Vulkan-Sim: A GPU Architecture Simulator for Ray Tracing

Mohammadreza Saed*, Yuan Hsi Chou*, Lufei Liu*, Tyler Nowicki†, and Tor M. Aamodt*

*University of British Columbia, Canada,
†Huawei Technologies, Canada,
mrsaed@ece.ubc.ca, yuanhsi@ece.ubc.ca, liulufei@student.ubc.ca, tyler.bryce.nowicki@huawei.com, aamodt@ece.ubc.ca

***Abstract*—Ray tracing can generate photorealistic images with more convincing visual effects compared to rasterization. Recent hardware advances have enabled ray tracing to be applied in real-time. Current GPUs feature a dedicated ray tracing acceleration unit, and game developers have started to make use of ray tracing APIs to bring more realistic graphics to their players. Industry cooperatively contributed to Vulkan, which recently introduced an open-standard API for ray tracing. However, little has been disclosed about the mapping of this API to hardware. In this paper, we introduce Vulkan-Sim, a detailed cycle-level simulator for enabling architecture research for ray tracing. We extend GPGPU-Sim, integrating it with Mesa, an open-source graphics library to support the Vulkan API, and add dedicated ray traversal and intersection units. We also demonstrate an explicit mapping of the Vulkan ray tracing pipeline to a modern GPU using a technique we call delayed intersection and any-hit execution. Additionally we evaluate several ray tracing workloads with Vulkan-Sim, identifying bottlenecks and inefficiencies of the ray tracing hardware we model. To demonstrate the utility of Vulkan-Sim we conduct two case studies evaluating techniques recently proposed or deployed by industry targeting enhanced ray tracing performance.**

***Keywords*-GPU; Ray Tracing; Computer Graphics; Modeling and Simulation;**

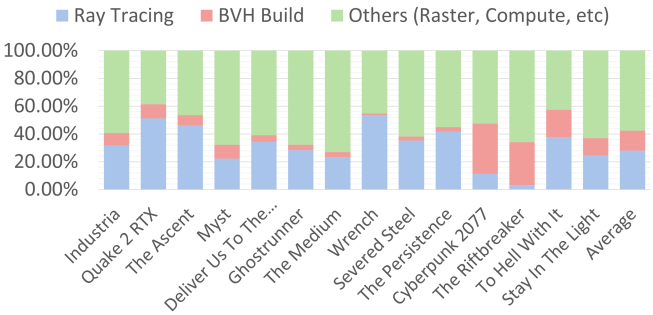


Figure 1: Game Frame Time Distribution on RTX 2080Ti

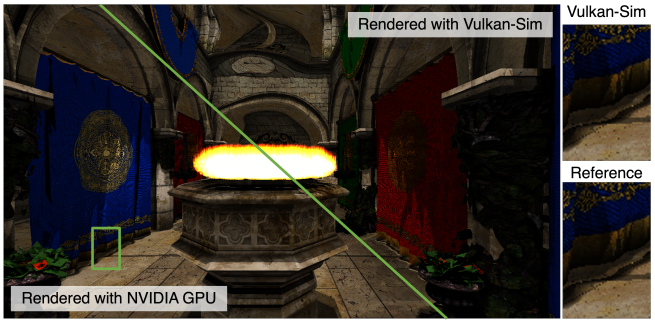


Figure 2: Sponza rendered with NVIDIA GPU and Vulkan-Sim. Right: Magnified comparison of green rectangle.

I. Introduction

Ray tracing has become more prevalent in recent graphics workloads. It is capable of creating more photorealistic renderings, including many visual effects that are difficult or impossible to create through rasterization. Consequently, ray tracing dominates in high-end graphical applications such as movies [20] and computer-aided design (CAD) renderings [4]. Moreover, as hardware advances ray traced effects are now being incorporated into a growing number of real-time workloads such as video games. However, computation requirements for ray tracing are still very high, even on a high end GPU. While games can benefit greatly from ray traced visuals, current graphics hardware only allow developers to implement a limited amount of ray tracing effects, such as soft shadows, reflections, or ambient occlusion, before frame rates dip below the realtime threshold of 60 frames per second [46].

Figure 1 shows the results of profiling existing GPU ray tracing accelerator enabled video games listed on NVIDIA's website [1]. The figure shows that ray tracing takes up 28% of the rendering time on average. While research on hardware accelerated ray tracing has been ongoing for decades [24], the recent adoption into consumer devices will likely increase demand for improved solutions in this area. By exploring ray tracing acceleration techniques architects can improve hardware to enable more complex scenes in real-time, thus further increasing demand for and prevalence of ray tracing. Architecture research is enabled by simulators, and for graphical workloads, hardware simulators model system behaviour better than software rendering tools [23]. However, industry simulation tools are generally proprietary and unavailable to academic researchers. Moreover, popular graphics applications tend to use cross-vendor APIs such as Vulkan [6] and OpenGL [2] which are more abstract than vendor-specific programming languages such as CUDA. Vulkan defines an abstract computation pipeline a developer can leverage to simplify the task of writing software while

Table I: Comparison of existing graphics / GPGPU simulators.

Simulator	Ray Tracing	Timing Model	GPU Model	Vulkan Support	Multi-Threaded	Execution Model
PBRT [46]	Yes	No	No	No	No	N/A
Emerald [27]	No	Yes	Yes	No	No	Execution Driven
TEAPOT [15]	No	Yes	Yes	No	No	Execution Driven
SimTRaX [50]	Yes	Yes	No	No	Yes	Execution Driven
Ray Predictor [37]	Yes	Yes	Yes	No	No	Execution Driven
GPGPU-Sim 3.x [16]	No	Yes	Yes	No	No	Execution Driven
Accel-Sim [33]	No	Yes	Yes	No	No	Trace Driven
GPUTejas [38]	No	Yes	Yes	No	Yes	Trace Driven
MGPUSim [53]	No	Yes	Yes	No	Yes	Execution Driven
Vulkan-Sim	*Yes*	*Yes*	*Yes*	*Yes*	No	Execution Driven

obtaining good performance across platforms. This pipeline requires the application to provide multiple programs, called shaders. These shaders are provided by the software developer and each shader performs a specific task.

To facilitate hardware research on ray tracing acceleration, this paper introduces Vulkan-Sim. Vulkan-Sim builds on top of GPGPU-Sim 4.0 [33] and provides the first publicly available Vulkan ray tracing hardware simulator. Vulkan-Sim models GPU accelerated ray tracing at cycle level and enables ray tracing research by providing a simulation infrastructure that supports the Vulkan API. Vulkan is a modern, industry standard, open-source graphics API similar to OpenGL, while providing lower level control over hardware to programmers. Enabling Vulkan support allows Vulkan-Sim to have access to a broad range of ray tracing workloads that are publicly available with little to no modifications at all.

Vulkan-Sim maps the execution of the Vulkan ray tracing pipeline to a GPU and integrates AccelWattch [32] to provide power and energy estimates. Figure 2 compares Sponza, a scene frequently studied in graphics research, rendered with Vulkan-Sim and with an NVIDIA GPU. The figure shows very few differences (only 0.3% of pixels differ) and serves to illustrate that Vulkan-Sim's Mesa based runtime and PTX based functional simulation engine together render images with high fidelity versus an industry validated implementation of Vulkan. Most prior graphics hardware research evaluates proposals using specially created simulators that are often not publicly available [13], [36], [41]. Others build FPGA prototypes [40], [56] that are less flexible for early stage exploration. Vulkan-Sim extends our prior work [37], which used a custom CUDA-based ray tracer, enabling the hardware accelerator model developed there to be used with a wider range of closer to industry standard ray tracing workloads. Table I compares existing graphics and GPGPU simulators and highlights the gap in Vulkan and ray tracing support.

We make the following contributions in this paper:

- We efficiently map the Vulkan ray tracing pipeline to a modern GPU using a technique we call *delayed intersection and any-hit execution*.
- We introduce Vulkan-Sim, an extension of GPGPU-Sim to support Vulkan ray tracing, with a performance model for a generalized ray tracing acceleration unit and show a correlation of 95.7% with NVIDIA RTX 2080 SUPER GPU. Vulkan-Sim renders Vulkan ray tracing workloads with high pixel color accuracy when compared against an NVIDIA GPU.
- We illustrate the use of Vulkan-Sim by conducting two case studies, evaluating a potential hardware optimization for warp divergence reduction and independent thread scheduling for ray tracing.

II. Background

This section gives an overview of ray tracing accelerators and an introduction to ray tracing with the Vulkan API.

A. Baseline GPU Architecture

Figure 3 illustrates the GPU architecture modeled in Vulkan-Sim. The GPU consists of multiple compute units called SMs in NVIDIA's terminology. The SMs are connected to memory partitions through an on-chip interconnect. The memory partitions include memory access scheduling logic to interface with off-chip DRAM (not shown). Inside each SM there are multiple execution units. Scalar threads in the application are organized into warps, a collection of 32 threads, which run together using Single-Instruction Multiple Thread (SIMT) execution. Threads in a warp execute the same instruction on different data. In the event of differing branch outcomes (known as branch divergence), the SM uses immediate post-dominator reconvergence [26] to serialize threads executing different paths. Warps within a SM are scheduled in Greedy-then-Oldest (GTO) fashion, which schedules from a single warp until it stalls. Each SM has its own L1 data, texture and constant caches which are connected to execution units through a crossbar.

While this architecture is tailored for compute applications and raster based graphics, manufacturers have recently added ray tracing accelerators to handle ray tracing graphics more efficiently by offloading computations from the SMs. To model such hardware Vulkan-Sim models a ray tracing accelerator in each SM, called the RT Unit. The RT Unit in Vulkan-Sim builds upon that introduced in Lu et al. [37] and will be further described in Section III-C. More details

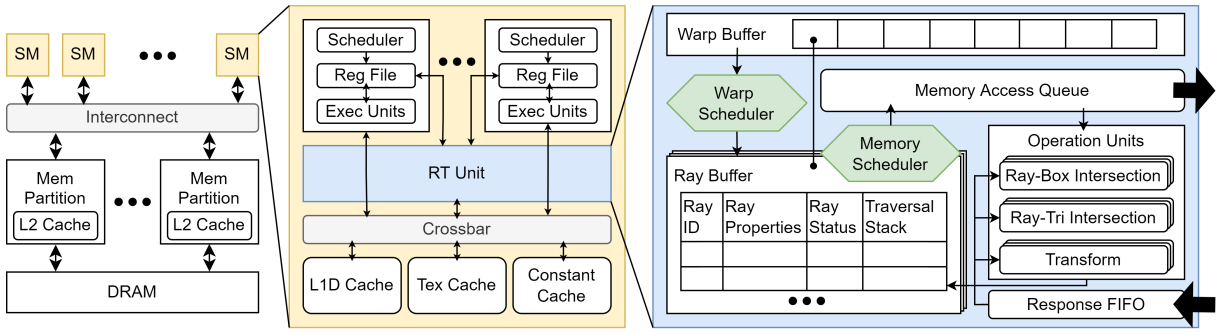


Figure 3: GPU performance model architecture

about general-purpose GPU architectures can be found in Aamodt et al. [12] and details specific to the GPGPU-Sim 4.0 baseline that Vulkan-Sim builds upon can be found in Khairy et al. [33].

B. Ray Tracing Accelerators

Modern GPUs, such as the NVIDIA RTX [18] and the Imagination PowerVR C-Series [17], feature hardware acceleration units for ray tracing. These ray tracing units collaborate with the GPU and perform two key roles. First, the RT Unit accelerates pointer chasing through a key data structure used in ray tracing, the bounding volume hierarchy (described in Section II-C). Second, the RT Unit computes intersections between rays and geometry in the scene using *Box Intersection Evaluators* and *Triangle Intersection Evaluators*. In Vulkan-Sim Vulkan `vkCmdTraceRaysKHR` calls within a shader initiate activity on the RT Unit. There are currently very few details publicly available of industry GPU implementations of ray tracing accelerators. Details lacking include how warps co-exist and how memory requests are scheduled. Thus, Vulkan-Sim employs a generic ray tracing accelerator performance model described in Section III-C. The RT Unit modeled in Vulkan-Sim employs techniques describe for the T&I Engine by Nah et al. [41] and SGRT by Lee et al. [36]. These designs follow a similar scheme that employ dedicated pipelined hardware units for acceleration structure traversal and ray intersection. We modify these designs to enable them to be combined with a GPU and integrate them in our RT Unit.

C. Ray Tracing With Vulkan

In this section we give an introduction to ray tracing and how it is performed in Vulkan. Ray tracing is a rendering technique for generating realistic lighting effects by simulating light rays as they propagate through and around objects in a scene [46]. Figure 4 shows a simple ray tracing example with a primary and a secondary ray. Primary rays originate from the camera, passing through the image plane and into the scene to calculate the color of that pixel (❶). If the primary ray hits an object, we can cast a secondary ray towards the light source to determine whether the point is shadowed (❷). If the ray's path is unoccluded, the pixel will be the cylinder's color, otherwise a shadow is rendered. By increasing the amount of rays cast per pixel, more parts of the scene are sampled, and light contribution from each ray is averaged to render a more accurate color for that pixel [31]. Each of these rays cast per pixel is called a sample in ray tracing. Having a high amount of samples per pixel comes at the cost of increased rendering time. To achieve better image quality under a constrained time budget, the scene which consists of many triangles and custom geometry is built into an acceleration structure (AS) as a bounding volume hierarchy (BVH) tree [39]. The AS reduces the amount of objects that a ray needs to test for intersections by partitioning the scene with axis-aligned bounding boxes (AABB) to reduce primitive search complexity.

Ray acceleration hardware has been incorporated into GPUs in recent years and graphics APIs such as Vulkan and DXR are starting to better support hardware accelerated ray tracing. This work focuses on Vulkan which unlike other popular graphics APIs, DirectX and OpenGL, is an open standard, low-level graphics API, targeting high performance real-time rendering by exposing hardware to developers and can be adopted by any vendor. Vulkan version 1.2.162 officially introduces extensions to the core API to support hardware accelerated ray tracing [7]. The `VK_KHR_acceleration_structure` extension handles AS building and management while `VK_KHR_ray_tracing_pipeline` is responsible for ray tracing shader stages and pipeline in Figure 5 which we explain later in this section.

A Vulkan ray tracing application consists of a CPU portion in C++ used to specify the rendering pipeline and GLSL shaders for the GPU to execute, similar to OpenGL. The C++ part of a Vulkan application performs multiple API calls for setting up the ray tracing pipeline. This includes requesting the `VK_KHR_ray_tracing_pipeline` extension to specify different ray tracing shader stages and allocating memory on the GPU used for the shader input and output [10]. The `VK_KHR_acceleration_structure` extension is used to build the AS, which is a tree data structure used to organize scene geometry into bounding volumes and reduces

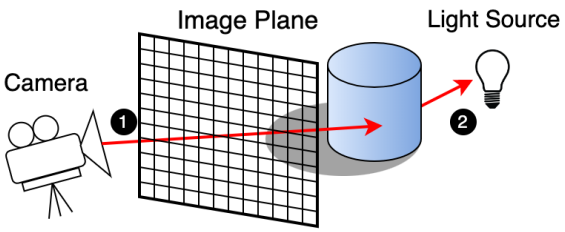


Figure 4: Ray Tracing Overview

ray traversal complexity logarithmically [9]. The Vulkan AS is defined in two levels as shown in Figure 6: one bottom-level AS (BLAS) for each unique object's geometry and a single top-level AS (TLAS) to position the objects within the scene with BLAS instances and its corresponding transformation matrix. Finally, once the CPU has finished setting up, the `vkCmdTraceRaysKHR` call is invoked to launch a ray tracing kernel on the GPU.

The ray tracing kernel follows the ray tracing pipeline defined by the Vulkan specification in Figure 5 and it describes the execution order of different types of ray tracing programmable shaders. There are multiple ways to implement the Vulkan ray tracing pipeline on a GPU which can vary between different GPU vendors. We expand on this further in Section III-A. In Vulkan-Sim, each GPU thread maps to a ray and begins at the ray generation shader where each thread uses its location on the screen to calculate the origin and direction of the ray it will trace and calls the `traceRayEXT` function to begin ray tracing. To determine what geometry the ray will intersect, a hardware traversal and intersection (T&I) unit, indicated by the green shapes in the pipeline, performs AS traversal and computes ray intersections with axis-aligned bounding boxes (AABB) in the AS and with triangles in the scene geometry [11]. Ray-triangle hits are validated by the any-hit shader and ray intersections with custom geometry are validated by the intersection shader. Intersection shaders are often used to implement user-defined or procedural geometry such as cylinders and determine if the ray hits the geometry or not. If the ray misses all objects in the scene, then a miss shader and otherwise a closest hit shader is executed. These shaders can be used to return the color of the intersected object or the color of the background. Details on how Vulkan-Sim simulates the Vulkan ray tracing pipeline is in Section III-B.

III. Simulating Vulkan Ray Tracing

In this section, we describe challenges of simulating Vulkan ray tracing kernels and how we address them in Vulkan-Sim's functional simulation and timing model. Vulkan-Sim is integrated with Mesa, an open-source implementation of graphics APIs such as OpenGL and Vulkan which is used for software emulation and hardware acceleration on GPUs. We briefly summarize how Mesa's Vulkan

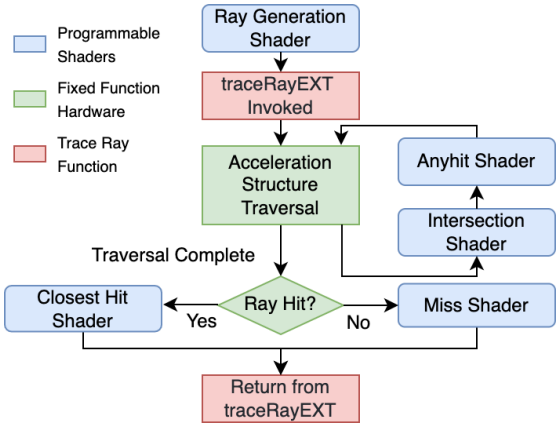


Figure 5: Vulkan Ray Tracing Pipeline

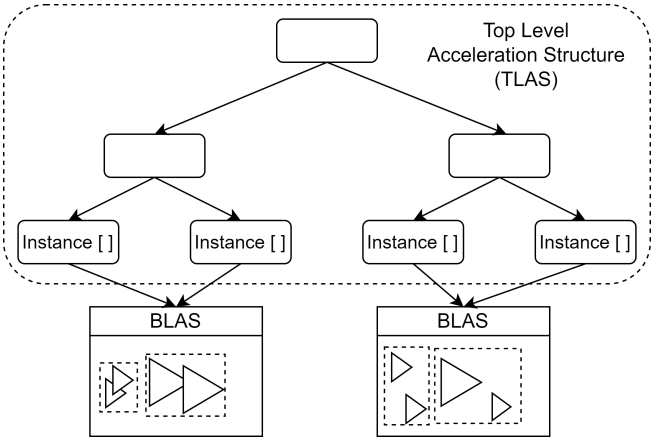


Figure 6: Vulkan Acceleration Structure

frontend is employed to enable Vulkan-Sim to support the Vulkan ray tracing API.

The rest of this section is organized as follows. Section III-A discusses the challenges of simulating Vulkan ray tracing workloads. Section III-B and III-C describes Vulkan-Sim's functional and timing model respectively. Section III-D gives an overview of Vulkan-Sim's software architecture.

A. Challenges of Simulating Vulkan

A challenge we tackle in this paper is finding a way to map the high-level ray tracing pipeline in Figure 5 onto a programmable GPU. To date, GPU manufacturers have released few details regarding their ray tracing implementations and open-source drivers, which may eventually provide greater insight into industry designs remain unavailable. Below are multiple methods for implementing the pipeline. In the first list we discuss how a GPU thread can map to a ray tracing workload, and the second list covers when the different shader stages in the Vulkan ray tracing pipeline should be executed.

- ***One thread per raygen shader***: In this method, each thread executes a raygen shader and continues to traverse

the BVH tree and call other shaders sequentially when a `traceRayEXT` function is encountered. Shader calls are treated as function calls. Thus, any thread divergence in the raygen shader continues in other shaders.

- ***One thread per shader***: In this method each shader is executed on a separate thread. When a shader call is needed, the calling thread writes to a work queue. One kernel is launched for each shader type which starts consuming from the work queue. This removes the flow of thread divergence from the raygen shader to other shaders as they will be grouped together in the working queues. However, it adds synchronization overhead.

During BVH traversal, intersection and any-hit shaders need to be executed to determine ray intersection with primitives. Rays in a warp can intersect multiple different leaf nodes, and each ray can execute a different set of shaders. We list various shader execution options below.

- ***Immediate execution***: In this method, when a thread reaches an any-hit or intersection shader during traversal, it calls the shader immediately. This causes high divergence when threads in a warp call different shaders.
- ***Thread spawning for intersection and any-hit***: Unlike miss and closest-hit shaders, any-hit and intersection shaders can be executed more than once per ray. We can use work queues similar to the ***one thread per shader*** method only for these shaders. This reduces thread divergence for scenes with multiple instances of these shaders by adding synchronization overhead.
- ***Delayed intersection and any-hit execution***: This method executes intersection and any-hit shaders after the traversal of all threads in the warp is completed. Afterwards, each thread executes all any-hit and intersection shaders for the ray sequentially. Delaying shader calls is allowed as Vulkan does not assume any order for intersection and any-hit shader execution. This method adds memory overhead as a list of intersection and any-hit shaders need to be stored during traversal for later execution.

In this work, we implement ***One thread per raygen shader*** for thread mapping as enough rays are launched to utilize all GPU cores due to the large pixel counts of high resolution images. For shader execution, We choose ***Delayed intersection and any-hit execution*** for its potential to reduce divergence. While other shader execution options may reduce divergence further due to executing the same shaders from different warps, they come at the cost of synchronization overhead.

B. Functional Simulation

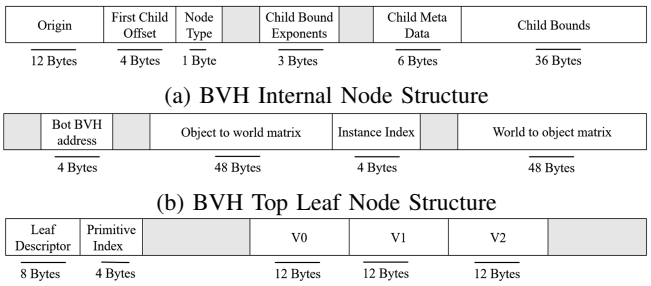
The functional simulation is Vulkan-Sim's implementation of the Vulkan ray tracing pipeline in Figure 5. In the following we elaborate our solution to mapping Vulkan RT applications employing multiple distinct RT shaders onto a SIMT pipeline supporting accelerated BVH tree traversal. The timing model, described in Section III-C, uses the functional simulation for accurate simulation on a GPU.

We break down the discussion of the function model into five parts organized to follow the flow of a Vulkan program: Acceleration Structure, Shader Translation, Shader Binding Table, Traversal and Intersection Implementation, and Kernel Invocation. The acceleration structure is a data structure created to represent the ray tracing scene and is traversed by the GPU during runtime. Shader Translation covers how Vulkan-Sim converts high level GLSL ray tracing shaders to low level instructions that are executable on a GPU. The translated shaders are stored in a Shader Binding Table specified by the Vulkan specification. Traversal and Intersection Implementation introduces how Vulkan-Sim traverses the acceleration structure and handles execution of shaders. Finally, Kernel Invocation describes how ray tracing is launched on a GPU.

1) Acceleration Structure: As mentioned in Section II-C, the acceleration structure is used to reduce the number of ray intersection tests logarithmically and is split into TLAS and BLAS instances. The acceleration structure implementation in Vulkan-Sim is a 6-wide BVH tree adopted from Mesa. The TLAS is made up of internal nodes and leaf nodes. Figure 7a shows the structure of internal nodes. Each internal node is 64 bytes and contains pointers to its children along with the AABB of each child node. Since child nodes of an internal node are stored consecutively in memory, we can reduce the memory footprint by only storing the pointer of the first child instead of storing all six child node pointers. Figure 7b shows the top level leaf nodes which are at the bottom levels of the TLAS and are 128 bytes each. They point to the root address of BLAS instances and also hold the object-to-world and world-to-object transformation matrices of the BLAS instance. Additionally, they hold user-defined instance indices that specify which closest-hit and intersection shaders should be executed if geometry in the corresponding BLAS is intersected.

Traversal of the BLAS is started if a leaf node in the TLAS is intersected. The BLAS consists of internal nodes, triangle leaves, and procedural leaves. The BLAS internal nodes are structured like the TLAS internal nodes, except they point to triangle leaves or procedural leaf primitives instead of BLAS root nodes. Figure 7c shows the structure of the triangle leaves which are 64 bytes each and store the leaf descriptor, primitive index, and triangle vertices. Leaf descriptor contains metadata such as the type of the node. Procedural leaves are used for rendering procedural geometry, which are 3D meshes generated by code. Each procedural leaf consists of a leaf descriptor and a primitive index that is addressed during geometry generation.

2) Shader Translation: A Vulkan application submits a ray tracing pipeline with a call to `vkCmdTraceRaysKHR`. This sends all the shaders in Figure 5 to Mesa for



(a) BVH Internal Node Structure

(b) BVH Top Leaf Node Structure

(c) BVH Triangle Leaf Node Structure

Figure 7: BVH Node Structures

compilation. Graphics shaders are written in the OpenGL Shading Language (GLSL), precompiled to SPIR-V, an intermediate representation used by Vulkan, and read in by the Vulkan application. These shaders are then compiled to an intermediate representation used exclusively by Mesa called NIR. NIR is designed to be backend-agnostic and used as an initial translation pass for optimization. NIR can then be translated to a hardware specific ISA by a Mesa backend module. As Mesa lacks such a backend module for ray tracing, in order to execute ray tracing shaders on Vulkan-Sim, we created a NIR to PTX translator to convert NIR shaders to PTX, a virtual GPU ISA created by NVIDIA that is compatible with GPGPU-Sim. We also extended the PTX ISA to include several instructions to accommodate the Vulkan API (Table II). Most NIR instructions are low-level instructions that easily map to one or a sequence of PTX instructions such as ALU or control flow instructions. However, NIR includes 15 high-level ray tracing specific instructions such as `traceRayEXT`, `loadRayWorldOrigin`, `loadRayLaunchId`, etc. that need to be translated to lower-level instructions. The most notable `traceRayEXT` instruction translates to PTX shown in Algorithm 1, described later in this section. Some instructions such as `loadRayWorldOrigin` can be translated to existing PTX instructions while others such as `loadRayLaunchId` are translated to a custom PTX instruction. Table II lists the new PTX instructions added for ray tracing. We verify the translation by comparing instruction results in GLSL shaders with an NVIDIA GPU.

One issue encountered during shader translation is how different shader stages in the Vulkan ray tracing pipeline should be executed by a GPU, as mentioned in Section III-A. RT shader stages need to be executed in a particular fashion defined by the Vulkan ray tracing pipeline in Figure 5. While all threads execute the raygen shader, some shader stages such as the miss, any-hit, or closest hit shader only conditionally execute when rays miss or intersect objects, and cannot simply be executed serially. Our solution handles this during shader translation by starting with tree traversal, collecting geometry hits, and then calling other shaders conditionally.

`traceRayEXT` is an important GLSL function in RT shaders, that starts AS traversal and executes other shaders as shown in Figure 5. This function can be be implemented as a software subroutine or offloaded to an RT accelerator and in Vulkan-Sim, we offload to a dedicated accelerator called the RT unit. Section III-C gives more details on how we model the RT unit to perform AS traversal. Algorithm 1 shows the PTX pseudocode that the `traceRayEXT` GLSL function translates to in the translated PTX shaders. `traverseAS()` is the instruction that calls a software subroutine or a hardware accelerator to perform AS traversal. The additional code after `traverseAS()` handles the correct conditions for executing the various shader stages of the ray tracing pipeline. During AS traversal (Line 1), traversal data of all intersected procedural leaf nodes are stored in a table. After traversal is completed, intersection shaders are called in a loop (Lines 2-11) as we employ the *Delayed intersection and any-hit execution* method, discussed in Section III-A. The *intersectionExit* instruction determines if the thread is done with all intersection shaders, after which the intersection shader is called in an if-else-if fashion. Next, if geometry is intersected by the ray, a closest-hit shader is called (Lines 12-18). Otherwise, a miss shader is executed (Line 20). Closest-hit shaders are called in an if-else-if fashion, based on the shader ID of the target shader. Algorithm 1 shows code for only two intersection and two closest-hit shaders, but can be extended beyond this. Mesa receives a list of shaders and their type as input when the ray tracing pipeline is created. Based on this information, the NIR to PTX translator generates the correct amount of intersection and closest-hit shader calls at runtime by recording the shader IDs of those shaders and inserting if statements to match the number of each shader type in the translation. While the pseudocode doesn't show execution of any-hit shaders, they should be executed in a loop similar to intersection shaders.

Traversal information, such as the closest hit geometry, is stored in a structure in main memory that can be accessed by specific shader instructions. Since the GLSL `traceRayEXT` function can be called in other shaders recursively, results of traversal are stored in a stack. The `endTraceRay` instruction pops the stack and clears the intersection shader (Line 22). Table II showcases the more important custom PTX instructions added in Vulkan-Sim to support Vulkan ray tracing.

3) Shader Binding Table: Vulkan uses a shader binding table to record all shaders in a ray tracing pipeline. While only one ray generation shader is required for a Vulkan ray tracing kernel, multiple closest-hit, miss, any-hit, and intersection shaders can be specified. Vulkan-Sim assigns an ID to each shader when it is registered, which is returned to the user program and used as shader handles. These handles are stored in the shader binding table and specify which shaders should be executed.

Algorithm 1 traceRayEXT PTX Shader Implementation

```
1:  traverseAS()
2:  intersectionIdx ← 0
3:  while intersectionExit(intersectionIdx) do
4:      shaderID ← getIntersectionShaderID()
5:      if shaderID == intersectionID0 then
6:          callIntersectionShader(shaderID)
7:      else if shaderID == intersectionID1 then
8:          callIntersectionShader(shaderID)
9:      end if
10:     intersectionIdx++
11: end while
12: if HitGeometry() then
13:     shaderID ←getClosestHitShaderID()
14:     if shaderID == closestHitID0 then
15:         callClosestHitShader(shaderID)
16:     else if shaderID == closestHitID1 then
17:         callClosestHitShader(shaderID)
18:     end if
19: else
20:     callMissShader()
21: end if
22: endTraceRay()
```

Table II: Notable Vulkan-Sim Custom PTX Instructions

PTX Instruction	Instruction Description
traverseAS	Traverse the acceleration structure
endTraceRay	Pop traversal results stack and clear intersection table
rt_alloc_mem	Allocate memory and load address for variables shared among different shaders
load_ray_launch_id	Load a unique ray ID for each thread

4) Traversal and Intersection Implementation: Another core component of Vulkan-Sim's functional model is the implementation of the `traverseAS` function in our translated shader in Algorithm 1. Based on the acceleration structure organization introduced in Section III-B1, rays will encounter internal nodes and leaf nodes when traversing the TLAS, and intersect internal nodes, triangle leaves, and procedural leaves when in the BLAS. A ray begins traversal at the root node of the TLAS and enters the while loop in Line 1 of Algorithm 2. Since the root node is also an internal node, the ray enters the while loop on Line 2 and traverses through internal nodes until it encounters a top level leaf node. This signals that it is entering the BLAS, so we apply the worldToObjectMatrix to the ray to transform it to the BLAS's coordinate system on Line 6. Similar to the TLAS, the ray traverses through internal nodes of the BLAS until it reaches a leaf node which can be either a triangle leaf or a procedural leaf. For triangle leaves, we perform ray-primitive intersection tests and update the closest-hit geometry if it intersects on Line 12.

Algorithm 2 Acceleration Structure Traversal

```
Require: Ray, AccelerationStructure
1:  while Ray Not Terminated do
2:      while Node.Type == TopInternalNode do
3:          Traverse to the next node
4:      end while
5:      while Node.Type == TopLeafNode do
6:          Apply worldToObjectMatrix to Ray
7:          while Node.Level == BottomLevel do
8:              while Node.Type == BottomInternalNode do
9:                  Traverse to the next node
10:             end while
11:             while Node.Type == BottomLeafNode do
12:                 if Node.LeafType == Triangle then
13:                     Perform primitive intersection test
14:                     Update closest-hit geometry
15:                 else
16:                     /* Node.LeafType == Procedural */
17:                     add intersection to intersectionBuffer
18:                 end if
19:             end while
20:         end while
21:     end while
22: end while
```

For procedural leaves, we add the primitive index and the intersection shader ID to the intersection buffer on Line 17. Every time a ray accesses a node or intersection buffer, we record memory addresses that are accessed with its size and data type to a transactions buffer, which is then sent to the timing model to simulate memory access latencies.

5) Kernel Invocation: The Vulkan binary launches ray tracing kernels by calling `vkCmdTraceRaysKHR` which invokes Vulkan-Sim through the Mesa frontend with shader binding tables and kernel dimensions being the inputs. Vulkan-Sim starts a kernel with block size of $(32, 1, 1)$ and grid size of ($launch\_width$ / 32, $launch\_height$, $launch\_depth$). Each thread executes a raygen shader, and in our case $launch\_width$ and $launch\_height$ correspond to image width and length, where each warp handles 32 pixels horizontally across the image starting from the top left pixel. This can be freely configured to explore different mappings. Afterwards, each thread executes the ray generation shader and follows the ray tracing pipeline. Shader input and output are handled through special Vulkan buffers called descriptor sets, which are sent to Vulkan-Sim through the Mesa frontend and accessed during shader execution.

C. Timing Model

As mentioned in Section II-A, we extend the existing GPGPU-Sim performance model illustrated in Figure 3 to include a ray tracing accelerator (RT unit) in a similar manner to the baseline GPU described by Liu et al. [37]. Although

conceptually alike, we model the full Vulkan ray tracing pipeline, including shader execution, instanced acceleration structures with varying data sizes, and storing of ray hits that were previously ignored. The original GPU model consists of a collection of streaming multiprocessors (SMs) connected through an interconnection network to various memory partitions. Within each SM, there are several cores that feature a warp scheduler, a dedicated register file (RF), and its own functional execution units. We treat the RT unit as an execution unit, except only one exists per SM rather than per core, and add this to the SIMT compute pipeline. When a trace ray instruction is decoded and ready to execute, the warp is routed to the RT unit, which requires a variable latency to complete, similar to the existing load/store units. Vulkan-Sim enables detailed evaluation for each modeled component and offers opportunities to explore different scheduling orders and other potential RT unit optimizations. Figure 3 shows the GPU architecture and zooms in on the RT unit in more detail.

1) RT Unit Overview: The performance model of the RT unit focuses on two sources of latency: BVH operations and memory accesses. Warps enter the RT unit during the execution stage in the SIMT compute pipeline and are tracked in the *Warp Buffer*. In each cycle, a warp is selected, and memory requests from the threads in the warp are scheduled to be issued to the L1 cache. The returning ray tracing data is directed into the *Response FIFO* to be processed, modeling the latency of memory accesses. Then, the *Operation Scheduler* determines the requesting thread and forwards its ray properties along with the returned geometry data to the *Operation Units*, modeling the latency of BVH operations such as ray-box intersections, ray-triangle intersections, and coordinate transformations. Finally, once the BVH operation is complete, the controller updates the appropriate threads with the results and proceeds down the BVH tree accordingly.

2) Warp Management: Up to eight warps can co-exist within the RT unit in our baseline configuration as a sweet spot for area overhead and performance. Each warp maintains a *Ray Buffer*, which tracks ray information such as its properties, current status, and traversal stack, similar to the Ray Store in the Imagination Ray Acceleration Cluster (RAC) [17]. Ray properties include the origin, direction, and t-parameters, which are required to perform intersection tests. The traversal stack is maintained as a short stack [29] with eight entries and spills into per-thread memory as described by Aila et al. [14]. For each new cycle, the *Warp Scheduler* selects a single warp following a greedy then oldest approach by prioritizing a single warp until it stalls waiting for memory requests. This approach is generally preferable over round robin, where warps are more likely to arrive at stalling memory accesses around the same time [42]. The number of active threads for each warp is tracked in *Warp Status*, which signals completion when there are no remaining threads traversing the BVH tree.

3) Memory Scheduling: Once a warp has been scheduled, the memory scheduler evaluates the *Ray Status* and the *Traversal Stack* in the *Ray Buffer* for each thread in the warp. The *Ray Status* indicates if the thread is ready to issue a memory request, and the next memory address is read from the *Traversal Stack*. Otherwise, the thread could be awaiting a previous request, performing an intersection test, or completed with its traversal. The *Memory Scheduler* collects each of these addresses from all the threads in the warp, merging any identical requests, and pushes the final unique set to the *Memory Access Queue*. By the end of the cycle, all threads should have pushed their respective requests into the *Memory Access Queue*, or remain in a ready state if the queue is full. The first request in the queue is also sent to the L1 data cache, or potentially, a dedicated RT cache. Larger data requests are broken into several 32B chunks to be processed over multiple cycles. On a cache miss, the memory request continues into the memory hierarchy and eventually returns to the *Response FIFO*, following the performance model of the latest GPGPU-Sim [33].

4) BVH Operations: At the beginning of each cycle, the RT unit pops from the *Response FIFO* if there is data and forwards it to the *Operation Units*. The *Operation Scheduler* evaluates the *Ray Status* and the top of the *Traversal Stack* to identify threads awaiting the current memory response and forwards the *Ray Properties*. There are three possible BVH operations: ray-box and ray-triangle intersection test, and coordinate transformations explained in Section III-B. A flag in the returned data determines which of the three pipelined hardware units should be used. We model our intersection units as described by Liu et al. [37], which is based on ray-box and ray-triangle intersection units in the T&I Engine [41], and our transformation units as a simple matrix multiplier. We include sufficient instances of these units to avoid the need for an additional queue and assume a fixed latency. Alternative configurations of operation units can easily be explored with Vulkan-Sim.

Once the BVH operation is complete, the controller updates the *Ray Status* and pops the top entry from the *Traversal Stack*, setting up the thread for the next node on the following cycle. On a primitive hit, the results are stored in memory and read back during the closest hit shader execution.

D. Software Architecture

Vulkan-Sim builds upon the Mesa graphics library with partial Vulkan ray tracing support. Figure 8 shows the software architecture of Vulkan-Sim. After launching a Vulkan ray tracing binary, the application communicates with Mesa to execute Vulkan API calls (❶). When the Vulkan binary registers SPIR-V shaders to the ray tracing pipeline with the API call `vkCreateRayTracingPipelinesKHR`, Mesa first compiles them to an intermediate representation called NIR that is used for optimization and translation for various

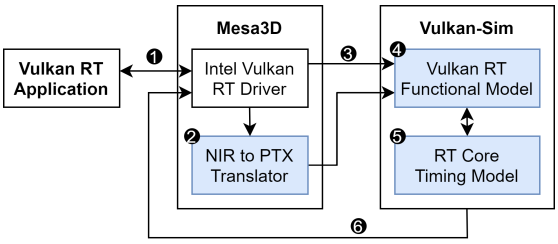


Figure 8: Vulkan-Sim Software Architecture

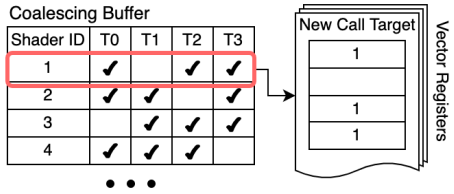


Figure 9: Function Call Coalescing, `getNextCoalescedCall` operation

backend drivers. The NIR-to-PTX Translator in Vulkan-Sim translates the NIR shaders to PTX shaders that are compatible with GPGPU-Sim and stores them for later use (❷). When the Vulkan binary makes the ray tracing API call `vkCmdTraceRaysKHR`, Mesa invokes Vulkan-Sim to begin simulation of the translated ray tracing PTX shaders that were generated in ❷. In addition to the translated shaders, we also send Vulkan-Sim all the arguments of `vkCmdTraceRaysKHR` and the descriptor sets that are used for shader input and output (❸). Once Vulkan-Sim is invoked, the functional model (❹) executes the ray tracing PTX shaders and communicates with the timing model (❺) to perform cycle level simulation. We explained this in more detail in Sections III-B and III-C respectively. Once the simulation is complete, it stores any results back to the corresponding image buffers or memory locations, and the `vkCmdTraceRaysKHR` API call is complete (❻).

IV. Case Studies

Using Vulkan-Sim, we present two case studies targeting scheduling in the RT units to improve ray tracing performance. First, we introduce function call coalescing, which attempts to optimize shader calls in the ray tracing pipeline. Then, we evaluate how independent thread scheduling can support ray tracing workloads.

A. Function Call Coalescing

A multitude of user-defined shaders can be specified in the Vulkan ray tracing pipeline, where a set of shaders can be specified for each BLAS instance in the AS. Although this gives the application flexibility, it may result in redundant function calls leading to poor SIMT efficiency. Considering that Vulkan does not define an order for ray hits (such as closest-first), the T&I unit can be optimized to process geometry in an any-order. Thus, at each intersection call-site of Algorithm 1 the threads of a warp are not guaranteed to invoke the same shader. Invoking a non-uniform function call on a SIMT architecture requires looping over the shader IDs. Redundant shader execution occurs when different threads invoke the same shader on different iterations of the outer loop of Algorithm 1. The thread divergence when the intersection shaders are called is unnecessary as reordering the shader execution can resolve the divergence. This also leads to lower than necessary SIMT efficiency and may explain the same issue observed by instrumented any-hit and intersection shaders for workloads with several non-opaque or procedurally defined objects [45].

We evaluate Function Call Coalescing (FCC), a method for eliminating shader call divergence and mitigating redundant shader invocations, proposed by Nowicki [43]. Function call coalescing includes a coalescing buffer that is conceptually a table in GPU memory where each row stores a shader ID and a flag for each SIMT thread that invokes the function. Additionally, each thread stores intersection data such as primitive index and instance index. FCC differs from baseline when (1) procedural intersections are inserted into the table in Algorithm 2 and (2) execution of the shaders.

The `intersectionBuffer` is replaced with `coalescing buffer` in Algorithm 2. When a new procedural intersection is found, the shader ID is inserted in the table by matching with an existing shader ID. This results in a set of shader IDs and a thread mask for their corresponding SIMT threads. The table could have multiple entries with the same shader ID if a thread hits the same procedural geometry multiple times. The operation is repeated until all hits are processed.

The details of the code generation for the trace ray function are shown in Algorithm 3. Each row in the buffer is read once by the `getNextCoalescedCall` command, producing a vector register for the function call as shown in Figure 9. Each thread in the warp will always invoke the same function, eliminating the function call divergence. We evaluate FCC on RTV6 scene using Vulkan-Sim in Section VI-E

B. Independent Thread Scheduling

GPUs follow a single instruction multiple threads (SIMT) paradigm that performs poorly during branch divergence when the warp is split and execution proceeds serially between the two warp splits. Using a default SIMT stack model to handle control flow divergence allows only one warp split to be scheduled even for long latency operations that leave the pipeline idle. As part of the Volta microarchitecture, NVIDIA introduced independent thread scheduling (ITS) to enable greater flexibility in their GPUs, allowing threads to diverge and execute in an interleaved fashion [19], [22]. This alternative approach is implemented in Vulkan-Sim as

Algorithm 3 traceRayEXT PTX Shader With FCC

```
traverseAS()
intersectionIdx ← 0
while intersectionExit(intersectionIdx) do
    shaderID ← getNextCoalescedCall(intersectionIdx)
    if shaderID then
        callIntersectionShader(shaderID)
    end if
    intersectionIdx++
end while
if HitGeometry() then
    shaderID ← getClosestHitShaderID()
    if shaderID == closestHitID0 then
        callClosestHitShader(shaderID)
    else if shaderID == closestHitID1 then
        callClosestHitShader(shaderID)
    end if
else
    callMissShader()
end if
endTraceRay()
```

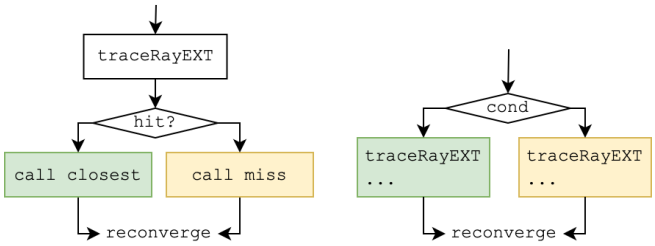


Figure 10: Example branch divergence in ray tracing.

a multi-path architecture by tracking reconvergence in tables rather than a stack to schedule warp splits independently [25].

Control flow divergence is very common in the ray tracing algorithm during BVH traversal, but it is mitigated by hardware RT units. However, there is also divergence in shader execution, specifically when a warp splits between the closest-hit and the miss shader for rays that hit or miss the scene, shown in Figure 10 (left). The raygen shader can also include branches around `traceRayEXT` to trace different variations of rays depending on some condition as shown in Figure 10 (right), which we observe in the Quake II RTX game [44]. In these cases, stack-based reconvergence executes the green block then the yellow block serially whereas ITS attempts to schedule the green and yellow blocks in parallel. Since `traceRayEXT` is a long latency instruction and independent between branches, the first green block can be scheduled and once it has entered the RT unit, the next yellow block can be scheduled without waiting for ray tracing to complete. Executing these blocks simultaneously should improve RT unit efficiency. We evaluate ITS with

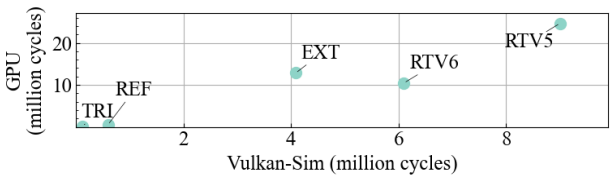


Figure 11: Correlation between Vulkan-Sim and NVIDIA RTX 2080 SUPER GPU.

Vulkan-Sim in Section VI-F.

V. Methodology

Vulkan-Sim is constructed from two components: the Vulkan frontend and the execution engine. For the Vulkan frontend, we extended Mesa 21.1.0-devel with preliminary ray tracing support and completed its ray tracing functionality through Vulkan-Sim's functional simulation. The MESA Vulkan frontend provides acceleration structure building, shader compiling, and Vulkan API interface, which we intercept in the frontend and send to GPGPU-Sim to invoke execution of shaders. For the execution engine, we extended GPGPU-Sim 4.0.0 [33] to model the RT unit mentioned in Section III-C and to support custom Vulkan specific instructions from translating NIR shaders to PTX.

Although sharing the same base timing model, Vulkan-Sim is not a trace-based simulator like Accel-Sim [33], which relies on NVIDIA's native GPU ISA (SASS). Extending AccelSim traces to simulate ray tracing is challenging because NVBit, the instrumentation tool used to generate execution traces, does not expose sufficient details of the ray tracing SASS instruction. Instead, Vulkan-Sim employs functional emulation. Vulkan-Sim is able to extract detailed memory level information and workload characteristics as demonstrated in our results (Table IV), including the bounding volume hierarchy (BVH) tree depth and node count. PTX level simulation provides important flexibility when studying ISA modifications like those we study in Section IV-A.

Table III shows the evaluated configurations. We allocate one RT unit per SM, similar to NVIDIA. The max number of warps allowed concurrently in the RT unit affects the area overhead of ray and warp buffers. The more concurrent warps, the higher the area overhead. The number of intersection units have less of an impact since memory is the main bottleneck for ray tracing. We choose 32 of each operation unit in the RT unit so it fully handles all 32 threads in a warp.

A. Evaluation Benchmarks

We select five workloads to evaluate, outlined in Table IV, with three official Vulkan samples released by KhronosGroup, one popular Vulkan ray tracing renderer, and a microbenchmark. TRI is a simple ray traced triangle consisting of only primary rays and REF has mirror reflections and shadows, which are rendered by secondary rays. EXT is the Sponza

Table III: GPGPU-Sim Configurations

	Baseline	Mobile
# Streaming Multiprocessors (SM)	30	8
Max Warps / SM	32	
Warp Size	32	
Warp Scheduler	GTO	
# Registers / SM	65536	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	
RT Unit Max Warps	4	
RT Unit MSHR Size	64	

scene, a commonly used model in the graphics community, which features more complex geometry, textures, and uses more types of rays such as secondary, shadow, and ambient occlusion (AO). RTV5 is from the *RayTracingInVulkan* workload [3], where we can load OBJ files such as the statue and create effects such as refractions, depth of field, and global illumination. We add the RTV6 scene to this workload for evaluating FCC, which also demonstrates the capability of Vulkan-Sim to handle multiple types of procedural geometry. Only one frame is rendered for RTV5 and RTV6 and the rendered images are noisy as low sample path tracing tends to be [28], [30]. While Vulkan-Sim currently lacks some extension support to run more complex workloads such as games like Quake II RTX [44], extensions can be added for better Vulkan compatibility. Additionally, RTV6 supports loading custom geometry and shaders to render more detailed scenes such as those found in modern games.

B. Simulator Validation

We validate Vulkan-Sim's functional simulation by comparing function results in ray tracing shaders against NVIDIA. Only 0.3% of pixels rendered in Figure 2's sponza scene differ from an NVIDIA GPU. We have attempted to validate Vulkan-Sim's timing model with RT accelerators in existing GPUs, but some issues stand in the way. The first challenge is the limited amount of provided documentation from companies. NVIDIA, AMD, and Imagination have provided high level diagrams or press releases about their RT acceleration, however the inner details of their architectures are unknown. Secondly, the trace ray instruction is a CISC-like instruction with an unknown amount of memory references and the underlying BVH structure that it operates off is not documented. While it may be possible to microbenchmark, it is not in the scope of this work. Despite these challenges, since they both support Vulkan's ray tracing API, in Figure 11 we compare execution cycles between Vulkan-Sim and an NVIDIA RTX 2080 SUPER GPU which should provide some sense of how the implementations relate. The datapoints in this figure corresponds to a correlation of 95.7% on our evaluation

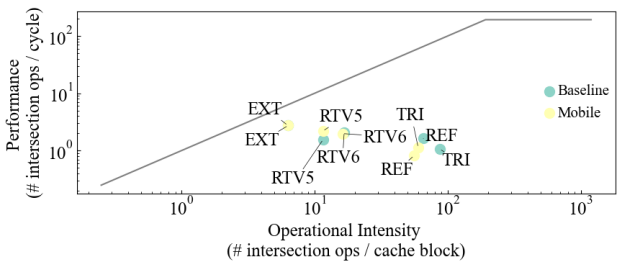


Figure 12: Roofline plot for the RT unit.

benchmarks. We note that the slope between hardware and simulator is about 2.58 (rather than 1.0) which we suspect is due to differences in RT unit design, such as having lower numbers of operation units or fewer warps per RT unit to save area. We also conduct a separate correlation study in Section VI-G which better matches simulation parameters to a RTX 2080 SUPER and provide some insights. Another reason is we captured execution times on a real GPU and multiplied by the GPU clock to get cycles, which could be inaccurate due to variable clock frequency of GPUs. Additionally, other companies may end up with very different solutions for RT acceleration so exactly matching NVIDIA's hardware is not a goal for us. Vulkan-Sim merely provides a Vulkan simulation framework and baseline RT unit for researchers to improve upon.

VI. Results

In this section, we evaluate the RT unit design in Section III and IV on five different Vulkan ray tracing workloads shown in Table IV, with images rendered using Vulkan-Sim without denoising. In these workloads, ALU operations account for 60% of the measured instruction type breakdown, followed by memory operations with 25%, and only around 1% trace ray instructions. While the workloads only execute a small number of trace ray operations, each of those operations contributes to a high percentage of memory accesses. EXT is the most realistic workload that we evaluate, where trace ray instructions make up around 60% of memory accesses, with the RT units active for 92% of total cycles on average.

A. Roofline Plot

To evaluate the performance bottlenecks in ray tracing, we adapt the Roofline Model [55] which relates performance to memory traffic using the configurations in Table III. In our ray tracing context, we consider intersection tests and ray transformations as *operations*, and we define *Operational Intensity* to be the total operations performed per cache block fetched. With this definition, *Performance* can be measured as operations per cycle (or the total number of pipeline stages occupied in the intersection and transformation units) with a maximum value defined as the $\#\ units \times \#\ stages\ per\ unit$. We plot the memory bound as one cache block per cycle.

Table IV: Summary of workloads (images rendered by Vulkan-Sim).

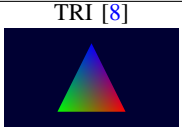
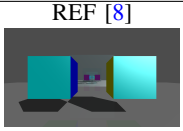
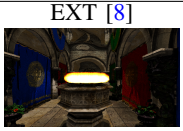
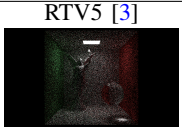
Scene	TRI [8]	REF [8]	EXT [8]	RTV5 [3]	RTV6
					
BVH Tree Depth	3	4	13	12	8
Average Nodes per Ray	1.5	4.3	73	7.3	19
Primitive Count	1	50	283265	448893	4080

Figure 12 illustrates where each workload fits on the roofline plot. All workloads fall under the memory bound but are far from both the memory and compute bound, implying that ray tracing performance is generally limited by memory accesses, but these workloads are underutilizing the available resources. EXT and RTV, which are more realistic workloads, are closer to the memory bound, almost fully utilizing the RT unit hardware. In a mobile GPU context, EXT and RTV5 are even more clearly memory bound. Since none of the points lie on the bounding lines, this plot highlights the need to either make workloads more efficient in software or improve the way workloads are executed in the RT units. Techniques such as warp scheduling, warp repacking, and cache replacement policies can be explored with Vulkan-Sim to bring workloads closer to the bounds and fully utilize GPU resources.

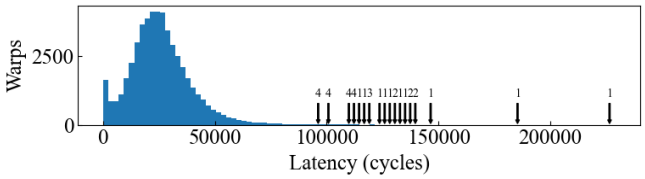


Figure 13: Per warp latency in RT units.

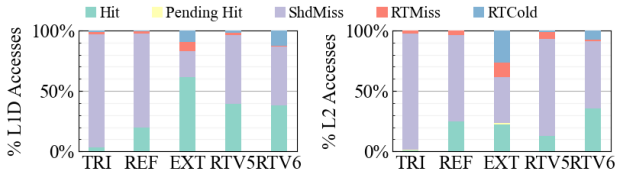


Figure 14: L1D and L2 cache accesses breakdown.

B. SIMT Efficiency

SIMT efficiency measures the proportion of active threads in a warp each cycle, which is a good measure of how well a workload executes on a GPU. TRI and REF workloads have nearly full SIMT efficiency, only rarely diverging at geometry boundaries where part of warp may hit while the remainder misses. EXT, RTV5, and RTV6 show more warp divergence, with more than 50% of warps executing only 1-4 threads out of the possible 32, as a result of the larger scene and more secondary rays. For example, primary rays only make up 15% of the workload in EXT. Secondary rays are generally incoherent with different ray origins and directions, which cause them to traverse different parts of the BVH tree and execute different paths in the shaders. This is especially prominent in RTV5 and RTV6 where secondary rays are generated by scattering randomly throughout the scene.

Similarly, SIMT efficiency in the RT unit measures the proportion of active rays in each cycle. Low SIMT efficiency in the RT units results from early terminating rays mixed with long tail effects, compounded with any low SIMT efficiency from the GPU. When a thread completes traversal, it idles until all threads from the same warp are also complete, then the instruction is committed and another warp can be scheduled to the RT unit. On average, SIMT efficiency is only 35% in the RT units, with RTV5 as low as 7%. Threads are idle for 59% of cycles on average, waiting for tailing threads to complete traversal. The low SIMT efficiency contributes to the inefficient use of GPU resources highlighted in the roofline plot. Several prior works have proposed solutions to mitigate this issue, but without a detailed simulator like Vulkan-Sim, we cannot confirm their effectiveness.

At a higher level, these long tail effects also result in low warp occupancy and influence the execution time of the entire workload. Figure 13 shows a distribution of warp latency in the RT units for the EXT workload. Some warps complete very quickly by benefitting from cache hits or missing the geometry in the scene entirely. The remainder of warps follow what resembles a log-normal distribution, with 95% of warps completing in 50000 cycles but a few trailing warps requiring around 4× more cycles. These trailing warps are indicated with arrows in Figure 13, totaling just 35 warps out of more than 50000. Vulkan-Sim shows that ray tracing performance cannot be improved unless tail effects are mitigated.

C. Memory

Figure 14 shows the breakdown of the L1D and L2 cache. Cache misses primarily result from shader loads with only a small portion coming from RT unit accesses, which aligns with the origin of the memory requests. Interestingly, shader loads show proportionally more misses than RT unit loads despite the random nature of BVH memory accesses. However, most of these are compulsory (cold) misses, which signify that the data is not reused and perhaps should not be cached. For RT unit loads, there is evidence of cache

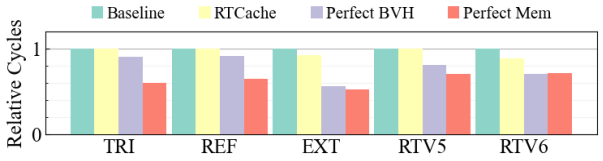


Figure 15: Execution times of memory configurations.

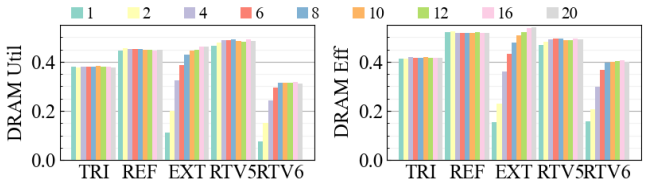


Figure 16: Comparison of DRAM for varying maximum number of warps per RT unit.

thrashing from the amount of capacity or conflict misses. Figure 15 shows that using a dedicated RT cache can improve performance.

Figure 15 also shows two limit studies for zero latency BVH node accesses (Perfect BVH) and zero latency DRAM accesses (Perfect Mem). Perfect BVH only targets memory accesses originating from the RT unit, so it has lower gains for TRI and REF where shader loads dominate. In EXT, RT unit loads dominate and Perfect BVH shows greater speedups, demonstrating the workloads are mainly memory bound.

Figure 16 evaluates the DRAM efficiency and utilization for these workloads across a range of maximum warp limits per RT unit from 1-20. These values are measured as the percentage of cycles where data was transferred out of the total execution cycles for DRAM utilization, and specifically out of cycles where there were DRAM requests at the memory access scheduler for DRAM efficiency. Our default configuration allows a maximum of eight warps to execute in each RT unit simultaneously, labeled as '8' in Figure 14. Raising this limit from a single warp provides more rays to schedule between and hide the memory latency but flattens at around eight warps when additional warps start to compete for available bandwidth. All the workloads we evaluate do not show high DRAM efficiency, averaging at 46%, likely from coherent memory accesses at the beginning of traversal. For example, memory requests for the BVH root node are merged together across all warps in an RT unit and represented by a single request to the DRAM. These merged requests are sequential in time as part of the BVH traversal process and potentially random in address depending on how the BVH tree is stored in memory, which produces both a low bank level parallelism and low row buffer locality in the DRAM. In a mobile GPU configuration with less DRAM bandwidth, the efficiency and utilization are higher at 77% and 75%, respectively. The DRAM behaviour in all configurations shows very similar efficiency and utilization, implying that there are almost constant memory requests to DRAM. This insight from Vulkan-Sim motivates future research for in-memory computing for ray tracing or solutions to reduce cache misses.

D. Energy

We use AccelWattch [32] to measure GPU power for our RT workloads and estimate power from RT units as described by Liu et al. [37]. The RT units average less than 1% of total GPU power since the most power intensive portion of ray tracing is accessing DRAM, accounting for 10%. The majority of power is dissipated as constant and static GPU power. Although not directly originating from RT units, energy can be reduced if RT unit performance improves because shorter execution times reduce energy from constant and static power.

E. Function Call Coalescing

Having more than one intersection shader in a workload can cause low SIMT efficiency in the GPU [45]. The `traceRayEXT` instruction calls different intersection shaders in line 6 and 9 of Algorithm 1 which causes thread divergence. FCC aims to increase SIMT efficiency by grouping the same intersection shaders together. To evaluate FCC, we added procedural cubes to RayTracingInVulkan and created a new scene RTV6, which contains both procedural spheres and cubes, with each shape having a different intersection shader. We tested FCC on a mobile GPU configuration in RTV6.

FCC improves average SIMT efficiency by 9% but decreases performance compared to baseline intersection table by 6%, shown in Figure 17. As described in Section VI-B, RTV6 has low SIMT efficiency. The low number of active threads are likely to hit the same procedural geometry. So the same intersection shader ID is added to the table and they don't cause extra thread divergence in Algorithm 1.

FCC has extra memory overhead compared to the baseline intersection table. Each thread with a new procedural intersection has to load the existing shader IDs in the coalescing table to check for a match. Upon a match, the corresponding thread mask is loaded. A new intersection is only added if the thread mask is not set. Consequently, FCC results in 11% more memory loads in the RT unit, which outweighs improvements in SIMT efficiency as the workload is memory bound. In RTV6, each warp requires 52 coalescing buffer entries, which is 13.4KB. In comparison, the baseline table needs 42 entries per warp, which is 10.9KB.

Even though RTV6 only has two intersection shaders, realistic workloads can have more intersection shaders and have any-hit shaders which will also be added to the coalescing table with intersection shaders. We expect the improvement of FCC to be larger in these cases as the thread divergence would be larger when shaders are executed in

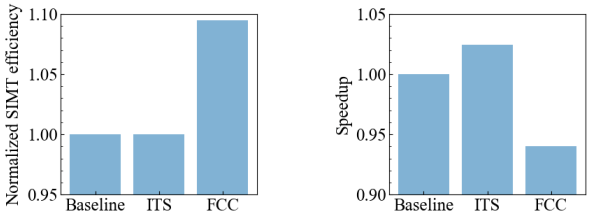


Figure 17: Speedup and SIMT utilization for FCC and ITS.

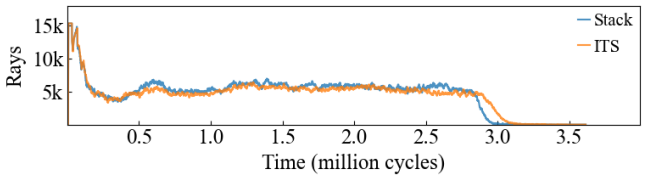


Figure 18: Comparison of combined RT unit occupancy for stack-based reconvergence and ITS.

Algorithm 1.

F. Independent Thread Scheduling

Compared to stack-based convergence, ITS only improves performance by up to 1% for our workloads because warps do not split on the lengthy `traceRayEXT` instruction. ITS performs best when warp splits are similar in length and both execute long latency instructions that create stalls, allowing ITS to schedule both splits to execute in parallel. However, unlike FCC, ITS aims to improve RT unit efficiency rather than SIMT efficiency in the GPU. Our workloads use simple closest-hit and miss shaders with more than 60% ALU instructions that do not benefit from ITS. Realistic workloads often feature more complex code where rays diverge based on their results and generate different `traceRayEXT` calls, such as in Quake II RTX [44]. We model this behaviour in a microbenchmark by injecting arbitrary warp divergence to AO rays in the EXT workload and find a 6% speedup with ITS, indicated in Figure 17 (right). We choose AO rays because they make up 59% of the rays in the workload. Also, they are incoherent, unlike primary rays which perform nearly equally well when executed serially due to better cache hits.

Ideally, ITS should halve execution time by perfectly scheduling two warp splits simultaneously instead of serially. Realistically, we find that RT unit resources are saturated and cannot always accommodate the second warp split, resulting in serial execution despite ITS. This effect appears in Figure 18 where ITS does not significantly increase the number of rays in the RT units because it is already executing the max number of warps. Increasing the max number of warps in each RT unit to fit these warp splits will negatively impact performance due to bottlenecks in the memory system.

The benefit we do observe in ITS is actually a result of better cache hits in the L1D and L2 caches. Even though warps may not necessarily be executed in parallel, ITS does alter the scheduling order by allowing warp splits to be scheduled simultaneously. This altered warp scheduling order results in slightly better warp occupancy in the RT units, which also creates better coalescing of memory requests between rays in each RT unit. Interestingly, because of the increased cache hits, there are fewer memory requests to the DRAM, causing lowered row buffer locality and bank level parallelism. In this microbenchmark, the average memory request latency reduces by around 7%, but the longest memory request latency increases, lengthening tail effects as observed in Figure 18.

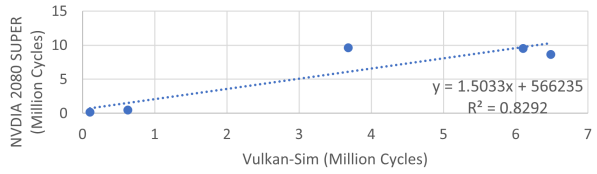
G. Correlation Study

To see if Vulkan-Sim's RT units can model the performance of ray accelerators found in real hardware, we tune our simulation configuration to match an NVIDIA RTX 2080 SUPER GPU in terms of cycle count on the benchmarks shown in Table IV. We start with the baseline configuration in Table III and match the clock and memory frequencies, SM count, cache size and number of RT cores from publicly available data online [5]. However Figure 19a shows this configuration is 1.5 times faster than the NVIDIA GPU in cycle count, indicated by the slope of the trendline. In Figure 19b we increased cache and dram latencies by referencing values from Khairy et al. [33] and Dalmia et al. [21] and also decreased the number of concurrent warps in the RT unit from 4 to 2, but the slope still remained at 1.5. Finally in Figure 19c, by decreasing the number of concurrent warps in the RT unit from 2 to 1, we manage to obtain a correlation graph with a slope of 0.88 and a correlation coefficient of 90% as our closest result.

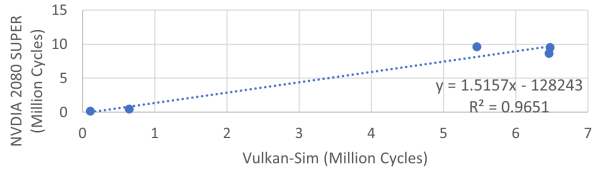
From the correlation plots, this may imply that NVIDIA's hardware only supports one warp in each RT Core. In Vulkan-Sim, to support multiple warps in the RT unit, the memory overhead per additional concurrent warp per RT unit is almost 2.4KB. This is mainly the memory needed for the ray buffer that consists of 32 rays per warp with each ray requiring (4B Ray ID + 32B Ray properties + 3 bit Ray status + 40B five entry short stack [54]), which is not an insignificant amount. Reducing this overhead may not be trivial and is be a potential research area to explore as increasing the number of concurrent warps in the RT unit can greatly increase ray tracing performance.

VII. Related Work

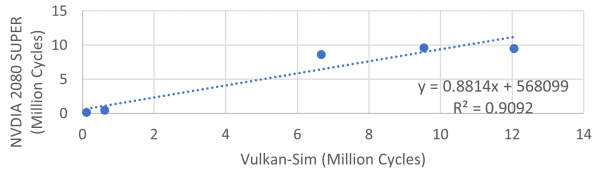
GPU / Graphics Simulators A recent simulator, Emerald [27], is a cycle level graphics simulator based on GPGPU-Sim and Mesa to support the OpenGL API. Emerald provides models for rasterization-based graphics hardware at a cycle level but only supports OpenGL and uses an older version of Mesa. Consequently, it does not support the Vulkan API or include models for ray tracing specific hardware. Liu et al. describe a simulator that models a generalized ray tracing



(a) Baseline config with matched parameters (4 warps in RT unit)



(b) Two warps in RT unit, Increased cache and DRAM latencies



(c) One warp in RT unit, Increased cache and DRAM latencies

Figure 19: Cycle Count Correlation verses RTX 2080 SUPER

accelerator [37], which we use as the foundation for our improved timing model. Their simulator does not support any existing ray tracing API like Vulkan and overlooks several components such as transforming rays between the TLAS and BLAS and storing hit points. GLTraceSim [48] is an OpenGL GPU simulator that replays traces to analyze memory accesses but does not model a particular GPU architecture. QSilver [49] is an older graphics simulator with an outdated simulation infrastructure. PBRT [46], is a popular graphics simulator for rendering ray traced images. While it can implement different software-based ray tracing algorithms, it lacks graphics hardware modeling used for hardware design.

Another category of GPU simulators focus on GPGPU applications. GPGPU-Sim [16], [33] is a cycle level GPGPU simulator that executes NVIDIA's virtual ISA, PTX, which Vulkan-Sim extends to include ray tracing specific hardware. GPUTejas parallelizes simulation at the threadblock level which can improve simulation speed.

Ray Tracing Accelerators Specialized accelerators for ray tracing have been explored in various works, each featuring hardware components targeting AS traversal and intersection test computation, which we model in Vulkan-Sim. Earlier works include SaarCOR [47] and RPU [56], both of which use Kd-tree acceleration structures in their implementation. More recently, Nah et al. proposed T&I engine [41], which we use to guide our timing model. RayCore [40] and SGRT [36] then build upon this, employing T&I cores to enable ray tracing on mobile devices. Other works such as Trax [52] and MIMD threaded multiprocessors (TM) [34] are MIMD processors capable of traversing incoherent rays. These accelerators all operate independently from the GPU. However, since rasterization is still crucial in real-time rendering, there is a disadvantage to decouple ray tracing from the existing GPU. RT Cores [18] from NVIDIA accelerate ray tracing from within the GPU, just as we model in Vulkan-Sim. Another field of hardware accelerated ray tracing focuses on different improvements. Kopta et al. [35] reduces energy usage and Shkurko et al. [51] reduces random memory accesses. Deng et al. [24] survey these works and acknowledge the importance of hardware accelerated ray tracing.

VIII. Conclusion and Future Work

This work presents Vulkan-Sim, a Vulkan ray tracing simulator. Vulkan-Sim integrates Mesa's Vulkan frontend with GPGPU-Sim to bring the first ray tracing architectural simulator driven by a state-of-the-art ray tracing API. Our functional simulation provides insight on how modern ray tracing APIs can be implemented in hardware, while the timing model provides cycle level simulation of a baseline ray tracing accelerator. This enables Vulkan-Sim to study ray tracing workloads in detail and explore accelerator design.

We also present two hardware optimization case studies that we evaluate in Vulkan-Sim: independent thread scheduling and function call coalescing. Independent thread scheduling improves RT unit efficiency by scheduling threads independently during control flow divergence, however it only provides up to 2% speedup for ray tracing. Our simulations show that while ITS provides the benefit of better cache hit rates, it does not significantly increase RT unit occupancy, and due to the increased cache hits, DRAM row buffer locality and bank level parallelism both decrease, canceling out the benefits that ITS provides. With many different shader stages in the Vulkan ray tracing API, executing different shaders on demand causes inefficiency due to warp divergence. By deferring shader execution, grouping up similar shader calls from different SIMT threads, and executing them together, functional call coalescing increases SIMT efficiency by 2%, however the memory overhead from accessing the shader IDs in the coalescing table causes a 6% slowdown instead.

For future work, we plan to support more Vulkan extensions for better compatibility and create a suite of ray tracing benchmarks that represent modern graphics applications.

Acknowledgment

The authors thank the anonymous reviewers for their feedback. We would also like to thank Jonathan Lew and Ningfeng Yang for their feedback on earlier drafts of this paper and Mabel Wang for help testing our artifact evaluation instructions. This research was funded in part by grants from Huawei Technologies. Tor M. Aamodt serves as a consultant for Huawei Technologies Canada Co. Ltd. and Intel Corp.

References

[1] NVIDIA RTX: List Of All Games, Engines And Applications Featuring GeForce RTX-Powered Technology. [Online]. Available: https://www.nvidia.com/en-us/geforce/news/nvidia-rtx-games-engines-apps/

[2] OpenGL Overview. [Online]. Available: https://www.khronos.org/opengl/

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

[4] Ray Tracing Rendering Software. [Online]. Available: https://www.autodesk.com/solutions/ray-tracing

[5] TechPowerUp - NVIDIA GeForce RTX 2080 SUPER. [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-rtx-2080-super.c3439

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

[7] Vulkan 1.2.162 Released With Ray-Tracing Support Promoted. [Online]. Available: https://www.phoronix.com/news/Vulkan-Ray-Tracing-Promoted

[8] Vulkan-Samples. [Online]. Available: https://github.com/KhronosGroup/Vulkan-Samples

[9] Vulkan Specification - Acceleration Structure. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-khr-extensions/html/vkspec.html#acceleration-structure

[10] Vulkan Specification - Ray Tracing. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-khr-extensions/html/vkspec.html#ray-tracing

[11] Vulkan Specification - Ray Traversal. [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-khr-extensions/html/vkspec.html#ray-traversal

[12] T. Aamodt, W. Fung, M. Martonosi, and T. Rogers, *General-Purpose Graphics Processor Architectures*. Morgan & Claypool Publishers, 2018.

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

[14] 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.

[15] 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.

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

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

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

[19] J. Choquette, O. Giroux, and D. Foley, "Volta: Performance and programmability," *IEEE Micro*, vol. 38, no. 2, pp. 42–52, 2018.

[20] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali, "Ray tracing for the movie 'cars'," in *IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 1–6.

[21] P. Dalmia, R. Mahapatra, and M. D. Sinclair, "Only buffer when you need to: Reducing on-chip gpu traffic with reconfigurable local atomic buffers," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2022, pp. 676–691.

[22] S. Damani, M. Stephenson, R. Rangan, D. Johnson, R. Kulkami, and S. W. Keckler, "Gpu subwarp interleaving," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2022.

[23] R. De Jong and A. Sandberg, "NoMali: Simulating a realistic graphics driver stack using a stub GPU," in *Proc. IEEE Symp. on Perf. Analysis of Systems and Software (ISPASS)*, 2016, pp. 255–262.

[24] Y. Deng, Y. Ni, Z. Li, S. Mu, and W. Zhang, "Toward real-time ray tracing: A survey on hardware acceleration and microarchitecture techniques," *ACM Computing Surveys (CSUR)*, vol. 50, no. 4, pp. 1–41, 2017.

[25] A. ElTantawy, J. W. Ma, M. O'Connor, and T. M. Aamodt, "A scalable multi-path microarchitecture for efficient GPU control flow," in *Proc. IEEE Symp. on High-Perf. Computer Architecture (HPCA)*, 2014, pp. 248–259.

[26] W. W. Fung, I. Sham, G. Yuan, and T. M. Aamodt, "Dynamic warp formation and scheduling for efficient gpu control flow," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2007.

[27] 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.

[28] J. Hasselgren, J. Munkberg, M. Salvi, A. Patney, and A. Lefohn, "Neural temporal adaptive sampling and denoising," *Computer Graphics Forum*, vol. 39, no. 2, 2020.

[29] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree GPU raytracing," in *Proc. ACM SIGGRAPH Symp. on Interactive 3D Graphics and Games (I3D)*, 2007, pp. 167–174.

[30] M. Işık, K. Mullia, M. Fisher, J. Eisenmann, and M. Gharbi, "Interactive monte carlo denoising using affinity of neural features," *ACM Transactions on Graphics (TOG)*, vol. 40, 2021.

[31] J. T. Kajiya, "The Rendering Equation," in *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH)*, 1986, p. 143–150.

[32] V. Kandiah, S. Peverelle, M. Khairy, J. Pan, A. Manjunath, T. G. Rogers, T. M. Aamodt, and N. Hardavellas, "AccelWattch: A power modeling framework for modern GPUs," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2021, pp. 738–753.

[33] 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.

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

[35] D. Kopta, K. Shkurko, J. Spjut, E. Brunvand, and A. Davis, "Memory considerations for low energy ray tracing," in *Computer Graphics Forum*, vol. 34, no. 1, 2015, pp. 47–59.

[36] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han, "SGRT: A mobile GPU architecture for real-time ray tracing," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2013, pp. 109–119.

[37] 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.

[38] G. Malhotra, S. Goel, and S. R. Sarangi, "Gputejas: A parallel simulator for gpu architectures," in *Int'l Conf. on High Performance Computing (HiPC)*, 2014, pp. 1–10.

[39] A. Marrs, P. Shirley, and I. Wald, Eds., *Ray Tracing Gems II*. Apress, 2021.

[40] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park, "RayCore: A ray-tracing hardware architecture for mobile devices," *ACM Transactions on Graphics (TOG)*, vol. 33, no. 5, pp. 1–15, 2014.

[41] J.-H. Nah, J.-S. Park, C. Park, J.-W. Kim, Y.-H. Jung, W.-C. Park, and T.-D. Han, "T&I engine: Traversal and intersection engine for hardware accelerated ray tracing," in *Proc. Int'l Conf. on Computer Graphics and Interactive Techniques in Asia (SIGGRAPH Asia)*, 2011, pp. 1–10.

[42] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, "Improving GPU performance via large warps and two-level warp scheduling," in *Proc. IEEE/ACM Symp. on Microarch. (MICRO)*, 2011, pp. 308–317.

[43] T. B. Nowicki and A. M. E. M. Eltantawy, "Methods and apparatuses for coalescing function calls for ray-tracing," patentus 17 008 437.

[44] NVIDIA, "Quake II RTX." [Online]. Available: https://github.com/NVIDIA/Q2RTX

[45] 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.

[46] M. Pharr and G. Humphreys, *Physically Based Rendering, Third Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2018.

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

[48] A. Sembrant, T. E. Carlson, E. Hagersten, and D. Black-Schaffer, "A graphics tracing framework for exploring CPU+GPU memory systems," in *Proc. IEEE Symp. on Workload Characterization (IISWC)*, 2017, pp. 54–65.

[49] J. W. Sheaffer, D. Luebke, and K. Skadron, "A flexible simulation framework for graphics architectures," in *Proc. ACM SIGGRAPH/EUROGRAPHICS Conf. on Graphics hardware (HWWS)*, 2004, pp. 85–94.

[50] K. Shkurko, T. Grant, E. Brunvand, D. Kopta, J. Spjut, E. Vasiou, I. Mallett, and C. Yuksel, "SimTRaX: Simulation infrastructure for exploring thousands of cores," in *Proc. Great Lakes Symp. on VLSI*, 2018, pp. 503–506.

[51] K. Shkurko, T. Grant, D. Kopta, I. Mallett, C. Yuksel, and E. Brunvand, "Dual streaming for hardware-accelerated ray tracing," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2017.

[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] Y. Sun, T. Baruah, S. A. Mojumder, S. Dong, X. Gong, S. Treadway, Y. Bao, S. Hance, C. McCardwell, V. Zhao, H. Barclay, A. K. Ziabari, Z. Chen, R. Ubal, J. L. Abellán, J. Kim, A. Joshi, and D. Kaeli, "Mgpusim: Enabling multi-gpu performance modeling and optimization," in *Proc. IEEE/ACM Int'l Symp. on Computer Architecture (ISCA)*, 2019, p. 197–209.

[54] K. Vaidyanathan, S. Woop, and C. Benthin, "Wide BVH Traversal with a Short Stack," in *Proc. ACM Conf. on High Performance Graphics (HPG)*, 2019.

[55] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[56] S. Woop, J. Schmittler, and P. Slusallek, "RPU: a programmable ray processing unit for realtime ray tracing," *ACM Transactions on Graphics (TOG)*, vol. 24, no. 3, pp. 434–444, 2005.

APPENDIX

A. Abstract

This artifact provides the complete source code to Vulkan-Sim along with the Vulkan binary traces used for evaluation in Table IV. We describe the installation procedure and workflow for the simulator to reproduce our results in Section VI. We also provide additional instructions on how to generate additional Vulkan binary traces to use with Vulkan-Sim.

B. Artifact check-list (meta-information)

- **Program:** Two programs provided: Vulkan-Sim, Trace Runner
- **Compilation:** gcc/g++, ninja, meson, cmake, cuda
- **Run-time environment:** Ubuntu 20.04
- **Hardware:** Intel CPU with integrated graphics required only if generating Vulkan binary traces. No CPU restrictions if only executing traces. Additionally the system may require a decent amount of memory. Some runs use over 5GB each.
- **Metrics:** Execution time, cache access breakdown and roofline plot
- **Output:** Simulation statistics
- **Experiments:** Provided scripts and some manual steps
- **How much disk space required (approximately)?:** Docker Image: 9.5 GB, Simulator tar: 5 GB
- **How much time is needed to prepare workflow?:** 1 to 2 hours
- **How much time is needed to complete experiments (approximately)?:** 2-3 days (in parallel), about a month (sequentially)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Archived (provide DOI)?:** Docker Image: https://doi.org/10.5281/zenodo.6941618 Simulator tar: https://doi.org/10.5281/zenodo.6929547

C. Description

1) How to access: Vulkan-Sim is available on Github, organized into three repos: Vulkan-Sim, Mesa, and Trace runner.
https://github.com/ubc-aamodt-group/vulkan-sim
https://github.com/ubc-aamodt-group/mesa-vulkan-sim
https://github.com/ubc-aamodt-group/trace-runner-vulkan-sim

We provide a docker image of our simulator along with benchmark traces used in this paper on Docker Hub or Zenodo: https://doi.org/10.5281/zenodo.6941618.

We also provide a tar of the simulator for those that want to generate their own traces to use with the simulator on Zenodo: https://zenodo.org/record/6941190

2) Hardware dependencies: There are no specific hardware requirements if only executing benchmark traces with Vulkan-Sim's trace runner. However, running a Vulkan application natively with Mesa or generating traces with the simulator requires an Intel CPU with integrated graphics. So far we have tested this on an Intel i7-7700HQ laptop CPU.

3) Software dependencies: We run our simulator on Ubuntu 20.04 and have not tested it on other platforms. Our application requires several dependencies to be installed:

- gcc/g++-9
- CUDA Toolkit 10 or 11
- Embree v3.12.0 or above (Artifact includes 3.13.4)
- VulkanSDK 1.2.162 or preferably newer

Install the remaining dependencies with the following:

```
sudo apt install -y build-essential git ninja-build meson libboost-all-dev
    xutils-dev bison zlib1g-dev flex libglu1-mesa-dev libxi-dev libxmu-
    dev libdrm-dev llvm libelf-dev libwayland-dev wayland-protocols
    libwayland-egl-backend-dev libxcb-glx0-dev libxcb-shm0-dev
    libx11-xcb-dev libxcb-dri2-0-dev libxcb-dri3-dev libxcb-present-dev
     libxshmfence-dev libxxf86vm-dev libxrandr-dev libglm-dev
```

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

4) Models: We include benchmark traces of the evaluated Vulkan workloads in this paper. These can be executed with the provided Vulkan trace runner using the docker image. We also provide access to the RayTracingInVulkan repo corresponding to RTV5 and RTV6 in the paper.

D. Installation

We provide two sets of experiments / artifacts described in separate sections here. The first one is a docker image containing Vulkan-Sim and benchmark traces used in this paper. The second artifact is a tar of the simulator to guide the user through trace generation with a sample workload. Instructions are also provided in each artifact.

1) Docker Image With Traces: Install docker, then pull the image using the command below from docker hub.

```
docker pull mohammadrezasaed/vulkan-sim
```

Alternatively, the docker image is uploaded to Zenodo. Use the following commands to unzip and load the image.

```
unzip vulkan-sim.zip
docker load -i vulkan-sim.tar
```

2) Trace Generation: Instructions here are also provided in the artifact along with additional troubleshooting tips.

1) Extract vulkan-sim-artifact.tar.gz and navigate to the extracted vulkan-sim-artifact/ folder.
2) Set environment variables with the following.

```
# Change to your own CUDA installation path
export CUDA_INSTALL_PATH=/usr/local/cuda
source embree-3.13.4.x86_64.linux/embree-vars.sh
```

3) Build vulkan-sim + mesa.

```
cd vulkan-sim/
source setup_environment debug
cd ../mesa/
meson --prefix="${PWD}/lib" build/
meson configure build/ -Dbuildtype=debug -D b_lundef=false
# Please ignore the build error about -lcudart and proceed on
ninja -C build/ install
export VK_ICD_FILENAMES=${PWD}/lib/share/vulkan/icd.d/
    intel_icd.x86_64.json
```

```
cd ../vulkan-sim/
make -j
cd ../mesa/
ninja -C build/ install
```

4) Build the Vulkan RT trace runner.

```
cd ../vulkan_rt_trace_runner/
make
```

E. Experiment workflow

Similar to installation, this section covers the workflow for the Docker image first, followed by trace generation.The docker can be used to reproduce the results in the paper.

1) Docker Image With Traces: The scripts for running the workloads and ploting the results are on Zenodo: https://doi.org/10.5281/zenodo.6941618.

The folder *runs* includes subfolders for each config which in turn include subfolders for each workload. These folders include two files, *gpgpusim.config* which is the GPU configuration and *run.sh* which is a single command to run the docker container for the current configuration and workload. The file *run_all.sh* is a simple python script that goes inside each folder and executes *run.sh* in the background, so it runs all configs and workloads in parallel. Change this script to account for the resources in your system or execute each *run.sh* file manually. *run.sh* files need to be executed from the directory they exist in. After all the executions have finished, run the python script *plot_all.py*. This script reads the output from each folder and plots Figures 11, 13, and 14 of this paper.

2) Trace Generation: We provide RTV6 for this trace dumping example.

First, modify Vulkan-Sim's dumping functions to match the application. These changes are already present in the artifact.

1) In vulkan-sim/src/cuda-sim/vulkan_ray_tracing.cc:106, change `bool use_external_launcher` to false.
2) Since RTV6 does not skip any descriptor sets, please comment out the `continue;` in both lines 2108-2114 and 2193-2199 of vulkan-sim/src/cuda-sim/vulkan_ray_tracing.cc
3) Compile vulkan-sim in vulkan-sim/ with `make -j`

Next, compile and run RTV6 to dump its trace.

1) Download RTV6 from our repo to a local folder

```
git clone https://github.com/ubc-aamodt-group/
    RayTracingInVulkan.git
```

2) Compile RTV6

```
cd RayTracingInVulkan/
sudo apt-get -y install cmake curl unzip tar libxi-dev libxinerama-
    dev libxcursor-dev xorg-dev
./vcpkg_linux.sh
./build_linux.sh
```

3) Copy gpgpusim.config to the binary directory

```
cp <vulkan-sim-root>/vulkan_rt_trace_runner/gpgpusim.config
    build/linux/bin/.
```

4) Run RTV6

```
cd build/linux/bin/
./RayTracer --scene 6 --height 320 --width 448
```

5) Wait for "Trace dumped" to show up in the terminal. Traces are in <vulkan-sim-root>/mesa/gpgpusimShaders/. You can terminate the program afterwards.
6) Copy the traces to another folder

```
# Change <vulkan-sim-root> to your own path
mkdir <vulkan-sim-root>/vulkan_rt_trace_runner/RTV6-trace
cd <vulkan-sim-root>/mesa/gpgpusimShaders/
cp * <vulkan-sim-root>/vulkan_rt_trace_runner/RTV6-trace
```

To replay the generated traces that are dumped from RTV6, use Vulkan-Sim's trace runner.

1) In vulkan-sim/src/cuda-sim/vulkan_ray_tracing.cc:106, change bool use_external_launcher to true.
2) Compile vulkan-sim with `make -j`
3) Move to vulkan_rt_trace_runner/ and `make`
4) Run the trace with the following command:

```
./vulkan_rt_runner RTV6-trace/ RTV6
# General Usage
./vulkan_rt_runner <path_to_trace_folder> <workload>
```

The workload argument is used to communicate with Vulkan-Sim about application specific settings for replaying the trace. There are none for RTV6.

F. Evaluation and expected results

1) Docker Image With Traces: Running the provided scripts in the Docker image will collect data to generate plots that closely match the figures in Section VI.

2) Trace Generation: Completing trace generation for RTV6 will yield application specific traces that can be re-simulated on any system using the Vulkan-Sim trace runner and bypass the need for Intel integrated graphics.

G. Experiment customization

The experiment can be customized by changing the parameters found in the `gpgpusim.config` file. Vulkan-Sim supports simulation of native Vulkan binaries by executing them on a native system with Intel integrated graphics. The simulator also supports generating traces of a Vulkan application and then replaying them with the provided trace runner.