Treelet Prefetching For Ray Tracing

Yuan Hsi Chou
yuanhsi@ece.ubc.ca
University of British Columbia
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 is traditionally only used in offline rendering to produce images of high fidelity because it is computationally expensive. Recent Graphics Processing Units (GPUs) have included dedicated accelerators to bring ray tracing to real-time rendering for video games and other graphics applications. These accelerators focus on finding the closest intersection between a ray and a scene using a hierarchical tree data structure called a Bounding Volume Hierarchy (BVH) tree. However, BVH tree traversal is still very costly due to divergent rays accessing different parts of the tree, with each ray following a unique pointer-chasing sequence that is difficult to optimize with traditional methods. To address this, we propose treelet prefetching to reduce the latency of ray traversal. Treelets are smaller subtrees created by splitting the BVH tree. When a ray visits a treelet root node, we prefetch the corresponding treelet, enabling deeper levels of the tree to be fetched in advance. This reduces the latency associated with pointer-chasing during tree traversal. Our approach uses a hardware prefetcher with a two-stack treelet based traversal algorithm, maximizing the benefits of treelet prefetching. Our simulation results show treelet prefetching on average improves performance of the baseline RT Unit in Vulkan-Sim by 32.1% while maintaining the same power consumption.

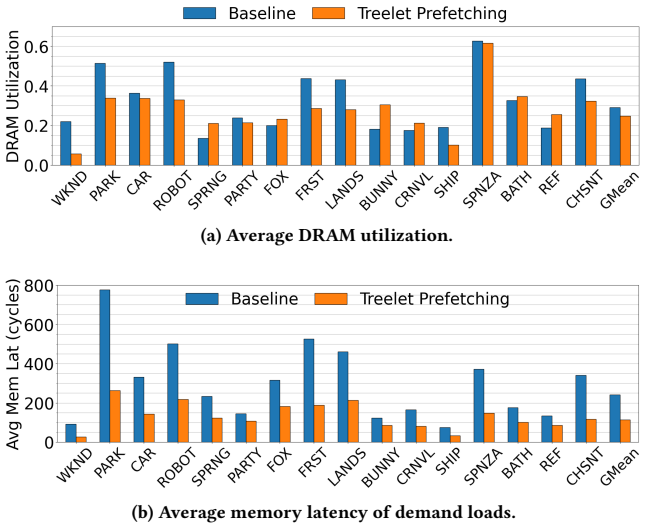
CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

ray tracing, graphics, prefetching, hardware accelerator, GPU

ACM Reference Format:
Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. 2023. Treelet Prefetching For Ray Tracing. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28–November 01, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3613424.3614288



(a) Average DRAM utilization.

(b) Average memory latency of demand loads.

Figure 1: Memory statistics for ray tracing comparing the baseline RT unit to our approach.

1 INTRODUCTION

Ray tracing is a rendering technique that generates realistic images by simulating the paths of light rays interacting with objects and materials in a scene. It was traditionally used for offline rendering such as in film production due to its high computational demands, which made real-time ray tracing applications on consumer hardware impractical. Modern GPUs, such as those made by NVIDIA, AMD, and Intel, come with specialized ray tracing accelerators. These accelerators allow for real-time ray tracing, which greatly improves the visual quality of graphics-intensive applications like video games. However, current hardware still struggles to fully render ray traced effects before frame rates drop below the desired real-time threshold of 60 frames per second [38].

The primary performance bottleneck in ray tracing is the cost of determining the closest intersection between a ray and a scene. While the scene is encoded as a tree data structure such as a Bounding Volume Hierarchy (BVH) tree to reduce the cost of finding intersections, traversing the BVH tree is still costly due to long memory latencies. This is because rays are usually incoherent (or lack locality), traveling from different locations and in various directions, and can traverse through different parts of the tree, causing divergent memory accesses and frequent cache misses. Figure 1 shows the average DRAM utilization and BVH memory access latency for ray tracing workloads on a baseline RT unit compared to our treelet prefetching approach. Ray tracing is a latency-bound application, meaning that threads are typically waiting on small data blocks to be fetched from memory, as seen from the low DRAM utilization and high memory latency in the baseline RT unit. Our approach reduces the memory latency of BVH accesses by 54% on average, resulting in a 32.1% IPC improvement in our results.

MICRO '23, October 28–November 01, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614288

GPUs rely on massive thread-level parallelism to hide memory latency. However, when there are significantly more memory requests per thread compared to the amount of compute, there will be insufficient warps for hiding memory latency. While increasing thread count can alleviate this, this comes at the cost of area overhead and increased register file usage. Prefetching techniques can be used to improve memory latency tolerance in GPGPU applications [22, 24, 42]. These techniques utilize tables to capture memory access locality and prefetch data into the cache. Unfortunately, they are not well suited for highly erratic and divergent memory accesses, as in the case of ray tracing. Prefetching techniques for graph applications have also been proposed [6, 17, 40, 45, 47], taking advantage of the predictable access offset patterns in graphs represented by adjacency matrices in a compressed sparse row (CSR) format. Although BVH trees are a type of graph, they contain more structure than arbitrary graphs. An n-node BVH tree only contains $n-1$ edges which if stored with an N-by-N adjacency matrix in CSR format introduces unnecessary complexity. BVH trees are also the industry standard for ray tracing as seen in NVIDIA Ada and Ampere whitepapers [2, 4]. Feng et al. [13] partition data structures to exploit parallelism, subdividing graphs and trees and distributing each partition to different threads to improve parallelism for CPUs. Recently, Liu et al. [27] worked on predicting ray intersections. However, their approach is limited to certain ray tracing effects such as ambient occlusion where any-hit intersections suffice and does not work well when closest-hit intersections are required for global illumination.

Aila et al. [5] proposed to use *treelets*, which are small subtrees of the overall BVH tree to speed up ray traversal. They explored using treelet queues to queue up rays that visit the same treelet and process them together to increase memory reuse. While an interesting idea, their simulated architecture is different from a programmable GPU and they lacked an actual hardware implementation. Adopting the queuing mechanism with current GPU threading models and modern ray tracing APIs is non-trivial. In this work, we build off the concept of treelets and propose prefetching for BVH trees at a treelet granularity. Tree traversal is an intensive pointer-chasing operation, requiring traversing to a node in the tree and finding the child pointers, before being able to find the child node addresses and issue loads. With treelet prefetching, as rays traverse the BVH tree and visit the root node of treelets, corresponding treelets can be prefetched to load deeper levels of the tree before they are needed. We combine treelet prefetching with a treelet based traversal algorithm in the ray tracing accelerator to further reduce ray traversal latency. From the limited available public information disclosed by GPU hardware manufacturers [2–4, 9, 11], it is unclear whether any commercial designs implement treelets and if so how.

We make the following contributions in this paper:

- We propose a treelet prefetching technique for ray tracing that can hide the memory latency of ray traversal.
- We propose a lightweight hardware implementation of a treelet based prefetcher by organizing BVH memory in a treelet based layout.
- We propose a treelet based traversal algorithm that is able to take advantage of treelet prefetching.

2 BACKGROUND

This section describes the GPU architecture with a dedicated ray tracing accelerator and GPU prefetching techniques.

2.1 Ray Tracing and BVH Traversal

2.1.1 Ray Tracing Overview. Ray tracing is a rendering technique that simulates the path of light in a scene by tracing rays from the camera through the scene [38]. Rays originate from the camera, pass through the image plane, and trace through the scene to find the closest intersection with a primitive. These rays, known as primary rays, are used to find the color of the pixel by sampling the color of the primitive at the intersection point. From the intersection point, secondary rays can be shot in the direction of the light sources to render shadows or towards nearby surfaces for global illumination. Additionally depending on the material of the intersected surface, if reflective or glossy, reflection rays can be traced to follow the path of the light bounce to render reflections. More rays can be cast per pixel and the results from each ray are accumulated and averaged to obtain the final pixel value [20]. As the number of rays traced per pixel is increased, more of the scene is sampled to produce better image quality. However, tracing a large number of rays is expensive and difficult to achieve under a constrained time budget. To improve render performance, the scene is built into an acceleration structure as a BVH tree to reduce the number of triangles or primitives a ray needs to test intersections with.

2.1.2 BVH Traversal. Ray tracing is a memory intensive task of traversing an acceleration structure in the form of a BVH tree to find the closest intersection between a ray and a primitive. A BVH tree is a hierarchical tree structure that organizes geometric primitives into multiple layers of bounding boxes. A node in the upper layers of the BVH tree contains a bounding box that encloses all bounding boxes and primitives of its child nodes. A BVH tree consists of two types of nodes: internal nodes and leaf nodes. Internal nodes contain axis-aligned bounding boxes (AABBs) and pointers to child nodes, and leaf nodes contain the actual primitives. BVH trees usually contain millions of triangles and are too large to fit entirely in cache.

The traversal process usually follows a depth-first (or breadth-first) search of the BVH tree while tracking unvisited intersected nodes in a traversal stack. We describe a depth-first traversal in the following. The traversal stack is first initialized with the BVH root node. Every loop iteration when the traversal stack is not empty, the front of the stack is popped and the popped node is checked for a ray intersection. If the intersected node is an internal node, its child nodes are pushed to the traversal stack. Otherwise, the intersected node is a leaf node, the primitive is tested for intersection with the ray, and the hit distance is recorded to identify the closest-hit primitive. This process repeats until there are no more nodes on the traversal stack. Ray tracing accelerators are designed to perform this traversal process as efficiently as possible.

2.2 GPU Architecture and RT Accelerators

GPUs are massively parallel processors containing tens of thousands of threads. Figure 2 illustrates the baseline GPU architecture with a dedicated ray tracing accelerator that we build on in this

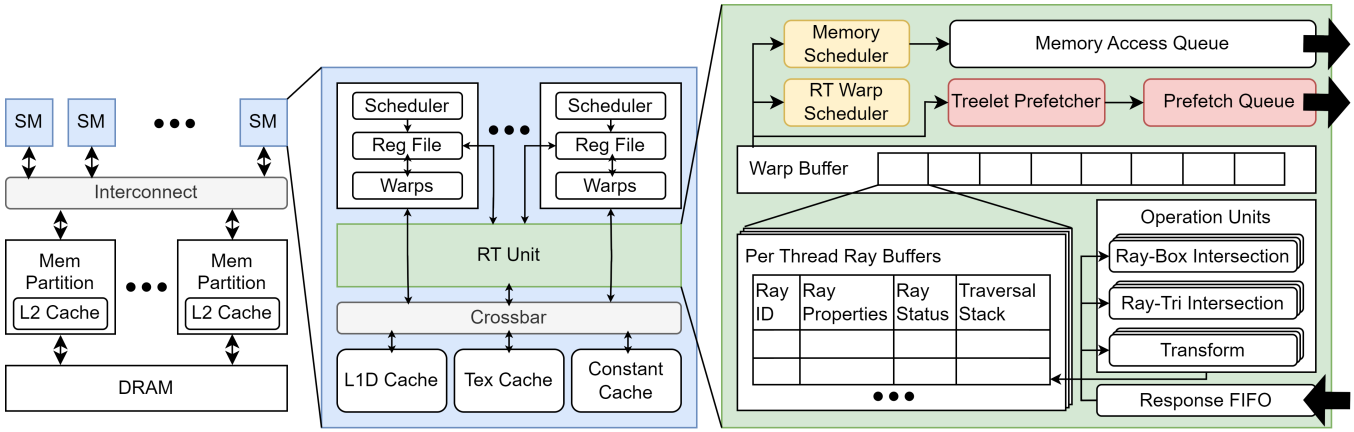


Figure 2: Baseline GPU architecture with RT Unit [41], with treelet components in red.

work. A GPU is composed of a collection of Streaming Multiprocessors (SMs) in NVIDIA's terminology, which serve as the GPU's computational units. Within each SM, there are multiple execution units responsible for executing shader programs. Threads in an SM are organized into warps which are groups of 32 threads that are executed in lockstep or a Single Instruction Multiple Thread (SIMT) fashion. Each SM also has multiple warp schedulers, each responsible for scheduling a group of warps in Greedy-then-Oldest (GTO) fashion. Additionally, every SM contains its own L1 cache, texture cache, and constant cache that is accessible to warps via a crossbar. All SMs are connected to a shared L2 cache which is divided into multiple memory partitions and communicates through an interconnect.

Modern GPUs now integrate a ray tracing accelerator within each SM which has dedicated hardware responsible for ray traversal and intersection testing. In Vulkan-Sim [41], the simulator we use for evaluation, the ray tracing accelerator is referred to as the RT unit. When a warp issues a trace ray instruction, it enters the RT unit during the pipeline's execute stage and is queued in the warp buffer, which holds ray metadata for all 32 threads of the warp. During each cycle, the RT unit selects a warp in the warp buffer for processing and issues memory requests to an L1 memory access queue. The returned data is read from the response FIFO and is used to perform ray-box and ray-primitive intersection tests with the operation units. Traversal results are written back to the warp buffer and the warp is cleared from the warp buffer when all the rays in the warp have been processed. Major GPU companies such as NVIDIA, AMD, Intel, and Imagination have published high level diagrams or press releases of their ray tracing hardware accelerators [2–4, 9, 11], however details of their microarchitecture are not publicly available. Besides Vulkan-Sim [41], we are not aware of any other publicly available GPU simulators that model the RT unit in detail.

We propose to add a treelet prefetcher to the RT unit to speed up ray traversal along with a prefetch queue to hold the issued prefetch addresses, both of which are highlighted in red in Figure 2. We elaborate more on the prefetcher in Section 3.

2.3 Hardware Prefetching Techniques

Prefetching techniques have been extensively researched for CPUs. There are also prior works on GPGPU application prefetchers with specific optimizations for GPU architecture.

Stream Prefetchers. Stream prefetchers record the access direction in certain memory regions [19, 37]. When an access is detected in the region, it prefetches subsequent memory locations following the current access. The prefetched data is usually stored in a stream buffer to avoid cache pollution.

Stride Prefetchers. Stride prefetchers record the local history of memory accesses by the same PC in a table to capture the stride or constant offset between memory addresses [12, 14]. When a stride is detected, it prefetches the next memory location according to the captured address delta. Stride prefetchers are well suited for arrays or structured data where memory is accessed with a constant offset.

GHB Prefetchers. Global history buffer prefetchers store the history of miss addresses in a global history buffer organized as a FIFO table [34, 35]. Each GHB table entry stores a miss address and a pointer to the next table entry, linking the miss addresses in temporal order. Upon encountering a miss, the GHB prefetcher finds the corresponding entry in the GHB table and uses the pointers to prefetch the subsequent miss addresses, allowing it to capture irregular access patterns.

GPGPU Specific Prefetchers. There is prior research on GPU prefetchers optimized for GPGPU applications. Lee et al. [24] proposed to use stride prefetching along with an inter-thread prefetching technique. Since GPGPU applications use a high number of threads, the execution length of each thread is usually short so the thread that issues a prefetch is unlikely to benefit from it. Inter-thread prefetching prefetches data for a later thread so that data will be ready when the thread is scheduled. To achieve this their prefetcher detects stride and stream access patterns on a per-warp basis. However, they still rely on stride and stream prefetching which are unable to capture irregular accesses found in ray tracing.

Koo et al. [22] proposed a CTA-aware prefetcher that works in conjunction with their scheduling scheme to address L1 cache misses that occur in a bursty manner for memory intensive applications. This leads to memory contention and impacts performance as

warps stall waiting for memory. They alleviate this by spreading out the bursty accesses over time through their proposed CTA-aware scheduler and prefetch data based on the leading warp of a CTA as threads within a CTA typically have stride accesses. However similar to the previous work, this technique does not apply to access patterns without a predictable base address and offset.

Irregular Workload Prefetchers. Ainsworth et al. [6] propose a prefetcher for graph applications to overcome the inability of traditional hardware prefetchers to capture irregular access patterns. These seemingly random memory accesses in graph workloads turn out to be well defined and predictable in advance due to graphs being stored in a standard format. Using data structure and application specific knowledge, they can know the traversal order of neighbor nodes ahead of time when a node is visited and prefetch node data. While BVH trees are a type of graph, they are not stored in a compressed sparse row format that is typically used in graph applications. Rather, nodes in a BVH tree are organized as a tree where nodes are connected through pointers. This makes subsequent node addresses difficult to predict with a base address and index offset. Other graph prefetchers are discussed in Section 7.

2.4 Ray Traversal Prefetching Challenges

As mentioned previously, ray traversal involves following a long sequence of pointer-chasing for each ray. Both stride and stream prefetchers are ineffective for irregular access patterns found in BVH traversal as the next node traversed by a ray is unlikely to follow a constant offset. While GHB prefetchers are effective for irregular access patterns, they are also not suitable for BVH traversal as each miss address sequence is often specific to a single ray. Guo et al. [17] evaluated a stride-based GHB prefetcher on a GPU in GPGPU-Sim [8]. They conclude that GHB prefetchers are ineffective for graph applications when using breadth-first search due to low prefetcher coverage and irregular strides in memory access patterns. Ray tracing workloads also exhibit irregular access patterns during ray traversal because each ray is used to sample different parts of the scene. As a consequence, rays are usually dispatched from various locations and cast in different directions. This characteristic is especially found in secondary and reflection rays which traverse drastically different parts of the BVH tree due to the different ray bounces. As a result, the memory access pattern of BVH tree traversal is highly irregular and unpredictable, making it difficult to predict the next memory access and prefetch the data in advance. To address this, we propose a treelet based prefetcher in Section 3 that is specifically designed for the divergent and irregular memory accesses of BVH traversal.

3 TREELET PREFETCHING

This section describes our proposed treelet prefetching technique for ray tracing. We first outline how treelets are formed from an existing BVH tree. Then we discuss how treelets are used to prefetch memory for ray traversal. Finally, we propose a treelet based traversal algorithm that improves ray tracing performance further when combined with treelet prefetching. The hardware implementation will be discussed in Section 4.

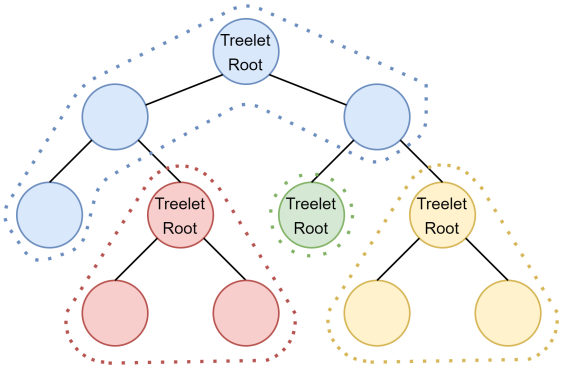


Figure 3: Example of a tree split into treelets with a maximum of 4 nodes similar to Figure 2 in [5]. Each circle represents a BVH node. Same colored nodes belong in the same treelet.

3.1 Treelet Based Formation

Inspired by Aila et al. [5] who proposed the concept of treelets, in this work we also divide up the BVH tree into connected subtrees called treelets to prefetch at a treelet granularity. Figure 3 shows an example of a two-wide BVH tree partitioned into treelets with a maximum size of 4 nodes each. In practice and in our evaluation, BVH trees are much larger and not limited to binary trees. We form treelets by grouping connected BVH nodes to maximize the size of each treelet. It is a greedy algorithm that starts from the BVH root node and greedily adds nodes to the current treelet until the maximum treelet size is reached. Three queues are used to track the formation progress. The *completedTreelets* queue contains the root address of completed treelets. The *pendingTreelets* queue contains the root address of treelets that have yet to be processed. Finally, the *stack* contains BVH nodes that still need to be traversed, similar to the traversal stack in the ray traversal algorithm.

Treelet formation initializes the *remainingBytes* to the maximum treelet size and adds the BVH root address to the *pendingTreelets* queue and traversal stack. The front of the *pendingTreelets* queue is the treelet we are currently forming. In every iteration, we traverse the tree and add the current node's children to the traversal stack, while calculating the remaining treelet space if the current node were to be added to the treelet. If there is still space, the node is added to the treelet and traversal continues. Once there is no more space, the current treelet root is complete and its root address is moved to the *completedTreelets* queue. If there are still nodes on the traversal stack, they are pushed into the *pendingTreelets* queue as new treelet root nodes that await formation. Treelet formation continues with the next treelet root node at the front of the *pendingTreelets* queue. The formation algorithm terminates when both the *pendingTreelets* queue and traversal stack are empty, and the *completedTreelets* queue contains the addresses of all treelet root nodes.

With this algorithm, treelets at upper levels will often be closer to the maximum desired size due to the greedy nature. This works to our advantage as the upper levels of the BVH tree are accessed

more frequently than the lower levels. In the next sections, we propose a treelet based ray traversal algorithm with an accompanying prefetcher.

3.2 Treelet Based Traversal

Ray traversal is typically done by traversing the BVH tree in a depth-first or breadth-first manner [15, 18, 31]. This causes memory accesses between rays to be spread out and hard to predict. We propose a treelet based traversal algorithm performed in the RT unit that transforms the sequence of memory accesses performed by each ray to be clustered within individual treelets. The high level idea is to have rays traverse nodes within a treelet first before moving on to nodes that belong to a different treelet. This potentially creates more node reuse opportunities between different rays and enables prefetching of BVH nodes.

Algorithm 1 describes the proposed treelet based traversal algorithm that will be executed by each ray. Different from a baseline depth-first traversal algorithm that only maintains one traversal stack, the treelet based traversal algorithm keeps track of two stacks: a *currentTreeletStack* and a *otherTreeletStack*. The *currentTreeletStack* is similar to the traversal stack in the baseline algorithm, recording which nodes still need to be visited in the current treelet. The additional *otherTreeletStack* keeps track of treelet root nodes that need to be traversed by the ray after it finishes intersecting with the nodes in the current treelet first. In contrast to our minimal changes to the traversal algorithm, the approach by Aila et al. [5] greatly alters how ray traversal is handled by using dynamically allocated treelet queues to hold rays that visit the same treelet and dedicated hardware for a ray launcher that dispatches rays in these treelet queues.

To begin, the *currentTreeletStack* is initialized with the root node of the BVH tree (Line 1). While there are still nodes in the *currentTreeletStack*, ray traversal will follow the baseline algorithm. The main difference is when adding child nodes that intersect with the ray, treelet based traversal checks if the child node belongs in the current treelet or not (Line 13). If the child node belongs in the current treelet, it is added to the *currentTreeletStack* (Line 14), otherwise, it is added to the *otherTreeletStack* (Line 16). When the *currentTreeletStack* is empty, the algorithm transfers the front of the *otherTreeletStack* over to the *currentTreeletStack* (Line 5) and resumes traversal as described above. Traversal ends when both the *currentTreeletStack* and the *otherTreeletStack* are empty.

3.3 Treelet Prefetching Overview

Treelet prefetching prefetches BVH nodes in treelet granularity to the GPU's cache. Ray tracing is a pointer-chasing application and memory accesses are divergent and hard to predict. With the treelet based traversal algorithm introduced previously, memory accesses are now clustered as individual treelets, making it possible to prefetch easily. As a ray visits a treelet root node, its subsequent memory accesses will also be to the nodes in the treelet since accesses to nodes from different treelets are deferred to the *otherTreeletStack*. Thus, we can prefetch the entire treelet to the GPU's cache and reduce the latency of accessing nodes in the current treelet. Ray traversal is a pointer-chasing operation where a ray travels down a node's child nodes repeatedly, creating a chain of dependent memory accesses where the latency is serialized. With treelet prefetching, while the pointer-chasing nature of ray traversal is still present, subsequent node accesses are confined within a treelet and can be fetched in advance without traversal. This reduces the node access latency during ray traversal as the nodes are already prefetched to the GPU's cache.

Algorithm 1 Treelet Based Traversal

```
Require: Ray, BVHRoot
1:  currentTreeletStack ← {BVHRoot}
2:  otherTreeletStack ← {}
3:  while currentTreeletStack ≠ ∅ || otherTreeletStack ≠ ∅ do
4:    if currentTreeletStack = ∅ then
5:      currentTreeletStack.push(otherTreeletStack.front())
6:      otherTreeletStack.pop()
7:    end if
8:    currentNode ← currentTreeletStack.front()
9:    currentTreeletStack.pop()
10:   if currentNode ≠ leaf node then
11:     for child in currentNode do
12:       if RayBoxTest(Ray, currentNode.AABB) then
13:         if child.treelet = currentNode.treelet then
14:           currentTreeletStack.push(child)
15:         else
16:           otherTreeletStack.push(child)
17:         end if
18:       end if
19:     end for
20:   else
21:     hitPrimitive ←
          RayPrimitiveTest(Ray, currentNode.primitive)
22:   end if
23: end while
```

4 PROPOSED ARCHITECTURE

This section discusses the architecture for the proposed treelet prefetcher and explores prefetch heuristics and scheduling policies.

4.1 Hardware For Treelet Prefetching

As mentioned in Section 2.2 when executing ray tracing shaders, warps issue the *traceRay* instruction to the RT unit, which adds rays in the warp to the RT unit's warp buffer. We propose to add a treelet prefetcher that prefetches treelets into the L1 cache of the GPU based on the rays in the warp buffer. Figure 2 shows the proposed architecture for treelet prefetching, which adds a treelet prefetcher and prefetch queue to the RT unit. The treelet prefetcher is connected to the warp buffer so it can identify treelets that will be traversed next. Ideally, we would prefetch a treelet for every ray in the RT unit given infinite memory bandwidth and cache capacity. However due to bandwidth limitations, the prefetcher needs to arbitrate between the different treelets in the warp buffer based on prefetch heuristics and add the corresponding treelet nodes to the prefetch queue. Often this is the most popular treelet that rays will visit next, but we explore different schemes in Ssection 4.2.

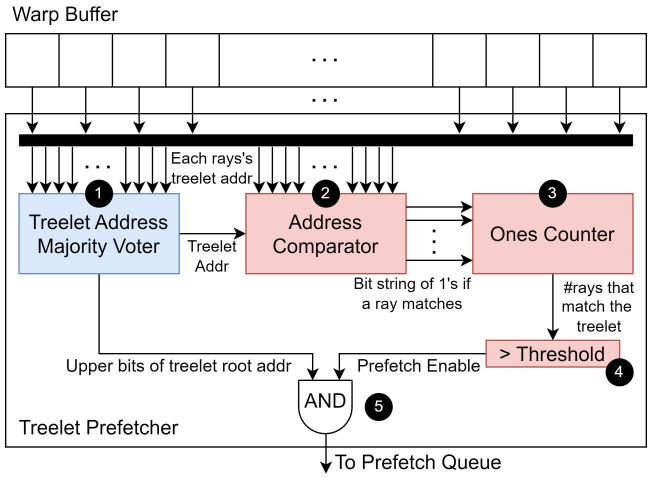


Figure 4: Hardware implementation of treelet prefetcher with treelet popularity threshold heuristic.

Figure 4 shows the high level block diagram of the proposed treelet prefetcher. There are two main components to the treelet prefetcher: the treelet address majority voter (❶) and the treelet popularity tracker (all red blocks). The treelet address majority voter (❶) identifies what treelet to prefetch next, usually the most popular treelet amongst rays in the warp buffer. It takes in the treelet address of each ray in the warp buffer and outputs the most popular treelet address. The treelet popularity tracker calculates the percentage of rays in the warp buffer that will traverse the most popular treelet. The treelet popularity tracker is used for various prefetch heuristics that we elaborate in the next section and it is made up of 3 sub-components. The address comparator (❷) compares the treelet address found by the treelet address majority voter to the treelet address of each ray in the warp buffer. It produces a bit string of size equal to the number of rays in the warp buffer where a bit is set if the corresponding ray matches the treelet address. Next, the ones counter counts the number of set bits in the bit string and converts it to a binary number which we refer to as treelet popularity (❸). The threshold comparator generates the prefetch enable signal if the treelet popularity is greater than a manually set threshold which ranges from 0 to the maximum number of rays in the warp buffer (❹). The prefetch enable is ANDed with the upper bits of the treelet root node address to generate the treelet prefetch address and sent to the prefetch queue to be processed (❺). We only require the upper bits of the treelet root node address because the treelets have a fixed maximum size and nodes within a treelet are organized to be packed together in memory. The treelet prefetcher also records the address of the last treelet it prefetches to avoid pushing duplicate treelet addresses to the prefetch queue and prefetching the same treelet multiple times in a row. A prefetch entry is processed from the prefetch queue every cycle when the RT unit's memory scheduler is not busy servicing demand loads.

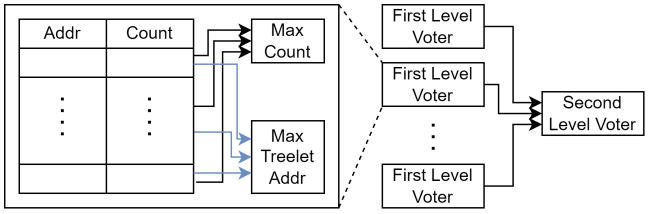


Figure 5: Treelet Address Majority Voter Design.

4.1.1 Prefetcher Implementation. The treelet address majority voter tracks the treelet occurrences of each thread in the warp buffer and is the core component of the prefetcher. However, completing this task in a single cycle is unfeasible in terms of both power and area costs since this requires a multi-ported 512-entry table to record treelet counts or a huge combinational structure. Instead, we propose a treelet address majority voter that tracks treelet occurrences over multiple cycles which is implemented as a two-level pseudo majority voter shown in Figure 5. The first-level voter finds the most popular treelet amongst threads in a warp, and the second-level voter finds the most popular treelet amongst the different warps. As there are 32 threads in a warp, the first-level majority voter requires a 32-entry table to record the treelet address and the count of each treelet. In each cycle, the treelet address of a thread is added to the table and the corresponding count is incremented. While tallying, we keep a rolling count of the occurrence of the most popular treelet and its address. Once every thread in the warp has been counted, the most popular treelet in the warp is found and the table is cleared to process the next warp. The second-level voter is similar in design but with only 16 entries to match the warp buffer size. This design requires 32 cycles to find the most popular treelet in the warp buffer if we duplicate the first level table for each warp. An alternative design is to reuse a single first-level voter for all warps and use a round-robin scheme to count the treelet occurrences of each warp to avoid having to make 16 copies of the first-level table. The address comparator is an optional component for other prefetch heuristics and can be implemented as a set of 32 comparators for each thread in the warp and reused across the different warps. We report prefetcher overheads and explore the impact of area optimizations in Section 6.5.

4.2 Prefetch Heuristics

The treelet prefetcher determines which treelet to prefetch while prefetch heuristics control whether a prefetch should be issued or not. We explore three heuristics for treelet prefetching: ALWAYS, POPULARITY, and PARTIAL. The **ALWAYS** heuristic always prefetches the most popular treelet by adding it to the prefetch queue as long as the treelet address is different from the previously prefetched treelet. The frequency of the prefetch decisions depends on the latency of the treelet prefetcher which we further explore in Section 6.5. While a simple approach, this heuristic may fetch an entire treelet even if only a few rays benefit from it, resulting in fetching excessive nodes for a small subset of rays instead of just traversing without prefetching. This usually happens in lower

levels of BVH traversal where rays are spread across multiple different treelets and the most popular treelet represents just a small majority of rays in the warp buffer.

The **POPULARITY** heuristic addresses the small majority issue by prefetching the most popular treelet only if its popularity ratio exceeds a manually set threshold between 0 and 1. The popularity ratio is the percentage of rays in the warp buffer that will traverse the most popular treelet, calculated by dividing the treelet popularity from the ones counter by the maximum number of rays. A threshold of 1 ensures a treelet is only prefetched if all rays in the warp buffer will traverse it, while a 0 is equivalent to the ALWAYS heuristic. This approach boosts the likelihood of reusing a prefetched treelet and alleviates memory pressure.

The **PARTIAL** heuristic attempts to address treelet overfetching by adaptively prefetching only a portion of the treelet based on popularity. Intuitively, when the popularity of the most popular treelet is high, there is a higher chance that more nodes in the treelet will be accessed. The idea is to prefetch the same fraction of nodes in the treelet as rays in this RT unit that access the treelet, starting from the front of the treelet. Since treelets are formed in a breadth-first manner, upper level nodes will come first in the treelet in the treelet and should be accessed more often than lower level nodes. For example, if all rays in the warp buffer are going to traverse the most popular treelet, then the entire treelet is prefetched. If only half of the rays traverse the most popular treelet, then only the first half of the treelet is prefetched. This ensures treelets will still be prefetched, and since nodes in the front of the treelet which are the upper level nodes of the treelet are prioritized, it can provide the benefit of reducing ray traversal latency while also preventing overfetching lower level nodes that will unlikely see much reuse between different rays.

4.3 Treelet Schedulers

The warp scheduler in the RT unit determines which warp in the warp buffer should be processed by the memory scheduler each cycle. The baseline scheduling policy schedules the oldest warp in the warp buffer that is not stalled; stalled meaning that all rays in a warp are either waiting for a memory response for a BVH node or waiting for ray intersection tests to finish. This scheduler focuses on reducing the warp latency of the oldest warp and attempts to free up the warp buffer as soon as possible so more warps can issue to the RT unit. With the introduction of treelet based traversal and prefetching, we propose two different scheduling schemes that optimize around treelets to improve performance. Since the RT unit spends resources to prefetch treelets, it is beneficial to issue warps that will traverse the prefetched treelet to maximize reuse.

The first proposed treelet scheduler (**O**ldest warp with **M**atching **R**ay, **OMR**) issues the oldest warp in the warp buffer that has a ray that matches the prefetched treelet. For rays in the warp that do not access the prefetched treelet, they still issue their memory requests to the RT unit's memory access queue as normal but receive no benefit from the prefetches. This scheduler aims to combine the benefits of the baseline scheduler while getting more reuse from the current prefetched treelet. The second proposed treelet scheduler (**P**rioritize **M**ost **R**ays, **PMR**) tries to maximize the reuse of the prefetched treelet by prioritizing the warp with the most rays that are going to traverse the prefetched treelet. While this policy maximizes reuse, it may result in longer latency as the different warps in the warp buffer all get scheduled more equally, since when the warp with the most matching rays gradually finishes traversing this treelet, another warp might take the ray majority and be prioritized.

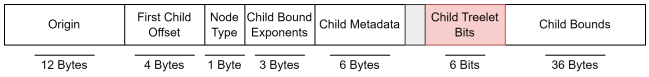


Figure 6: BVH node changes to support treelet based traversal, highlighted in red.

4.4 Node Layout For Treelet Prefetching

To support treelet prefetching, we need to identify what treelet a BVH node belongs to during traversal. This is also required by the prefetcher to determine what treelet node addresses to prefetch. One way to accomplish this is to modify the BVH node layout to store this information. Fortunately, this is a relatively simple change that can be done without modifying the BVH construction algorithm or increasing node sizes. Figure 6 shows the changes to the BVH node layout highlighted in red. The original BVH node layout is allocated to be 64 bytes and stores the bounding box, pointers, and other metadata for the child nodes of a 6-wide BVH tree. Since there are 2 unused bytes, we use 1 bit for each of the six children to indicate whether the corresponding child node belongs in the same treelet as its parent node, for a total of 6 bits. This fits in the unused space and the node size remains unchanged. While adding treelet child bits alone addresses treelet identification during traversal, it does not help the prefetcher identify what node addresses it should prefetch. One solution is that the BVH tree needs to be repacked into a treelet layout where nodes belonging to the same treelet are packed together in memory. This makes identifying what treelet a node belongs to trivial by just comparing the upper bits of the node address. An alternative is to store a mapping table from node ID to treelet ID. While this method does not require repacking the BVH tree and allows it to function with existing BVH tree builders, it requires additional memory to store the mapping table and mapping table loads. We evaluate these two approaches in Section 6.4.

5 METHODOLOGY

We extend Vulkan-Sim [41] to simulate our proposed treelet based prefetcher. The code is available at https://github.com/ubc-aamodt-group/treelet-prefetching-for-rt. The treelet based traversal algorithm is modeled in functional simulation to provide the RT unit in the timing model with the sequence of memory addresses. The treelet prefetcher and schedulers are modeled in the RT unit's timing model to process memory requests and schedule prefetches. We sweep prefetcher latencies to characterize timing impact in Section 6.5. Table 1 shows our evaluated simulation configuration. For power measurements, we used AccelWattch [21]. The prefetcher consumes extra power primarily with extra prefetch loads which is already captured by the power model. We synthesize our design with FreePDK45 for area results.

Table 1: Vulkan-Sim Configuration

# Streaming Multiprocessors (SM)	8
Max Warps / SM	32
Warp Size	32
Warp Scheduler	GTO
# Registers / SM	32768
Instruction Cache	128KB, 16-way assoc., 20 cycles
L1 Data Cache + Shared Memory	64KB, Fully assoc. LRU, 20 cycles
L2 Unified Cache	3MB, 16-way assoc. LRU, 160 cycles
Core, Interconnect, L2 Clock	1365 MHz
Memory Clock	3500 MHz
# RT Units / SM	1
RT Unit Warp Buffer Size	16

We simulated both the treelet based memory layout where treelet nodes are repacked together and an unmodified BVH tree with a node-to-treelet mapping table described in Section 4.4. With an unmodified BVH tree, a load to the mapping table is required before being able to identify the treelet a node belongs in. The design space to schedule mapping table loads is large, thus we model only two extreme cases. In the first case (Loose Wait), the mapping table load is inserted in the prefetch queue before the prefetches and simulated as an extra prefetch load. While unrealistic, it is the best case scenario for the prefetcher as the mapping table metadata could be loaded in advance if the prefetcher can identify the next treelet to prefetch early. The second (Strict Wait) is the worst case scenario where prefetches are only allowed to be added to the prefetch queue after mapping table loads return.

5.1 Evaluation Benchmarks

We evaluate our proposed approach on ray tracing scenes from LumiBench [28] and the BVH tree statistics for each scene are outlined in Table 2. The scene acceleration structures are BVH trees built by Intel Embree 3.12.2 [46] which use a highly compressed node format. Vulkan-Sim [41] describes the BVH node structures in more detail. These benchmarks are rendered at 1 sample per pixel (SPP) which is a common setting for real-time ray tracing, which is also the default setting in Unreal Engine 5 [1]. Higher SPP counts usually cost too much time to render in real-time and are only used for offline rendering. To reduce simulation time we simulate the benchmarks at a resolution of 32x32. However, we have also tested some scenes at a higher resolution of 96x96 and the speedups remain consistent. Principal Kernel Analysis by Baddouh et al. [7] finds that the IPC of most GPU kernels stabilizes around the final average, even in some irregular applications like graph processing. Using a smaller kernel sample or resolution can still be representative if metrics stabilize, as is the case for our benchmarks.

6 RESULTS

This section presents the results of our proposed treelet prefetcher. Section 6.1 breaks down the performance of treelet based traversal and treelet prefetching. Sections 6.2 and 6.3 evaluate the proposed prefetch heuristics and treelet schedulers. Section 6.4 and 6.5 discuss the results of BVH repacking and the hardware overhead of our proposed prefetcher. Lastly, Sections 6.6 and 6.7 evaluate the impact of treelet sizes and present prefetch effectiveness.

Figure 7 is the overall speedup and energy results of our proposed treelet prefetcher using the ALWAYS heuristic, PMR scheduler, and with a 512B maximum treelet size. Treelet traversal combined with treelet prefetching achieves an average speedup of 32.1% over the baseline RT Unit in Vulkan-Sim [41] while maintaining the same power consumption. Two scenes, WKND and PARTY, do not benefit from our approach. WKND is a simple scene with a small BVH tree that already fits in cache. PARTY loses performance due to treelet based traversal accessing more nodes than the baseline DFS traversal (see Section 6.1), however treelet prefetching is still beneficial and closes the performance gap. We also compare against Lee et al. [24] in Figure 8. We optimistically implement their prefetcher with infinite table structures, but results show their method is ineffective for ray tracing as it does not fetch many useful BVH nodes.

6.1 Treelet Based Traversal

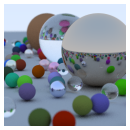
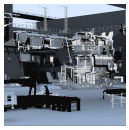
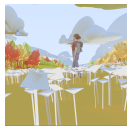
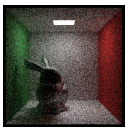
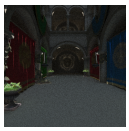
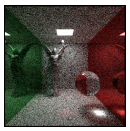
Figure 9 shows the overall performance broken down into treelet based traversal on the bottom and additional speedups gained from treelet prefetching on the top. Treelet based traversal alone results in a 3.7% slowdown over the baseline DFS traversal due to traversing extra nodes. However, treelet prefetching boosts the performance of treelet based traversal by 35.8%, bringing the overall speedup to 32.1%. From Figure 9 we can see that the treelet based traversal algorithm does not always yield better traversal results over the baseline DFS traversal order. Table 3 summarizes the average and maximum number of nodes traversed per ray for each workload. On average treelet based traversal reduces the number of nodes traversed per ray by 2.12% and 0.28% for the longest ray, demonstrating its ability to have a positive impact on the traversal performance while also reducing tail latency. However, there are some cases where the treelet based traversal algorithm performs worse than the baseline DFS traversal algorithm. This is because with DFS traversal rays reach the primitives in leaf nodes faster and can get a hit distance to identify closest-hit intersections. Once found, rays can omit intersection tests with nodes further from the current hit distance, known as early ray termination, and reduce node accesses during ray traversal. Treelet traversal on the other hand, prioritizes traversing nodes in a treelet first, which can lead to extra node accesses that reduce performance.

6.2 Treelet Prefetch Heuristics

Figure 10 compares the prefetch heuristics to the baseline RT unit. The ALWAYS heuristic is the most aggressive, always prefetching the most popular treelet. The POPULARITY heuristic builds off the ALWAYS heuristic but uses a popularity threshold for throttling. A larger threshold requires a higher percentage of rays in the warp buffer before the treelet is prefetched, reducing overfetching. The PARTIAL heuristic is another variation of the ALWAYS heuristic but attempts to prevent overfetching by only prefetching a portion of a treelet. Overall, the ALWAYS heuristic performs the best with a 31.9% average speedup, followed by POPULARITY with 27%, and PARTIAL with 16%.

Figure 11 shows the normalized L2 BW relative to no prefetching. The ALWAYS and POPULARITY heuristics successfully limit the

Table 2: Summary of evaluation scenes from LumiBench [28]. The maximum treelet size is set as 512B.

Scenes	WKND	PARK	CAR	ROBOT	SPRNG	PARTY	FOX	FRST
								
Tree Size (MB)	0.2	501.9	1,233.6	1,721.3	164.3	143.8	597.8	348.6
Tree Depth	7	14	16	18	14	14	15	14
Total Treelets	519	3,946,335	10,186,555	13,532,923	1,286,479	1,137,508	4,638,757	2,764,433
Scenes	**LANDS**	**BUNNY**	**CRNVL**	**SHIP**	**SPNZA**	**BATH**	**REF**	**CHSNT**
								
Tree Size (MB)	279.2	12.2	37.3	0.5	22.0	104.2	37.1	25.5
Tree Depth	12	11	16	12	16	16	13	12
Total Treelets	2,293,559	71,424	299,373	4,323	176,804	821,975	305,404	204,634

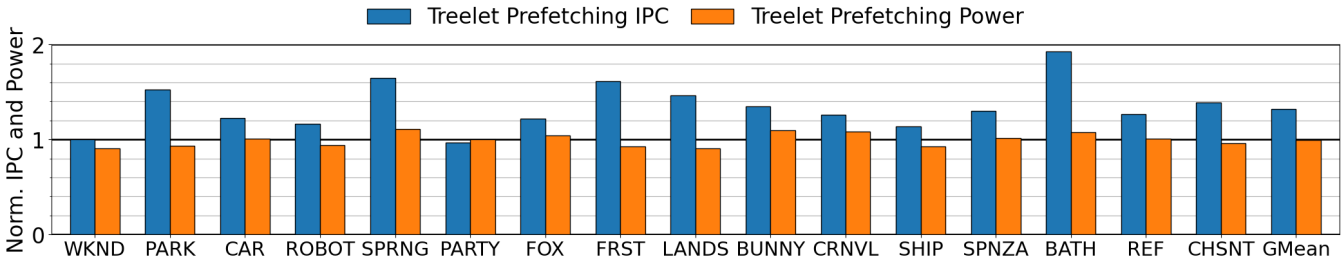


Figure 7: Overall speedup and power consumption of treelet prefetching with the ALWAYS heuristic, PMR scheduler and a 512B max. treelet size.

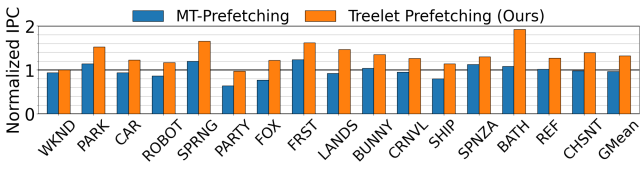


Figure 8: Performance comparison to prior work.

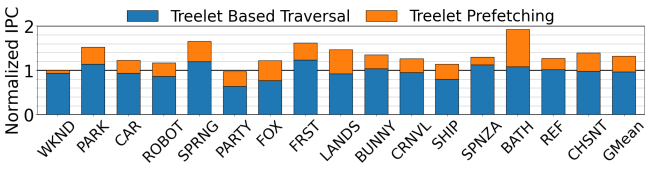


Figure 9: Speedup breakdown of treelet prefetching using the ALWAYS Heuristic with the baseline scheduler.

amount of prefetching, as seen from the L2 BW decrease. While there was concern about overfetching when proposing these heuristics, the performance results show that the benefits of bringing more nodes into the cache outweigh the downsides of the increased load on the memory system from additional prefetches. To support this claim, Figure 12 plots the breakdown of L1 cache statistics for the different prefetch heuristics. The bars starting from the bottom to the top are the ratio of node cache hits brought in by prefetches, cache hits brought in by demand loads, pending hits, and cache misses respectively. We find that the ALWAYS heuristic has a much higher ratio of node hits that were brought in by prefetches compared to POPULARITY with various thresholds. For the POPULARITY heuristic, the low performance is likely because the prefetches are not timely enough to be useful. While upper level nodes are more likely to be reused, they are also closer to the root of the treelet, there may not be enough time for the prefetch to arrive before the ray reaches the nodes.

6.3 RT Unit Treelet Schedulers

Figure 13 compares different treelet schedulers to the baseline RT unit without prefetching. All three schedulers perform similarly, with the PMR scheduler edging out slightly with a 32.1% speedup, followed by the baseline scheduler with 31.9% and the OMR scheduler in last at 31.8%. However, the required modifications outweigh the small performance benefits. There is a tradeoff between prioritizing the oldest warp to free up warp buffer space versus maximizing treelet reuse. The OMR and PMR schedulers both result in slightly

Table 3: Comparison of baseline DFS ray traversal to treelet based traversal. Lower values are better.

	Avg Nodes Per Ray			Max Nodes Per Ray		
Scenes	DFS	**Treelet Trav.**	Diff.	DFS	**Treelet Trav.**	Diff.
WKND	13.3	13.3	0.00%	61	61	0.00%
PARK	171.6	141.5	-17.50%	1389	1402	0.94%
CAR	47.2	47.9	1.47%	249	214	-14.06%
ROBOT	99.2	104.2	4.97%	1051	884	-15.89%
SPRNG	31.4	32.0	2.02%	216	139	-35.65%
PARTY	45.8	46.3	1.07%	422	466	10.43%
FOX	85.5	81.0	-5.30%	433	383	-11.55%
FRST	34.7	34.0	-2.12%	355	232	-34.65%
LANDS	30.4	29.7	-2.46%	223	344	54.26%
BUNNY	11.6	11.1	-4.15%	117	100	-14.53%
CRNVL	42.3	43.3	2.46%	489	495	1.23%
SHIP	42.3	43.1	1.94%	207	188	-9.18%
SPNZA	38.3	42.0	9.72%	112	218	94.64%
BATH	19.0	18.9	-0.55%	405	465	14.81%
REF	11.3	11.1	-2.19%	169	209	23.67%
CHSNT	52.8	42.8	-18.96%	212	202	-4.72%
GMean			-2.12%			-0.28%

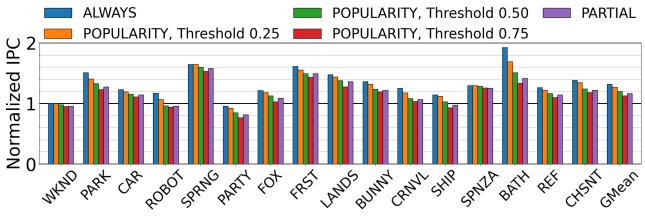


Figure 10: Performance comparison of different prefetch heuristics to the baseline RT unit.

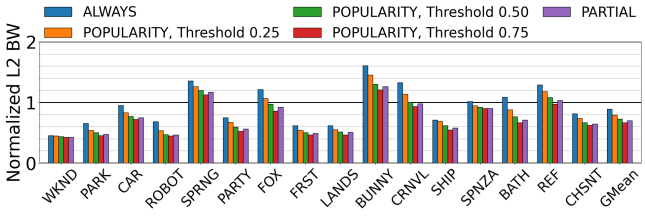


Figure 11: L2 BW comparison of different prefetch heuristics to the baseline RT unit.

longer thread latency in the RT unit due to the schedulers switching warps to match the current prefetched treelet more often, but the overall warp latency ends up shorter due to warps being scheduled in a more balanced manner. We notice that some scenes are slower with the proposed schedulers. However this is more so due to the ALWAYS heuristic as it is more aggressive, overfetches too many nodes and switches the prefetched treelet too often. This causes cache thrashing and hurts performance.

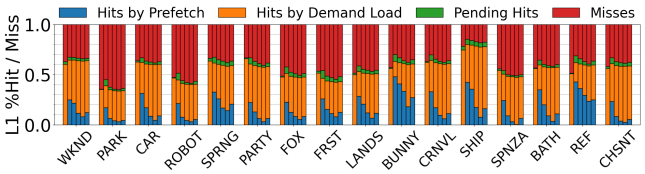


Figure 12: L1 cache stats of prefetch heuristics. For each workload, bars from left to right are: Baseline, ALWAYS, POPULARITY: 0.25, POPULARITY: 0.5, POPULARITY: 0.75, PARTIAL.

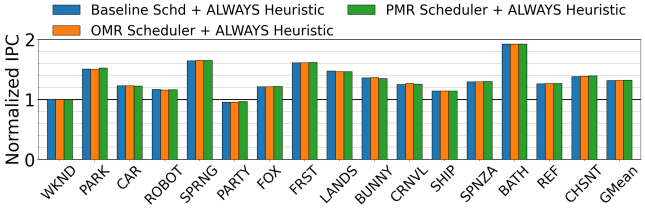


Figure 13: Performance of different treelet schedulers.

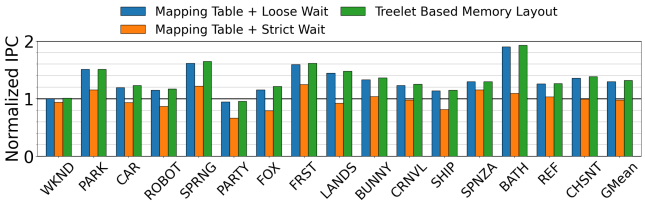


Figure 14: Performance of different treelet BVH options.

6.4 Treelet BVH Repacking

In Section 4.4 we described different BVH options to enable treelet prefetching, including repacking the BVH tree in a treelet based memory layout or using a node-to-treelet mapping table. Figure 14 shows the performance of the different options detailed in Section 5 compared to the baseline RT unit. With 512B treelets, the treelet based memory layout performs best with a 31.9% speedup over the baseline. Loose and Strict Wait represent a rough upper and lower bound performance when using an unmodified BVH tree that requires mapping table loads, falling behind the repacked BVH with only a 29.7% speedup and 2.5% slowdown respectively. Strict Wait results in a slowdown because it not only issues extra metadata loads for the prefetcher, but the prefetches also arrive too late to be useful. In both cases, the mapping table requires 4B of storage per BVH node address. Since a full node is 64B on average, the table is roughly 1/16th the size of the BVH tree. This significant overhead is another reason why BVH repacking is the better option.

6.4.1 Load Balancing. Naively repacking the BVH tree into a treelet based memory layout can result in unbalanced DRAM accesses across different DRAM chips. Figure 15 shows the performance of two identical treelet memory layouts using 512B treelets, with the only difference being that one adds a constant 256B stride between different treelet roots, separating the treelet roots 768B

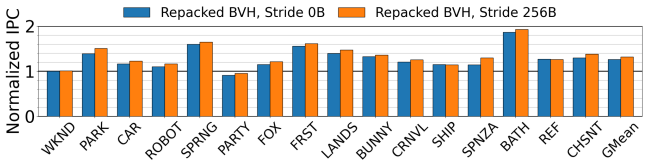


Figure 15: DRAM Load balancing effects of a repacked BVH using 512B treelets.

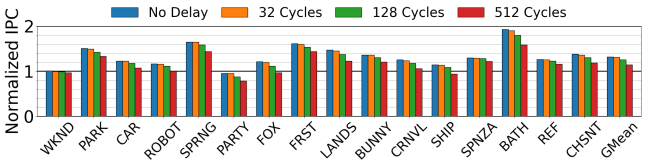


Figure 16: Performance impact of prefetcher latency.

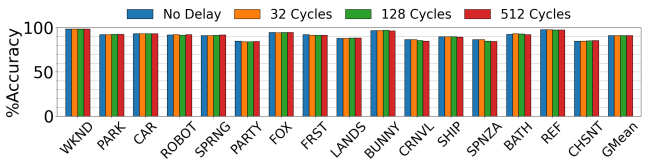


Figure 17: Decision accuracy of pseudo majority voter.

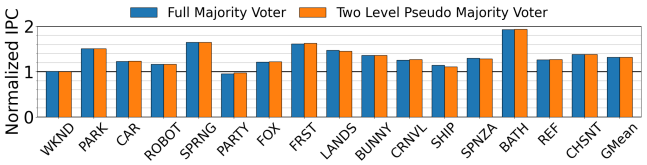


Figure 18: Performance of two-level-pseudo majority voter.

6.5 Prefetcher Design and Area Overhead

In Section 4.1, we suggested a more practical pseudo-two-level design for the treelet address majority voter. For the first-level voter, a 32-entry table tallies treelets of a warp and finds its most popular treelet. In each cycle, a thread's treelet is recorded in the table. Two sequential elements track the current maximum treelet and its current count. We synthesized the sequential part of the majority voter with FreePDK45 (45nm) which requires 461 μm^2 of area [44]. This table can be reused for each warp in the warp buffer or can be duplicated at the cost of area to shorten prefetcher latency. The per warp results are fed to a 16-entry table in the second-level voter, similar to the first-level voter, and it records the most popular treelet across all warps. One latency optimization is once any treelet's count exceeds half the table size, it is immediately declared the most popular, thus the 32-entry table only requires 4 bits to count occurrences and 3 bits for the 16-entry table. Since the treelet root address is aligned to 512B, only 23 bits are needed for the address field. This totals to 108B per first-level table and 52B for the second-level table. One idea is that the majority voter resembles the function of a GPU memory access coalescer [36], which finds accesses to the same cache lines among threads of a warp and we could repurpose it for the majority voter or reference its design. As the base prefetch heuristic is already effective, we do not suggest building the address comparators for the other proposed heuristics (red blocks in Figure 4) as they will only increase the area with minimal upsides. We acknowledge that design complexity is a limitation of our prefetcher and is a topic for further exploration.

The two-level design requires latency for the prefetcher to identify the most popular warp. Figure 16 sweeps prefetcher latency from 0 to 512 cycles and shows the performance impact. A 512-cycle latency is when the address majority voter consists of only 1 first-level table and counts 1 thread per cycle. On average, the two-level majority voter performs 1% worse (30.9% speedup versus baseline) when the prefetcher has a 32-cycle delay compared to having no delay. When increasing the latency to 512 cycles, performance drops to only 17% over the baseline, indicating that using only 1 table may be insufficient. A 128-cycle latency simulates having 4 first-level tables and only drops performance by 6.6% (25.3% speedup versus baseline) which seems like a feasible implementation. We also show the accuracy of the pseudo-two-level majority voter compared to a full majority voter in Figure 17 across different prefetcher delays. We measure this in simulation by comparing how often the pseudo-majority voter and full majority voter agree on the most popular treelet. On average the pseudo voter agrees with the full voter 91.2% of the time. We believe the accuracy loss mainly occurs at lower levels of the tree where rays are spread across multiple treelets and there is no obvious majority. However, Figure 18 shows that the accuracy loss of the pseudo-majority voter does not impact performance at all.

6.6 Treelet Sizes On Performance

Figure 19 shows the impact of treelet sizes on prefetch performance. We sweep treelet sizes of 256, 512, 1024, and 2048, and compare them to an unmodified RT unit. Using 512B treelets yields the best performance with a 31.9% speedup over the baseline followed by 29.4% with 1024B treelets, 30.4% with 2048B treelets and 24.8% with 256B treelets. While smaller treelets can reduce overfetching as it is more likely to prefetch nodes that are never used when prefetching larger treelets, it also reduces the prefetching effectiveness of loading deeper-level BVH nodes in advance. Larger treelets on the other hand can cause more memory traffic leading to more stalls and also reduced L1 hit rates. However, the performance decrease with larger treelets may be attributed to their formation in a breadth-first style, resulting in more nodes at the same depth and a minimal increase in tree depth. A potential solution can use heuristics or metadata to identify if a node should be prefetched.

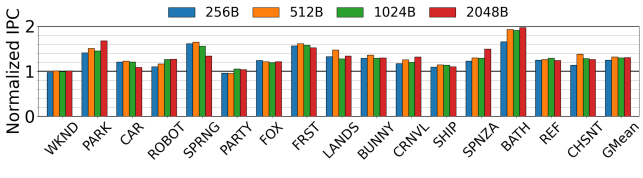


Figure 19: Performance with different treelet sizes.

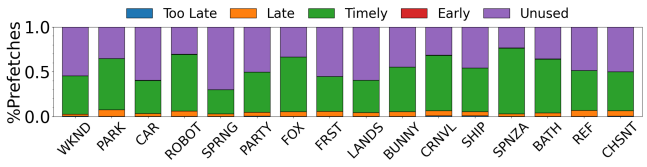


Figure 20: Prefetch effectiveness for 512B treelets, baseline scheduler, ALWAYS heuristic.

6.7 Prefetch Effectiveness

Figure 20 shows the prefetch effectiveness of treelet prefetching. A prefetch is **Too Late** if it hits in L1 but the data was fetched by a previous demand load. A **Late** prefetch is when it returns a pending hit. **Timely** prefetches are when a demand load hits in cache and the data is brought in by a prefetch. An **Early** prefetch is when the fetched data is evicted before being read by a demand load. **Unused** prefetches are never accessed by demand loads. Timely prefetches make up 47.8% of prefetches. However, 43.5% of prefetches are unused which is definitely an area for improvement.

7 RELATED WORK

7.1 GPGPU Prefetchers

Many-Thread Aware prefetching [24] uses threads to prefetch data for other threads rather than for itself. They also use throttling to prevent harmful prefetches from flooding the memory system. While it does not address irregular accesses, throttling may work for treelet prefetching. APOGEE [42] uses adjacent threads to identify access patterns and improve prefetch timeliness. This can work for ray traversal in upper BVH levels but is less effective later as rays diverge. Liu et al. [29] propose a self-tuning adaptive prefetcher to dynamically adjust prefetch modes, which could be applied to prefetch heuristics.

7.2 Graph Application Prefetching

GraphPulse [40] is an event-driven accelerator for graph processing with a graph node prefetcher. A coalescing queue combines events during graph processing and accurately prefetches data based on it. Prodigy [45] exploits the compressed sparse format of graphs to perform prefetching. A compiler pass first generates a data indirection graph (DIG) containing the layout and access patterns of key data structures. Data is prefetched when Prodigy observes loads to a data structure with a trigger edge on the DIG. Wang et al. [47] proposed shared memory prefetching to tackle irregular access patterns in graph applications. Threads are assigned to process vertex data from a partition in shared memory that is prefetched in advance.

7.3 Data Structure Partitioning

Feng et al. [13] take advantage of data structure partitioning to exploit parallelism. Within parallel regions during traversal identified by the programmer with compiler pragmas, they subdivide graphs and trees and distribute each partition to be processed by different cores to improve parallelism for CPUs. Locality improves by having the same cores process the same partitions repeatedly. While potentially useful for ray tracing, it may leave multiple processors idle as it is difficult to predict how often a partition is intersected by rays.

7.4 Treelet Based Ray Tracing Techniques

Navratil et al. [33] tackled incoherent rays by collecting rays into queue points to process together, reducing ray and scene data swapping. Aila et al. [5] improved the previous idea and first proposed the concept of treelets. They collect rays at treelet queues for processing to maintain high cache hit rates and reduce memory traffic. However it is unclear how one would implement treelets on a GPU, as supporting the large treelet queues is crucial to their technique. STRaTa [23] applied treelets to a multiple instruction multiple data (MIMD) based ray accelerator. They introduce ray stream buffers by configuring a part of L2 cache to have large amounts of rays on-chip and keep treelet queues populated. Dual Streaming [43], built upon STRaTa [23] by minimizing data transfers by reorganizing ray tracing into two separate scene and ray data streams and modifying the traversal algorithm to fit this paradigm. The ray stream is a collection of all in-flight rays and the scene stream contains the corresponding treelet data. However, as rays are duplicated across different treelet queues, performing early ray termination is not trivial. While an interesting solution it is difficult to compare our proposed solution to their work as their implementation is under the context of their custom MIMD accelerator. An equivalent GPU implementation encounters additional challenges such as how threads are spawned to process each ray queue. In their architecture, each Thread Processor acts independently to fetch rays from a ray staging buffer which might require non-trivial shader modifications to realize on a GPU and are not discussed in their paper.

7.5 Ray Traversal Acceleration Techniques

Ray Sorting. Ray sorting improves ray coherency by grouping rays that traverse similar parts of the AS. Pharr et al. [39] reordered ray computation to improve ray coherency and cache utilization. Garanzha and Loop [16] sorted rays based on ray origin and direction before processing in packets. Moon et al. [32] sorted rays with their final hit points. Meister et al. [30] improved sorting heuristics to minimize ray divergence. Ray sorting can be applied orthogonally to our work. However modern ray tracing APIs such as Vulkan and DXR generate rays dynamically in the ray generation shader, thus rays are not readily available to sort before the ray tracing kernel.

Acceleration Structure Optimizations. Ylitie et al. [48] explored wide BVH trees to increase SIMD utilization. Lin et al. [26] restructured BVH nodes with node splitting, reducing memory footprint. Benthin et al. [10] and Liktor et al. [25] perform BVH compression for memory bandwidth reduction. BVH optimizations benefit our work as more nodes fit into the same memory footprint, making prefetching more effective.

Ray Prediction. Liu et al. [27] predict ray intersections with a hash function. On correct predictions ray traversal is eliminated. On an incorrect prediction, rays traverse the BVH tree normally. However their method only works well for anyhit rays and is not obvious how it extends to other ray types such as closest-hit rays that are used for global illumination.

8 CONCLUSION

This work presents a treelet prefetching scheme to improve ray traversal performance. Conventional prefetchers like stride and stream prefetching are inadequate for ray tracing due to irregular access patterns during BVH traversal. Ray accesses exhibit little overlap and can be highly divergent, sampling independent scene areas and traversing different parts of the tree. Our solution instead prefetches at treelet granularity.

We use the concept of treelets which are connected subpartitions of a BVH tree. By traversing all nodes a ray intersects with in a treelet, BVH nodes gain more reuse between rays for effective prefetching. To further reduce traversal latency, we propose a treelet prefetcher, taking advantage of clustered memory accesses within individual treelets. When a ray visits a treelet we prefetch the entire treelet, reducing the latency of accessing lower level nodes. Treelet prefetching also removes the need for rays to visit a node before fetching its child nodes, minimizing pointer-chasing dependencies. Our simulations show treelet based traversal reduces performance slightly by 3.7% over a DFS baseline. However, when combined with treelet prefetching, the overall speedup reaches 32.1% while maintaining the same power consumption. Many directions for future work include optimizing treelet formation with statistical metrics, exploring effective treelet schedulers, and applying inter-thread aware mechanisms from prior works. We believe this work extends beyond ray tracing and has potential in other domains such as graph applications since treelets can capture the locality between nodes and allow for prefetching while skipping load dependencies.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback. We would also like to thank Jonathan Lew and Lufei Liu for their feedback on earlier drafts of this paper. This research is funded in part by grants from Huawei Technologies. Tor M. Aamodt recently served as a consultant for Huawei Technologies Canada Co. Ltd and Intel Corp.

REFERENCES

[1] 2020. *Unreal Engine 4 Ray Tracing Features Settings.* Retrieved April 22, 2023 from https://docs.unrealengine.com/4.26/en-US/RenderingAndGraphics/RayTracing/RayTracingSettings/#:~:text=Ray%20Tracing-,Samples%20Per%20Pixel,sample%20per%20pixel%20by%20default.

[2] 2021. *NVIDIA AMPERE GA102 GPU ARCHITECTURE.* Retrieved April 27, 2023 from https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

[3] 2022. *Real-Time Ray Tracing on Intel Arc Graphics.* Retrieved April 27, 2023 from https://game.intel.com/story/intel-arc-graphics-ray-tracing/

[4] 2023. *NVIDIA ADA GPU ARCHITECTURE.* Retrieved April 27, 2023 from https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

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

[6] Sam Ainsworth and Timothy M. Jones. 2016. Graph Prefetching Using Data Structure Knowledge. In *Proc. ACM Conf. on Supercomputing (ICS).*

[7] Cesar Avalos Baddouh, Mahmoud Khairy, Roland N Green, Mathias Payer, and Timothy G Rogers. 2021. Principal kernel analysis: A tractable methodology to simulate scaled GPU workloads. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO).*

[8] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS).* 163–174.

[9] Kristof Beets. 2021. Rays Your Game: Introduction to the PowerVR Photon Architecture. https://imaginationtech.com/products/gpu/graphics-architecture/powervr-photon/

[10] Carsten Benthin, Ingo Wald, Sven Woop, and Attila T. Áfra. 2018. Compressed-Leaf Bounding Volume Hierarchies. In *Proc. ACM Conf. on High Performance Graphics (HPG).*

[11] John Burgess. 2020. RTX on—the NVIDIA Turing GPU. *IEEE Micro* 40, 2 (2020), 36–44.

[12] Tien-Fu Chen and Jean-Loup Baer. 1995. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers (TOC)* (1995).

[13] Min Feng, Changhui Lin, and Rajiv Gupta. 2012. PLDS: Partitioning Linked Data Structures for Parallelism. *ACM Transactions on Architecture and Code Optimization (TACO)* (2012).

[14] J.W.C. Fu, J.H. Patel, and B.L. Janssens. 1992. Stride Directed Prefetching In Scalar Processors. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO).*

[15] Kirill Garanzha and Charles Loop. 2010. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum* (2010).

[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] Hui Guo, Libo Huang, Yashuai Yashuai Lü, Jianqiao Ma, Cheng Qian, Sheng Ma, and Zhiying Wang. 2018. Accelerating BFS via Data Structure-Aware Prefetching on GPU. *IEEE Access* (2018).

[18] Michael Guthe. 2014. Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics 2014 - Short Papers.*

[19] N.P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA).*

[20] James T. Kajiya. 1986. The Rendering Equation. In *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH).* 143–150.

[21] Vijay Kandiah, Scott Peverelle, Mahmoud Khairy, Junrui Pan, Amogh Manjunath, Timothy G Rogers, Tor M Aamodt, and Nikos Hardavellas. 2021. AccelWattch: A Power Modeling Framework for Modern GPUs. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO).* 738–753.

[22] Gunjae Koo, Hyeran Jeon, Zhenhong Liu, Nam Sung Kim, and Murali Annavaram. 2018. CTA-Aware Prefetching and Scheduling for GPU. In *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS).*

[23] 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. 47–59.

[24] Jaekyu Lee, Nagesh B. Lakshminarayana, Hyesoon Kim, and Richard Vuduc. 2010. Many-Thread Aware Prefetching Mechanisms for GPGPU Applications. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO).*

[25] Gabor Liktor and Karthik Vaidyanathan. 2016. Bandwidth-Efficient BVH Layout for Incremental Hardware Traversal. In *Proc. ACM Conf. on High Performance Graphics (HPG).*

[26] 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.

[27] 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 *Proc. IEEE/ACM Symp. on Microarch. (MICRO).* 709–723.

[28] Lufei Liu, Mohammadreza Saed, Yuan Hsi Chou, Davit Grigoryan, Tyler Nowicki, and Tor M. Aamodt. 2023. LumiBench: A Benchmark Suite for Hardware Ray Tracing. In *Proc. IEEE Symp. on Workload Characterization (IISWC).*

[29] Peng Liu, Jiyang Yu, and Michael C. Huang. 2016. Thread-Aware Adaptive Prefetcher on Multicore Systems: Improving the Performance for Multithreaded Workloads. In *ACM Transactions on Architecture and Code Optimization (TACO).*

[30] 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.

[31] Daniel Meister, Shinji Ogaki, Carsten Benthin, Michael J. Doyle, Michael Guthe, and Jirí Bittner. 2021. A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum* (2021).

[32] 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).

[33] Paul Arthur Navratil, Donald S. Fussell, Calvin Lin, and William R. Mark. 2007. Dynamic Ray Scheduling to Improve Ray Coherence and Bandwidth Utilization.

In *IEEE Symposium on Interactive Ray Tracing*. 95–104.

[34] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. 2004. AC/DC: an adaptive data cache prefetcher. In *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*.

[35] K.J. Nesbit and J.E. Smith. 2004. Data Cache Prefetching Using a Global History Buffer. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*.

[36] Lars Nyland, John R. Nickolls, Gentaro Hirota, and Tanmoy Mandal. 2008. Systems and methods for coalescing memory accesses of parallel threads. Patent No. US20090240895A1, Filed Mar. 24th., 2008, Issued Dec. 27th., 2011.

[37] S. Palacharla and R.E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*.

[38] Matt Pharr and Greg Humphreys. 2018. *Physically Based Rendering, Third Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc.

[39] 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.

[40] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*.

[41] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. 2022. Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*.

[42] Ankit Sethia, Ganesh Dasika, Mehrzad Samadi, and Scott Mahlke. 2013. APOGEE: Adaptive prefetching on GPUs for energy efficiency. In *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*.

[43] 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)*.

[44] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajaiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *IEEE International Conference on Microelectronic Systems Education*.

[45] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, John Magnus Morton, Agreen Ahmadi, Todd Austin, Michael O'Boyle, Scott Mahlke, Trevor Mudge, and Ronald Dreslinski. 2021. Prodigy: Improving the Memory Latency of Data-Indirect Irregular Workloads Using Hardware-Software Co-Design. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*.

[46] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (TOG)* (2014).

[47] Pengyu Wang, Lu Zhang, Chao Li, and Minyi Guo. 2019. Excavating the Potential of GPU for Accelerating Graph Traversal. In *Proc. IEEE Int'l Parallel and Distributed Processing Symp. (IPDPS)*.

[48] 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)*.