

# Treelet Accelerated Ray Tracing on GPUs

Yuan Hsi Chou

yuanhsi@ubc.ca

University of British Columbia

Vancouver, Canada

Tor M. Aamodt

aamodt@ubc.ca

University of British Columbia

Vancouver, Canada

## Abstract

Despite advances in hardware acceleration, ray tracing use in real-time rendering is limited and often lowers frame rates, leading users such as video game players to disable the feature entirely. Prior work has shown that dividing the BVH tree into smaller subtrees (treelets) and traversing all rays that visit a treelet before switching treelets can significantly reduce memory traffic on a specialized accelerator, but there are many challenges to applying treelets to GPUs. We find that a naive treelet implementation is ineffective and propose optimizations to improve performance. Virtualized Treelet Queues consist of two main components. Ray virtualization increases the number of concurrent rays in flight to create more cache reuse opportunities by terminating raygen shaders that have already issued their trace ray instruction, reclaiming CUDA cores and allowing more raygen shaders to be executed. To take advantage of the increased concurrent rays, we propose a dynamic treelet queue architecture that dynamically switches between traversal modes to increase efficiency. We also find that performing warp repacking boosts SIMT efficiency of warps in the RT unit which is crucial to achieving good traversal performance with treelet queues. Our simulations show virtualized treelet queues achieve on average 95% speedup compared to a baseline GPU with ray tracing acceleration across all scenes in LumiBench rendered with path tracing at one sample per pixel with three max bounces per ray.

## ACM Reference Format:

Yuan Hsi Chou and Tor M. Aamodt. 2025. Treelet Accelerated Ray Tracing on GPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*, March 30–April 3, 2025, Rotterdam, Netherlands. ACM, New York, NY, USA, 14 pages. <https://doi.org/10.1145/3676641.3716279>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org). *ASPLOS '25, Rotterdam, Netherlands*.

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1079-7/25/03

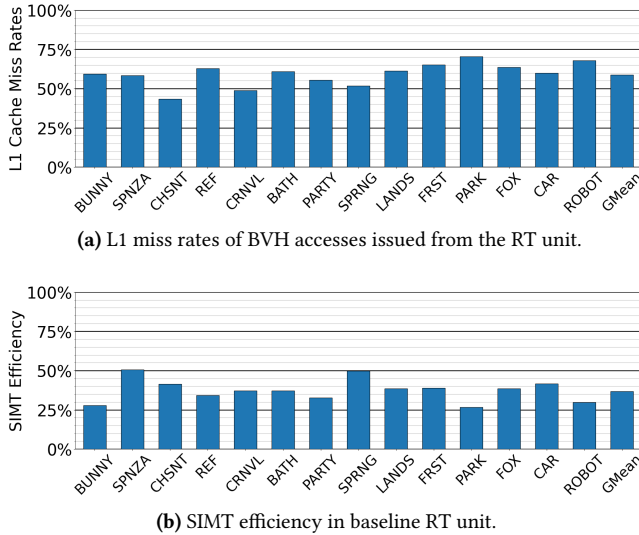
<https://doi.org/10.1145/3676641.3716279>

## 1 Introduction

Ray tracing can create photorealistic images, and is traditionally used in offline rendering such as for movie production. However, its high computation cost limits it from being widely used in real-time applications. While modern GPUs from companies such as NVIDIA, AMD, and Intel feature dedicated ray tracing hardware to improve rendering speed, they can still only achieve limited ray-traced effects within real-time frame rates of 60 frames per second [27].

In ray tracing applications, rays are traced through a scene to identify intersections with geometry. Ray tracing is a memory-intensive workload because of the vast amount of geometry that must be searched through. While scene geometry is already encoded into an acceleration structure as a bounding volume hierarchy (BVH) tree to reduce search complexity, traversing the BVH tree is still costly due to long memory latencies and high cache misses. These challenges are exacerbated by the fact that rays are usually incoherent (or lack locality), traversing through different parts of the BVH tree, and causing memory divergence [18]. Additionally, BVH trees are large and do not fit in on-chip caches, leading to frequent off-chip memory accesses and high memory traffic. Figure 1a shows L1 miss rates of only BVH accesses issued from the RT unit across all scenes in LumiBench for path tracing [22] measured in Vulkan-Sim [28], a cycle-level simulator. The average L1 miss rate is 58% and reaches as high as 70%, implying caches are ineffective at capturing BVH node locality and promoting data reuse. Additionally, we observe that the baseline RT unit also has low SIMT efficiency, as shown in Figure 1b, indicating that there are many inactive threads in the RT unit during traversal which could be used to improve performance.

To address this, researchers have proposed various techniques to improve ray tracing performance. Aila et al. [5] proposed to subdivide the BVH tree into smaller subtrees, or treelets, that fit in the processor's cache to reduce memory traffic. They group up and process rays that will traverse the same treelet first before switching to a different treelet and its corresponding rays and reduced memory traffic by 50-75%. While no performance results were shown, Aila et al.'s study motivates the possibility of a 2× or higher performance improvement. However, Aila et al.'s evaluation only considered memory traffic and did not appear to factor in the memory latency of accessing data at different levels of the memory hierarchy, which is crucial for latency-bound workloads like ray tracing. We attempted to implement treelets



**Figure 1.** Performance bottlenecks of the baseline RT unit. Scenes are sorted by ascending BVH size.

on a modern GPU in Vulkan-Sim, but found that a naive treelet implementation did not result in a speedup due to insufficient rays in treelet queues to promote data reuse and low SIMT efficiency. Kopta et al. [19] adopted the treelet idea to STRaTA, a custom MIMD architecture, and observed up to 43% lower DRAM energy. Shkurko et al. [30] improved upon Kopta et al. by reorganizing the traversal algorithm, transforming memory accesses during ray traversal into two predictable data streams: one for the BVH tree and one for the ray data. This improved traversal performance and reduced DRAM energy compared to STRaTA, however both STRaTA and Shkurko et al. increased memory bandwidth consumption. MIMD architectures also do not experience SIMT divergence like GPUs do, and we find that maintaining high SIMT efficiency is key to achieving good traversal performance with treelets on GPUs. Chou et al. [8] proposed a treelet-based prefetcher on ray tracing capable GPUs and observed a 30% speedup in ray tracing performance. However, Chou et al. report 43.5% of prefetches are unused which wastes memory bandwidth. We build on the treelet traversal order by Chou et al. to implement our virtualized treelet queues. Intel has also described a mechanism at a high level similar to what we call ray virtualization in their GPUs [1] to reorder hit shaders after traversal, but we adopt it to increase concurrent rays in flight for better treelet performance.

In this work, we introduce virtualized treelet queues, an efficient architecture that provides the benefits of treelets on modern ray tracing capable GPUs and is compatible with modern ray tracing APIs. Virtualized treelet queues comprise three parts: ray virtualization, dynamic treelet queues, and warp repacking. Ray virtualization greatly increases the number of concurrent rays in flight which would normally

be limited by the amount of warp slots in the RT unit since each ray is executed by a thread. We achieve this by terminating raygen shaders after a thread issues its trace ray instruction to the RT unit, allowing the GPU to reclaim the CUDA cores and launch new threads / raygen shaders from already queued up Cooperative Thread Arrays (CTAs). To take advantage of the increase in concurrent rays, we implement dynamic treelet queues in the GPU’s RT unit to group up rays that access the same treelet together, achieving better cache locality and reduced miss rates. During later phases of ray traversal when rays diverge more, treelet queues end up underpopulated and it becomes inefficient to process rays in treelets. We propose to group up these underpopulated treelet queues and traverse them regularly instead. However, this leads to a sharp decrease in SIMT efficiency due to varying BVH node access counts from different rays. We apply warp repacking to regroup active rays together into a new warp, boosting SIMT efficiency and performance. With all optimizations, virtualized treelet queues achieve up to 2.55× better path tracing performance under usage scenarios comparable to video games on ray tracing capable GPUs.

We make the following contributions in this paper:

- We propose a ray virtualization technique to increase the number of concurrent rays in flight (Section 3.1).
- We design a dynamic treelet queue architecture on modern ray tracing capable GPUs, improving ray tracing performance by dynamically switching between traversal modes (Section 3.2).
- We find maintaining high SIMT efficiency is crucial to traversal performance with treelets and propose to group up underpopulated treelet queues and warp repacking to achieve this goal (Section 4.4, 4.5).

## 2 Background and Motivation

This section provides an overview on ray tracing and the Vulkan ray tracing API, the baseline GPU and RT unit architecture, and prior work on treelets. We also discuss the challenges of implementing treelets on GPUs.

### 2.1 Ray Tracing and Modern Ray Tracing APIs

We first describe ray tracing at a high level and then discuss how modern ray tracing APIs handle ray tracing on GPUs.

**2.1.1 Ray Tracing Overview.** Ray tracing, specifically path tracing, renders images by tracing light rays through a scene to simulate the physical behavior of light. Primary rays are traced initially from the camera’s viewpoint through each pixel on the screen and are tested for intersections with scene geometry. When an intersection is found, the ray’s color is computed based on the material properties of the intersected object and the light sources in the scene. Further secondary rays can be generated from the intersection point to render reflection, refraction, and shadows. This is repeated

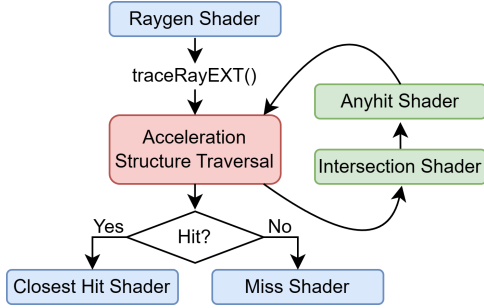


Figure 2. Vulkan Ray Tracing Pipeline.

until a maximum depth is reached or the ray’s contribution to the final image is negligible.

**2.1.2 Modern Ray Tracing APIs.** Modern ray tracing APIs such as Vulkan and DXR define a ray tracing pipeline to handle the different GPU shader stages during ray traversal [2]. Figure 2 shows the Vulkan ray tracing pipeline, but DXR follows a similar structure. When the host program invokes ray tracing kernels on the GPU, the ray generation (raygen) shader first executes to generate rays. Next, the `traceRayEXT()` function is called in the raygen shader to trace rays through the scene’s acceleration structure to find ray intersections. The acceleration structure (AS) is a bounding volume hierarchy (BVH) tree consisting of hierarchical axis-aligned bounding boxes (AABBs) encapsulating scene geometry to reduce search complexity. Rays traverse through the BVH tree’s internal nodes, usually in depth-first order and intersect with AABBs until reaching leaf nodes. Ray-triangle intersection tests are performed at the leaf nodes where rays intersect with scene geometry but intersection shaders can be used for custom intersection tests. Anyhit shaders are optional shaders that execute when a ray intersects with a geometry object. After traversal is complete, depending on whether the ray intersected with geometry or not, either the closest hit shader or the miss shader is executed and the thread returns to the raygen shader where the `traceRayEXT()` function was called.

## 2.2 GPU and RT Unit Architecture

GPUs are massively parallel processors optimized for throughput. Figure 3 shows the organization of a GPU consisting of multiple streaming multiprocessors (SMs) connected to the memory system through an interconnect. GPU kernels are executed by threads that are grouped into multiple cooperative thread arrays (CTAs). The host processor issues kernel invocations to the kernel management and dispatch unit which sends the CTAs to the CTA scheduler. The CTA scheduler assigns CTAs to available SMs where each CTA is processed as warps which are groups of 32 threads that execute in lockstep or single instruction multiple thread (SIMT) fashion. Each SM contains multiple execution units or CUDA

cores responsible for executing shader instructions of the assigned CTA. Each SM also has its own L1 cache and register file which are connected via a crossbar. All SMs share the L2 cache which resides in multiple memory partitions connected through an interconnect.

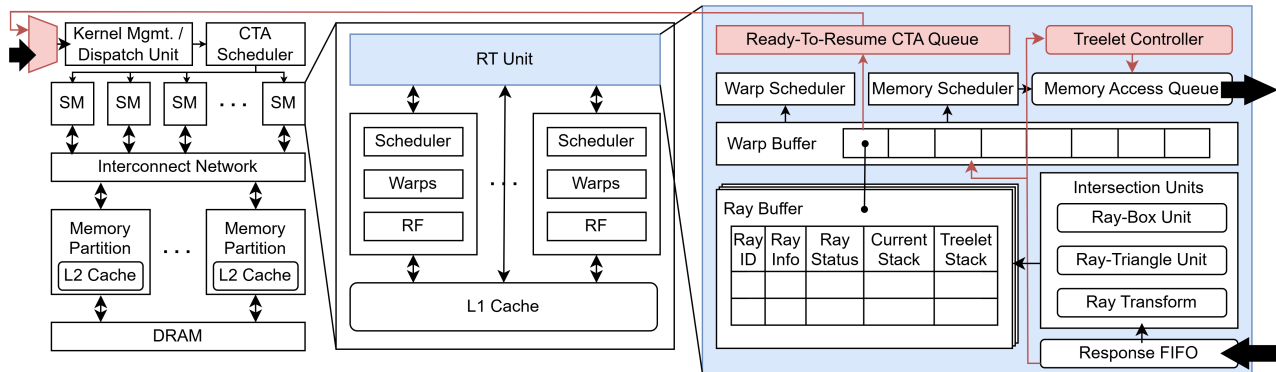
For modern GPUs, each SM also contains a ray tracing accelerator to offload the `traceRayEXT()` function from the CUDA cores and speed up BVH traversal and ray intersection tests. In this work, we build upon the RT unit architecture in Vulkan-Sim [28] which is a cycle-level simulator for the Vulkan ray tracing pipeline that has been microbenchmarked against real GPU hardware with a correlation of 95.7%.

Figure 3 illustrates the baseline RT unit architecture in Vulkan-Sim. After the raygen shader issues the `traceRayEXT()` function to the RT unit, the ray information of threads in the warp is written to each ray buffer entry in the warp buffer, including the ray ID, ray origin and direction, and its status. Each ray buffer also records the current traversal stack and treelet stack which was proposed by Chou et al. [8] to support the treelet traversal order that we will use to enable treelet queues. Each cycle the RT unit processes a warp from the warp buffer and the memory scheduler pushes a BVH address to the memory access queue to fetch from memory. When data is returned to the RT unit’s response FIFO, rays perform intersection tests with the fixed function operation units and traversal results are written back to the warp buffer. A warp is completed when all rays in the warp finish traversal and resume execution in the raygen shader.

## 2.3 Prior Work On Treelets

While ray tracing seems embarrassingly parallel as every ray can be processed independently, implementing it on parallel architectures is challenging, primarily due to the large BVH sizes and memory divergence. Secondary rays are incoherent and access very different parts of the BVH tree, and coupled with the fact that the entire tree is too large to be fully stored in on-chip caches causes frequent off-chip memory accesses and high cache miss rates.

To address these challenges, prior work proposed various memory optimization techniques to improve ray tracing performance. Aila et al. [5] proposed to subdivide the BVH tree into smaller subtrees, or treelets (such as in Figure 4), that fit in the processor’s cache to reduce memory traffic. To benefit the most from loading a treelet, they prioritize executing rays that will traverse the same treelet first before fetching a new treelet. They achieve this using dynamically allocated queues per treelet in memory to group up rays that will traverse the same treelet which resulted in significant memory traffic reduction. This treelet approach is similar in motivation to tile-based rendering where an entire scene is subdivided into tiles and rendered separately to reduce working set sizes and improve cache locality. While their simulated architecture resembled the resources of a GPU, one major limitation is they did not consider nor simulate memory latency. With

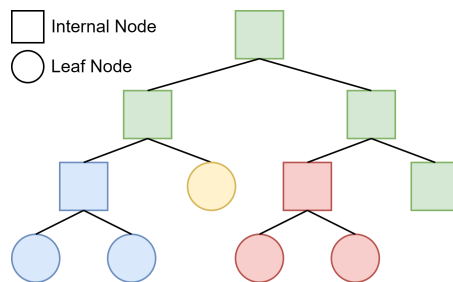


**Figure 3.** Baseline RT Unit Architecture of Vulkan-Sim [28] with modifications to support treelet traversal order [8]. Additional hardware to enable the proposed dynamic treelet queues are highlighted in red.

ray tracing being latency-bound [13], data placement in different parts of the memory hierarchy greatly impacts overall performance. For example, Aila et al. assumed ray data and traversal stacks are stored in DRAM, whereas in modern ray tracing GPUs, this data is stored directly on-chip in the RT unit’s warp buffer for quick access. As treelet queues increase auxiliary memory traffic (treelet queues, ray data, traversal stacks) significantly as noted by Aila et al., it may offset the BVH memory traffic reduction when implemented on a modern GPU with ray tracing acceleration.

Shkurko et al. [30] implemented treelet queues by reorganizing memory accesses into treelet and ray data streams. They coordinate loading both data streams from DRAM so that the treelet data is loaded alongside the rays that will traverse that treelet. This improved traversal performance and reduced DRAM energy on their custom MIMD-based accelerator, however Shkurko et al. suffer from a much higher memory bandwidth requirement due to needing to constantly fetch ray data from DRAM. However, the MIMD architecture used by Shkurko et al. is fundamentally different from a programmable GPU as MIMD threads do not need to consider SIMT divergence and can execute independently. In our work, we find that managing SIMT efficiency of warps is crucial for treelet queues to be effective on a GPU. Additionally, ray data on modern GPUs is stored in the RT unit instead of in DRAM so streaming ray data is costly.

Chou et al. [8] is the closest work to ours that explores how to adopt the treelet concept to a GPU with ray tracing acceleration. They proposed a prefetching scheme that first identifies the most popular treelet amongst all rays in each RT unit and prefetches that treelet entirely into the cache, resulting in a 30% speedup in ray tracing performance. Chou et al. modified BVH traversal to follow a treelet traversal order using two stacks: a current stack for traversal within a treelet, and a treelet stack to track pending treelets that a ray needs to visit. A ray traverses all nodes within its current stack first before moving on to the next treelet in the treelet

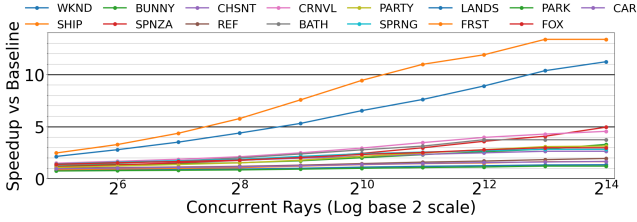


**Figure 4.** Example of a BVH tree divided into treelets with 4 max nodes. Nodes of the same color are in the same treelet.

stack. By comparing the treelets at the front of each ray’s treelet stack amongst rays in the RT unit, the RT unit can determine which treelet to prefetch next. However, Chou et al. do not fully implement the treelet queue concept as proposed by Aila et al. [5] as Chou et al. still allow different threads within the RT unit to traverse different treelets. This results in low SIMT efficiency within a warp as rays in the same warp can be traversing different treelets and potentially thrashing the cache especially if rays in the RT unit are spread across multiple different treelets evenly, causing the prefetcher to constantly switch between prefetching different treelets. Chou et al. also mention that 43.5% prefetches are never used as it is impossible to know which nodes within a treelet will be visited by a ray, costing memory bandwidth.

## 2.4 Motivation

To understand whether treelets are viable for GPUs, for this section only, we construct a simple standalone analytical model to estimate the potential performance gain of adopting treelets on GPUs for just ray traversal at different degrees of concurrent rays in flight before diving into the actual design details. For the rest of the paper, we evaluate our proposal on Vulkan-Sim, a detailed cycle-level simulator and not with this analytical model. We hypothesize that with more concurrent



**Figure 5.** Potential speedup of treelets in LumiBench with increasing concurrent rays from analytical modeling. Workloads are listed in Table 2.

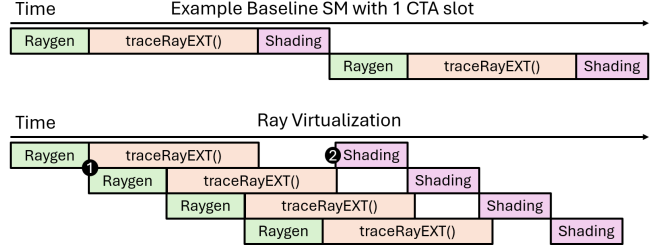
rays in flight, more rays can reuse the fetched treelet data, reducing memory accesses and lowering traversal latency.

To get these estimates, we first record every BVH memory access made by each ray during ray traversal as input to the analytical model. We do not model any caching of BVH nodes for both the baseline and treelet queues to simulate every access being a miss. With this assumption, the cycles needed to traverse every ray in the baseline RT unit roughly equals the total amount of nodes traversed by every ray multiplied by the memory latency. For treelet queues, we introduce the concept of concurrent rays, where all concurrent rays are processed in the same batch and rays in the same batch can reuse the fetched treelet data at no latency cost once fetched. The cycles required for treelet queues will be the product of the number of unique treelets in each batch of concurrent rays, the number of nodes in a treelet, memory latency, and the total number of ray batches. Supporting more concurrent rays in a batch reduces the total number of unique treelets the GPU has to fetch and reduces cycle count.

Figure 5 plots the estimated performance gains of treelets as we increase the number of concurrent rays across LumiBench scenes. As the number of concurrent rays increases, the potential gain from treelets due to memory savings also increases and can reach 3 to 4x speedup for most scenes. WKND and SHIP stand out for having the smallest BVH sizes and having higher chances of rays traversing similar treelets. While these are rough estimates, they motivate the need to support a large number of concurrent rays in flight on a GPU for a performant treelet queues implementation.

### 3 Virtualized Treelet Queues

This section presents the high-level design of virtualized treelet queues on GPUs. The goal is to enable efficient ray traversal on GPUs by increasing the number of concurrent rays in flight and using treelet queues to extract more data reuse opportunities. To achieve this, our design consists of two main parts: ray virtualization and treelet queues, which we will expand upon in the following sections.



**Figure 6.** Example of ray virtualization in a GPU with one CTA slot per SM.

#### 3.1 Ray Virtualization

Ray virtualization aims to increase the number of concurrent rays processed by the RT unit. In the baseline system, threads executing the raygen shader issue their ray to the RT unit and are stalled until the RT unit completes ray traversal. The raygen shader cannot be terminated after issuing rays to the RT unit since ray traversal results are used for shading afterwards. The anyhit, intersection, closest hit and miss shaders are also called from the raygen shader after traversal. Without terminating the raygen shader, the GPU cannot issue more new rays due to CTA scheduling hardware constraints. Due to the above, the RT unit can only trace a limited amount of concurrent rays bounded by the number of warp slots in the RT unit’s warp buffer and the number of concurrent threads processed by the CUDA cores. To support more concurrent rays, the GPU needs to handle the execution of more concurrent CTAs and increase the RT unit’s warp buffer size, both of which require high area overheads. However, since threads are stalled after issuing their `traceRayEXT()` function to the RT unit, we can exploit this to reclaim stalled CTAs and launch new raygen shaders to increase the number of concurrent rays in flight.

Figure 6 shows a simplified example of ray virtualization in a GPU with one CTA slot compared to a baseline system. For simplicity, only one SM is processing the CTAs and there is only one warp per CTA. Initially, both GPUs launch their first CTA to an SM which executes the raygen shader. After the raygen shader issues the `traceRayEXT()` function to the RT unit, threads in the baseline GPU stall until the RT unit completes all ray traversals in the warp (Figure 6 top). Once done, the warp continues raygen shader execution to perform shading calculations using the traversal results. With ray virtualization (Figure 6 bottom), the GPU can terminate the raygen shader after all threads in the CTA issue `traceRayEXT()` to the RT unit, and reclaim the CTA slot. This is similar to what Intel has done on their ARC GPUs from their high-level description [1], but instead of using it to reorder hit shaders after traversal, we adopt it to enable more concurrent rays in flight for treelets. While the RT unit processes the rays of the just terminated CTA, the GPU can launch new raygen shaders by issuing pending CTAs to the

freed up CTA slot (❶), increasing the number of concurrent rays in the RT unit. The extra ray data from the newly launched raygen shaders can be stored in either dedicated partitions of the cache or in the register file for quick access by the RT unit. This process repeats until no more CTAs remain to be processed. Meanwhile, as rays complete traversal in the RT unit (❷), warps can resume execution if any CTA slots of the SM are free. We prioritize resuming CTAs that have completed traversal in the RT unit to avoid queuing up too many rays and running out of resources. However, ray virtualization alone does not provide much benefits as the RT unit is still limited by the warp buffer size.

### 3.2 Dynamic Treelet Queues

To benefit from increased concurrent rays from ray virtualization, the second part of our design is the treelet queues. In ray tracing, there are two ways to organize traversal which we refer to as ray stationary and treelet stationary. Traditional ray tracing architectures are **Ray Stationary**, where the same rays are kept on-chip and the RT unit fetches BVH nodes from memory, prioritizing ray data reuse at the cost of BVH locality. In contrast, treelet approaches such as Aila et al. [5] are **Treelet Stationary**, where a treelet is kept on-chip and rays are fetched from memory to reuse the same treelet, trading off reduced BVH loads for increased ray data loads. The best choice of approach changes throughout the different phases of ray tracing execution so we propose to dynamically switch between both modes, maximizing data reuse and avoiding costly ray data fetches when possible.

Next, we describe the operation of dynamic treelet queues at a high level as depicted in Figure 7. The operation of dynamic treelet queues can be roughly split into three phases in chronological order: an initial traversal phase in ray stationary mode, a treelet stationary mode, and a final phase in ray stationary mode. In the **Initial Traversal Phase** as warps of rays enter the RT unit, they traverse the BVH tree in ray stationary mode until rays within a warp start to diverge past a threshold (❶). We start with ray stationary mode since rays are coherent initially and would exhibit high cache hit rates and low memory divergence with either traversal mode, and also avoids the overhead of fetching ray data.

Once rays start diverging, they are spread across more treelets and the RT unit switches to **Treelet Stationary Mode** (❷). Memory-diverged warps are terminated and write their rays to the corresponding treelet queues based on what treelets they need to traverse next. Once a treelet queue accumulates enough rays, the RT unit starts processing it by fetching ray data for rays in the queue to the warp buffer and treelet data to the L1 cache. Since rays in the queue will only access nodes in that treelet, cache accesses should always hit and reduce off-chip memory accesses. Rays traverse the treelet until they reach the boundary of the treelet and are pushed to the next treelet queue based on which treelet they

intersect next. A queue is emptied before switching to the next treelet queue to maximize reuse.

This process repeats until the largest treelet queue falls below a certain threshold, usually during the later half of ray traversal when rays are too diverged or there are not enough rays to fill up the queues. At this point, it is no longer beneficial to remain in treelet stationary mode as there are not enough rays to make use of fetching an entire treelet (❸). We group up the stray rays in the underpopulated treelet queues to form warps and these warps are processed in **Ray Stationary Mode** until all rays in it finish traversal.

## 4 Proposed Architecture

This section describes the architectural changes to support ray virtualization and treelet queues in hardware (Section 4.1 and 4.2). To reduce the latency of loading data from memory for treelet queues, we discuss how to preload treelet and ray data in Section 4.3. However, we find that a naive treelet implementation is ineffective in the later half of traversal as rays start diverging more, and we propose grouping up underpopulated treelet queues (Section 4.4) and warp repacking (Section 4.5) to address this issue.

### 4.1 Ray Virtualization Implementation

As discussed in Section 3.1, ray virtualization increases the number of concurrent rays seen by the RT unit through reclaiming CTA slots of an SM once threads issue their `traceRayEXT()` function. We will now describe the implementation of ray virtualization in more detail.

When the host program invokes a ray tracing kernel, the raygen shader kernel, which consists of multiple CTAs (a collection of threads), is launched on the GPU. CTAs are scheduled by the GPU’s CTA scheduler and bound to different SMs for execution and each SM has a limit on how many CTAs can execute concurrently. Warps in the raygen shader first generate rays and then issue them to the RT unit with the `traceRayEXT()` function. Once all threads in the CTA submit their rays, the CTA can be terminated allowing new CTAs to be launched. To resume the terminated CTA later when its rays have finished traversal, we allocate memory and store the CTA’s state which includes the registers of every thread and SIMT stack of all warps in the CTA. The kernel management unit also needs to store a table of terminated CTA IDs with their allocated CTA state address and size. Once a CTA is terminated, a new CTA can be launched on the SM until all raygen shader CTAs are issued or the desired number of concurrent rays is reached.

While new CTAs are executing, the RT unit traverses the submitted rays of the terminated CTAs. When all rays from a CTA complete traversal, the CTA can now resume. However, the baseline GPU can only launch CTAs from the CTA scheduler and not from an SM’s RT unit. To support this, in Figure 3 we introduce an additional path from each SM’s

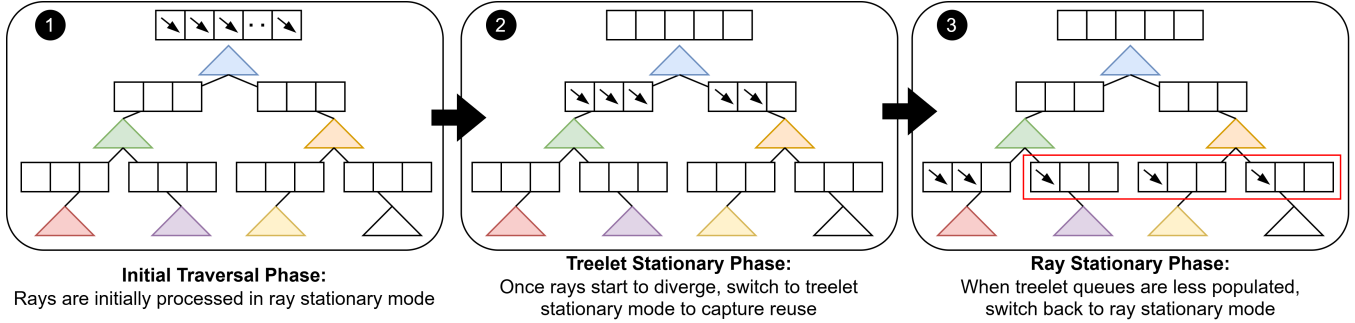


Figure 7. Dynamic treelet queue overview. Each triangle is a treelet and boxes above each treelet are its treelet queue.

RT unit to the CTA scheduler, allowing the RT unit to inject ready-to-resume CTAs back into the GPU’s CTA scheduler and signal it to prioritize the injected CTAs. Once an SM is available for the injected CTAs to resume, the CTA’s state needs to be restored by loading each thread’s register data and SIMT stacks from memory. We quantify the performance impact of CTA resuming in Section 6.6. Afterwards, the CTA can continue execution from where it left off.

#### 4.2 Dynamic Treelet Queue Implementation

To take advantage of increased concurrent rays from ray virtualization, we introduced dynamic treelet queues in Section 3.2. Figure 8 shows RT unit changes to enable treelet queues highlighted in red. When warps issue rays to the RT unit, the ray data is stored in memory and resides in a reserved portion of the L2 cache (1). Warps also enter the RT unit’s warp buffer to begin their **Initial Traversal Phase** (2). Once a warp’s rays start to diverge, the warp is terminated the RT unit updates the **Treelet Count Table** in the **Treelet Controller** by incrementing the ray count for a corresponding treelet that each terminated ray needs to traverse next (3). To identify what rays belong in which treelets, the treelet controller also updates the **Treelet Queue Table** in the L1 cache with the ray IDs and inserts them into the corresponding treelet queue entry (4). Figure 9 shows the layout of the treelet queue table. Each entry stores a treelet address, a ray count, and an array of ray IDs that belong in the treelet up to 32 rays. We choose 32 rays per entry as the whole array of rays can form a full warp. To insert a ray into the treelet queue, the treelet controller finds the corresponding treelet queue entry and appends the ray ID to the array. If a treelet queue entry is full, duplicate treelet entries are allowed in the table. The treelet queue table is implemented as a hash table and collisions are handled with chaining. We use a simple hash function that XORs groups of 2 LSB bits or 4 MSB bits of the treelet address and in our experiments the max collisions for a key is only two. The treelet controller has a state machine to handle reads and modifications to the table. Since the maximum concurrent rays in an RT unit are determined by the degree of ray virtualization, the treelet

queue table can be sized to accommodate most of the rays in the RT unit by dividing the maximum rays by the warp size. If the table overflows, which can happen when too many treelets queues are underpopulated, excess entries are stored in memory and fetched when needed.

While rays in the warp buffer traverse, the treelet controller identifies the largest treelet in the treelet count table and adds it to the list of current treelets if it has enough rays and does not exceed the maximum allowed concurrent processed treelets (4). This design parameter depends on the warp buffer size and how large treelets are sized relative to the cache size. Once a treelet is added to the list of current treelets, the treelet controller loads the treelet from memory to the L1 cache and fetches ray data from the L2 cache (5). Ray data loads bypass the L1 cache to avoid evicting treelet data. The returned ray data is stored in the warp buffer to form a treelet warp (6). Once all rays in a treelet warp traverse to the boundary of a treelet or finish traversal, the treelet controller updates the treelet count table and treelet queue table to reflect the current treelet queue state. Traversal is done when the treelet count table is empty.

#### 4.3 Treelet and Ray Data Preloading

A large portion of the treelet queue overhead comes from loading treelet and ray data when a treelet queue is initially dispatched. One optimization to reduce this latency is to preload the treelet into the L1 cache and preemptively fetch ray data. The treelet controller monitors the treelet count table and when the number of rays in the current treelet queue falls below a threshold, it will send a preload request to fetch the next largest treelet and the corresponding ray data of the largest treelet queue. When preloading a new treelet and its ray data, ideally they should arrive when all rays in the current treelet queue have just finished, meaning the preload needs to happen when the remaining traversal cycles of the current treelet queue equals the memory latency. However, we also do not want early preloads to evict existing treelet data in the cache. With treelet queues, BVH accesses should be cache hits and ray intersection tests have constant latency, thus we can estimate a preload timing based on how many

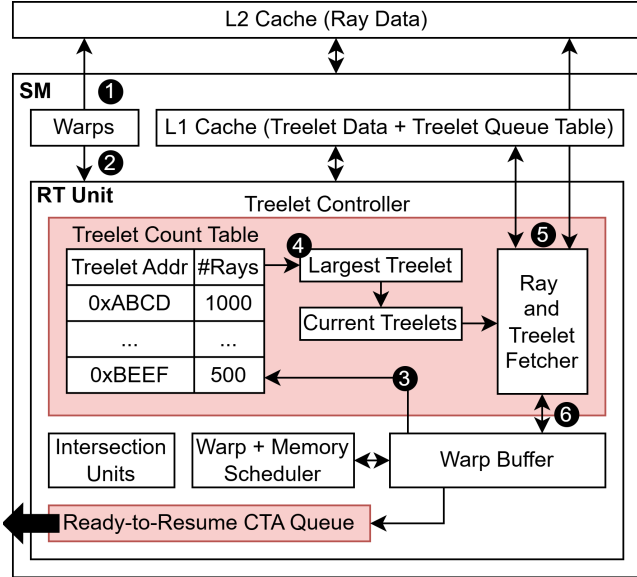


Figure 8. Treelet Queue Implementation.

Treelet Address (Duplicate treelets allowed)	Ray Count	Ray IDs (Max 32 rays per entry)
Treelet A	32	0, 1, ..., 31
Treelet A	32	32, 33, ..., 63
...	...	...
Treelet N	20	4064, ..., 4083

Figure 9. Treelet queue table layout to track what rays belong in which treelet.

warps worth of rays remain in the current treelet queue. The estimated cycles required to process the remaining rays will be the product of the number of remaining warps, the ray intersection latency, and the average depth of a treelet (proxy for nodes intersected per treelet).

An alternative idea to work around preloads evicting current treelet data is to reserve a portion of the cache to hold preloaded treelet data and use smaller treelets instead. For example, sizing treelets to be half the cache size allows rays to traverse one treelet while the next treelet is preloaded. While smaller treelets lessen the opportunity for data reuse since the treelet queues will be smaller, in practice we find the benefit of treelet preloading outweighs this downside. Ray data can also be preloaded similarly.

#### 4.4 Grouping Underpopulated Treelet Queues

One major issue with treelet queues is that individual queues can become underpopulated when the entire ray population starts to further diverge later during ray traversal. Fetching an entire treelet for an underpopulated treelet queue is inefficient when these rays could instead fetch individual BVH nodes in fewer memory accesses. This situation is identified

when the largest treelet queue falls below a certain threshold and there are no more rays left in the current treelet queue to further popular the treelet queues. At this stage, we switch from treelet stationary to ray stationary mode and form warps with rays in the underpopulated treelet queues. To do this, the treelet controller selects treelet queues starting from the first treelet count table entry to acquire enough rays to fill up the warp buffer. Afterward, the RT unit traverses those rays like the baseline until all of them have finished traversal completely before fetching new rays to process.

#### 4.5 Warp Repacking

While processing rays in underpopulated treelet queues, some rays may finish traversal before others and sit idle. Since a warp cannot terminate until its longest ray finishes, SIMT efficiency is greatly impacted lowering memory level parallelism and ray traversal performance. We address this with warp repacking where when half (or a set threshold) of the rays in a warp are inactive, the warp is terminated and the treelet controller will fetch more rays from the underpopulated treelet queues to fill the warp up with new rays. Repacking a warp with new rays is only made possible with ray virtualization as the RT unit can easily fetch new rays from memory. This is similar to the issue observed in the ray predictor by Liu et al. [21] where rays that successfully predict the final hit point can terminate traversal early whereas rays that mispredict need to perform ray traversal fully. This causes the mispredicted rays to hold up the entire warp and Liu et al. repack those rays into new warps to increase SIMT efficiency. Since repacking is done entirely within the context of the RT unit, there is limited data movement overhead and can be done quickly. The data movement overhead of our warp repacking technique primarily comes from loading rays from memory to refill warps, which loads ray data from memory to the warp buffer.

Our warp repacking technique is different from Liu et al. [21] as we repack warps with new rays from memory to increase SIMT efficiency and performance whereas Liu et al. repack warps with rays that already exist inside the warp buffer. Warp reformation methods such as Dynamic Warp Formation [10] and Thread Block Compaction [9] are also related to our warp repacking method but operate by rearranging threads from different warps that are running concurrently. This requires complex modifications to the GPU register file to allow independent register file indexing per lane. This is different and more complex and expensive than warp repacking which merely fetches new rays from memory to fill a portion of a warp when it contains many inactive threads and is again solely contained within the RT unit.



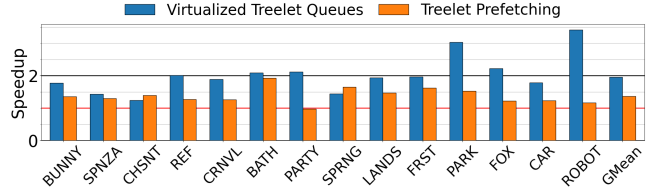
**Table 1.** Vulkan-Sim Configuration.

# Streaming Multiprocessors (SM)	16
Max Warps per SM	32
Warp Size	32
Max CTA per SM	16
Warp Scheduler	GTO
# Registers / SM	32768
Instruction Cache	128KB, 16-way assoc., 39 cycles
L1 Data Cache + Shared Memory	16KB, Fully assoc. LRU, 39 cycles
L2 Unified Cache	128KB, 16-way assoc. LRU, 187 cycles
Core, Interconnect, L2 Clock	1365 MHz
Memory Clock	3500 MHz
# RT Units / SM	1
RT Unit Warp Buffer Size	1

## 5 Methodology

We extend Vulkan-Sim [28] to simulate our virtualized treelet queue architecture. Vulkan-Sim is a cycle-based simulator that models the Vulkan ray tracing pipeline and RT unit architecture. To model ray virtualization, we terminate a CTA after all its threads have executed `traceRayEXT()` which offloads ray traversal to the RT unit. The CTA state, thread registers, and SIMT stack of all warps in the CTA are stored in memory. The number of saved registers is from running `ptxas`, NVIDIA’s PTX shader assembler, on the raygen shader which gives the maximum amount of required registers. This is a conservative number as in practice, only live registers need to be saved. When the CTA is ready to resume, the CTA is injected back into the GPU’s CTA scheduler and loads are issued to retrieve the saved state before the CTA is scheduled to an SM for modeling timing overheads. We set ray virtualization to allow up to 4096 rays in flight per SM. To simulate treelet queues, we used the treelet partitioning code by Chou et al. [8] and size treelets to be half the L1 cache size. This allows storing two treelets in the L1 cache at a time, enabling the RT unit to preload a treelet while currently processing one. Ray data is held in the L2 cache in a reserved section, and also stored in memory if evicted by other rays. Ray data loads bypass the L1 to not evict treelet data. The data movement latency and energy cost of warp repacking comes from fetching rays from memory to refill warps, which loads ray data from memory to the warp buffer. The latency is captured by memory loads in the timing model and load energy is captured by `Accelwattch` in Vulkan-Sim.

Table 1 shows our simulated GPU configuration. We use a warp buffer size of one according to the profiling from Vulkan-Sim [28]. Cache latencies are referenced from the NVIDIA RTX 3080 configuration in `Accel-Sim` [17]. While modern GPUs feature much larger sizes than what we simulate (128KB L1, 72MB L2 for RTX 4090 GPU), BVH trees in real-world applications are also much larger than our benchmarks, reaching hundreds of MB and far exceeding the cache capacity [4]. Prior works show it is possible to estimate the performance of a full system through scale-model simulation



**Figure 10.** Overall speedup of Virtualized Treelet Queues (4096 concurrent rays) compared to Treelet Prefetching [8].

with good accuracy for both GPGPU workloads [29] and ray tracing [12]. The baseline GPU uses the treelet traversal order by Chou et al. [8]. We compare our proposal to Treelet Prefetching [8] which is the most recent work employing treelets on ray tracing capable GPUs and the implementation is available.

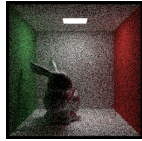
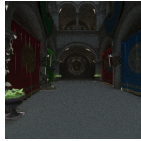

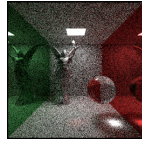


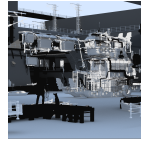
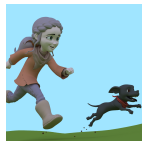
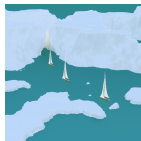
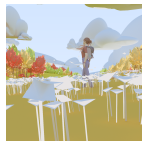

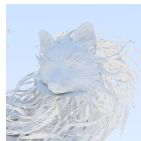
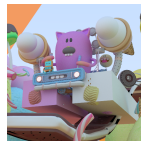

### 5.1 Evaluation Benchmarks

We evaluate our proposal on scenes from `LumiBench` [22] and Table 2 shows summary statistics of the scenes sorted by BVH size. The BVH trees range from 13.18MB to over 1GB in size and contain up to 20.6M triangles. We use a 4-wide BVH tree built by Intel Embree [31] and is then repacked into the BVH format from Benthin et al. [7] by Vulkan-Sim. We simulate all scenes at 256x256 resolution at one sample per pixel (SPP) with three max bounces or until the secondary ray’s contribution to the final pixel color is too small. The scenes are path traced where primary rays are traced from the camera to the image pixel and secondary rays are traced from the intersection point of the primary ray and scattered based on the material at the hit point. While we are aware most real-time ray tracing applications obtain the primary hit point of a ray using rasterization and not by tracing primary rays, the simulation framework provided by `LumiBench` and Vulkan-Sim does not support rasterization. Also, low SPP counts are often used in real-time ray tracing to achieve interactive frame rates.

## 6 Results

Figure 10 shows the overall speedup of Virtualized Treelet Queues with 4096 concurrent rays compared to Treelet Prefetching [8]. Scenes are sorted by ascending BVH size. Dynamic treelet queues when combined with grouping underpopulated treelet queues and warp repacking achieve a 95% speedup over the baseline and outperforms treelet prefetching [8] by 43%. Two scenes have less speedup compared to others. In SPINZA, SIMT efficiency is already high so our optimizations of grouping underpopulated treelet queues and warp repacking which boost SIMT efficiency are less effective. CHSNT does not have high L1 miss rates in the baseline so it benefits less from treelets.

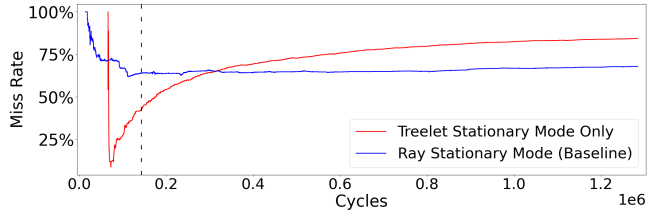
**Table 2.** Summary of evaluation scenes from LumiBench [22].

Scenes	BUNNY	SPNZA	CHSNT	REF	CRNVL	BATH	PARTY
							
BVH Size (MB)	13.18	22.84	28.28	40.36	60.67	112.79	156.05
Triangle Count	144.1K	262.3	313.2K	448.9K	449.6K	423.6K	1.7M
Scenes	SPRNG	LANDS	FRST	PARK	FOX	CAR	ROBOT
							
BVH Size (MB)	177.96	303.48	380.51	542.53	648.48	1,328.23	1868.95
Triangle Count	1.9M	3.3M	4.2M	6.0M	1.6M	12.7M	20.6M

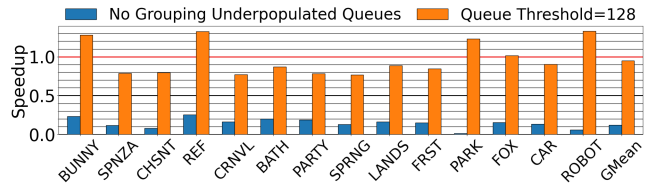
### 6.1 Treelet Stationary Phase Performance

To evaluate whether treelet queues and the treelet-stationary mode are effective at improving the memory system performance, we compare the L1 cache miss rates of BVH accesses in the baseline GPU and treelet queues. Figure 11 shows the L1 cache miss rate over time for the LANDS scene in LumiBench. The blue line represents the L1 cache miss rate of the baseline GPU operating in ray-stationary mode where the cache miss rates start out high and plateau at around 60%. The red line shows the L1 miss rates of the RT unit if it were to operate permanently in treelet-stationary mode. Initially treelet queues provide a large reduction in cache miss rates, going as low as 9% where almost every ray in the RT unit is traversing the same treelet and each treelet fetch is amortized over many rays. However, as rays diverge and treelet queue sizes decrease, the cache miss rate increases and eventually plateaus at around 75% to 80%, surpassing the baseline GPU’s cache miss rate. This is because when treelet queues are small, the RT unit fetches more BVH nodes than required, leading to more cache misses. This suggests that treelet queues are very effective but only when the queues are large enough. The dashed vertical black line in Figure 11 indicates the threshold where the treelet queues start becoming underpopulated and the RT unit groups these treelet queues together to process in ray-stationary mode to avoid the high cache miss rates.

While treelet-stationary mode decreased cache miss rates, the miss rate of the overall system increases after transitioning back to ray-stationary mode with warp repacking due to ray data loading and CTA restores for ray virtualization. Despite this, both SIMT efficiency and the amount of overlapped memory accesses double, which resulted in speedups and are shown in the next sections.



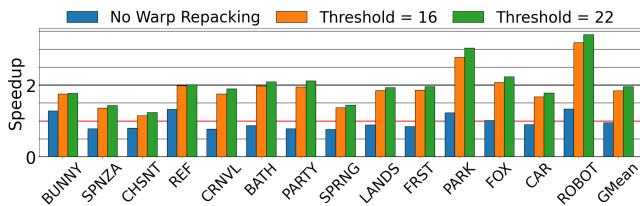
**Figure 11.** L1 cache miss rate over time under treelet-stationary mode compared to the baseline RT unit for the LANDS scene.



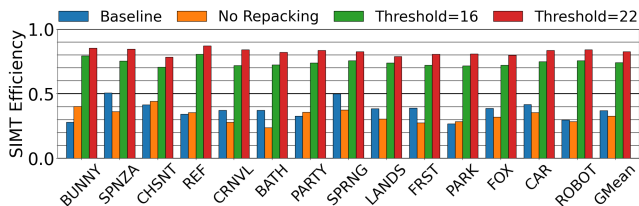
**Figure 12.** Speedup of grouping underpopulated treelets.

### 6.2 Grouping Underpopulated Treelet Queues

Figure 12 shows a speedup of treelet queues with no optimizations and treelet queues that group underpopulated queues into warps, compared to the baseline GPU. The queue threshold specifies the minimum number of rays in a treelet queue before it can be considered underpopulated. As a result, when we group underpopulated treelet queues into ray stationary warps, we see a 8× performance increase over the naive treelet queue implementation when using a threshold of 128 rays. However, this optimization alone is not enough to yield good treelet performance, being on average 5% slower than the baseline and requires additional optimizations.



(a) Speedup comparison of treelet queues with warp repacking.



(b) SIMT efficiency comparison of treelet queues with warp repacking.

Figure 13. Performance effects of warp repacking.

### 6.3 Warp Repacking

Figure 13 shows the speedup of warp repacking at different repack thresholds. A threshold of 16 means if a warp has less than 16 active rays, it will be repacked into a new warp. Speedups are normalized to the baseline GPU (indicated by the red line) and all variants group up underpopulated treelet queues. Without warp repacking, treelet queues has a 5% slowdown compared to the baseline. Warp repacking with a 16-thread threshold provides an 84% speedup and a 22-thread threshold at 95% speedup. Without warp repacking, treelet queues cannot maintain high SIMT efficiency due to the high variance in the number of BVH nodes each ray needs to traverse. This causes some rays to finish traversal earlier, leading to few active rays in a warp and lowering memory level parallelism as there are fewer rays to issue memory accesses from. This is evident from Figure 13b which plots the SIMT efficiency of treelet queues with and without warp repacking alongside the baseline. SIMT efficiency is the ratio of active threads to the total number of threads in a warp, 1 being a fully active warp. Both the baseline and treelet queues without warp repacking have similar SIMT efficiencies of 0.37 and 0.33, indicating that only 10 to 12 rays are active in a warp on average. By grouping rays from underpopulated treelet queues, we may even worsen the problem by potentially executing rays with distant hit points together. Enabling warp repacking increases SIMT efficiency up to 0.82 with a 22-thread threshold, which more than doubles the baseline and correlates to the speedup.

### 6.4 Traversal Mode Breakdown

As mentioned in Section 3.2, dynamic treelet queues operate in three phases: an initial traversal phase in ray stationary

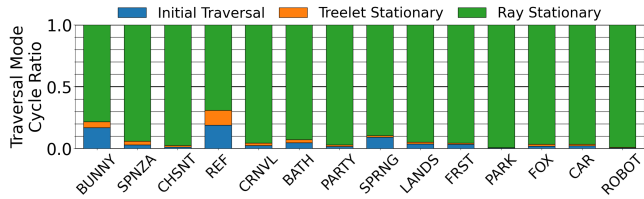


Figure 14. Cycle distribution of ray traversal modes.

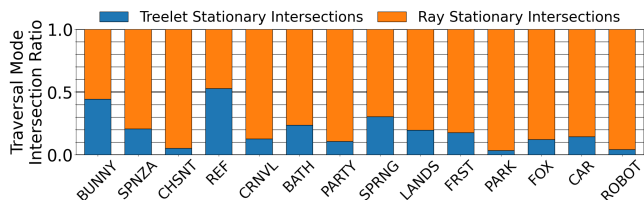


Figure 15. Ratio of ray intersection tests processed under different ray traversal modes.

mode, a treelet stationary mode, and a final phase where underpopulated treelet queues are processed in ray stationary mode. Figure 14 shows the cycle ratio of the three phases when both grouping underpopulated treelet queues and warp repacking are enabled. The bar height is proportional to cycles in the optimized system and not operations in baseline. For all scenes, after a short initial traversal, the RT unit spends the majority of cycles in ray stationary mode. The large ray stationary cycle ratio is due to the high divergence of rays in the later stages of traversal, causing treelet queues to be underpopulated and inefficient to process. Additionally, most rays in the ray stationary phase are secondary rays, which there are less of compared to primary rays, making it harder to populate treelet queues. This suggests treelet queues are more beneficial in early stages of ray traversal and more optimizations on improving ray stationary traversal which contains mostly diverged rays is needed to further improve performance. While the treelet stationary phase seems short, it does not mean treelet queues are ineffective. We experimented with skipping the treelet stationary phase by setting the treelet queue threshold to zero and immediately starting ray stationary phase and on average scenes took 4-6 times more cycles to complete compared to the baseline.

Figure 15 plots the ratio of ray intersection tests processed under different traversal modes. The treelet stationary phase processes up to 52% of ray intersection tests with an average of 15%, while ray stationary mode processes the rest. This ratio greatly depends on the scene and how divergent the rays are. With more divergent rays such as tracing more ray bounces, the treelet stationary phase is expected to process fewer intersection tests. When tracing less divergent batches of rays such as when tracing more samples per pixel, the treelet traversal mode ratio increases.

## 6.5 Area Overheads

This section breaks down the size of various storage structures in our design. For the **Treelet Count Table** in the RT unit, for each table entry we store a treelet address and its corresponding ray count. Since our treelet is sized to half the L1 cache size which is 8KB and the treelets can be packed together in memory as suggested by Chou et al. [8], we only need to save the most significant 19 bits of the treelet address. To track ray counts for a maximum of 4096 rays, we need 12 bits. We track only treelets whose ray count exceeds the underpopulation threshold as we process underpopulated treelet queues in ray stationary mode. If the treelet count table is full when inserting a new treelet, we evict the smallest treelet queue and process those rays in ray stationary mode later. In our experiments with a 127 ray threshold, the maximum amount of treelet queues before switching to ray stationary traversal did not exceed 549. Amongst those, only a maximum of 13 queues exceeded the threshold at any given time. Thus 600 entries is enough to capture treelet ray counts and is a 2.2KB storage overhead in the RT unit.

We store the complete ray data in memory, consisting of the ray origin, ray direction, tmin and tmax values which is 32B per ray. Since the RT unit processes a maximum of 4096 concurrent rays, it totals 128KB and fits in the L2 cache. The **Treelet Queue Table** (Figure 9) tracks treelet addresses and their corresponding ray IDs and is stored in the L1 cache. Each treelet address is 19 bits and each ray ID is 12 bits, and we allocate 128 table entries to accommodate when all 4096 rays are in warps (such as when all rays are in the root treelet). The total size of the treelet queue table is  $(19 + 32 \times 12 \text{ bits}) \times 128 \text{ entries}$  which is 6.29KB, meaning the L1 cache fits both the treelet data and the treelet queue table.

## 6.6 Ray Virtualization Overheads and Energy

Ray virtualization suspends stalled CTAs to increase the number of concurrent rays in flight. The primary overhead of ray virtualization is the cost of storing the suspended CTA state in memory and resuming CTAs by loading the saved state before they can be executed. According to ptxas, the raygen shader in LumiBench uses a maximum of 10 32-bit registers per thread. We also need to save a 32-bit SIMT mask, a program counter (PC), and a reconvergence PC per SIMT stack depth for each warp in the CTA. Figure 16 shows the overhead of ray virtualization is on average a 10% slowdown. Figure 17 shows the energy results with and without virtualization compared to the baseline. Treelet queues provides 60% in energy savings compared to the baseline which is primarily from the reduced cycles needed to complete the ray traversal. Ray virtualization consumes 11% of the total energy in our design, predominantly from the increased memory accesses to save and load CTA states.

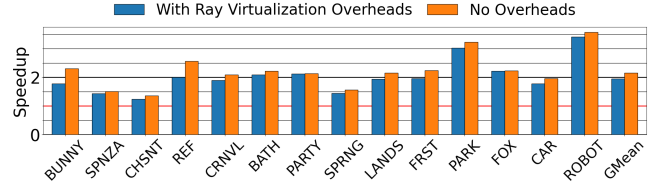


Figure 16. Ray virtualization performance overheads.

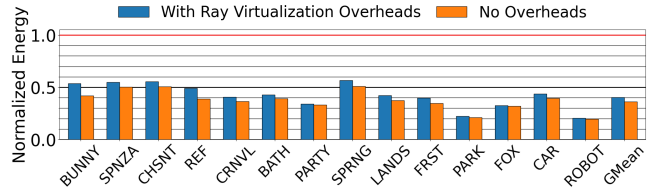


Figure 17. Ray virtualization energy costs.

## 7 Related Work

### 7.1 GPU Resource Virtualization

Prior research explored how to oversubscribe GPU resources to improve performance. Yoon et al. [33] improved the performance of GPGPU workloads due to hardware underutilization. They proposed a virtual thread mechanism to schedule CTAs up to the scheduling limit that bypasses hardware limits. To handle CTA overscheduling, they swap out stalled CTAs and swap in new CTAs, effectively increasing thread level parallelism. Jeon et al. [16] proposed a GPU register file virtualization scheme to accommodate more threads while maintaining the same register file size. They allow multiple warps to share the same registers by releasing dead registers from a warp and reallocating them to another warp that is scheduled later. Intel [1] also described their technique at a high level, similar to what we call ray virtualization, where they terminate raygen shaders after issuing traceRayEXT() to enable sorted hit shader execution for better coherence during their Games Developer Conference (GDC) 2022 talk.

### 7.2 Ray Traversal Acceleration

**7.2.1 Ray Sorting.** Sorting rays before traversal improves ray coherency. Garanzha and Loop [11] sorted rays into coherent packets based on ray origin and direction before processing them. Moon et al. [25] sorted rays by their first intersection point with the scene. Meister et al. [24] estimate ray termination points to achieve better secondary ray coherency. However these works suffer from high sorting overhead, taking almost as long as ray traversal itself. Treelet queues essentially achieve a similar goal by grouping up rays based on their accessed treelet, but without the high overhead.

**7.2.2 Hardware Optimizations.** Lu et al. [23] addressed SIMT divergence in CUDA-based ray tracing by regrouping rays with the same traversal state (internal nodes/leaf nodes/inactive) together. We are inspired by Lu et al. to regroup warps with many inactive rays during ray stationary mode. Liu et al. [21] proposed a ray predictor to predict the intersecting leaf node of a ray to skip BVH traversal. However, it only works well for anyhit rays and is challenging to apply to closest-hit rays. We adopt their warp repacking technique to improve SIMT efficiency for treelet queues.

### 7.3 BVH Tree Memory Optimizations

BVH compression and memory optimizations have been explored to reduce memory traffic and can be used in conjunction with our proposal for even larger performance improvements. Gabor et al. [20] introduced a novel memory layout and node addressing scheme to reduce the memory footprint of child node pointers in the BVH. Ylitie et al. [32] presented a compressed wide BVH layout reducing the size of the BVH and memory traffic significantly. Benthin et al. [7] proposed a different BVH compression method focused on compressing leaf nodes and is the BVH layout used in Vulkan-Sim [28]. NVIDIA introduced a displaced micromesh engine in their Ada Lovelace GPUs to reduce BVH build times and storage costs, reporting 2× faster ray-triangle intersection tests [3].

## 8 Conclusion

This work explores how to efficiently implement treelet queues on modern ray tracing capable GPUs. We propose Virtualized Treelet Queues, an architecture that increases the number of concurrent rays in flight and treelet queues that dynamically switch between treelet and ray stationary traversal modes to increase efficiency. We find that a naive treelet implementation is ineffective as rays diverge quickly and cause treelet queues to be underpopulated leading to low SIMT efficiency. Through grouping underpopulated treelet queues together and warp repacking to boost SIMT efficiency, we achieve on average 95% speedup against a baseline GPU with ray tracing acceleration.

Recent research have explored ways to accelerate general purpose workloads on RT units by reformulating their algorithms to fit the ray tracing pipeline, for instance database indexing in RT-DBSCAN [26] and RTIndex [15] and nearest neighbor search in RTNN [34]. Since these modified workloads transform their data into a BVH tree and the search query into a ray, this work can be potentially applied to accelerate these workloads. With proposals from Ha et al. [14] and Barnes et al. [6] that extend ray tracing accelerators in GPUs to support more general tree traversal workloads, we believe the hardware modifications and treelet queue optimizations in this work in conjunction to accelerate general tree traversal workloads on GPUs as well.

## References

- [1] A Quick Guide to Intel’s Ray Tracing Hardware (Presented by Intel). <https://gdcvault.com/play/1027816/A-Quick-Guide-to-Intel>.
- [2] Vulkan. <https://www.vulkan.org/>.
- [3] NVIDIA ADA GPU ARCHITECTURE, 2023. <https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf>.
- [4] Raytracing on AMD’s RDNA 2/3, and Nvidia’s Turing and Pascal, 2023. <https://chipsandcheese.com/2023/03/22/raytracing-on-amds-rdna-2-3-and-nvidias-turing-and-pascal/>.
- [5] Timo Aila and Tero Karras. Architecture considerations for tracing incoherent rays. In *Proc. ACM Conf. on High Performance Graphics (HPG)*, pages 113–122, 2010.
- [6] Aaron Barnes, Fangjia Shen, and Rogers Timothy G. Extending GPU Ray-Tracing Units for Hierarchical Search Acceleration. 2024.
- [7] Carsten Benthin, Ingo Wald, Sven Woop, and Attila T. Áfra. Compressed-Leaf Bounding Volume Hierarchies. In *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2018.
- [8] Yuan Hsi Chou, Tyler Nowicki, and Tor M. Aamodt. Treelet prefetching for ray tracing. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2023.
- [9] Wilson W. L. Fung and Tor M. Aamodt. Thread block compaction for efficient simt control flow. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2011.
- [10] Wilson W.L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2007.
- [11] Kirill Garanzha and Charles Loop. Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. *Computer Graphics Forum*, 29(2):289–298, 2010.
- [12] Davit Grigoryan, Yuan Hsi Chou, and Tor M. Aamodt. Zatel: Sample Complexity-Aware Scale-Model Simulation for Ray Tracing. In *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2024.
- [13] Michael Guthe. Latency Considerations of Depth-first GPU Ray Tracing. In *Eurographics 2014 - Short Papers*, 2014.
- [14] Dongho Ha, Lufei Liu, Yuan Hsi Chou, Seokjin Go, Won Woo Ro, Hung-Wei Tseng, and Tor M Aamodt. Generalizing ray tracing accelerators for tree traversals on gpus. 2024.
- [15] Justus Henneberg and Felix Schuhknecht. RTIndex: Exploiting hardware-accelerated GPU raytracing for database indexing. *arXiv preprint arXiv:2303.01139*, 2023.
- [16] Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim, and Murali Annavaram. Gpu register file virtualization. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2015.
- [17] Mahmoud Khairy, Zhesheng Shen, Tor M Aamodt, and Timothy G Rogers. Accel-Sim: An extensible simulation framework for validated GPU modeling. In *Proc. IEEE/ACM Int’l Symp. on Computer Architecture (ISCA)*, pages 473–486, 2020.
- [18] Hansung Kim, Angie Wang, Sizhuo Zhang, and Sophia Shao. Cost of Divergence in Ray Tracing: Performance Characterization on CPU and GPU. In *Int’l Workshop on Domain Specific System Architecture (In conjunction with Int’l Symp. on Computer Architecture (ISCA))*, 2023.
- [19] Daniel Kopta, Konstantin Shkurko, Josef Spjut, Erik Brunvand, and Al Davis. Memory considerations for low energy ray tracing. In *Computer Graphics Forum*, volume 34, pages 47–59, 2015.
- [20] Gabor Liktor and Karthik Vaidyanathan. Bandwidth-Efficient BVH Layout for Incremental Hardware Traversal. In *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2016.
- [21] Lufei Liu, Wesley Chang, Francois Demoullin, Yuan Hsi Chou, Mohammadreza Saed, David Pankratz, Tyler Nowicki, and Tor M Aamodt. Intersection Prediction for Accelerated GPU Ray Tracing. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, pages 709–723, 2021.
- [22] Lufei Liu, Mohammadreza Saed, Yuan Hsi Chou, Davit Grigoryan, Tyler Nowicki, and Tor M. Aamodt. LumiBench: A Benchmark Suite

- for Hardware Ray Tracing. In *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2023.
- [23] Yashuai Lü, Libo Huang, Li Shen, and Zhiying Wang. Unleashing the power of gpu for physically-based rendering via dynamic ray shuffling. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2017.
- [24] Daniel Meister, Jakub Boksansky, Michael Guthe, and Jiri Bittner. On Ray Reordering Techniques for Faster GPU Ray Tracing. In *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, pages 1–9, 2020.
- [25] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-Oblivious Ray Reordering. *ACM Transactions on Graphics (TOG)*, 2010.
- [26] Vani Nagarajan and Milind Kulkarni. RT-DBSCAN: Accelerating DBSCAN using ray tracing hardware. *arXiv preprint arXiv:2303.09655*, 2023.
- [27] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Third Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2018.
- [28] Mohammadreza Saed, Yuan Hsi Chou, Lufei Liu, Tyler Nowicki, and Tor M. Aamodt. Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing. In *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2022.
- [29] Hossein SeyyedAghaei, Mahmood Naderan-Tahan, and Lieven Eeckhout. GPU Scale-Model Simulation. In *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2024.
- [30] Konstantin Shkurko, Tim Grant, Daniel Kopta, Ian Mallett, Cem Yuksel, and Erik Brunvand. Dual Streaming for Hardware-Accelerated Ray Tracing. In *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2017.
- [31] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (TOG)*, 2014.
- [32] Henri Ylitie, Tero Karras, and Samuli Laine. Efficient Incoherent Ray Traversal on GPUs through Compressed Wide BVHs. In *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2017.
- [33] Myung Kuk Yoon, Keunsoo Kim, Sangpil Lee, Won Woo Ro, and Murali Annavaram. Virtual thread: Maximizing thread-level parallelism beyond gpu scheduling limit. In *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2016.
- [34] Yuhao Zhu. RTNN: accelerating neighbor search using hardware ray tracing. In *Proc. ACM Symp. on Prin. and Prac. of Par. Prog. (PPoPP)*, pages 76–89, 2022.