LumiBench: A Benchmark Suite for Hardware Ray Tracing

Lufei Liu[1] Mohammadreza Saed[1] Yuan Hsi Chou[1] Davit Grigoryan[1]
Tyler Nowicki[2] Tor M. Aamodt[1]

[1]*University of British Columbia, Canada*
[2]*Huawei Technologies, Canada*

Abstract

Ray tracing as a graphics rendering method is becoming increasingly popular in real-time applications, supported by dedicated accelerator cores in the latest generation GPUs. However, the high computational intensity of the ray tracing algorithm still limits the visual effects that can be produced while maintaining a high frame rate. To improve ray tracing hardware, it is important to understand the underlying characteristics of ray tracing workloads and identify performance bottlenecks. In this paper, we present LumiBench, the first benchmark suite for evaluating ray tracing hardware performance in modern GPUs designed to execute on the Vulkan-Sim GPU simulator. LumiBench features a diverse set of scenes and shaders that are representative of real applications but simple enough to be simulated in a reasonable amount of time. We first evaluate LumiBench against Rodinia to highlight the difference between ray tracing versus general purpose workloads and demonstrate the need for a dedicated benchmark suite. Then, we characterize the workloads included in LumiBench, which are organized into several clusters targeting different aspects of the ray tracing pipeline, and provide insights for future architectural research.

1. Introduction

Ray tracing is a rendering technique used in computer graphics over decades for its ability to produce photorealistic images. However, ray tracing is computationally expensive, and thus primarily used in offline rendering applications, such as animated movies and computer-aided design (CAD). More recently, advances in hardware and software have made it possible to use ray tracing in real-time applications, including video games. Hardware support for ray tracing is growing and many of the latest generation GPUs now feature specialized accelerators. However, performance is still far from ideal, allowing only a few rays per pixel to be traced. Experts in ray tracing agree that further improvements in hardware are crucial to widespread adoption and improved visual quality in real-time applications [56]. The photorealistic nature of ray traced images also make them ideal for applications such as virtual and augmented reality (VR/AR), but current hardware struggles to maintain sufficiently high frame rates for an immersive experience. Studying the performance of ray tracing workloads is important to understand bottlenecks and support the architectural design of future hardware.

Although the newly-released Vulkan-Sim GPU architecture simulator [50] now provides a platform for studying the performance and bottlenecks of ray tracing hardware, there is still a lack of a standardized benchmark suite for academic research. Many existing ray tracing benchmarks are too complex to be simulated in a reasonable amount of time in the range of hours to days and are often not publicly available. Other common benchmarks such as MediaBench [42] and Rodinia [33] do not include ray tracing workloads. As a result, computer graphics researchers typically hand-select scenes to study from available open-source projects without a full understanding of the underlying characteristics. For example, recent research on hardware ray tracing such as the ray intersection predictor [44] and Mach-RT [54] evaluate their ideas using six and seven scenes with no description of the scene characteristics or selection methods and only one scene is shared between the two works.

In this paper, we present LumiBench, a benchmark suite for hardware ray tracing. LumiBench is designed to be representative of real-world ray tracing workloads and to be easily simulated in a reasonable amount of time. We expect LumiBench to be useful for identifying performance bottlenecks and for evaluating architectural design choices targeting the performance of the ray tracing pipeline. Our contributions are as follows:

- We propose a set of benchmark workloads to study the performance of hardware ray tracing accelerators on GPUs.
- We identify useful metrics for evaluating the ray tracing workloads and use them to analyze the similarity between benchmark scenes.
- We characterize the workloads using an updated version of Vulkan-Sim and highlight insights for architectural research.

2. Background and Motivation

We design LumiBench specifically to support the modern ray tracing pipeline. In this section, we provide basic background information on ray tracing and Vulkan-Sim, and discuss the motivation for our benchmark suite.

2.1. Ray Tracing

The ray tracing algorithm models the propagation of light in a 3D scene by tracing rays from the camera through each pixel of the image plane and following the path of the rays as they interact with objects in the scene. The final color of each pixel is determined by objects that the ray intersects in its path. In modern ray tracing, the base geometric primitives of the scene are organized into an acceleration structure, most commonly the bounding volume hierarchy (BVH), to reduce the search space for the intersection of a ray with the scene. The BVH is a spatial tree structure where each node is an axis-aligned bounding box (AABB) that tightly encloses the geometric primitives in its subtree. Modern ray tracing APIs divide the BVH structure into two levels: the bottom-level acceleration structure (BLAS) built using the primitives and the top-level acceleration structure (TLAS) which contains instances of the BLAS. This configuration avoids duplicating geometry when the same object appears multiple times in a scene and instead uses a ray transformation to map the ray intersection test to the appropriate BLAS instance.

To traverse the BVH tree, a ray is tested for intersection with each AABB in the tree, starting from the root node. When the ray reaches a TLAS leaf, a matrix transformation is applied to the ray to bring it to the BLAS coordinate system. Then, once the ray reaches a BLAS leaf node, it is tested against the enclosed geometric primitives to determine the closest intersection. The traversal and intersection procedure commonly follows a while-while loop [20] and we refer readers to [51] for more details on ray tracing.

Rendering using ray tracing also involves other steps, which are organized into shader stages in the modern ray tracing pipeline, different from the raster pipeline, illustrated in Figure 1. Each of these shaders executes as kernels on the GPU. The programmable ray generation shader is responsible for defining ray properties including origin and direction, then launching the `traceRay` command. The green stages in Figure 1 implement the traversal and intersection algorithm, abstracted away in hardware. Although the traversal occurs entirely in fixed-function hardware, the optional intersection and anyhit programmable shaders allow for custom intersection tests, discussed more in Section 3.1.4. Once traversal is complete, either the closest hit or miss shader executes to determine the final color of the pixel. Finally, the `traceRay` command terminates and returns to the ray generation shader.

2.2. Graphics Processing Units (GPUs)

Although GPUs were initially designed for graphics rendering, their high throughput and parallelism provide an advantage in many other applications. As GPUs gained popularity in general purpose computing, benchmarks such as Rodinia [33] were developed to evaluate performance and aid in the architectural design of GPUs. Insights from Rodinia show that many optimization techniques are not intuitive when working with the unique architecture of GPUs, and a representative benchmark suite is crucial for evaluating different optimizations. This problem is exacerbated when the GPU architecture is augmented with a specialized ray tracing unit (RT unit), which is still relatively new and not widely explored. Moreover, the majority of ray tracing programs execute specifically inside this dedicated RT unit, which may benefit from different optimizations than the general purpose GPU cores.

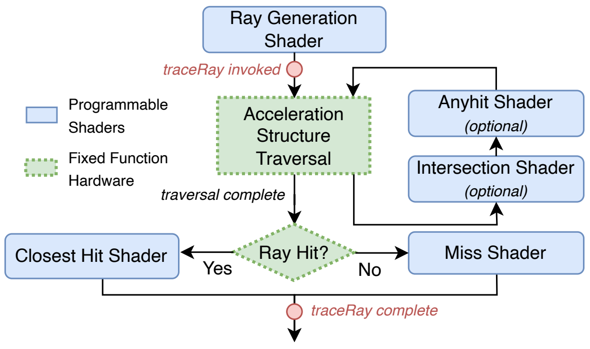


Figure 1: The ray tracing pipeline.

Since the introduction of the RT Core [31] from NVIDIA in 2018, interest in hardware support for ray tracing has been growing. Specialized ray tracing hardware is now available in most of the latest generation GPUs, including the Intel ARC GPU with Ray Tracing Units (RTU) [39], the Ray Accelerator (RA) from AMD [17], and Imagination Technology's Ray Acceleration Cluster (RAC) [26].

However, these ray tracing accelerators are relatively new and many architectural optimizations have not been fully explored. For example, rays are grouped into warps or wavefronts for execution on the GPU, but rays easily diverge and often invoke different shaders, leading to inefficiencies. New ideas like Shader Execution Reordering (SER) [48], the Thread Sorting Unit (TSU) [39], and the Coherency Engine [26] have been introduced into the updated ray tracing accelerators, which aims to group rays accessing the same shaders together to improve performance. These solutions are just the beginning of architectural research for ray tracing hardware. Other inefficiencies, such as divergence during ray traversal, remain unsolved.

2.3. Vulkan-Sim

Vulkan [16] is an open-source graphics API similar to OpenGL that provides programmers with lower level control over hardware resources. The Vulkan API standardizes ray tracing across different hardware vendors and provides a common interface, ideal for our benchmarking purposes. There are many existing academic tools designed for Vulkan, including Vulkan-Vision [49] and Vulkan-Sim [50], which allow for detailed analysis of Vulkan ray tracing workloads.

Vulkan-Vision supports workload characterization by intercepting Vulkan API calls through an intermediate layer and analyzing the collected statistics, but requires the workload to execute on real hardware. Although running applications on real hardware may provide the most accurate performance results, architecture-dependent characterization can hide underlying, inherent behavior of the workload [38]. Vulkan-Sim is a cycle-level simulator for Vulkan ray tracing workloads and best fits our goal of studying the performance of ray tracing hardware and evaluating new hardware changes. Previous graphics simulators such as TEAPOT [23] and Emerald [35] were designed for rasterization-based rendering and GPU simulators such as Accel-Sim [40] do not model ray tracing hardware. Vulkan-Sim includes five scenes that serve to validate the simulator rather than to represent real-world workloads. Of those five scenes, *TRI* and *REF* are toy scenes with only one and 50 triangles respectively, and the authors do not assess the diversity of the remaining three scenes.

3. LumiBench Benchmark Suite

LumiBench introduces a set of benchmark scenes and shaders that can be combined to create different types of workloads. We choose our benchmark scenes to stress different aspects of the ray tracing hardware. In computer graphics, the worst-case scenarios are often the most important to consider because of the strict frame rate requirements for 60 frames per second (FPS) of real-time applications. Each frame must be rendered within a certain amount of time to maintain the desired frame rate, and thus the slowest ray or pixel determines the frame rate. This requirement differs from most compute workload evaluations, which are often designed to be representative of the average case.

The performance of ray tracing applications is highly dependent on the shader that is executed and the scene that is rendered. In this section, we describe our shader and scene selections for LumiBench.

3.1. Stress Cases

We consider three major stress case scenarios in our benchmark scenes, similar to the approach used in the Benchmark for Animated Ray Tracing (BART) [43]. Most stress scenarios in BART are specific to animated scenes and analyzing software algorithms so we do not include them in our benchmark. However, large working sets, overlap of bounding volumes, and different quantities of light sources apply to both animated and static scenes. We adapt these stresses for our benchmarking purposes and add special cases that make use of the optional shaders in the ray tracing pipeline. Although these scenarios stress the system, they are all commonly found in real-world applications.

3.1.1. Large working set. The most obvious stress case for ray tracing is a large working set of geometric primitives that do not fit in the cache. The BVH structure size scales logarithmically with the number of scene primitives due to the tree structure and correlates directly to the number of memory accesses during ray traversal. Ray tracing is a memory-bound application, with random and divergent memory accesses during traversal, often thrashing the cache as shown in Section 5.3.

A subclass of this stress case is scenes with many BLAS instances. Although the overall memory footprint is reduced, ray traversal becomes inefficient because primitives in different BLAS instances cannot be mixed and optimized together during BVH construction. Compared to a flat, single-level BVH structure, ray traversal performance can typically be $2\times$ slower [27] from visiting unnecessary nodes.

3.1.2. Long and thin primitives. When scene primitives such as triangles and curves are long and thin, such as in hair and ropes, the BVH structure is inefficient at pruning the search space for intersections. The BVH structure is optimized when the bounding boxes enclose the primitives as tightly as possible, but these boxes are axis-aligned and leave a large empty space when long and thin primitives do not align with the box. As a result, bounding volumes overlap more often and rays are more likely to traverse down unnecessary subtrees.

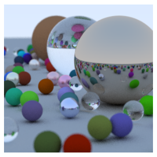
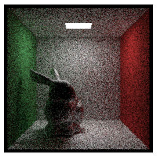
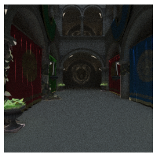
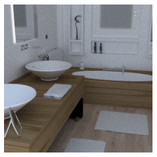
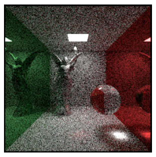
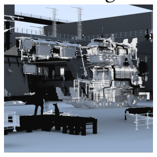
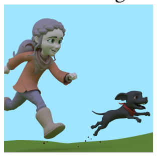
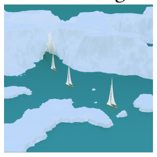
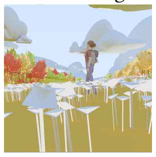
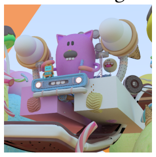
3.1.3. Indoor and enclosed. Rendering an enclosed scene means that every ray must intersect with at least one object in the scene, and traverse the BVH structure from the root to a leaf node. Open scenes, however, include rays that can miss the scene entirely and skip most of the BVH traversal. Also, lighting in enclosed scenes is more complex than open scenes because, physically, the light likely bounces between multiple surfaces before reaching the camera. In the context of ray tracing, this indirect lighting is created through global illumination, which simulates each bounce of light by tracing additional rays from the intersection point of the original ray.

3.1.4. Intersection and anyhit shaders. Some scenes are composed of procedural geometries that are defined through equations rather than the typical triangle meshes. Procedural geometry cannot use the hardware ray-triangle intersection units and requires custom intersection tests instead, which are defined in the *intersection shader* stage of the ray tracing pipeline.

Also, some applications require *anyhit shaders*, most commonly used for alpha testing. For example, alpha masking is a technique where complex shapes, like leaves, are defined by a texture cutout with an alpha channel rather than a detailed triangle mesh. When a ray intersects a texture-masked triangle during traversal, the anyhit shader is invoked to fetch the texture and check the alpha channel to determine if the intersection is valid. The NVIDIA Ada architecture GPUs introduce additional hardware support for alpha testing [48], but applications may still require anyhit shaders for specific cases.

Both of these shaders execute on GPU compute cores instead of in the RT unit and introduce additional overheads of passing data between GPU cores and the ray tracing accelerator. Anyhit shaders may also require texture accesses, which can further stress the memory system. The specific implementations of these optional shader invocations are not disclosed by any hardware vendors. However, in Vulkan-Sim, the shader invocations are queued and then executed after ray traversal is complete, instead of interleaving them with traversal as they are generated as the ray tracing pipeline diagram in Figure 1 might suggest. Vulkan-Sim also attempts to coalesce calls to the same shader [47].

TABLE 1: LumiBench scenes.

							
WKND	**SHIP**	**BUNNY**	**SPNZA**	**CHSNT**	**BATH**	**REF**	**CRNVL**
0 triangles (procedural)	6.3K triangles	144.1K triangles	262.3K triangles	313.2K triangles	423.6K triangles	448.9K triangles	449.6K triangles
							
FOX	**PARTY**	**SPRNG**	**LANDS**	**FRST**	**PARK**	**CAR**	**ROBOT**
1.6M triangles	1.7M triangles	1.9M triangles	3.3M triangles	4.2M triangles	6.0M triangles	12.7M triangles	20.6M triangles

3.2. Scene Selection

Many scenes used to evaluate ray tracing research are not publicly available because of the artistic IPs involved, making it difficult to reproduce results and compare different techniques. To address this issue, several publicly available scene repositories have been created, including the Utah 3D Animation Repository [15], the NVIDIA ORCA [11] repository, the Blender demo files [2], and the Disney Moana Island Scene [3]. Scenes published by Casual Effects [46] and Benedikt Bitterli [30] are also popular in ray tracing research. We take advantage of these public scenes to create our benchmark suite, which is designed to be a representative sample rather than a compilation of all available scenes. Table 1 shows the scenes we evaluate and Table 2 lists the subset we have selected for LumiBench.

LumiBench includes White Lands (*LANDS*), Red Autumn Forest (*FRST*), Splash Fox (*FOX*), PartyTug (*PARTY*), Spring (*SPRNG*), Procedural (*ROBOT*) and Racing Car (*CAR*) from the Blender demo files [2] for their high primitive counts, which are similar to those found in modern video games. We add *SHIP* [29] for long and thin geometry, then *BATH* [30] and *REF* [12] for indoor scenes with reflections, and we include a few scenes popular in computer graphics research, which are *BUNNY* [46], Sponza (*SPNZA*) [46] and Carnival (*CRNVL*) [41]. We also add Ray Tracing in One Weekend (*WKND*) [12] and Horse Chestnut Tree (*CHSNT*) [46] because they make use of the *intersection* and *anyhit* shaders respectively. Lastly, we created a synthetic Park scene as a mix of publicly available grass [5], trees [13], human characters [7], mountains [10], and a car [8], which has a high primitive count and features long and thin grass.

We also evaluate several game maps from the Counter-Strike: Global Offensive (CS:GO) [53] game, which are not included in our final benchmark suite due to proprietary restrictions but serve as a point of comparison between our benchmark scenes and real-world game scenes. We encourage researchers interested in including these scenes in their evaluation to reach out to the game developers and request permission for research purposes as we have done.

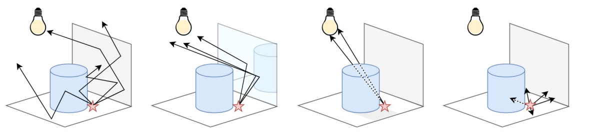

(a) Path Tracing (b) Reflections (c) Shadows (d) Ambient Occlusion

Figure 2: Illustration of different types of rays (redrawn from [32]).

3.3. Shader Selection

Different types of ray tracing shaders also stress the hardware differently. The most common types of ray-traced effects are global illumination, reflections, shadows, and ambient occlusion, described later in this section. Currently, full ray tracing is too computationally expensive for real-time applications even when using specialized hardware accelerators. Thus, most games use a hybrid approach that combines rasterization with layers of ray-traced effects to achieve the desired image quality. Each effect can be implemented separately as shaders in the ray tracing pipeline and enabled or disabled depending on the desired visual quality and performance. Therefore, in LumiBench, we implement each effect individually so that they can be studied independently or as a combined workload.

In real applications, shaders are far more complex than the shaders we use in LumiBench. However, the shaders are still generating rays that fundamentally follow the same pattern as the different ray types we include in Figure 2. Moreover, shader execution happens within the regular GPU cores rather than the ray tracing accelerator, which is different from the focus of LumiBench. By including a selection of different shaders, we can still gain sufficient insights into the ray tracing pipeline and particularly the transitional costs between the GPU cores and the RT unit.

3.3.1. Path Tracing. Path tracing (*PT*, Figure 2.a) is a fully ray-traced rendering technique that begins with *primary rays* from the camera and generates new *secondary rays* in random directions at each intersection point to simulate light bounces. This technique is often used to create the visual effect of global illumination. The rays are traced recursively until they reach a light source, meet the maximum depth, or miss the scene. As a result, these rays are likely to diverge as they bounce in different directions and terminate at different depths, stressing SIMT efficiency in the hardware. The final color of each pixel is determined by the color of the closest intersection point, so each ray cannot terminate until all intersections have been identified.

3.3.2. Reflections. When a *primary ray* intersects with a reflective surface, reflections can be generated by tracing a *secondary ray* in a new direction following the Law of Reflection. Reflection rays (Figure 2.b) are usually more coherent because they follow a similar reflection direction. We do not implement a separate reflections shader in LumiBench since not all scenes are reflective, but reflection rays are generated for any reflective surface in the path tracing shader, such as in the *BATH* scene.

3.3.3. Shadows. Shadows (*SH*, Figure 2.c) are created by shooting a ray from the primary intersection point towards a light source and testing for occlusion. Shadow rays are not always coherent because they may shoot toward different light sources and originate from very different intersection points but they are not random like path tracing rays. Occlusion tests are also more efficient than finding the closest intersection because identifying any intersection is sufficient to determine that a point is in shadow and traversal can terminate.

3.3.4. Ambient Occlusion. Ambient occlusion (*AO*, Figure 2.d) is an effect that models the shadows formed in crevices from ambient lighting. AO rays are short rays that are shot in random directions from the primary intersection point and the number of occluded rays determines the amount of shadowing. These rays also benefit from the efficiency of occlusion tests but are more divergent than shadow rays because they are shot in random directions.

3.4. Diversity Analysis

Our stress scenes can be combined with any of the three shaders, with the exception of *CHSNT* which only supports *PT*, producing 46 unique workloads. We conduct a diversity analysis following the principles of Microarchitecture-Independent Workload Characterization (MICA) [38] to assess the similarity between our workloads and prune to a representative subset.

We begin by identifying a comprehensive set of 35 GPU metrics covering the instruction mix, memory hierarchy behavior, SIMT efficiency, and performance collected by Vulkan-Sim. We then add 29 additional metrics specific to the hardware RT unit, which includes intersection tests executed and memory access behavior during ray traversal. Lastly, we include 23 scene and shader characteristics, such as the number of primitives, type of rays generated, and executed shaders in the ray tracing pipeline. Using a large set of metrics allows us to better capture the behavior of the workloads from different perspectives and identify the most important metrics for workload characterization.

TABLE 2: Selected subset of LumiBench

	Scene	Shader	Stress
SPNZA_AO	SPNZA	Ambient Occlusion	Indoor and enclosed, textures
BUNNY_AO	BUNNY	Ambient Occlusion	Indoor and enclosed
WKND_PT	WKND	Path Tracing	Procedural intersections
SHIP_SH	SHIP	Shadows	Long and thin primitives
ROBOT_SH	ROBOT	Shadows	Large working set
BATH_PT	BATH	Path Tracing	Reflective surfaces, textures
PARK_PT	PARK	Path Tracing	Long and thin primitives
CHSNT_PT	CHSNT	Path Tracing	Anyhit texture alpha masking

Although MICA proposes to use only microarchitecture-independent characteristics, we included all metrics in our analysis to capture both the underlying inherent program behavior as well as the effects of the microarchitecture. We apply Principal Component Analysis (PCA) to our characterization data using the same criteria of data reduction as MICA and plot a dendrogram to visualize the similarity between the workloads in Figure 3. Benchmarks are grouped by horizontal lines and the location of the line on the y-axis represents the similarity between different clusters.

The PCA results help us select a representative subset of the workloads by choosing one workload from each cluster in the dendrogram. In our selection, we prioritize including a diversity of shaders and stress cases, also avoiding any duplicated scenes. Table 2 lists the workloads we have selected and the stress cases they represent.

We find that the PCA analysis is quite successful at clustering ray tracing workloads, clearly separating cases like the only workload with anyhit shaders (*CHSNT_PT*) from all others. However, each cluster often includes multiple stress scenarios, implying that PCA is able to capture a much more comprehensive set of factors that distinguish the workloads. Noticeably, scenes from the CS:GO game are clustered together, indicating they do differ from our test scenes. However, a few workloads in LumiBench fall into the same clusters as CS:GO, which we include in our representative subset in order to capture some real-world game behaviors.

We additionally validate our selection by simulating the workloads using a different hardware configuration that adjusts the number of cores, cache size, intersection latencies, and the number of warps per RT unit. We find the speedups of our subset to be consistent with the full set, matching both minimum and maximum speedups and showing an average with only a 1% difference. Although we have selected eight representative workloads for LumiBench, this subset can be considered a default selection and we encourage researchers to choose different subsets of workloads more specific to their research interests.

Because PCA mixes the metrics through linear combination to reduce dimensionality, it cannot produce meaningful principal components. Thus, we also use the genetic algorithm as described in MICA to identify the eight most representative metrics (Table 3) and plot Kiviat diagrams in Figure 4 to further compare the LumiBench workloads. We also include *DUST2* from CS:GO in this comparison, showing the gaps between LumiBench and real-world game

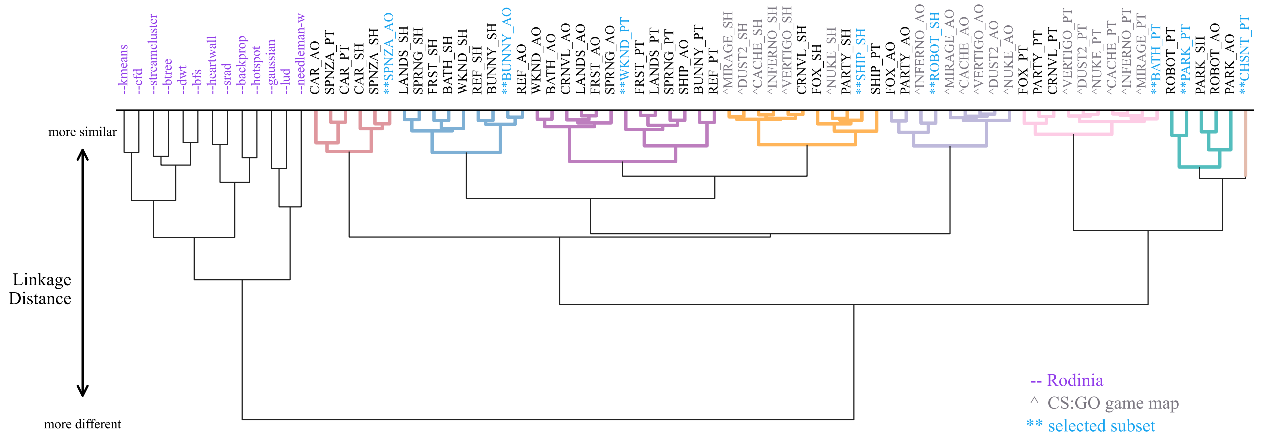


Figure 3: Dendrogram of workload similarity in LumiBench with colors indicating clusters identified by PCA.

TABLE 3: SELECTED SIMILARITY CHARACTERISTICS

	Characteristic	Architecture	RT	Category
1	Avg DRAM row buffer locality	Dependent		Memory
2	Avg DRAM utilization	Dependent		Memory
3	BVH tree depth	Independent	✓	Scene
4	RT memory writes	Dependent	✓	Shader
5	L1D RT read hits	Dependent	✓	Memory
6	% TLAS leaf node accesses	Independent	✓	Scene
7	% BVH node accesses	Independent	✓	Scene
8	Avg RT active cycles	Dependent	✓	Shader

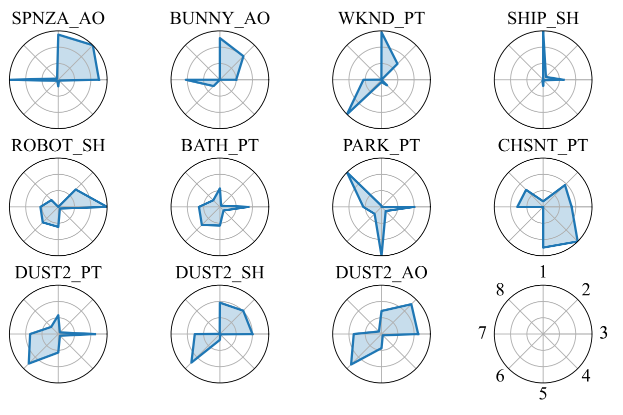


Figure 4: Kiviat diagrams for each workload illustrating the eight most representative characteristics.

scenes. Interestingly, there is a lot of diversity in the selected metrics, which implies that the workloads are not only different in terms of the stress cases but also in terms of the underlying program behavior. The selected metrics include both microarchitecture-independent and microarchitecture-dependent characteristics, which further supports our decision to include all metrics in our analysis.

3.4.1. Comparison to Rodinia. The Rodinia [33] benchmark suite was developed to address the lack of representative workloads for heterogeneous systems with GPUs or other accelerators and aims to cover a diverse set of applications. The suite spans many of the dwarves in the Berkeley Dwarf Taxonomy [24] but still fails to cover any applications with similar behavior to ray tracing. We evaluated 13 Rodinia workloads that can execute on Vulkan-Sim to collect the same set of metrics used for LumiBench, excluding any metrics specific to ray tracing. The result of PCA on the combined set of data clusters all Rodinia workloads together and clearly separates them from the LumiBench even though ray tracing metrics are excluded.

Che et al. [34] identifies *BFS* as the closest matching workload to ray tracing, but our PCA shows it being very different from our workloads. Furthermore, the *raytrace* workload in PARSEC [28] is fundamentally different from the modern ray tracing pipeline and is not representative of real-time applications. Most importantly, LumiBench aims to stress GPUs with a specialized ray tracing accelerator, which is not considered in any previous benchmark suites.

4. Evaluation Methodology

We characterize LumiBench using an updated version of Vulkan-Sim and simulate the workloads using a modified version of the *RayTracingInVulkan* [12] application. We use the *RayTracingInVulkan* application because it is open-source and provides a simple interface for implementing different shaders and loading in OBJ scene files. The hardware configuration we use in Vulkan-Sim is outlined in Table 4, taken from the original Vulkan-Sim work to represent a mobile GPU configuration. We also evaluate LumiBench using the desktop GPU configuration from Vulkan-Sim for comparison but using a mobile configuration is ideal for LumiBench to match the scaled-down nature of the workloads. We expect ray tracing in mobile GPUs to become increasingly popular to support VR/AR devices and as ray tracing technology matures.

LumiBench workloads can also be executed on real GPUs to evaluate performance. However, our workloads are simplified to run on Vulkan-Sim and are not designed

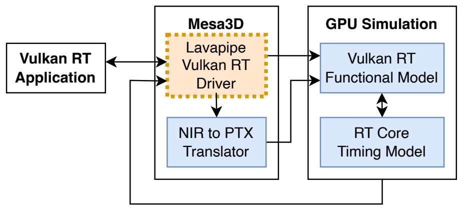


Figure 5: Updated Vulkan-Sim software architecture.

TABLE 4: 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
Compute Core Clock	1365 MHz
Interconnect Clock	1365 MHz
L2 Clock	1365 MHz
Memory Clock	3500 MHz
# RT Units / SM	1
Max Warps / RT Unit	4

to evaluate real hardware because the workloads do not sufficiently saturate desktop GPUs. Existing commercial ray tracing benchmarks like 3DMark [1] are more suitable for this purpose, but they are not open source and cannot provide the detailed architecture insights collected using Vulkan-Sim that are necessary for research.

4.1. Vulkan-Sim Updates

Figure 5 shows the software architecture of the updated Vulkan-Sim. Vulkan-Sim relies on the Mesa3D [9] graphics driver to implement the Vulkan API, managing the necessary resources and compiling the shaders to the NIR intermediate representation. These shaders are then translated to PTX instructions and the Mesa3D driver launches shader execution on the modified GPGPU-Sim [40] simulator. However, Vulkan-Sim is constrained by the requirement of an Intel CPU with integrated graphics to execute any workloads beyond the few provided traces. In order to simulate the LumiBench workloads, we integrate with *Lavapipe* [6], which provides a software Vulkan implementation that can execute without a physical GPU and removes the Intel dependency from Vulkan-Sim.

Lavapipe is an open-source project maintained as part of Gallium [4] in Mesa3D and works as a frontend to connect Vulkan applications to our GPU simulator. However, Lavapipe is designed as a software-only rasterizer and does not support the ray tracing pipeline. We implement all the necessary Vulkan API support for *RayTracingInVulkan*, enabling Vulkan-Sim×Lavapipe to match Vulkan-Sim×Intel. We also add *anyhit* shader support that was previously missing. The change in Vulkan-Sim to support Lavapipe is illustrated by the orange block in Figure 5.

4.2. Types of Workload

In our evaluation, we focus on real-time applications, choosing to simulate a single frame of each workload. However, complex workloads can also be represented by LumiBench by adjusting parameters such as image resolution, number of samples per pixel, or maximum ray depth. Current real-time applications usually cannot support more than 1-2 samples per pixel, but improving performance through new optimizations may allow for higher sample counts in the future and researchers are encouraged to increase this value to evaluate new ideas.

Also, as new algorithms are introduced, shaders may be modified. LumiBench can be used to evaluate the performance of these new shaders by simply replacing the original shader with the new one and running the same scenes. Fundamentally, the shaders in LumiBench already cover a wide range of ray types and new techniques are likely to be similar or a combination of existing ray types. For example, the behavior of the Dynamic Diffuse Global Illumination (DDGI) [45] technique can be modeled by *SH* as the underlying ray types are well-matched.

4.3. Representative Sampling

Simulating the scenes and shader combinations in LumiBench at a high screen resolution is ideal but too computationally expensive and time consuming. Although modern games are designed to run at a resolution of 1080p or higher, we choose to simulate the workloads at a lower resolution of 128 $\times$ 128 pixels with two samples per pixel. Despite the unrealistic nature of the reduced resolution, the impact on the ray tracing hardware is still representative of real workloads because graphics rendering is usually completed in sequential tiles of parallel rays [18].

We use AerialVision [22] to visualize the dynamic architectural behavior in the simulated GPU over time for *PARK_PT*, *BUNNY_AO*, and *SHIP_SH*, and include an overview for a few metrics in Figure 6. The top row of Figure 6 shows the number of active warps per RT unit reaches the max, indicating our resolution is still high enough to saturate the ray tracing hardware and can be treated as the equivalent of sampling a few tiles of a higher resolution image.

The issue of sampling for architectural simulation to maintain a reasonable simulation time is common across all domains. The *Principal Kernel Projection* approach shows that a smaller sample can still be representative when the rolling average and standard deviation of the metric of interest is stable [25]. Figure 6 also demonstrates that performance metrics like the number of instructions per cycle (IPC) and L1 data cache (L1D) miss rates stabilize for our benchmarks despite initial fluctuations, and behavior of larger resolutions can be projected from the smaller sample.

Figure 6 also includes *SHIP_SH* at 1080p resolution, which confirms that key performance metrics follow the same trends as the lower resolution. However, the L1D miss rate is higher at 1080p because additional rays in the larger resolution bounce in more directions and access more parts

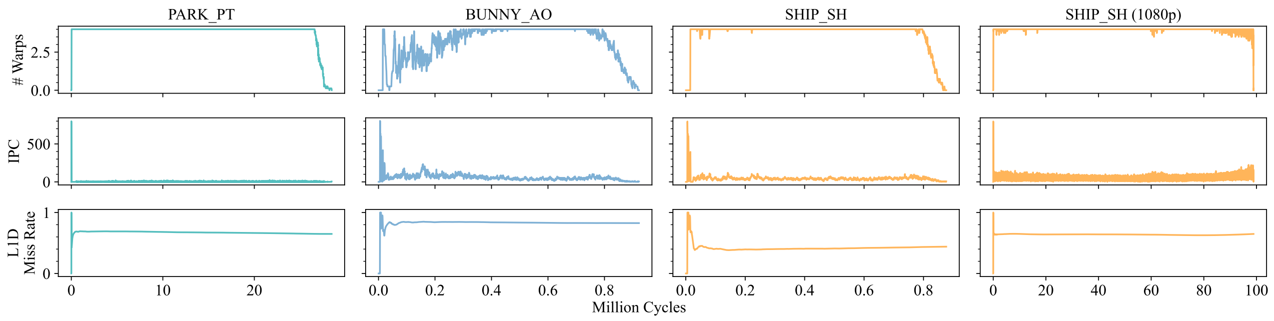


Figure 6: Overview of the number of warps per RT unit, IPC, and L1D miss rate over time for select workloads.

of the scene, which increases the working set size and strains the cache. Although some workload behaviors do differ slightly between resolutions, we believe LumiBench can still adequately evaluate ray tracing hardware designs because of the diverse workloads included. The specific simulation details are generally less significant compared to the overall trends of the program, which is highlighted by a study that compares the accuracy of simulating with different intermediate representations and concluded with nearly the same correlation to real hardware [36].

5. Characterization Results

We use Vulkan-Sim to characterize LumiBench workloads from several perspectives, including scene structure, shader composition, and memory system behavior, all of which appear in our top eight similarity metrics. Each of these aspects is important to consider when applying optimizations to the ray tracing pipeline and designing future hardware.

5.1. Workload Structure

5.1.1. Scene Geometry. Scene structure is important to understand the underlying algorithmic behavior of the workloads. Figure 7 (top) shows the breakdown of geometry in each scene and the resulting BVH structure. The scenes are sorted by the number of triangles they include, which may seem to represent the complexity of the scene, but the ability in ray tracing to create instances of the same geometry makes evaluation results more complex. For example, although *PARTY* has relatively few triangles compared to other workloads, it has a large number of instances, resulting in a more complex scene.

Moreover, Figure 7 (bottom) shows the BVH structure depth and path tracing shader execution time for each scene. Even though *FRST* uses more triangles, the overall BVH structure is not much deeper than *SPRNG*, implying the average length of ray traversals may not be much longer. In contrast, the minimum length of a ray traversal to hit geometry in *CRNVL* is nearly as long as *CAR* despite having significantly less geometry. The overall execution time is not directly correlated to any of these factors individually.

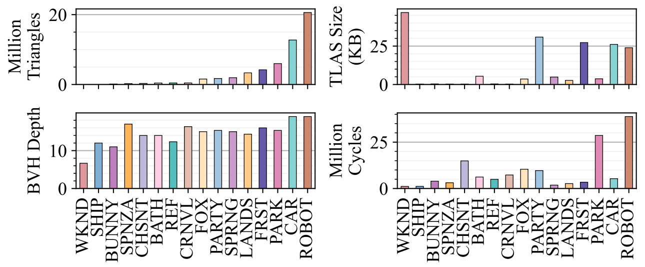


Figure 7: Breakdown of BLAS and TLAS structures, BVH depth, and path tracing execution time for each scene.

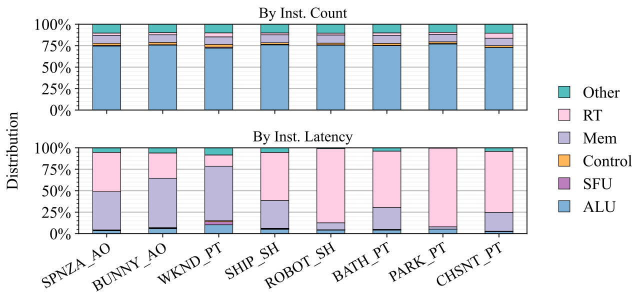


Figure 8: Instruction type distribution by dynamic instruction count versus simulated latency for each scene.

5.1.2. Instruction Mix. Figure 8 shows the distribution of instruction types in LumiBench shaders by dynamic instruction count (top) and simulated latency (bottom) for each scene. Our shaders are primarily dominated by ALU instructions with only a few `traceRay` instructions per shader, which is reflected in the instruction count distribution. However, `traceRay` is a very expensive instruction covering the entire ray traversal algorithm, causing RT type instructions to dominate the instruction latency distribution.

The other dominating factor in latency is the Mem type load and store instructions. In complex workloads like *PARK_PT*, the ray traversal is the primary bottleneck, but in simpler workloads, memory accesses by the shader become the bottleneck. This effect is even more evident in *WKND_PT* where the ray traversal is very short because the scene is entirely composed of procedural geometry,

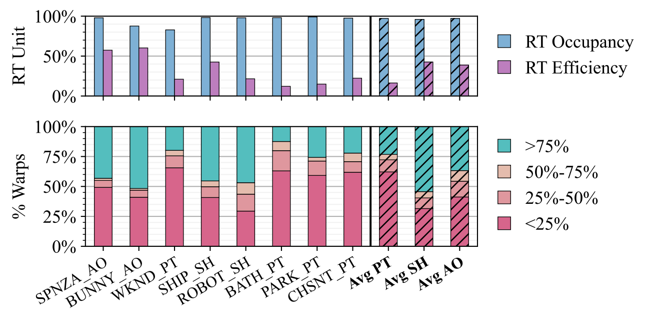


Figure 9: Warp occupancy and efficiency for RT unit (top) and SIMT efficiency (bottom) for each workload, with averages for each shader type.

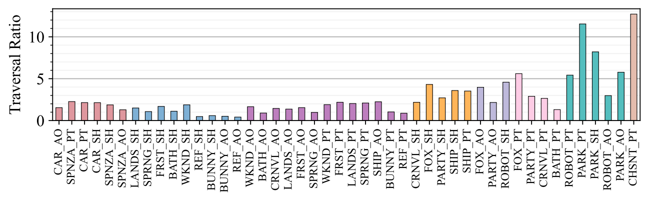


Figure 10: Ratio of BVH depth to average traversal length for each scene.

so the majority of the execution time is spent on memory accesses in the shader to fetch all the necessary information required to compute an intersection.

5.2. Hardware Efficiency

5.2.1. RT Unit Efficiency. The top of Figure 9 shows the average occupancy and efficiency of the RT unit for each shader type, defined as the average number of active warps and the average number of active rays per warp in the RT unit respectively. The average occupancy is high across most workloads, which deceptively appears like the RT unit is well utilized. However, when considering the efficiency, it indicates ray tracing suffers greatly from load imbalance.

The average efficiency is especially low for *PT* workloads, caused by the increasingly divergent rays that bounce to various extents before being terminated. There is also a problem with straggler rays that take significantly longer to execute in *PT* workloads and the warps cannot continue until all rays in the warp have finished. The efficiency of *SH* and *AO* workloads is much better because the rays are terminated after the first hit, with *SH* workloads being the highest because *SH* rays are more coherent and shoot in similar directions. *BUNNY_AO* shows exceptionally high RT unit efficiency because AO rays terminate quickly and the simple scene does not provide any opportunities for straggler rays with lengthy traversals. In contrast, *PARK_PT* creates many straggler rays because the BVH structure cannot efficiently bound the long and thin geometry, causing the RT unit to be very underutilized. Naively increasing the RT unit size would not improve performance.

5.2.2. SIMT Efficiency. On the bottom of Figure 9, we plot the SIMT efficiency for each shader type, which shows very similar trends as the RT unit efficiency. SIMT efficiency is measured independently of the RT unit and shows that divergent threads and load imbalance are not exclusive to the `traceRay` instruction, but persist through the entire program. Optimizations targeting the RT unit alone will be insufficient to improve the performance of these ray tracing workloads, and a more holistic approach considering the entire ray tracing pipeline is necessary.

5.2.3. Traversal Ratio. Another consideration is the ratio of the average number of nodes traversed per ray to the depth of the BVH structure in each workload, shown in Figure 10. A high *traversal ratio* implies the BVH structure is not pruning the search space for intersections effectively, such as in *CHSNT_PT* which requires the anyhit shader to confirm intersections. On the other hand, low ratios like in *BATH_AO* can indicate a well-structured BVH, but can also be caused by empty space, large geometric features, or a camera position where rays miss the scene entirely. Shader implementation details, such as choosing to search for any intersection rather than the closest intersection, are also reflected in the traversal ratio. Simply comparing the average number of nodes traversed per ray does not provide all the necessary insights into the bottlenecks, but the traversal ratio is a much better indicator of performance.

5.3. Memory Behavior

5.3.1. Cache Accesses. Ray tracing is commonly considered to be a memory latency-bound application, implying that optimizing the cache is an excellent approach to improving performance. Figure 11 shows the distribution of all L1D accesses for each scene and breaks down `traceRay`-related accesses versus other shader accesses. The average cache miss rate for `traceRay` accesses in the L1D is 50%, and surprisingly only increases to 66% for relatively large scenes like *PARK_PT*, with the L2 cache exhibiting similar behavior. Divergent memory access patterns observed in *AO* ray traversals are likely related to the higher cache miss rate in *BUNNY_AO*, but misses are primarily incurred by the shader implying ray traversal is relatively simple compared to shading. In *PARK_PT*, the opposite is true, with the majority of cache misses being incurred by ray traversal due to a far more complex scene structure. In general, compulsory cold misses are a small part of the total cache misses, confirming that the scenes do not fully fit in the cache and thrashing occurs from the random memory access pattern of the ray traversal.

5.3.2. DRAM. From the DRAM perspective in Figure 12, the bandwidth utilization and efficiency of the different workloads in LumiBench differ greatly. DRAM efficiency, defined as the cycles with data transfers relative to cycles with DRAM requests at the memory access controller, ranges from 48% to 88% for a mobile GPU configuration. Low DRAM efficiency is likely due to the poor memory access patterns observed with ray traversals, particularly in

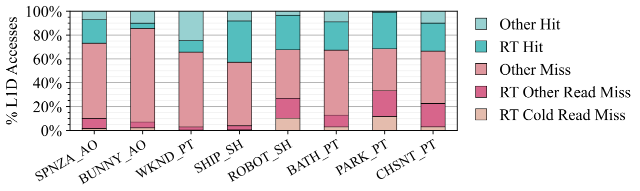


Figure 11: Distribution of L1D cache accesses for each scene.

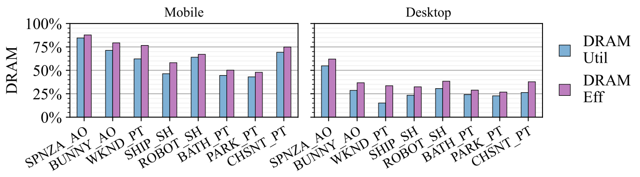


Figure 12: DRAM utilization and efficiency of each workload.

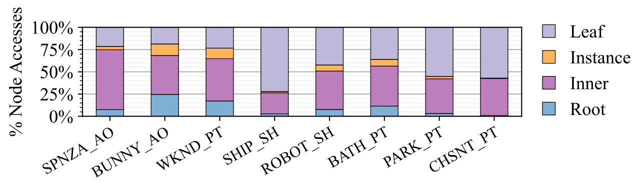


Figure 13: Distribution of data types fetched to the RT unit for each scene.

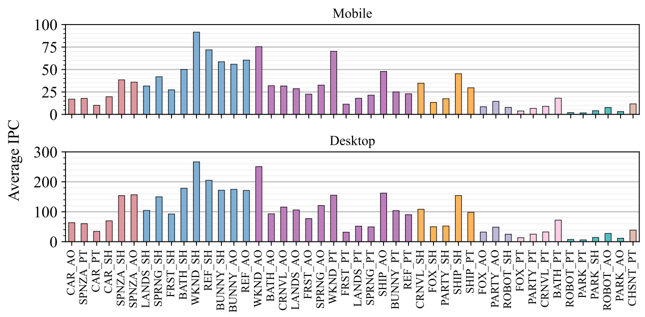


Figure 14: Average IPC for each workload.

complex scenes like *PARK_PT*, which leads to long latency data fetches. Design considerations to improve ray tracing performance for complex scenes should focus on latency much more than bandwidth concerns since the bandwidth cannot be efficiently utilized. A simple experiment with *PARTY_PT*, which has a very low DRAM efficiency of 37%, shows that changing DRAM bandwidth has a minimal impact on performance because memory is primarily latency-bound. Also, compared to DRAM utilization, defined as the percentage of cycles where data was transferred relative to total program cycles, the average difference is only around 7%. This small difference implies memory requests to DRAM are nearly constant and dominate the program, making DRAM an important optimization target.

For a desktop GPU configuration, both the DRAM utilization and efficiency are significantly lower because memory accesses are latency-bound and cannot fully utilize the increased bandwidth. However, the trends are similar, with *PARK_PT* still exhibiting the lowest DRAM efficiency and *SPNZA_AO* the highest. These trends also highlight that although *ROBOT* has the most geometry, it is not the most complex because only a portion of the geometry is accessed during traversal, resulting in its low DRAM utilization.

5.3.3. Data Mix. Figure 13 shows the distribution of data types fetched by the RT unit for each workload. While mostly similar, *SHIP_SH* and *PARK_PT*, which stress long and thin geometry, have a much higher proportion of leaf nodes fetched. This behavior is likely caused by inefficient bounding boxes that include a large amount of empty space. Optimizations focused on improving the BVH structure or the traversal ratio would be beneficial for these workloads.

5.4. System Behavior

Figure 14 plots the average IPC for each workload, showing the cumulative effect of the different factors on the system. IPC also matches well with our PCA results because the metric is comprehensive and captures the overall performance of the system. The plot highlights particularly difficult workloads that provide the best optimization opportunities because ray tracing performance is bound by the worst-case scenario.

Evaluating the performance of LumiBench workloads on the desktop GPU configuration shows similar trends to the mobile GPU configuration, also illustrated in Figure 14. The desktop GPU reports higher IPCs as a result of the increased resources.

5.4.1. Simulation Time. In our characterization study, simulation time ranged from around 15 minutes to under 13 hours per workload, with an average of 1.5 hours. We simulate LumiBench at 128×128 pixels to maintain a reasonable simulation time because large resolutions significantly increase computational effort. For example, simulating a single frame of *SHIP_SH* at 128×128 resolution requires only 30 minutes, versus 36 hours at 1080p resolution when evaluating multiple workloads and hardware configurations. Complex workloads like *PARK_PT* can require more than a week to simulate at 1080p.

5.5. Analytical Modeling

Analytical studies are especially useful for extrapolating performance results to higher resolutions and larger scenes that are infeasible to simulate. We evaluate the analytical model described in Hong and Kim [37]. They represent the performance of a GPU kernel with two metrics: Memory Warp Parallelism (MWP) and Computation Warp Parallelism (CWP). MWP is the maximum number of memory requests that can be issued in parallel by an SM, and CWP is the number of warps that an SM can execute during a memory warp waiting period. A limitation of this model is the lack of any cache effects due to high cache hit rates for the instruction, texture, and constant caches exhibited by

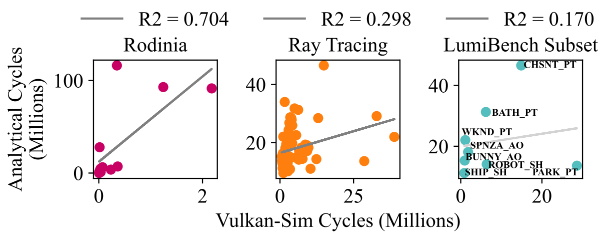


Figure 15: Comparison for analytical model

GPGPU applications, as well as the lack of a cache for global memory in the modeled G80 GPU architecture. We instead multiply the DRAM latency by the L1 cache miss rate to estimate the average memory latency.

Existing analytical models do not fit ray tracing workloads, as shown in Figure 15. The R^2 value for linear regression over our data is 0.704 for Rodinia workloads, but only 0.298 for ray tracing workloads and lower when applied to the LumiBench subset.

6. Related Works

Previous works have primarily focused on evaluating techniques and optimizations for the ray tracing algorithm itself, rather than the systematic evaluation of ray tracing hardware. Industry benchmarks that do evaluate graphics hardware do not focus on ray tracing. This section outlines the related works that are most relevant to LumiBench.

6.1. Graphics Benchmarks

Benchmark suites used to evaluate computer graphics techniques are generally a set of scenes that stress different aspects of the rendering algorithms. BART [43] was the first benchmark suite developed to evaluate ray tracing workloads and the most similar to LumiBench. However, BART was still designed to evaluate algorithms rather than hardware, and the proposed scenes are now outdated compared to modern applications. Other ray tracing benchmark scenes have been proposed since but focus on stressing specific algorithm features, such as TauBench [21] which stresses temporal reuse techniques for ray tracing.

Several industry benchmarks exist for evaluating graphics hardware, such as 3DMark [1] and SPECviewperf [14]. These benchmarks are proprietary and designed to evaluate commodity graphics hardware, which is not suitable for use with academic simulators in the research community. Many of the scenes and applications in these benchmarks are far too complex to be simulated in a reasonable amount of time. The open-source Quake II RTX game offers an excellent alternative to industry benchmarks for graphics researchers, but the application is also too complex for Vulkan-Sim.

6.2. Ray Tracing Characterization

Several works also characterize ray tracing workloads, providing insights into performance bottlenecks and opportunities for optimization. Early works from Aila et al. [19] evaluate the performance of a software ray tracing implementation on a GPU and identify the memory bandwidth as the main bottleneck, but predate current ray tracing accelerators. More recently, Vasiou et al. [55] use SimTRaX to evaluate a set of four scenes from the McGuire Computer Graphics Archive [46] selected for their variation in geometric complexities only. Their study mostly focuses on the effects of a growing number of ray bounces on the memory system, however, the high numbers of ray bounces Vasiou et al. evaluate are not representative of real-time applications. Also, SimTRaX is designed to simulate the specialized TRaX [52] accelerator, which does not integrate with the GPU and therefore does not operate using the industry-standard ray tracing pipeline.

Pankratz et al. [49] introduced Vulkan-Vision, using the tool to evaluate the performance of ray tracing workloads on real hardware. Although Vulkan-Vision identifies bottlenecks in the ray tracing pipeline, the tool requires applications to execute on real hardware so it cannot be used to evaluate hardware optimizations. LumiBench provides the first publicly available benchmark suite for ray tracing hardware used with a cycle-level simulator.

7. Conclusion

In conclusion, we present LumiBench as a benchmark suite for evaluating ray tracing hardware performance in modern GPUs. LumiBench is designed to be representative of real-world ray tracing workloads while still being able to complete simulation using the Vulkan-Sim GPU simulator in a reasonable amount of time. We characterize the workloads in LumiBench using Vulkan-Sim and provide insights for architectural research. Our updates to Vulkan-Sim are available at *https://github.com/ubc-aamodt-group/vulkan-sim* and our modified version of the RayTracingInVulkan application is available at *https://github.com/ubc-aamodt-group/RayTracingInVulkan* on GitHub. Detailed instructions to run LumiBench are included in the Appendix.

We expect LumiBench to be a first step towards a standardized benchmark suite for hardware ray tracing, however, there are still many opportunities for future work. We plan to continue adding more scenes and shaders to LumiBench to increase the diversity of the workloads and target different aspects of the ray tracing hardware, which requires additional improvements to Vulkan-Sim. Furthermore, we hope to add support for animations and dynamic scenes to better study temporal effects and the performance of the ray tracing pipeline over time.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We also wish to thank all the developers of the open-source scenes and projects that made LumiBench possible. This research is funded in part by grants from Huawei Technologies and the Natural Sciences and Engineering Research Council of Canada (NSERC).

Appendix

1. Abstract

The included artifact provides the source code for the updated version of Vulkan-Sim, which uses the Lavapipe driver. We also provide our modified version of the RayTracingInVulkan application and all the benchmark scenes and shaders used in LumiBench. We describe the installation procedure and workflow to reproduce our results in Figure 14 and package the artifact as a Docker container.

2. Artifact check-list (meta-information)

- **Program:** Vulkan-Sim simulator and RayTracingInVulkan application
- **Compilation:** GCC/G++, CMake, CUDA, Ninja, Meson
- **Binary:** source code and compiled binary both provided
- **Model:** OBJ models included
- **Run-time environment:** tested on Ubuntu 20.04
- **Hardware:** >12 GB RAM
- **Metrics:** execution time, cache access breakdown, ray tracing application statistics
- **Output:** simulator metrics and PPM images
- **How much disk space required (approximately)?:** 18GB uncompressed Docker image or 8GB from source (including simulator and benchmark scenes)
- **How much time is needed to prepare workflow (approximately)?:** 1-2 hours
- **How much time is needed to complete experiments (approximately)?:** 3-5 days if benchmarks are executed sequentially
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Data licenses (if publicly available)?:** CC BY-NC-SA
- **Archived (provide DOI)?:** Zenodo: *https://doi.org/10.5281/zenodo.8267898*

3. Description

3.1. How to access. We package both the modified version of Vulkan-Sim as well as all of the scenes and shaders for LumiBench in a Docker container, which is available from *https://doi.org/10.5281/zenodo.8267898* on Zenodo. Alternatively, our updates are available from *https://github.com/ubc-aamodt-group/vulkan-sim* for Vulkan-Sim and *https://github.com/ubc-aamodt-group/RayTracingInVulkan* for LumiBench on GitHub.

3.2. Hardware dependencies. There are no specific hardware dependencies for this project. However, the simulation of larger models or resolutions can range from hours to several days and require more RAM. For example, the PARK scene requires around 12GB of RAM to simulate in our experiments.

3.3. Software dependencies. The provided Docker container includes all of the software dependencies for Vulkan-Sim and the benchmark scenes and shaders.

For users who wish to build the simulator from source, the following dependencies are required:

- Vulkan SDK with ray tracing
- CUDA Toolkit 10
- Embree3
- Mesa
- All original dependencies of Vulkan-Sim
- All original dependencies of RayTracingInVulkan
- `matplotlib`, `numpy`, `pandas`, `scikit-learn` and `scipy` Python packages are required to produce the figures in this paper

3.4. Models. We redistribute the benchmark scenes used in LumiBench in the Docker container, which includes the following models:

- White Lands (LANDS)
- Red Autumn Forest (FRST)
- Splash Fox (FOX)
- PartyTug (PARTY)
- Spring (SPRNG)
- Procedural (ROBOT)
- Racing Car (CAR)
- Ship (SHIP)
- Bathroom (BATH)
- Reflective Cornell Box (REF)
- Bunny (BUNNY)
- Crytek Sponza (SPNZA)
- Carnival (CRNVL)
- Ray Tracing In One Weekend (WKND)
- Horse Chestnut Tree (CHSNT)
- Park (PARK)

4. Installation

We provide a Docker container for users to reproduce our results, which is available from *https://doi.org/10.5281/zenodo.8267898* on Zenodo. After downloading the container, users can run the following command to start the container:

```
docker load < lumibench.tar.gz
docker run -it vulkansim:lavapipe /bin/bash
```

Vulkan-Sim and RayTracingInVulkan are both installed and precompiled in the container, ready for use. Alternatively, users can build the simulator from source by following the instructions below.

```
cd /home/vulkansim/mesa
meson --prefix="${PWD}/lib" build -Dvulkan-
    drivers=swrast -Dgallium-drivers=swrast -
    Dplatforms=x11 -D b_lundef=false -D
    buildtype=debug
cd build
ninja -C build/ install
```

The first compilation of Mesa generates files necessary for Vulkan-Sim, which must be compiled before Mesa will compile successfully.

```
cd /home/vulkansim/gpgpu-sim_emerald
source setup_environment
make -j
cd /home/vulkansim/mesa
ninja -C build/ install
```

The benchmark application can be compiled by following the instructions below.

```
cd /home/vulkansim/RayTracingInVulkan
./vcpkg_linux.sh
./build_linux.sh
```

5. Experiment workflow

After starting the Docker container, navigate to the RayTracingInVulkan executable directory and use the following Python script to run the benchmark:

```
cd /home/vulkansim/RayTracingInVulkan/build/
    linux/bin
python3 run_benchmark.py
```

A message of `Simulation complete!` will be printed to the console when the benchmark is complete and simulator log files for each workload will be saved in the `./logs` directory. In addition, PPM images corresponding to the simulation timestamp of each workload will also be saved in the same directory. We provide additional Python scripts to parse the simulator logs once the benchmark is complete, generate a CSV file containing a table of metrics for each workload and reproduce Figure 14. In order to generate the figure, some additional Python packages are required.

```
pip3 install matplotlib numpy pandas scikit-
    learn scipy
python3 generate_results.py
```

A second included script will produce a dendrogram similar to Figure 3, but the figure will be different because it does not include CS:GO scenes or Rodinia workloads. Run this script after generating the CSV results to create a dendrogram.

```
python3 plot_dendrogram.py
```

6. Evaluation and expected results

Successfully executing the Python scripts will produce a CSV file containing a table of metrics for each workload and Figure 14. If insufficient memory is available in the Docker container, some of the larger workloads may fail to execute.

7. Experiment customization

Experiments can be customized by modifying the `gpgpusim.config` file included in the executable directory. This file describes the GPU configurations as supported by Vulkan-Sim. For example, the max number of warps per ray tracing unit can be modified by changing the `gpgpu_rt_max_warps` parameter.

Additionally, workloads can be executed individually by modifying the `run_benchmark.py` script or by executing the workload directly.

```
./RayTracer --scene <scene number> --width <
    width> --height <height> --samples <
    samples per pixel>
```

For shadow shaders, add `--shader-type <type number> --shadow-rays <number of rays>` to the command line arguments, with type 1 for direct shadows and 2 for ambient occlusion. The anyhit shader can be invoked with `--shader-type 5`.

8. Notes

Scenes from CS:GO are not included in our benchmark suite and thus also not included in this artifact because they are not publicly available. However, we encourage researchers to reach out to Valve for permission to include these scenes for their own research.

9. Methodology

Submission, reviewing and badging methodology:

- *https://www.acm.org/publications/policies/artifact-review-badging*
- *http://cTuning.org/ae/submission-20201122.html*
- *http://cTuning.org/ae/reviewing-20201122.html*

References

[1] 3DMark. [Online]. Available: https://www.3dmark.com

[2] Blender demo files. [Online]. Available: https://www.blender.org/download/demo-files/

[3] Disney Moana Island Scene. [Online]. Available: https://disneyanimation.com/resources/moana-island-scene/

[4] Gallium. [Online]. Available: https://docs.mesa3d.org/gallium/index.html

[5] "Grass 3D Model." [Online]. Available: https://free3d.com/3d-model/grass-74284.html

[6] "Lavapipe." [Online]. Available: https://gitlab.freedesktop.org/mesa/mesa/-/tree/main/src/gallium/frontends/lavapipe

[7] "Male Base Mesh 3D Model." [Online]. Available: https://free3d.com/3d-model/male-base-mesh-6682.html

[8] "Mercedes Benz GLS 580 2020 3D Model." [Online]. Available: https://free3d.com/3d-model/mercedes-benz-gls-580-2020-83444.html

[9] Mesa 3d. [Online]. Available: https://www.mesa3d.org/

[10] "Mountain 3D Model." [Online]. Available: https://free3d.com/3d-model/mountain-183041.html

[11] ORCA: Open Research Content Archive. [Online]. Available: https://developer.nvidia.com/orca

[12] Ray Tracing In Vulkan. [Online]. Available: https://github.com/GPSnoopy/RayTracingInVulkan

[13] "Realistic Trees Scene 3D Model." [Online]. Available: https://free3d.com/3d-model/realistic-tree-pack-3-trees-95419.html

[14] SPECviewperf. [Online]. Available: https://gwpg.spec.org/bm-specviewperf/

[15] Utah 3D Animation Repository. [Online]. Available: http://www.sci.utah.edu/~wald/animrep/index.html

[16] Vulkan. [Online]. Available: https://www.vulkan.org/

[17] Advanced Micro Devices Inc. (2022) AMD RDNA architecture. [Online]. Available: https://www.amd.com/en/technologies/rdna

[18] A. T. Áfra, C. Benthin, I. Wald, and J. Munkberg, "Local shading coherence extraction for SIMD-efficient path tracing on CPUs." in *High Performance Graphics*, 2016, pp. 119–128.

[19] T. Aila and T. Karras, “Architecture considerations for tracing incoherent rays,” in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2010, pp. 113–122.

[20] T. Aila and S. Laine, “Understanding the efficiency of ray traversal on GPUs,” in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2009, pp. 145–149.

[21] J. Alanko, M. Mäkitalo, and P. Jääskeläinen, “TauBench: Dynamic benchmark for graphics rendering,” 2022.

[22] A. Ariel, W. W. L. Fung, A. E. Turner, and T. M. Aamodt, “Visualizing complex dynamics in many-core accelerator architectures,” in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2010, pp. 164–174.

[23] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, “TEAPOT: A toolset for evaluating performance, power and image quality on mobile graphics systems,” in *Proc. ACM Conf. on Supercomputing (ICS)*, 2013, pp. 37–46.

[24] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, “The landscape of parallel computing research: A view from Berkeley,” 2006.

[25] C. Avalos Baddouh, M. Khairy, R. N. Green, M. Payer, and T. G. Rogers, “Principal kernel analysis: A tractable methodology to simulate scaled GPU workloads,” in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 724–737.

[26] K. Beets, “Rays Your Game: Introduction to the PowerVR Photon Architecture,” 2021. [Online]. Available: https://imaginationtech.com/products/gpu/graphics-architecture/powervr-photon/

[27] C. Benthin, S. Woop, I. Wald, and A. T. Áfra, “Improved two-level BVHs using partial re-braiding,” in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2017, pp. 1–8.

[28] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The PARSEC benchmark suite: Characterization and architectural implications,” in *Proc. IEEE/ACM Conf. on Par. Arch. and Comp. Tech. (PACT)*, 2008, pp. 72–81.

[29] J. Bikker, “Real-time ray tracing through the eyes of a game developer,” in *Symp. on Interactive Ray Tracing*, 2007.

[30] B. Bitterli, “Rendering resources,” 2016, https://benedikt-bitterli.me/resources/.

[31] J. Burgess, “RTX on—the NVIDIA Turing GPU,” *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020.

[32] A. Burnes. (2019) Ray tracing, your questions answered: Types of ray tracing, performance on GeForce GPUs, and more. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/geforce-gtx-dxr-ray-tracing-available-now/

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2009, pp. 44–54.

[34] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, “A characterization of the Rodinia benchmark suite with comparison to contemporary CMP workloads,” in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2010.

[35] A. A. Gubran and T. M. Aamodt, “Emerald: Graphics modeling for SoC systems,” in *Proc. IEEE/ACM Int’l Symp. on Computer Architecture (ISCA)*, 2019, pp. 169–182.

[36] A. Gutierrez, B. M. Beckmann, A. Dutu, J. Gross, M. LeBeane, J. Kalamatianos, O. Kayiran, M. Poremba, B. Potter, S. Puthoor *et al.*, “Lost in abstraction: Pitfalls of analyzing GPUs at the intermediate language level,” in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2018, pp. 608–619.

[37] S. Hong and H. Kim, “An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness,” in *Proc. IEEE/ACM Int’l Symp. on Computer Architecture (ISCA)*, 2009.

[38] K. Hoste and L. Eeckhout, “Microarchitecture-independent workload characterization,” *IEEE Micro*, vol. 27, no. 3, pp. 63–72, 2007.

[39] Intel Corporation. (2022) Introduction to the Xe-HPG architecture. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-the-xe-hpg-architecture.html

[40] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, “Accel-Sim: An extensible simulation framework for validated GPU modeling,” in *Proc. IEEE/ACM Int’l Symp. on Computer Architecture (ISCA)*, 2020, pp. 473–486.

[41] D. Konieczka, “3D render lighting challenges,” https://www.3drender.com/challenges/.

[42] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, “Mediabench: A tool for evaluating and synthesizing multimedia and communications systems,” in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 1997, pp. 330–335.

[43] J. Lext, U. Assarsson, and T. Moller, “A benchmark for animated ray tracing,” *IEEE Computer Graphics and Applications*, vol. 21, no. 2, pp. 22–31, 2001.

[44] L. Liu, W. Chang, F. Demoullin, Y. H. Chou, M. Saed, D. Pankratz, T. Nowicki, and T. M. Aamodt, “Intersection prediction for accelerated GPU ray tracing,” in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 709–723.

[45] Z. Majercik, J.-P. Guertin, D. Nowrouzezahrai, and M. McGuire, “Dynamic diffuse global illumination with ray-traced irradiance fields,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 8, no. 2, 2019.

[46] M. McGuire, “Computer graphics archive,” 2017, https://casual-effects.com/data.

[47] T. B. Nowicki and A. M. E. M. Eltantawy, “Methods and apparatuses for coalescing function calls for ray-tracing,” US Patent 17 008 437.

[48] Nvidia Corporation. (2022) Nvidia Ada GPU architecture. [Online]. Available: https://images.nvidia.com/aem-dam/Solutions/geforce/ada/nvidia-ada-gpu-architecture.pdf

[49] D. Pankratz, T. Nowicki, A. Eltantawy, and J. N. Amaral, “Vulkan Vision: Ray tracing workload characterization using automatic graphics instrumentation,” in *Proc. IEEE/ACM Symp. on Code Generation and Optimization (CGO)*, 2021, pp. 137–149.

[50] M. Saed, Y. H. Chou, L. Liu, T. Nowicki, and T. M. Aamodt, “Vulkan-Sim: A GPU architecture simulator for ray tracing,” in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2022, pp. 263–281.

[51] P. Shirley, “Ray tracing in one weekend,” *Amazon Digital Services LLC*, vol. 1, p. 4, 2018.

[52] J. Spjut, A. Kensler, D. Kopta, and E. Brunvand, “TRaX: A multicore hardware architecture for real-time ray tracing,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 28, no. 12, pp. 1802–1815, 2009.

[53] Valve, “Counter-Strike: Global Offensive,” Video Game, 2012, accessed on June 9, 2023, on Microsoft Windows. [Online]. Available: https://store.steampowered.com/app/730/CounterStrike_Global_Offensive/

[54] E. Vasiou, K. Shkurko, E. Brunvand, and C. Yuksel, “Mach-RT: A many chip architecture for high performance ray tracing,” vol. 28, no. 3, pp. 1585–1596, 2020.

[55] E. Vasiou, K. Shkurko, I. Mallett, E. Brunvand, and C. Yuksel, “A detailed study of ray tracing performance: render time and energy cost,” *The Visual Computer*, vol. 34, pp. 875–885, 2018.

[56] C. Yuksel, “Ray tracing in video games,” 2023. [Online]. Available: https://blog.siggraph.org/2023/05/siggraph-spotlight-episode-65-tracing-the-future-exploring-the-revolutionary-technology-of-ray-tracing.html/