of memory accesses, 5.5% of which are wasteful mispredictions on average. However, the net result is a 12% reduction of interior node accesses for ray-box intersection tests and 2% reduction of primitive accesses despite mispredictions. Combined, net memory accesses decrease by 13%, improving performance and diminishing memory system stress. Table 5 shows this result is similar to memory savings calculated using Equation 1 using averages across all scenes.

Approximately 7% of energy is saved using the predictor, and a full breakdown is included in Table 4. Although the predictor requires additional power, DRAM energy still dominates, decreasing the overall energy with shortened execution time.

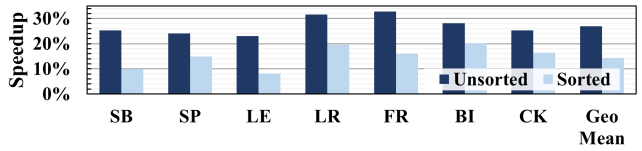


Figure 12: Speedup of proposed ray intersection predictor (with repacking) over baseline RT unit.

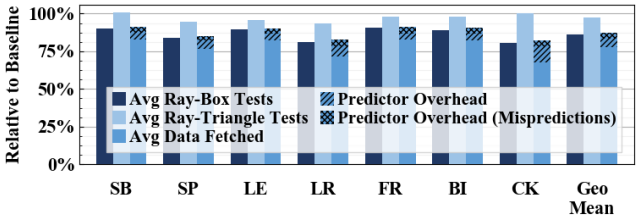


Figure 13: Memory accesses and predictor overheads compared to baseline RT unit.

Table 5: Estimated vs Actual Reductions in Node Accesses

v	n	p	k	m	Estimated	Actual
0.246	28.382	0.955	1	2.810	**4.298**	**3.726**

Table 6: Speedups for different table sizes

Number of Entries	Number of Nodes per Entry		
	1	2	4
512	24.8%	24.6%	23.7%
1024	25.8%	25.3%	24.2%
2048	25.4%	24.9%	23.4%

6.1 Predictor Table Configurations

This section explores varied predictor table configurations.

6.1.1 Table Size. Table 6 compares the number of entries and nodes per entry in the predictor table. The optimal size occurs at 1024 entries with one node per entry. Verified rays increase with the number of nodes, but results worsen because predictions are more expensive. Increasing the number of entries also does not necessarily improve the predictor since the goal is to maximize collisions between similar rays. With 1024 entries and 1 valid bit + 15 tag bits + 1 node per entry with 27 bits = 43 bits per entry, the area overhead of the predictor table is roughly 5.5 KB per SM. To achieve

Table 7: Comparison of placement policies

Policy	Speedup	Predicted	Verified
Direct-mapped	15.9%	58.7%	15.1%
2-way	23.1%	86.3%	22.7%
4-way	25.8%	95.5%	24.6%
8-way	25.5%	96.2%	23.5%

Table 8: Speedups for different hash functions

(a) Grid Spherical Hash Function

Number of Origin Bits	Number of Direction Bits				
	1	**2**	**3**	**4**	**5**
3	19.1%	19.4%	19.3%	16.1%	21.5%
4	20.4%	21.9%	24.0%	22.9%	18.3%
5	22.3%	24.7%	25.8%	23.3%	14.0%

(b) Two Point Hash Function

Number of Origin Bits	Estimated Length Ratio			
	0.05	**0.15**	**0.25**	**0.35**
3	18.8%	19.3%	18.2%	18.2%
4	19.4%	24.7%	20.7%	17.0%
5	23.1%	15.5%	9.0%	6.8%

similar speedups without a predictor, the L1 cache would have to be around 6× larger at 384KB for our benchmarks, as indicated in Figure 1 (right).

6.1.2 Placement Policy. We compare different placement policies in the predictor table: direct-mapped, 2-way, 4-way, and 8-way set-associative. In the direct-mapped predictor table, a tag is still used so that rays with the same index but different hashes will not use the same entry. Table 7 compares speedup between these policies as well as the percentage of predicted and verified rays. For all cases, we use LRU as the replacement policy and find that 4-way set-associative performs best. A larger ratio of verified to predicted rays would be ideal, but the benefits of skipping most of the traversal when verified already outweigh the low cost of evaluating mispredicted nodes.

6.1.3 Node Replacement Policy. As predictions are added to entries in the predictor table, older predictions must be evicted. Although we use one node per entry in our configurations, we test different replacement policies for other configurations that may use multiple nodes per entry. We compare three policies: Least Frequently Used (LFU), LRU, and LRU-K, which keeps track of the last K references [43], and find that the differences between them are insignificant.

6.1.4 Hash Functions. Table 8a and 8b show the results of the two hash functions. Using five origin bits and three direction bits for Grid Spherical gives a good balance in the tightness of the hash function and the largest speedup. Two Point gives comparable results.

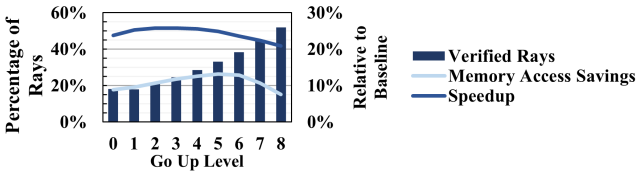


Figure 14: Increased verified rate versus decreased memory access savings for different Go Up Levels.

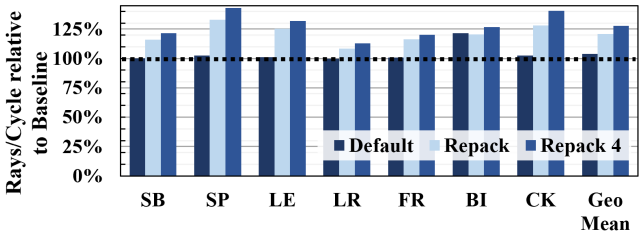


Figure 15: Performance with warp repacking (*Repack*), repacking with four additional warps (*Repack 4*), and no repacking (*Default*) relative to the baseline RT unit.

6.2 Other Predictor Configurations

6.2.1 Go Up Level. Increasing the Go Up Level creates more opportunities to make useful predictions at the expense of more costly mispredictions and reduced memory savings per prediction. When predicted, the ray must still traverse through some interior nodes, adding additional delay on mispredictions. Figure 14 illustrates this tradeoff. The percentage of verified rays increases with Go Up Level, but eventually the memory savings peak. Any further increase of the Go Up Level does not improve performance, and we find that a Go Up Level of 3 performs best. Despite the highest memory savings with level 5 in Figure 14, level 3 performs better because Go Up Level also influences the order and composition of memory requests, which affects the cache hit rate and DRAM utilization.

6.2.2 Warp Repacking. Figure 15 demonstrates the effect of warp repacking on the benchmark scenes. In the *Default* setting, the predictor sometimes causes a slowdown because the elongated mispredicted threads delay completion for the entire warp. After repacking the threads into separate warps (*Repack*, Section 4.4.1), performance improves by a geometric average of 17% from *Default*. The additional performance gains beyond memory access savings arise from a better interleaved mixture of interior and leaf node memory requests in repacked warps, which creates more random accesses and improves bank parallelism in the DRAM by 41%. By allowing additional warps, the performance can be increased by an additional 7% for four additional warps (*Repack 4*, Section 4.4.2).

6.2.3 L1 Cache. Since the majority of ray tracing memory accesses are not unique, implying they could be cached given enough capacity, the cache is crucial. Interfacing the RT unit with the L1 cache performs well when there are no competing memory operations from the rest of the SM pipeline. Alternatively, a specialized RT cache can be placed in the RT unit. We found diminishing returns

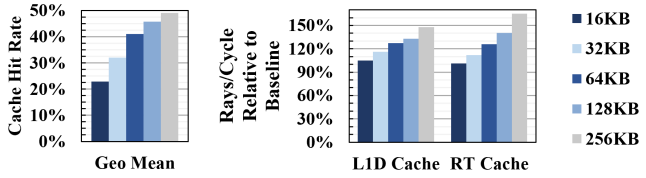


Figure 16: Cache hit rates (left) and speedup (right) for varying cache configurations.

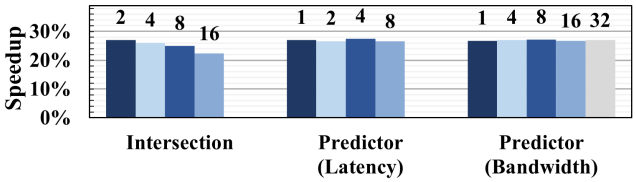


Figure 17: Latency sensitivity for intersection tests, and predictor lookup latency and bandwidth, averaged over all scenes.

for L1 cache sizes larger than 64KB. A full comparison is included in Figure 16.

6.2.4 Latency Sensitivity. Guthe [17] found latency has a significant effect on ray tracing workloads executing on a GPU, and thus we study the impact of the latency of different portions of the RT unit. The leftmost group of bars in Figure 17 show the effect of increasing the latency of intersection tests and the resulting lowered speedups. The numbers over the bars indicate the latency in cycles of the intersection test unit. We also explore varying access latency and bandwidth (rays per cycle) for the predictor (next two sets of bars). The results indicate latency is less important for the predictor than for intersection tests. Similarly, increasing the bandwidth of the predictor has little benefit. Only one prediction is performed per ray while many intersection tests per ray are required for rays without predictions or with mispredictions.

6.2.5 GPU Configurations. Since the predictor table is local per SM, GPU configurations with more SMs perform worse because prediction opportunities are reduced from segregated rays. However, 90% of the reported savings are still retained for up to six SMs. This effect can likely be mitigated by tuning the predictor. We evaluate our predictor in the mobile GPU context but find similar benefits apply to a desktop GPU similar to the NVIDIA RTX 2080 Ti. Using the same predictor configurations, memory accesses decrease by 5% on average.

6.3 Limit Study

Figure 2 plots the results from a limit study of ray prediction. The left graph shows memory savings with both our implementable predictor and three idealized predictors. The right graph shows the percentage of rays verified. Using realistic configurations (*Predictor*) leads to around 13% memory savings and 27% verified rays. However, given the ability to always select a predictor table entry that results in a verified ray (oracle lookup, *OL*, left graph; *Potential Prediction (5.5KB)*, right graph), the percentage of verified rays increases by 11% to 38%. By avoiding overheads of mispredicted rays, memory savings with *OL* nearly double to 24% on average. If we can train the predictor assuming an unbounded predictor table where we always find a BVH node if that node was intersected by a prior ray (oracle training, *OT*, left graph; *Potential Prediction (∞)*, right graph), we can reach up to 58% memory savings. Lastly, if we allow threads to immediately update this unbounded predictor table without accounting for the latency of BVH tree traversal (oracle updates, *OU*), we observe another 0.25% memory savings.

6.4 Other Applications

We also evaluate the predictor for global illumination (GI) applications. For these closest-hit rays, we use predicted intersections to trim the ray's maximum length before traversal rather than predicting the final hit point. Despite being designed for occlusion rays, the predictor achieves 4% average speedup on our scenes for GI with three ray bounces.

7 RELATED WORKS

This paper builds upon preliminary studies of ray prediction by our group [11, 12] that explored idealized predictor structures without considering their implementation in a GPU pipeline. Ray tracing is also a well explored area and this section surveys a variety of works that are related to accelerating ray tracing.

Hardware Accelerated Ray Tracing. Specialized hardware for ray tracing has long been explored. Earlier designs such as SaarCOR [50] and RPU [61] included similar traversal and intersection structures. However, these are implemented as independent devices and do not cooperate with the GPU. Since rasterization is still important for real-time rendering and ray tracing can be used to augment it, ray tracing hardware should not be entirely decoupled from the existing GPU.

In more recent work, Trax [54] and MIMD threaded multiprocessors (TM) [26] are MIMD processors capable of traversing incoherent rays. Nah et al. introduced T&I engine [39], dedicated hardware for traversal and intersection on KD-trees. RayCore [38] and SGRT [28] employ T&I cores to enable ray tracing on mobile devices. Deng et al. show the differences between these architectures in [13]. Kopta et al. [25] change GPU architectures to enable reconfigurable pipelines to reduce energy usage. Shkurko et al. [53] focus on avoiding random memory accesses during ray traversal. They divide acceleration structure into segments and store rays inside the acceleration structure for each segment which enables them to achieve perfect prefetching. Ni et al. [41] partition KD-tree into sub-trees and build a scheduling mechanism to regroup rays on each sub-tree into warps. These works rely on improving performance by speeding up each individual traversal step, whereas we propose to reduce the workload by completely skipping over parts of the traversal. We build on top of a baseline ray tracing unit with similar performance to these works and improve it with a microarchitectural technique of speculative traversal elision[3].

[3]This term is inspired by Speculative Lock Elision [47], except we elide BVH tree traversal as opposed to locks.

GPU Traversal Algorithms. GPU ray tracing greatly benefits from more efficient traversal algorithms. Aila et al. [2] proposed grouping rays into ray packets and traversing rays together, improving memory coherence. Aila et al. [1] improved upon this idea and introduced treelets during tree traversal, batching together rays traversing the same treelets and further reduced divergence. Prior works have also explored GPU ray tracing with a short-stack [22] or without keeping a traversal stack [7, 20], which reduce memory traffic during traversal. These works improve performance via software methods without changing the underlying architecture.

Ray Sorting / Reordering. Ray sorting improves ray coherency and reduces divergence during GPU traversal. Pharr et al. [44] introduced the idea of reordering ray computation to improve ray coherency and increase cache utilization. Garanzha and Loop [16] sorted rays based on ray origin and direction before processing them in packets. Moon et al. [37] took a different approach by sorting rays based on their final hit points. Meister et al. [34] improved on sorting heuristics to further minimize ray incoherence. These works complement our proposed ray intersection predictor.

Acceleration Structure Optimizations. Some works optimize the structure of BVH trees to improve performance. Ylitie et al. [62] explored wide BVH trees to increase SIMD utilization. Lin et al. [30] restructured BVH tree nodes with a novel node splitting technique, reducing the memory footprint of the AS. These techniques should also work in parallel with our proposed ray intersection predictor.

Warp Repacking. Warp repacking aims to reorganize threads in diverged warps to regain SIMT efficiency. Authors in [14, 55] explored this idea in a GPU compute context. Wald [58] took the idea and extended it to a CUDA path tracer. Lu et al. [31] repacked threads from different warps with the same traversal state for higher ray coherence. We took inspiration from these works and extended them for our predictor.

8 CONCLUSION

In this paper, we presented a ray intersection predictor to accelerate ray-traced AO. By storing a history of ray hashes and their corresponding ray intersection results in a predictor table, we can skip BVH nodes for future rays. Naively implementing the predictor in SIMD execution leads to incoherence, and thus warps are restructured after predictions, separating out predicted rays. Simulation results show that the predictor achieves, on average, a 26% speedup over the baseline RT unit and reduces memory accesses by 13%.

There are still many exciting directions to explore on ray intersection prediction. The limit study we presented indicates potential for improvements through better predictor structures. Moreover, in this paper we mainly focused upon AO, one of the many lighting effects that benefit from ray tracing. While we presented results for an extension of the predictor to global illumination, which requires closest-hit points, more can presumably be achieved. Exploring the predictor on dynamic scenes and animations is also likely a compelling avenue. Predictor states could potentially be preserved between frames and the predictor retrained only for dynamic elements.

ACKNOWLEDGMENTS

This research is funded in part by grants from Huawei Technologies, Activision, and the Natural Sciences and Engineering Research Council of Canada (NSERC). Tor Aamodt serves as a consultant for Huawei Technologies Canada.

A ARTIFACT APPENDIX

A.1 Abstract

We model the baseline ray tracing unit and ray intersection predictor in GPGPU-Sim. We evaluate our model on various machines running Ubuntu 18.04 and collect results to generate our figures. In this section, we provide detailed instructions on how to use our simulator and generate Figure 12, which contains the main results of the paper. In addition, we provide a ready-to-run Docker image with all dependencies installed and the simulator compiled. There are no specific hardware requirements for this process.

A.2 Artifact check-list (meta-information)

- **Program:** provided
- **Compilation:** gcc/g++, cuda
- **Model:** graphical models, included
- **Run-time environment:** Ubuntu 18.04, no root access required
- **Hardware:** CPU with >30GB RAM
- **Metrics:** execution time
- **Output:** file, plot generated with provided scripts
- **Experiments:** provided scripts and some manual steps
- **How much disk space required (approximately)?:** 30GB
- **How much time is needed to prepare workflow (approximately)?:** less than 1 hour
- **How much time is needed to complete experiments (approximately)?:** less than 1 day (in parallel), around a week (individually)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** GPGPU-Sim (BSD-3)
- **Data licenses (if publicly available)?:** Model files (CC BY 3.0)
- **Archived (provide DOI)?:** 10.5281/zenodo.5147579

A.3 Description

A.3.1 How to access. Our simulator can be found on Github: https://github.com/ubc-aamodt-group/ray-intersection-predictor The ray tracing benchmark, models, and simulator are also all included in our Docker image located in Zenodo: https://doi.org/10.5281/zenodo.5147579

A.3.2 Hardware dependencies. There are no specific hardware dependencies for this project. However, the simulation of larger ray files can take multiple hours for each model and require more RAM.

A.3.3 Software dependencies. We run our simulator on Ubuntu 18.04 and have not tested it on other platforms. Our application requires several dependencies to be installed, including all the dependencies of GPGPU-Sim:

- gcc/g++
- CUDA Toolkit 10.0
- Embree v3.13.0
- libtbb
- zlib
- flex
- bison

- makedepend
- python3

We also require Docker to be installed to run the Docker container. These dependencies are all already installed in our Docker image.

A.3.4 Models. We include `.obj` files for the graphics models we use in our paper. These can also be obtained from [33]. We also include `.ray_files`, which contain the specific rays we simulated in our paper. These rays are randomly generated as described in Section 5.2.

A.4 Installation

The entire system and all dependencies are included in our Docker image, which can be downloaded from Zenodo:
https://doi.org/10.5281/zenodo.5147579
Load the Docker image and start the container with a bash terminal.

```
sudo docker load < rtpredictorimage.tar.gz
sudo docker run -it rtpredictor:latest /bin/bash
```

The environment variables can be set up manually if necessary:

```
export CUDA_INSTALL_PATH=/usr/local/cuda-10.0
export LD_LIBRARY_PATH=/home/tools/embree-3.13.0.x86_
64.linux/lib:/home/tools:$LD_LIBRARY_PATH
source /home/simulator/setup_environment debug
```

A.5 Experiment workflow

Inside the Docker container are the ray tracing application and simulator (`/home/rtao` and `/home/simulator`, respectively). Sample scripts to run various configurations are included in the `/home/performance_sweep` folder. In general, the configurations are set in the `gpgpusim.config` file, and the application can be simulated with the following command:

```
./magic_CWBVH --anyhit -m $MODEL -f $RAY_FILE
```

Alternatively, two shell scripts have been prepared to sweep models for the performance results.

```
/home/performance_sweep/sweep_scenes_small.sh
/home/performance_sweep/sweep_scenes_large.sh
```

A.6 Evaluation and expected results

Sweeping the models for performance results will collect data that generates Figure 12. These results show the performance gains of using our ray intersection predictor, relative to the baseline ray tracing unit, for both sorted and unsorted rays. Run the script to plot the data into a bar chart saved as `results.png` that matches Figure 12.

```
python3 /home/performance_sweep/plot_results_bar.py
```

A.7 Experiment customization

The experiment can be customized by adjusting the parameters found in the `gpgpusim.config` file to control different parameters discussed in the paper. Other graphical models can also be used, provided with a `.obj` file matching the format of the existing sample models in the Docker image.

REFERENCES

[1] Timo Aila and Tero Karras. 2010. Architecture considerations for tracing incoherent rays. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 113–122.

[2] Timo Aila and Samuli Laine. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 145–149.

[3] Martí Anglada, Enrique de Lucas, Joan-Manuel Parcerisa, Juan L Aragón, Pedro Marcuello, and Antonio González. 2019. Rendering elimination: Early discard of redundant tiles in the graphics pipeline. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*. 623–634.

[4] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2014. Eliminating Redundant Fragment Shader Executions on a Mobile GPU via Hardware Memoization. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*. 529–540.

[5] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[6] Colin Barré-Brisebois, Henrik Halén, Graham Wihlidal, Andrew Lauritzen, Jasper Bekkers, Tomasz Stachowiak, and Johan Andersson. 2019. Hybrid rendering for real-time ray tracing. In *Ray Tracing Gems*. 437–473.

[7] Nikolaus Binder and Alexander Keller. 2016. Efficient Stackless Hierarchy Traversal on GPUs with Backtracking in Constant Time. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 41–50.

[8] Blizzard Entertainment. 2021. Engineer's Workshop: Enabling Ray-Traced Shadows in Shadowlands. Retrieved April 11, 2021 from https://worldofwarcraft.com/en-us/news/23494819/engineers-workshop-enabling-ray-traced-shadows-in-shadowlands

[9] John Burgess. 2020. Rtx on—the nvidia turing gpu. *IEEE Micro* 40, 2 (2020), 36–44.

[10] Jamison Collins, Suleyman Sair, Brad Calder, and Dean M Tullsen. 2002. Pointer cache assisted prefetching. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 62–73.

[11] Francois Demoullin, Ayub Gubran, and Tor M Aamodt. 2019. Hash-Based Ray Path Prediction: Skipping BVH Traversal Computation by Exploiting Ray Locality. *arXiv preprint arXiv:1910.01304* (2019).

[12] Francois M Demoullin. 2020. *Hybrid rendering: in pursuit of real-time raytracing*. Master's thesis. University of British Columbia.

[13] 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 Computing Surveys (CSUR)* 50, 4 (2017), 1–41.

[14] Wilson WL Fung and Tor M Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*. 25–36.

[15] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. 2007. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 407–420.

[16] Kirill Garanzha and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum* 29, 2 (2010), 289–298.

[17] Michael Guthe. 2014. Latency Considerations of Depth-first GPU Ray Tracing.. In *Eurographics (Short Papers)*. 53–56.

[18] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*. 37–47.

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. arXiv:1602.01528

[20] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. 2011. Efficient Stack-Less BVH Traversal for Ray Tracing. In *Proc. Spring Conference on Computer Graphics (SCCG)*. 7–12.

[21] Hodgson, David. 2019. Modern Warfare Initial Intel: Call of Duty: Modern Warfare's game engine is put through its paces. Retrieved April 14, 2021 from https://blog.activision.com/call-of-duty/2019-06/Initial-Intel-Call-of-Duty-Modern-Warfares-game-engine-is-put-through-its-paces

[22] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. 2007. Interactive K-d Tree GPU Raytracing. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*. 167–174.

[23] Alexander Keller, Timo Viitanen, Colin Barré-Brisebois, Christoph Schied, and Morgan McGuire. 2019. Are We Done with Ray Tracing?. In *ACM SIGGRAPH Courses*.

[24] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. 2020. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*. 473–486.

[25] Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. 2015. Memory considerations for low energy ray tracing. In *Computer Graphics Forum*, Vol. 34. Wiley Online Library, 47–59.

[26] D. Kopta, J. Spjut, E. Brunvand, and A. Davis. 2010. Efficient MIMD architectures for high-performance ray tracing. In *Proc. IEEE Conf. on Computer Design (ICCD)*. 9–16.

[27] Samuli Laine. 2010. Restart trail for stackless BVH traversal. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 107–111.

[28] Won-Jong Lee, Youngsam Shin, Jaedon Lee, Jin-Woo Kim, Jae-Ho Nah, Seokyoon Jung, Shihwa Lee, Hyun-Sang Park, and Tack-Don Han. 2013. SGRT: A mobile GPU architecture for real-time ray tracing. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 109–119.

[29] Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling Energy Optimizations in GPGPUs. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*. 487–498.

[30] Daqi Lin, Konstantin Shkurko, Ian Mallett, and Cem Yuksel. 2019. Dual-Split Trees. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*. Article 3, 9 pages.

[31] Yashuai Luü, Libo Huang, Li Shen, and Zhiying Wang. 2017. Unleashing the power of GPU for physically-based rendering via dynamic ray shuffling. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 560–573.

[32] Scott McFarling. 1993. *Combining branch predictors.* Technical Report. WRL Technical Note TN-36.

[33] Morgan McGuire. 2017. Computer Graphics Archive. https://casual-effects.com/data

[34] Daniel Meister, Jakub Boksansky, Michael Guthe, and Jiri Bittner. 2020. On Ray Reordering Techniques for Faster GPU Ray Tracing. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*. 1–9.

[35] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J Doyle, Michael Guthe, and Jiří Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. In *CGF*, Vol. 40. 683–712.

[36] Microsoft. 2021. DirectX Raytracing (DXR) Functional Spec: TraceRay control flow. Retrieved April 11, 2021 from https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html

[37] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. 2010. Cache-Oblivious Ray Reordering. *ACM Transactions on Graphics (TOG)* (2010).

[38] 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 (TOG)* 33, 5 (2014), 1–15.

[39] 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 *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*. 1–10.

[40] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N Patt. 2011. Improving GPU performance via large warps and two-level warp scheduling. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 308–317.

[41] Yufei Ni, Yangdong Deng, and Zonghui Li. 2021. Agglomerative Memory and Thread Scheduling for High Performance Ray Tracing on GPUs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2021).

[42] NVIDIA. 2020. NVIDIA OptiX 7.2 - Programming Guide. Retrieved April 11, 2021 from https://raytracing-docs.nvidia.com/optix7/guide/index.html#device_side_functions

[43] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record* 22, 2 (1993), 297–306.

[44] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. 1997. Rendering Complex Scenes with Memory-Coherent Ray Tracing. In *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*. 101–108.

[45] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proc. ACM Conf. on Arch. Support for Prog. Lang. and Op. Sys. (ASPLOS)*. 743–758.

[46] J. Power, M. D. Hill, and D. A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*. 568–578.

[47] R. Rajwar and J. R. Goodman. 2001. Speculative lock elision: enabling highly concurrent multithreaded execution. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 294–305.

[48] Timothy G Rogers, Mike O'Connor, and Tor M Aamodt. 2012. Cache-conscious wavefront scheduling. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 72–83.

[49] Schmid, Jan and Deligiannis, Johannes. 2019. It Just Works: Ray-Traced Reflections in 'Battlefield V'. Retrieved April 11, 2021 from https://www.gdcvault.com/play/1026282/It-Just-Works-Ray-Traced

[50] J Schmittler, I Wald, and P Slusallek. 2002. SaarCOR: a hardware architecture for ray tracing. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics hardware (HWWS)*. 27–36.

[51] André Seznec. 2011. A 64-Kbytes ITTAGE indirect branch predictor. In *JWAC-2: Championship Branch Prediction*. JILP.

[52] Peter Shirley. 2016. Ray tracing in one weekend. *Amazon Digital Services LLC* 1 (2016).

[53] Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. 2017. Dual Streaming for Hardware-Accelerated Ray Tracing. In *Proc. ACM Conf. on High Performance Graphics (HPG)*.

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

[55] Michael Steffen and Joseph Zambreno. 2010. Improving simt efficiency of global rendering algorithms with architectural support for dynamic micro-kernels. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*. 237–248.

[56] Geeky Gaming Stuff. 2021. What Are AAA Games? A Guide To Unofficial Terminology. https://geekygamingstuff.com/what-are-aaa-games/

[57] The Khronos Vulkan Working Group. 2021. <Vulkan 1.2.174 - A Specification(with KHR extensions): Ray Result Determination. Retrieved April 11, 2021 from https://www.khronos.org/registry/vulkan/specs/1.2-khr-extensions/html/chap33.html

[58] Ingo Wald. 2011. Active thread compaction for GPU path tracing. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. 51–58.

[59] Turner Whitted. 2005. An improved illumination model for shaded display. In *ACM SIGGRAPH Courses*. 4–es.

[60] Sascha Willems. 2019. Vulkan examples for ray traced shadows and reflections using VK_NV_ray_tracing. Retrieved November 4, 2020 from https://www.saschawillems.de/blog/2019/04/27/vulkan-examples-for-ray-traced-shadows-and-reflections-using-vk_nv_ray_tracing/

[61] Sven Woop, Jörg Schmittler, and Philipp Slusallek. 2005. RPU: a programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (TOG)* 24, 3 (2005), 434–444.

[62] Henri Ylitie, Tero Karras, and Samuli Laine. 2017. Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs. In *Proc. ACM Conf. on High Performance Graphics (HPG)*. Article 4, 13 pages.