Data-Driven Analysis and Design of Vulkan Ray-Tracing Applications using Automatic Instrumentation

by

David Pankratz

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

© David Pankratz, 2021

Abstract

Modern graphics Application Programming Interfaces (APIs) provide first-class support for ray tracing. Hardware vendors implement drivers for the graphics API including a black-box compiler. The black-box compiler creates architecture-specific binaries that leverage ray-tracing hardware acceleration. Ray-tracing support in modern APIs allows all geometry and shaders to be specified for a single execution. Thus, ray tracing is more complex and difficult to reason about than rasterization, a traditional rendering method. Ray-tracing developers must contend with the unknowns of an inscrutable GPU binary and a monolithic execution model. The increase in complexity from rasterization to ray tracing has not been accompanied by commensurate tooling.

This thesis first presents Vulkan Vision (V-Vision). V-Vision is a framework for developing instrumentation passes for shaders in the Vulkan graphics API. V-Vision handles the commonalities of generating, analyzing, and presenting instrumentation data. Specifically, V-Vision provides instrumentation primitives to capture a complete inter-shader and intra-shader ray-tracing execution trace. Instrumentation utilities implemented using V-Vision are operating-system, vendor, and architecture agnostic. V-Vision does not require source-code modification or recompilation. V-Vision's out-of-the-box instrumentation utilities demonstrate the ability to gather fine-grained execution data. Moreover, V-Vision's utilities are capable of measuring microarchitectural effects, such as independent thread scheduling. The execution data enables limit studies at hardware, compiler, and application levels. V-Vision's annotated shader and

heatmap representations enable productive debugging and profiling. V-Vision has been accepted into the MindInsight tool family.

Next, this thesis presents RayScope. RayScope automatically captures application-agnostic ray-tracing execution data and geometry data from Vulkan applications. RayScope provides an interactive visualizer, populated using the ray tracing and geometry data. Therefore, RayScope can be understood as a set of tools that enables understanding, debugging, profiling, and designing through visualization of application execution data. RayScope implements an instrumentation pass and analysis using V-Vision but also implements Vulkan-API call instrumentation. RayScope's outputs are human-readable to encourage integration with other visualization and debugging tools. RayScope assisted in identifying longstanding bugs in Vulkan ray-tracing applications. RayScope further assisted in uncovering poorly defined minimum collision distances causing wasted computations in multiple ray-tracing applications. RayScope also helped identify geometry construction problems causing visual artifacts and wasteful computation in the well-known model Sponza. Finally, RayScope automatically identified a misconfiguration of Vulkan geometry flags and recommended a solution for one ray-tracing application. Applying the recommendation results in a reduction of 96.8% of any-hit shader executions.

The level of information provided to the developer has a large impact on the quality of the application that they develop. Changes motivated by the information provided by V-Vision and RayScope are often minimal but have tangible implications for performance and correctness. The effectiveness of V-Vision and RayScope indicates that tooling, and the knowledge it provides, was lacking for real-time hardware-accelerated ray tracing in Vulkan. The work presented in this thesis improves the tooling landscape by releasing V-Vision and RayScope as open-source projects, and improves the body of knowledge by sharing common pitfalls in real-time hardware-acceleration ray tracing.

Preface

Chapter 3 of this thesis is published as **D. Pankratz**, T. Nowicki, A. ElTantawy, J. N. Amaral, "Vulkan Vision: Ray Tracing Workload Characterization using Automatic Graphics Instrumentation", *CGO 2021: Proceedings of the 19th ACM/IEEE International Symposium on Code Generation and Optimization*, in February 2021. I presented this work virtually at CGO on March 1st, 2021. To develop V-Vision I devised, prototyped, implemented, and evaluated the project. T. Nowicki and A. ElTantawy proposed many studies that were made possible by V-Vision and I formulated others. For each study I devised, prototyped, implemented, and evaluated an instrumentation utility. I planned and drafted the manuscript under the guidance of A. ElTantawy. A. ElTantawy characterized the interactions and features of V-Vision and created the main architecture figure. T. Nowicki authored the background section discussing details of Vulkan and ray tracing. All authors collaborated in editing and polishing the writing of the final manuscript. J. Nelson Amaral ensured a high-quality manuscript with his feedback and edits.

Chapter 4 of this thesis has been submitted for review toward publication as **D. Pankratz**, T. Nowicki, A. ElTantawy, J. N. Amaral, "RayScope: Interactive Visualizations of Vulkan Ray-Tracing Applications Using Automatic and Application-Agnostic Instrumentation". I conceptualized, prototyped, implemented, and evaluated the tool. A. ElTantawy and T. Nowicki assisted in determining an appropriate venue and drafting the manuscript. T. Nowicki and A. ElTantawy provided valuable feedback on the visualizations and the manuscript. T. Nowicki wrote the background section and part of the related work section. J. Nelson Amaral guided the research evaluation methodology and improved the cohesiveness of the writing.

Acknowledgements

I wish to thank my manager at Huawei, Ahmed ElTantawy, for his immense and ongoing support. I feel very lucky to have worked under and learned from Ahmed, and I greatly admire his technical expertise, high-level vision, and leadership. I am honoured that Ahmed gave me the opportunity to present to the Huawei hardware architecture team and the Khronos Vulkan and SPIR groups.

I would like to thank my supervisor at Huawei, Tyler Nowicki, for his advice and enlightening conversations. I learned a considerable amount in a short time thanks to Tyler's encyclopedic knowledge of Vulkan, GPU architectures, compilers, and computer graphics.

I also want to thank my supervisor at the University of Alberta, Dr. J. Nelson Amaral, for his guidance and for the opportunities he afforded me as his graduate student. I greatly appreciate his support and blessing in exploring a wide range of opportunities and collaborations.

Finally, I want to thank Wyatt Praharenka for his solidarity and friendship. I greatly admire Wyatt's lateral thinking and work ethic. I am thankful to have worked with him in my first foray into compilers.

This research has been funded in part by the University of Alberta Huawei Joint Innovation Collaboration (UAHJIC) and Graduate Research Assistantship Fellowship (GRAF).

Contents

List of Tables

List of Figures

List of Acronyms

AABB Axis Aligned Bounding Box.

AO Ambient Occlusion.

API Application Programming Interface.

AS Acceleration Structure.

BLAS Bottom-Level Acceleration Structure.

BVH Bounding Volume Hierarchy.

CFG Control Flow Graph.

CUDA Compute Unified Device Architecture.

FPS Frames Per Second.

GLSL OpenGL Shading Language.

GPU Graphics Processing Unit.

IR Intermediate Representation.

ITS Independent Thread Scheduling.

PC Program Counter.

SBT Shader Binding Table.

SIMD Single Instruction, Multiple Data.

SIMT Single Instruction, Multiple Thread.

SM Streaming Multiprocessor.

SPIR-V Standard Portable Intermediate Representation.

TLAS Top-Level Acceleration Structure.

Chapter 1

Introduction

Performance and quality of real-time graphics applications—gaming, as well as augmented and virtual reality—have a direct impact on end-user experience and are therefore considered as a key differentiator in several markets, including gaming consoles, mobile devices, and high-end desktops. To achieve high performance, the process of translating a scene description into an image on a screen has long been standardized in graphics Application Programming Interfaces (APIs)—such as Direct3D and OpenGL [5], [61]—as a GPU kernel, referred to as a graphics pipeline. Characterizing the performance bottlenecks of a graphics pipeline is challenging because the graphics programming model is hardware agnostic, making it more obscure than a low-level programming model such as Compute Unified Device Architecture (CUDA) [45]. CUDA application developers benefit from numerous program characterization tools using profiling and instrumentation [46], [52], [59]. The infrastructure for graphics applications is less developed, but the need to create high quality and performant GPU code remains.

In comparison to traditional software, where call-graph traces reveal the flow of execution, the implicit data and control flow in a graphics pipeline is obscure to application and system developers. Inscrutability is compounded by recent extensions, such as ray tracing, with more complex pipelines. Thus, system developers have less support to improve performance in a graphics pipeline. The programmable shaders significantly influence performance, yet the run-time behaviour (e.g. execution trace, control flow, hotspots) of these

shaders is not provided. There is a need for instrumentation tools to analyze graphics pipelines. The efficacy of instrumentation has been recognized in the compute domain of GPUs that use APIs such as CUDA [45] and OpenCL [38].

Rasterization, the traditional method for real-time rendering, renders each object in the scene separately. In contrast, ray tracing renders scenes by simulating the paths of millions of rays of light moving through a scene to the eye or to the camera. In ray tracing, visual quality is directly related to the number of rays traced by the application. The final pixel value is an accumulation of indirect light that originated from multiple light sources and was reflected or refracted, potentially many times, on surfaces in the scene. As a consequence of this complex process, ray tracing is prone to hidden functional and performance pitfalls.

Ray tracing as a rendering technique has been around for decades with a wide range of applications, algorithms, and implementations. Ray-tracing pitfalls can be identified by visual or automatic inspection of the ray paths. However, until recently it was not practical to develop a standard approach for ray-path collection that works across applications and hardware platforms. Over time, ray-tracing visualizers adopted an approach requiring manually inserted profiling calls [15], [28], [53]. These techniques are also only applicable to CPU-based ray tracers. Therefore, a need exists to bring debugging capabilities to GPU-based ray tracers to provide recommendations and insights to developers, resulting in improved productivity and better applications.

The sophisticated software-hardware stack present in Vulkan applications comes with many unknowns. Runtime behaviour varies by application, operating system, driver, vendor, and architecture. To contend with these challenges, game studios and game engines create custom profiling and debugging solutions. Further, hardware vendors develop custom solutions that are only compatible with certain architectures. These approaches are admirable, but fragmented. Standardized and open-source ecosystems like Vulkan have great potential for unifying these efforts. Moreover, specialized tooling can be created based on first-class features in Vulkan, such as ray tracing, to automatically provide recommendations and insights to developers.

The contributions of this thesis are as follows:

1. Vulkan Vision (V-Vision), a framework for shader instrumentation in Vulkan applications. It extends Vulkan validation layer instrumentation capabilities to enable meaningful performance and behaviour-centric instrumentation. V-Vision is open source under MindInsight at `https://gitee.com/mindspore/mindinsight`.

2. A set of instrumentation utilities enabling automatic out-of-the-box instrumentation of pipelines in Vulkan applications. These utilities assist Vulkan application developers in profiling and debugging their applications. The utilities also facilitate limit studies of applications, compilers, and hardware to guide their design.

3. An analysis of ray-tracing applications using V-Vision. V-Vision's data collection and visualization provide an unprecedented view into the runtime behaviour of GPU ray tracing that manifests as actionable feedback for developers.

4. RayScope, an open-source platform-independent tool with a workflow that automatically captures detailed ray-tracing information and features an interactive visualizer to display this information. RayScope's visualizer accurately represents procedural geometry using runtime execution information that prior state-of-the-art tools approximate.

5. A characterization of application-independent geometry information, ray information, and ray-geometry interaction information in the Vulkan API. RayScope automatically collects this ray-tracing information using Vulkan API and shader instrumentation. RayScope emits the ray and geometry data as human-readable files to encourage use with other visualization tools.

6. An automatic analysis of the ray-tracing information that enables Rayscope to produce performance recommendations for common issues, such as Vulkan API misuse and poor dynamic interactions of rays with geometry.

For example, applying one of RayScope's recommendations to fix Vulkan API misuse results in a 15.71% FPS increase on a GTX 1660ti and a 3.15% FPS increase on a RTX 3080.

7. Case studies that demonstrate that RayScope is effective in assisting debugging and supporting application-design tasks. These case studies demonstrate that RayScope: produces visualizations that allow long-standing bugs to be quickly identified; provides effective performance recommendations resulting in tangible performance benefits; and allows emergent issues with well-studied models, such as Sponza, to be identified.

Chapter 2

Background

Rasterization is the preeminent real-time rendering technique. The reason for rasterization's popularity is that it is readily hardware accelerated and therefore performant. However, rasterization approximates every pixel and requires techniques like anti-aliasing to produce a high-quality image. Rasterization does not account for indirect effects of light and often relies on precomputing shadow maps based on a particular light position. Fundamentally, rasterization operates in the screen space and discounts the effect of light on objects outside the current view [9].

Ray tracing is a computationally expensive rendering technique that improves the visual quality of graphics applications by implementing sophisticated interactions of light. Ray tracing accounts for the effect of objects on light, including objects that are culled by rasterization.

Figure 2.1 shows an image rendered using ray tracing and a visualization of the ray paths to generate the image. The image in the right of Figure 2.1 shows a clear sphere rendered using ray tracing. The image in the left of Figure 2.1 shows a visualization of a subset of ray paths proceeding through the clear sphere. The orange rays collide with the sphere. At this point, the ray-tracing application must determine the result of the collision. In this case, the application determines the angle of refraction for each of the orange rays. The application continues the ray paths by launching the pink rays through the inside of the sphere. The pink rays collide with the back of the sphere, and the application again determines the angle of refraction to produce the

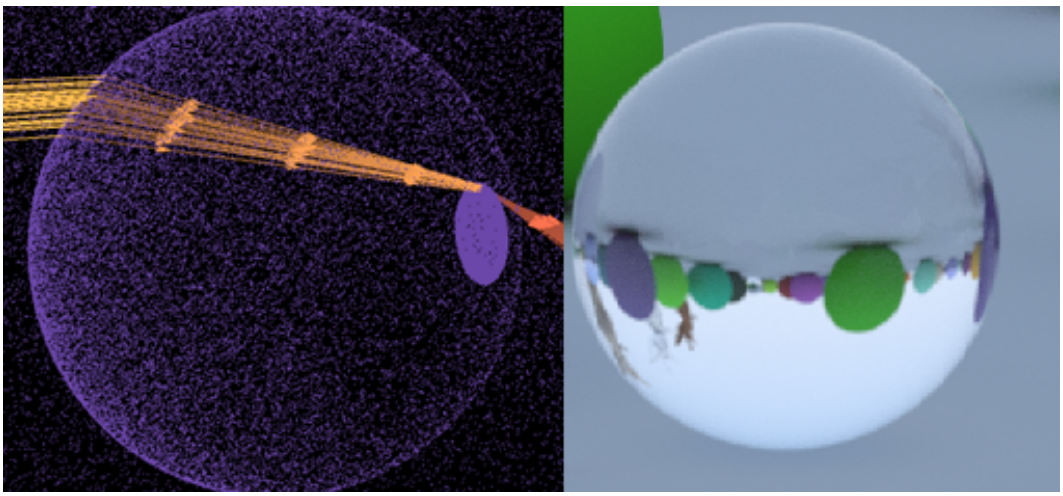

Figure 2.1: **Left:** Ray paths of backwards ray-tracing implementation. Rays are travelling left-to-right, and each colour represents a distinct segment in the ray path. The ray-tracing implementation determines ray behaviour at each collision site based on type of material and the desired visual effect. In this example, the rays are refracted after each collision because they are colliding with a clear sphere. **Right:** Image of a clear sphere rendered using backwards ray tracing. Note that the sphere inverts the image because rays that enter at the top of the sphere are directed to the bottom and rays that enter at the bottom are directed to the top. The effect of the sphere on the rays is analogous to the human eye's effect on light.

purple rays. This example illustrates how ray tracing emulates the physical behaviour of light. More generally, ray-tracing applications decide on the set of materials and visual effects that they will support. Each effect or material may be modelled differently in terms of ray behaviour. For example, the green sphere shown in the background of the image in the right of Figure 2.1 has a different material than the clear sphere in the foreground. The implementation may decide that rays that collide with the green sphere bounce off the sphere for indirect lighting or alternatively terminate the ray.

Ray-tracing applications use acceleration structures to elide unnecessary ray-intersection tests. Acceleration structures are recursive trees that partition space. Acceleration structure nodes have the property that if a ray misses the parent volume, the ray must also miss all of the childrens' volumes. For the purposes of this thesis, the acceleration structure[1] refers to a Bounding Volume Hierarchy (BVH), as that is the current industry standard [54].

Ray tracing can be performed online or offline. In offline ray tracing, indirect lighting is achieved using Monte-Carlo sampling of ray paths [53]. This approach has considerable latency and thus is not applicable to real-time ray tracing. Advances in machine-learning-based denoising have enabled indirect lighting in real-time ray tracing [8].

In anticipation of Graphics Processing Unit (GPU) vendors supporting real-time ray tracing, ray-tracing interfaces have been standardized in modern graphics APIs [5], [19]. Real-time ray tracing in modern applications is based on backwards ray tracing [6]. In this formulation of ray tracing, rays begin at the camera and are shot through all the pixels of the screen. Conceptually, shooting rays through each pixel determines what the pixels *see* and thus renders a complete image. Rays mimic the behaviour of physical light, and because of this, ray tracing can generate photorealistic images.

[1]Acceleration structure in Vulkan parlance actually refers to a two-level scene graph describing geometry and its instances. BVHs are constructed in a vendor-specific black-box driver based on the two-level scene graph.

2.1 Instrumentation

Ray-tracing applications' visual quality depends on the number of rays that are traced and therefore the performance of the implementation. However, implementing and optimizing ray-tracing application is a challenging process. Instrumentation of ray-tracing applications has been shown to be effective in assisting the developer in understanding the ray-tracing implementation and detecting issues [15], [28], [53].

Instrumentation is the process of inserting code into an existing program, typically to gain meta-information about the program's execution. Meta-information about program execution is valuable because it informs developer decisions. Developers may use instrumentation outputs to assist in debugging, profiling, or to understand the dynamic behaviour of their program. Instrumentation may be performed manually [30], ahead of time [34], or dynamically [59]. Instrumentation can be added to the source code [30], to library procedure calls [55], in the compiler [52], or in the executable binary [59]. Instrumentation that is automatically inserted can study aspects of program execution that would be prohibitively time consuming to study with manual instrumentation. For example, determining the runtime instruction mix of a binary execution requires, at the very least, inserting sophisticated assembly in every basic block [59].

Figure 2.2 shows an example of instrumentation affecting the performance of a program. In the first copy of the loop, the number of iterations is known at compile time. The compiler deduces that j will equal 100 following the loop and that running the loop is unnecessary. The second copy of the loop shows the instrumented version of the program. The function call `Instrument` has side effects, and the compiler can no longer safely remove the loop. Executing the instrumented version of the program will take significantly longer since the loop must be fully evaluated. Therefore, adding instrumentation may have non-trivial effects on the program's performance. Note that program semantics are not changed, so the variable j will be set correctly in both loops.

The effect on down-stream compilers is important when instrumenting an

```
1  // Copy 1
2  int j = 0;
3  for (int i = 0; i < 50; i++){
4    // compiler removes loop
5    j += 2;
6  }
7
8  // Copy 2
9  int j = 0;
10 for (int i = 0; i < 50; i++){
11   // compiler cannot remove loop
12   Instrument();
13   j += 2;
14 }
```

Figure 2.2: The potential impact of instrumentation on downstream compiler transformations. In this example, the loop spanning lines 3-6 is evaluated statically by the compiler. The instrumentation call on line 12 prevents the same loop from being evaluated statically. This issue can be mitigated by instrumenting after optimization passes.

application's source code. One method to mitigate the challenge shown in Figure 2.2 is to add instrumentation after the compiler has run its optimization passes. Instrumenting after the compiler has run its passes has the downside of being architecture specific. Therefore, when gathering data with instrumentation it is important to consider whether architecture agnosticism or detailed performance measurements are more important.

2.2 GPU Execution Model

Graphics applications are naturally parallel because each stage in the graphics pipeline is executed over a multi-dimensional buffer of independent elements, allowing each element to be evaluated by its own thread. Post-processing effects, such as convolution, that have dependencies between elements are typically implemented using a compute pipeline. For example, each pixel in the frame is independent, allowing it to be evaluated by separate threads running in parallel. The independent property of work items in a graphics pipelines naturally extends to rays in ray tracing. Ray tracing can therefore be

accelerated using a GPU. However, ray tracing poses additional challenges to GPU-based execution. Incoherent ray bounces and BVH traversal algorithms cause disparate work allocation and memory accesses between threads. These challenges have been partially mitigated using a dedicated hardware unit for ray traversal [47].

Threads are grouped into *warps*—also known as subgroups—where threads in each warp execute in lockstep on different data on Single Instruction, Multiple Data (SIMD) hardware units. This execution paradigm reduces instruction fetching, decoding, and scheduling overheads and enables coalesced memory accesses. Control flow divergence reduces the efficiency of Single Instruction, Multiple Thread (SIMT) execution due to threads becoming inactive in portions of the execution path. In the worst case of divergence, the execution of threads in a warp is completely serialized. The utilization of SIMD hardware units can be assessed using SIMT efficiency, the average percentage of active threads per warp.

2.3 Vulkan

Similar to recent low-level graphics APIs, such as Metal [3] and DirectX 12 [36], Vulkan gives the application full control over the GPU devices and their resources [31]. A thorough review of Vulkan is not within the scope of this discussion. This section discusses the API and several key interfaces that relate to ray tracing, including ray-trace commands, validation layers, the ray-tracing pipeline, data collection by shader instrumentation, the acceleration structure, memory buffers, and collection of ray-tracing information.

The following steps initialize Vulkan for rendering: creating an instance of Vulkan, identifying a GPU for rendering, integrating with the windowing system, initializing at least one render pass with a command buffer, and creating a frame buffer. Vulkan applications populate buffers for each geometric object with the object's geometry data, such as vertices and indices for triangular geometry. A rendering pipeline is created by specifying its shaders and their variables. Pipelines include the Vulkan commands that trigger the GPU to

render the scene. To render an image, applications bind the required memory buffers and schedule pipelines.

Vulkan requires Standard Portable Intermediate Representation (SPIR-V) for each programmable shader stage, and the driver is responsible for just-in-time compiling SPIR-V to machine code. SPIR-V is generated by compiling OpenGL Shading Language (GLSL) shaders, although other shader languages can also be compiled to SPIR-V. GLSL specifies the built-in variables and functions available within each shader stage.

To reduce runtime overhead, the Vulkan driver performs little error checking. The expensive task of validating each API call occurs outside the driver within a shared library, known as a validation layer. The validation layer is invoked by the driver twice: once before each Vulkan API call, to evaluate the function arguments, and once after the call, to evaluate the result. The source code of the Vulkan application is not required. The method for specifying and enabling a validation layer differs slightly for each OS but usually requires one or more environment variables and a configuration file.

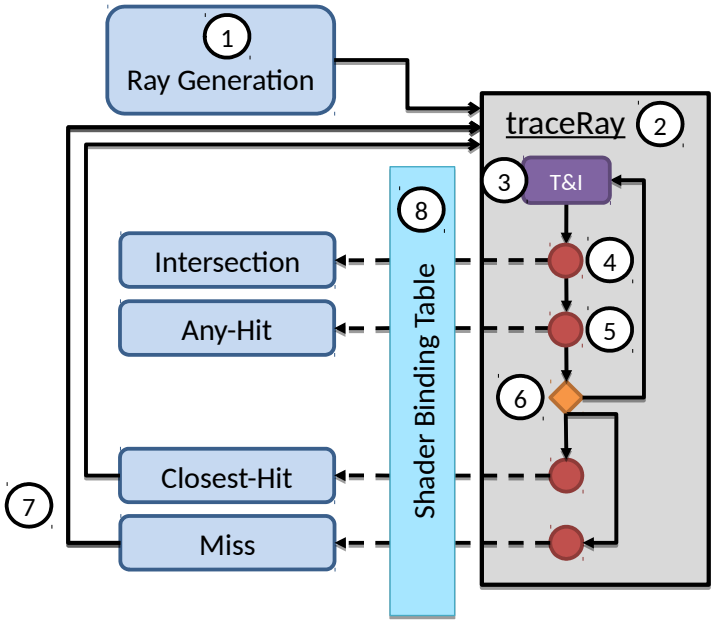

Figure 2.3: Ray-tracing pipeline in the Vulkan ray-tracing extension.

Vulkan Ray-Tracing Pipeline: The Vulkan ray-tracing pipeline, shown in Figure 2.3, is executed by all threads in a warp. It is distinct from the raster

graphics pipeline that consists of vertex, tessellation, geometry, and fragment shaders. The ray-tracing pipeline is recursive and nonlinear, unlike the linear execution of the raster graphics pipeline.

Execution of the ray-tracing pipeline begins at the ray generation shader ①. The ray generation shader creates new rays and traces them through a scene of 3D objects by invoking `traceRay` ②. A `traceRay` call is translated by a mixture of compiler transformations and hardware acceleration into the flow described next.

First, `traceRay` performs traversal and intersection ③ given the ray origin, ray direction, and minimum intersection distance as function arguments. Traversal and intersection takes a ray definition and traverses the acceleration structure to determine ray collisions. An object may be defined with an Axis Aligned Bounding Box (AABB) that encloses a procedural shape. If a collision with an AABB is detected, then an intersection shader is invoked by `traceRay` ④. Other objects, composed of triangles, may invoke an any-hit shader upon collision ⑤. The condition ⑥ checks whether all intersections have been processed. In the case that there are remaining intersections, steps ③, ④, ⑤ are repeated. In the case that all intersections have been processed, this may have happened in one of two ways: at least one collision occurred and the closest-hit shader is invoked, or there were zero intersections and the miss shader is invoked. The miss and closest-hit shaders may perform a recursive call to `traceRay` ⑦. Ray collisions trigger a lookup in the Shader Binding Table (SBT) ⑧, to determine what shader, if any, is associated with the collision.

Figure 2.4 shows a characterization of `traceRay's` operation. The intersection shader is executed to evaluate the validity of the ray intersecting with a procedural object. The any-hit shader is executed to re-evaluate the validity of the intersection. The intersection and any-hit shaders are classified under conditional geometry because they dynamically determine validity of intersections with geometry, based on a programmable condition. If the set of confirmed intersections is nonempty, the closest-hit is invoked; otherwise a miss shader is invoked. These shaders receive the final result of the `traceRay` and thus determine whether the ray should continue. Thus the closest hit and miss are

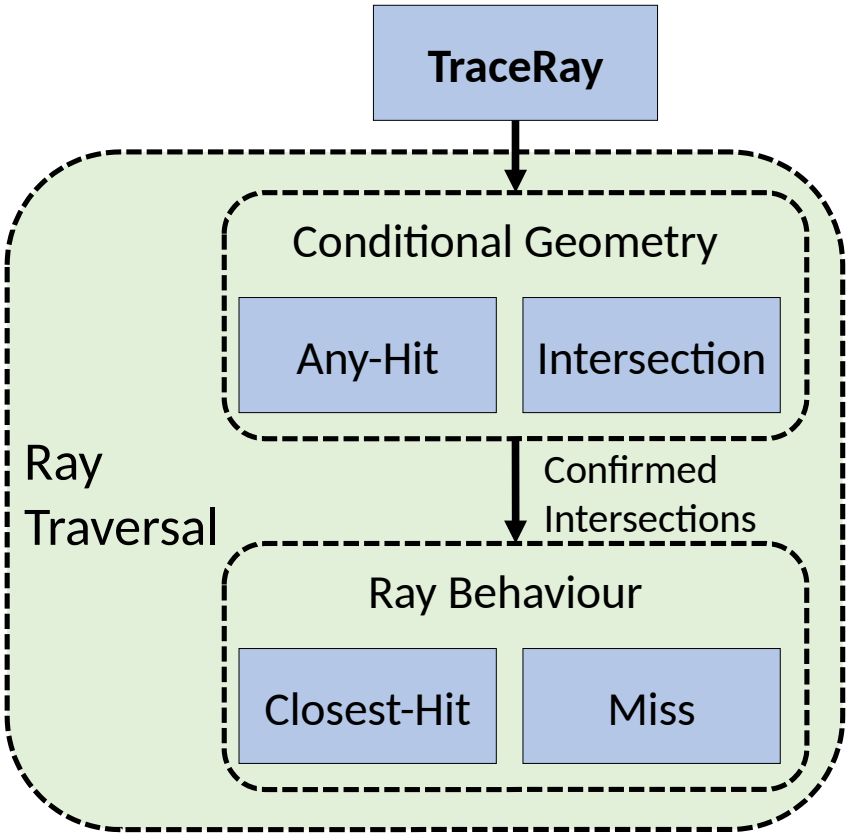

Figure 2.4: Overview of `traceRay` operation defined in Vulkan. The `traceRay` operation traces a single ray through the scene. Application-provided shaders (blue) define how geometry and the ray interact. The conditional-geometry shaders allow intersections for the ray to be discarded programmatically. This mechanism builds the set of confirmed ray-geometry intersections for the ray. The ray-behaviour shaders determine how the ray should behave based on the confirmed ray-geometry intersections.

classified as defining ray behaviour.

In ray tracing, usually each thread traces a different ray. When executing `traceRay`, rays allocated to threads within the same warp may hit different objects and execute different shaders. The result is divergent indirect function calls that implement the flow of execution. Upon identifying different ray hits, each thread accesses the SBT with its hit information—which objects and materials a thread has hit—to obtain the addresses of the shaders to be executed. Fully divergent indirect function calls, where each thread in a warp executes a different shader, serializes the execution of these functions. If recursion occurs within the called shader, SIMT utilization can be further impacted.

A single ray-tracing pipeline execution issued with the Vulkan API is able to render an entire scene consisting of multiple objects with different textures, materials (e.g. matte, gloss, semi-gloss, eggshell, etc.), geometry types (e.g. triangles, implicit surfaces, etc.) and transforms. Contrast that with rasterization, where each draw command renders a single geometry that may not even be a whole object. The application specifies the dimensions of the ray-tracing task, referred to as the launch size, and the ray-tracing pipeline is executed for each element. For most common uses of ray tracing, the launch dimensions will correspond to the dimensions of the rendered scene.

Acceleration Structure: The Vulkan API defines an Acceleration Structure (AS) to contain and organize scene geometry for ray tracing. A brief description of the Vulkan API methods applicable to building acceleration structures are provided in Table 2.1. An AS is broken down into top and bottom levels. A Bottom Level Acceleration Structure (BLAS) defines a set of geometry primitives. These primitives may either be triangles or bounding boxes. A Top Level Acceleration Structure (TLAS) defines a set of BLAS instances and transform matrices to specify the final geometry in the scene. For ray tracing, the application creates BLAS nodes that refer to one or more geometry buffers. The application then creates TLAS nodes with references to BLAS nodes to populate the scene with objects, similar to a two-level scene graph. To position and orient the objects in the scene, the application specifies

Name	Description
`vkCmdBuildAccelerationStructuresKHR`	Builds a set of acceleration structures (AS). Specifies Top-Level Acceleration Structure (TLAS) or Bottom-Level Acceleration Structure (BLAS), the geometry type—triangular or bounding box, the buffers containing the geometry data, and the number of primitives to expect.
`vkMapMemory`	Returns a host-accessible pointer for a range of device memory. The application may copy data from host to device through the pointer.
`vkUnmapMemory`	Releases the host-accessible pointer provided by vkMapMemory.
`vkBindBufferMemory`	Binds a range of device memory to a buffer.
`vkCmdCopyBuffer`	Copies memory from one buffer to another. This is commonly used to populate staging buffers on the device that hold vertex and index data.

Table 2.1: Description of key Vulkan API calls for constructing acceleration structures.

a geometry-to-object transformation for each BLAS and an object-to-world transformation for each TLAS.

This chapter introduced ray tracing, instrumentation, GPU execution, and Vulkan. These technical details are requisite background for Chapter 3 and Chaper 4. Chapter 3 introduces a SPIR-V instrumentation framework for capturing detailed runtime information from Vulkan pipelines executing on the GPU. Chapter 4 presents a ray-tracing visualization tool based on SPIR-V instrumentation and Vulkan-API call instrumentation.

Chapter 3

Vulkan Vision: Ray Tracing Workload Characterization using Automatic Graphics Instrumentation

Instrumentation is a technique that enables developers to make informed decisions. Instrumentation collects meta-information about program execution in addition to the regular program output. The adoption of real-time hardware-accelerated ray-tracing is driving innovation at the hardware, driver, compiler, and application levels. However, no existing framework facilitates shader instrumentation for Vulkan applications, as shown in Table 3.1.

This chapter answers the following research problem: *To what extent can the underlying hardware execution be measured through instrumentation, and can the execution data be used for limit studies, characterization, and profiling of the instrumented applications?* To solve this problem, this chapter introduces V-Vision, an open-source framework to capture fine-grained GPU execution data, independently of vendor-specific compilers.

The Vulkan API is a widely adopted graphics and compute API that minimizes driver overhead. This API elides default error checking and instead offers an optional validation layer to provide debugging capabilities. Validation layers support analysis and instrumentation of SPIR-V shader code. SPIR-V is a vendor-independent pre-compiled intermediate representation for device code required by Vulkan [24]. The existing support for SPIR-V instrumentation

Tool	**Open Source**	**Shader Instrumentation**	**Automatic Profiling**	**Target Agnostic**	**API**
NSight Graphics [43]	No	No	Yes	no	DirectX OpenGL Vulkan
Intel GPA [17]	No	No	Yes	no	DirectX OpenGL Vulkan Metal
PIX [37]	No	No	Yes	Yes	DirectX
Strengert *et al.* [55]	Yes	Yes	No	Yes	OpenGL
V-Vision	Yes	Yes	Yes	Yes	Vulkan

Table 3.1: Landscape of graphics profiling.

provides only error checking and the printf debugging extension.

V-Vision is a framework for graphics shader instrumentation in Vulkan applications. V-Vision extends existing validation layer instrumentation functionality by providing a framework with a set of ready-to-use instrumentation primitives, instrumentation utilities, and instrumentation analytics. V-Vision is open source under MindInsight at `https://gitee.com/mindspore/mindinsight`.

Instrumentation primitives fall into two groups: (1) unique-identification primitives, and (2) buffer-update primitives. Unique-identification primitives reveal accurate and fine-grained shader runtime behaviour and relate it to static data generated from the SPIR-V shader, such as the Control Flow Graph (CFG). Buffer-update primitives provide alias and race-free write operations to the instrumentation *StorageBuffer*. The instrumentation utilities specify program points to emit instrumentation primitives at and the values to provide to the utilities. The utilities provided by V-Vision produce both runtime and static data which are combined to produce a characterization of pipeline execution, such as dynamic instruction execution counts (hotspots).

The output of instrumentation utilities are consumed by the analytics in V-

Vision to characterize runtime behaviour and identify performance bottlenecks, such as serialized indirect function calls. The analytics include simulating improvements to performance bottlenecks, such as thread compaction to reduce the impact of serialization [13]. Lastly, the analyses produce visualizations within the shader code to streamline refactoring and assist in solving performance issues.

The rest of this chapter is organized as follows: Section 3.1 explains the architecture of V-Vision. Section 3.2 discusses the instrumentation primitives provided in V-Vision. Section 3.3 characterizes ray-tracing Vulkan applications using instrumentation utilities. Section 3.4 evaluates V-Vision's overhead and compares binary and Intermediate Representation (IR) instrumentation. Finally, we conclude in Section 3.5.

3.1 V-Vision Architecture

V-Vision provides programming interfaces to perform automated shader instrumentation prior to execution, and to analyze the corresponding output data. V-Vision further provides built-in functionality for common instrumentation and analysis tasks. This section details V-Vision's architecture, interface, execution and out-of-the-box features.

Vulkan Validation Layers: Vulkan performs error checking and other debugging capabilities in validation layers. Validation layers are intermediaries between Vulkan applications and the driver and require no code modification or recompilation [26].

Validation layers support use cases such as in-game overlays [58], and game traces captured with the *gfxreconstruct* layer [32]. These examples are possible because validation layers transparently compose with any standards-conforming Vulkan application across major operating systems and GPU architectures. Validation layers therefore support profiling any Vulkan application as no changes are required from the application developer.

Tool Architecture: V-Vision extends the standard Vulkan validation layer that provides error checking and printf. Figure 3.1 depicts V-Vision's

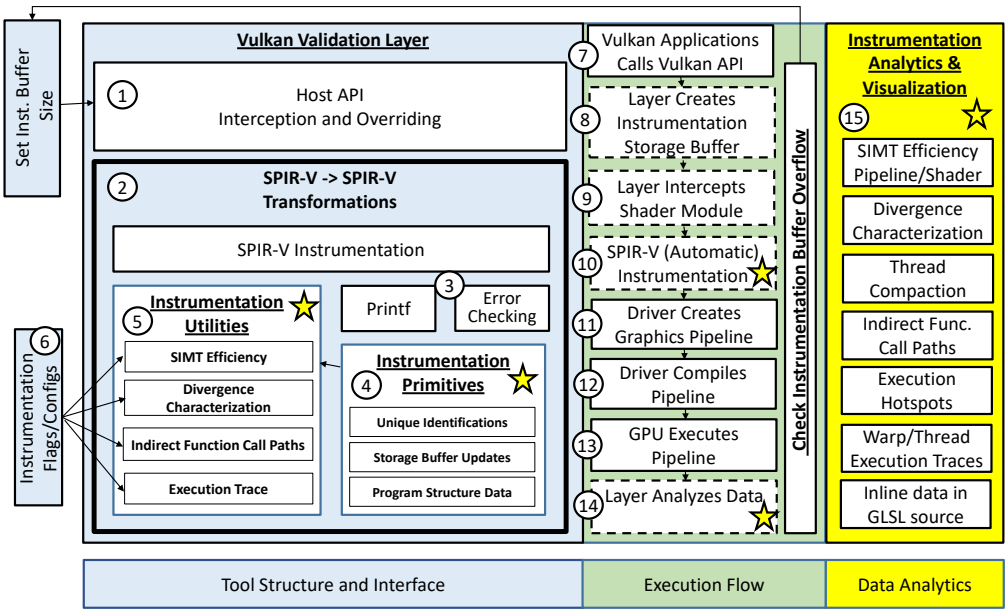

Figure 3.1: Vulkan Vision Architecture, Interface, and Execution Flow. The stars in the figure mark which modules are a novel contribution. The dotted boxes in Execution Flow represent the steps when using a validation layer for instrumentation.

architecture, interface and execution-flow. V-Vision's contributions are identified with stars. Vulkan validation layers intercept and modify the host-code APIs ① and device-code SPIR-V kernels ②. Transformations of the device code performs dynamic error checking and supports debugging with the `debugPrintfEXT` built-in [33] ③. By extending the instrumentation framework, V-Vision supports generic application profiling.

V-Vision's instrumentation primitives ④ systematically address challenges in instrumenting a graphics pipeline. As detailed in Section 3.2, these primitives are categorized into two groups: unique-identification primitives, and buffer-update primitives. Instrumentation passes generate primitives to record per-warp values, such which threads in a warp are active, and per-thread values, such as a thread's value of a variable. These primitives support customization of variables and program points that are instrumented.

V-Vision provides ready-to-use utilities ⑤ built using the instrumentation primitives: i) SIMT Efficiency: measure active threads at every basic block in a graphics pipeline; ii) Divergence Characterization: measure respective impacts

of control-flow, indirect function call and early-return divergence. iii) Indirect Function Call Paths: reveal all execution traces leading to indirect function calls; and iv) Execution Trace: complete warp and thread execution traces throughout the graphics pipeline. These utilities are exposed as flags (6). The utilities can be used by application developers and hardware designers to expose bottlenecks, such as poor SIMT Efficiency due to serialized indirect function call execution, and opportunities to improve them, such as thread compaction or shader refactoring.

Execution Flow: The workflow of V-Vision is shown in Figure 3.1 through the black arrows that represent the order of events. The dotted boxes represent the components of a Vulkan validation layer performing instrumentation and the boxes marked with a star are novel contributions of V-Vision. Removing all the dotted boxes results in the operation mode of a Vulkan application without validation. The flow begins with the execution of Vulkan API calls (7). Next, the validation layer intercepts the application's device-buffer creation to add a *StorageBuffer*—a readable and writable type of storage that is visible anywhere in the graphics pipeline—where the instrumentation data will reside (8). The validation layer receives the SPIR-V source for every shader module that the Vulkan application creates (9). The validation layer triggers V-Vision's SPIR-V automatic-instrumentation pass. This pass adds the desired instrumentation, according to the instrumentation utility, to each SPIR-V module source, thus creating new writes to the storage buffer (10). Thereafter, events outside of the validation layer occur: a graphics pipeline is created as a sequence of shaders (11), the pipeline is compiled into a GPU-specific binary (12), and the pipeline is run on the GPU (13). The last event provides the instrumentation data to the analyses in V-Vision (14).

Instrumentation Analytics: V-Vision provides analytics that transform the results of instrumentation utilities into feedback and visualizations (15): i) SIMT Efficiency: compute the SIMT Efficiency of shaders and graphic pipeline by reconstructing warp behaviour; ii) Divergence Characterization: characterize control-flow, indirect-function-call and early-return divergence by analyzing inactive threads; iii) Thread Compaction: upper-bound of thread

compaction benefit when applied to indirect function calls based on an oracle's knowledge [13], [60]; iv) Indirect-Function-Call Paths: enables the study of the SIMT efficiency of ray generation `traceRay` by reconstructing runtime thread paths; v) Execution Hotspots: visualize graphics pipeline hotspots based on dynamic instruction execution counts using program source code information; vi) Warp and Thread Execution Traces: study work allocation on thread and warp granularities by tracing execution; and vii) Inline Data Representation: present data inline at the point it was captured from in GLSL source. This chapter demonstrates the insights that are revealed by applying V-Vision to applications that implement Vulkan ray-tracing pipeline in Section 3.3.

3.2 Instrumentation Primitives

In general, there are common problems with instrumentation utilities, such as complicated logic to parse variable-size data and inefficient utilization of the instrumentation buffer. This section outlines V-Vision's primitives that mitigate these challenges and details their operation. The instrumentation primitives are provided by V-Vision for developers to create their own instrumentation utilities.

```
1  void ThreadUpdate(uint arg1, ...) {
2     uint i = atomicAdd(buf[1], entry_size);
3     if (i + entry_size >= buf.len())
4        return;
5     buf[i + 0] = thread_work_id;
6     buf[i + 1] = instrumentation_id;
7     buf[i + 2] = arg1;
8     ...
9     buf[i + 2 + k] = <arg k>;
10    ...
11 }
```

Figure 3.2: GLSL representation of ThreadUpdate primitive in V-Vision. The ThreadUpdate primitive appends an entry to the StorageBuffer, `buf`, for each thread that invokes it.

Buffer Updates Primitives: The core of V-Vision's auto-instrumentation is `ThreadUpdate`, shown in Figure 3.2 using GLSL. `ThreadUpdate` safely writes

an entry to the StorageBuffer, denoted as `buf` in the figure. Line 2 atomically adds the number of words to write, stored in `entry_size`, to a special location in the StorageBuffer, `buf[1]`.

The variable `i` receives the value of `buf[1]` *before* `entry_size` is added to it. The `if` statement on lines 3-4 checks whether the write will overflow the buffer, and if it will, aborts the write. The number of words written is updated before the function is aborted to determine how many words the instrumentation could write. This mechanism allows V-Vision to report if the StorageBuffer was too small, and exactly how large it should be. Line 5 writes the unique work identifier that is assigned to the thread, for example the LaunchID of the thread in ray tracing. Line 6 writes the identifier assigned to the instrumentation callsite that is used to lookup the instrumentation type and entry size. Lines 7-10 represent writing the arguments passed to the `ThreadUpdate` function, the exact number of which may vary. This primitive is called `ThreadUpdate` because every thread that executes it will append a new entry in the StorageBuffer.

```
1 void WarpUpdate(uint arg1, ...) {
2    if(subgroupElect())
3       ThreadUpdate(arg1,...);
4 }
```

Figure 3.3: GLSL representation of WarpUpdate primitive. The WarpUpdate primitive appends an entry to the storage buffer for each warp that invokes it.

The next primitive provided by V-Vision, `WarpUpdate`, writes an entry in the StorageBuffer for each warp that executes it. A GLSL representation of `WarpWrite` is shown in Figure 3.3. The `if` statement on line 2 differentiates `WarpUpdate` from `ThreadUpdate` using the GLSL builtin `subgroupElect`. The builtin will return true for the lowest-numbered thread in a warp, and false for all other threads. Thus, calling `ThreadUpdate` only if `subgroupElect` is true, results in one entry being written to the buffer. A special case of `WarpUpdate` is to compose it with the GLSL builtin `subgroupBallot(true)` to measure how many threads are active. `subgroupBallot(true)` evaluates the predicate `true` for all active threads in the warp. Thus, a bitmask is recorded that has

the property: bit_i is set iff $thread_i$ is active.

Unique-Identification Primitives: The unique identification primitives in V-Vision allow execution to be traced in control-flow, inter-procedurally and across pipeline stages.

Unique Warp ID: GLSL provides an abstract interface that differs from GPGPU APIs, such as CUDA, that provide a programming model that closely matches the hardware. GLSL is designed to develop shaders in isolation that will be connected by the compiler. Each shader type has rules for what data it is allowed to access.

An issue in analyzing the generated instrumentation data is the lack of attribution from the instrumentation data to the warp that created it. Two insights can lead to the creation of a mapping to allow for correct attribution for data created throughout the graphics pipeline: (1) every shader type has a *thread-work id* that is unique to each thread and always available within its respective shader stage; (2) any thread that will be active anywhere in a shader module, must also be active at the shader-module entry point. Based on these insights, V-Vision provides a primitive that generates a *warp id* enabling the creation of a mapping from *thread-work id* to *warp id*.

```
1 void CreateWarpId() {
2    uint warp_id = 0;
3    if(subgroupElect())
4       warp_id = atomicAdd(buf[0],1);
5    warp_id =subgroupBroadcastFirst(warp_id);
6    ThreadUpdate(warp_id);
7 }
```

Figure 3.4: GLSL representation of V-Vision's instrumentation primitive CreateWarpId. The primitive creates a warp id and every thread in the warp writes it to the buffer. `buf[0]` is a dedicated location in the StorageBuffer for creating warp ids.

The GLSL representation of the primitive `CreateWarpId` is shown in Figure 3.4. This primitive is executed by every thread in a warp. In line 2 each thread creates a copy of `warp_id`. The `subgroupElect` call in line 3 returns true to the lowest-numbered thread in the warp and returns false to all others.

The only thread that executes line 4 receives the current next available id, stored in position zero of the StorageBuffer, and increments it atomically. Thus, each warp receives a unique id. The GLSL builtin `subgroupBroadcastFirst` on line 5 is a synchronization point with two distinct functions. When invoked by the lowest-numbered thread's value of `warp_id` it broadcasts this value to all other threads. When invoked with zero by all other threads, it returns the unique `warp_id` broadcasted by the the lowest-numbered thread. Thus, after line 5 all the threads in the warp have the same value in their local `warp_id`. In line 6 each thread calls `ThreadUpdate` to write the value of `warp_id` to the StorageBuffer. `ThreadUpdate` also writes the *thread-work id* of each thread, and thus enables the creation of a complete mapping from *thread-work id* to *warp id*. An alternative use of this instrumentation is determining how the driver assigns *thread-work ids* to warps. In a ray-tracing application, it revealed an 4×8 zig-zag assignment rather than a linear assignment.

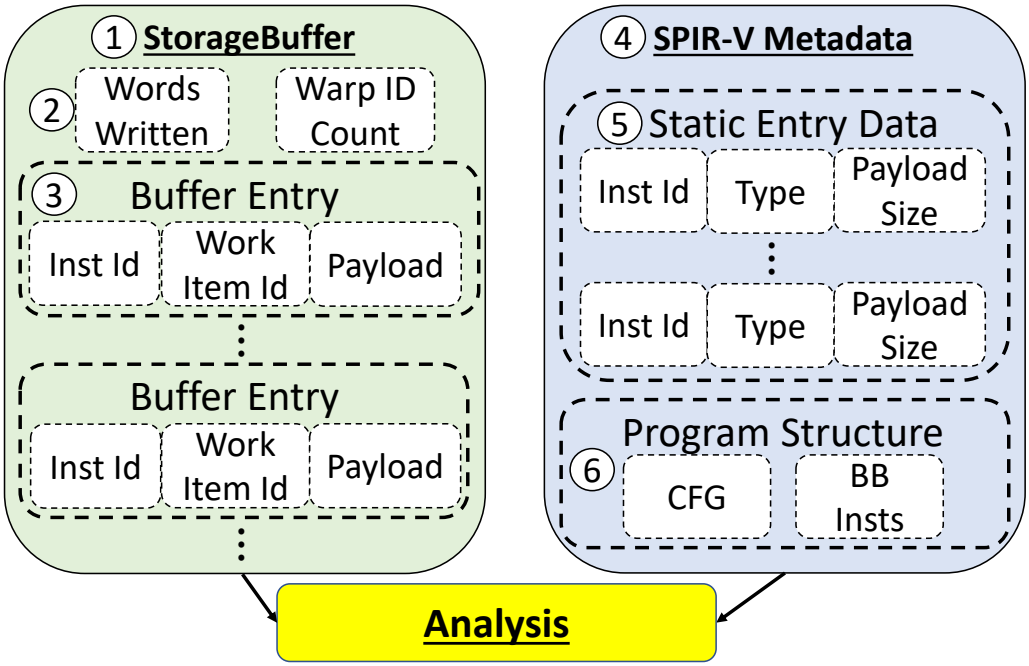

Figure 3.5: V-Vision's layout of StorageBuffer containing runtime instrumentation data and SPIR-V metadata used to complement the runtime data.

Unique SPIR-V Operation ID: Figure 3.5 shows the organization of

the data created by V-Vision. In the StorageBuffer ①, *Words Written* ② is the number of words that the instrumentation primitives write when `ThreadUpdate` or `WarpUpdate` are called. V-Vision reports this value to the user if it exceeds the capacity of the StorageBuffer. The user may then rerun the application with a larger buffer. *Warp ID Count* is atomically incremented by `CreateWarpId` to assign each warp a unique id. Buffer entries ③ are appended to the buffer by `ThreadUpdate` and `WarpUpdate`. The *inst id* value identifies which instrumentation call site produced the data. *thread-work id* identifies each thread and *payload* contains the customizable entry data.

V-Vision improves the utilization of the StorageBuffer by only writing runtime data. Statically-known data, represented as SPIR-V metadata ④ can be of two types: static entry data ⑤ is associated with the instrumentation call sites using *inst id*; program structure ⑥ records the control flow graph and information about individual basic blocks. With millions of entries written per frame, this distinction between static and runtime data prevents duplication in recording and leads to efficient profiling. For example, entries in the storage buffer have variable sizes. Instead of recording the size of the entry in the runtime entry, this size is obtained through a lookup in the SPIR-V metadata. Program structure data is included in instrumentation utilities on an as-needed basis. For example, the graphics pipeline hotspot analysis needs the number of instructions in each basic block; and tracking thread paths that lead to indirect function calls requires the shader module CFG.

```
1 /* execution count = 22072 histo=32:22072*/
2 WarpUpdate(subgroupBallot(true));
3 if(gl_LaunchIDNV.z != 0){
4 /* execution count = 11036 histo=32:11036*/
5   WarpUpdate(subgroupBallot(true));
6   ...
7 }
```

Figure 3.6: Data captured from V-Vision's SIMT Efficiency instrumentation utility. Presented as inline comments in the GLSL representation of the shader.

The instrumentation callsite ids support a visualization of analyses in V-Vision. `OpLine` is an SPIR-V debug instruction designed to encode file

Frame	Application	3D Obj
Hairball	ChameleonRT [57]	Hairball [35]
Sponza	ChameleonRT [57]	Sponza [35]
Sky	Quake II RTX [48]	
Window	Quake II RTX [48]	
Cornell Box	RayTracingInVulkan [14]	
Reflective Ball	RayTracingInVulkan [14]	
Robot	VkRaytrace [11]	

Table 3.2: Frames studied with V-Vision and the application they were captured from. 3D Obj files were only required for Chameleon RT.

information throughout the SPIR-V module. V-Vision repurposes `OpLine` to present data throughout the GLSL representation of each shader by transforming line directives into GLSL comments. The result, shown in Figure 3.6, is warp-execution trace information presented as inline comments. Lines 1 and 4 show the total number of warps that executed each instrumentation call along with histograms of the number of active threads in each warp. This inline presentation leverages code understanding when examining the profile data.

3.3 Ray-Tracing Insights

This section applies utilities and analyses provided by V-Vision to Vulkan ray-tracing applications. The insights are organized into insights for hardware, compiler, and application developers respectively.

3.3.1 Methodology

The data for each case study was collected on an NVIDIA Turing 1660Ti with beta driver version 451.79 that supports the `VK_RAY_TRACING_KHR` extension. Table 3.2 shows the frames captured in each application studied and, where applicable, the 3D object from Casual Effects [35]. Except for ChameleonRT, frames are captured using NSight Graphics 2020.3 C++ capture. For ChameleonRT, NSight Graphics failed to create a capture. Instead, the first frame of running ChameleonRT on Hairball and Sponza models was instrumented. For all measurements, each frame is executed 5 times. All experiments

are combined in a list and executed once, the list is scrambled before the next execution. This method ensures that temporary variations in the execution environment that are not under control will increase the variability of the measurements but not insert biases. The frames captured from Quake II RTX, Sky and Window, each execute the ray-tracing pipeline 5 times: Primary Rays, Direct Rays, Reflection Refraction 1, Reflection Refraction 2, Indirect Rays.

3.3.2 Hardware Insights

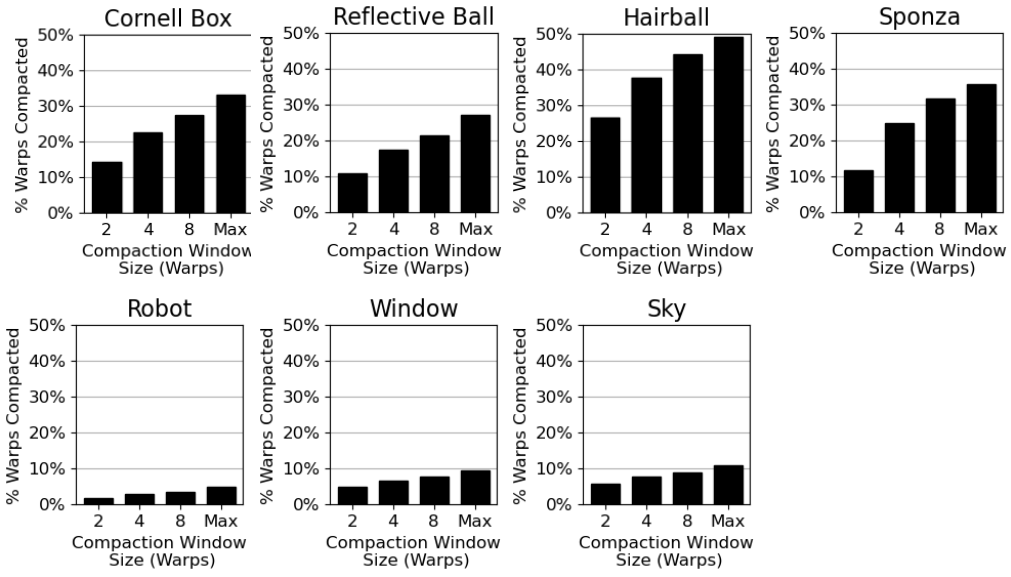

Figure 3.7: Impact of Thread Compaction on number of warp executions of `traceRay`. Windows are composed of consecutive warps.

Thread Compaction: Thread compaction is a proposed hardware modification to increase SIMT efficiency by repacking threads [13]. Thread compaction may create warps composed only of inactive threads that do not need to execute. A relevant analysis question is how much thread compaction can be used to improve ray tracing for each frame. V-Vision provides an upper-bound on the improvement provided by thread compaction by simulating the process of thread compaction using data captured through instrumentation.

Figure 3.8 presents an example of the instrumentation for Thread Compaction, V-Vision's instrumentation utility that tracks the dynamic thread

```
1  for (num_samples < MAX_SAMPLES){
2     ray = get_ray();
3     for(num_bounces < MAX_BOUNCES){
4        // PreTraceRay for traceRay 0
5        WarpUpdate(subgroupBallot(true));
6        traceRay(ray, ..., payload);
7        if(payload.missed)
8          break;
9     }
10    // Sync Point for traceRay 0
11    WarpUpdate(subgroupBallot(true));
12 }
```

Figure 3.8: Pseudocode for a global-illumination style ray-generation shader with instrumentation for Thread Compaction analysis.

paths executing `traceRay`. Using the instrumentation on line 5 to capture the threads executing the `traceRay` call on line 6, the analysis counts how many times each thread executes that inner-loop call. The instrumentation on line 11 captures threads exiting the inner loop. The analysis builds a *thread path*—a bit vector with a bit for each execution of `traceRay`—for each thread. A one in the thread path indicates that the thread is active for that execution. Examining all thread paths, the analysis counts the number of threads active for each `traceRay` execution. The estimation for the maximum compaction is the ceiling of the number of the number of threads active for a given execution of `traceRay` divided by the warp size. A more realistic estimate limits thread compaction to a number of consecutive warps, the number of warps included is the *compaction window*.

Figure 3.7 presents the % warps compacted—rendered inactive through receiving all inactive threads—for each frame. The majority of the warps rendering the Robot frame process rays that hit a skybox. These warps are very coherent and do not benefit much from compaction. Window and Sky perform both deterministic tracing and random tracing. Warps executing random path tracing benefit from thread compaction. The divergence in Cornell Box, Reflective Ball, Hairball and Sponza is significantly improved with two warps in a compaction window, increasing the window size has diminishing

benefits. This analysis illustrates how V-Vision estimates the potential for compaction, but it does not take into consideration work assignment or the memory subsystem. Thread compaction requires register migration between threads leading to additional hardware dedicated to relaying thread ids.

3.3.3 Compiler Insights

```
1 if (cond){
2    TraceRay(...);
3 }
4 //Pre-volta sync point
```

Figure 3.9: Pseudocode that triggers independent thread scheduling on NVIDIA's Turing architecture.

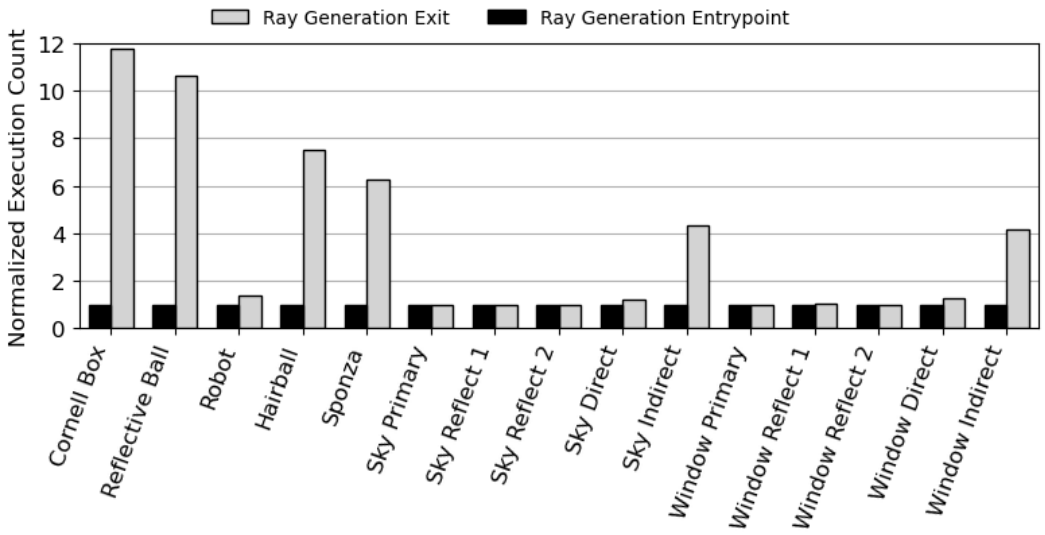

Figure 3.10: Evidence of independent thread scheduling in ray tracing. Execution count of ray generation shader exit normalized to entry execution count.

In the NVIDIA Turing and Volta architectures, each thread has its own Program Counter (PC)—in earlier GPUs all threads in a warp had the same PC. Per-thread PCs allow Independent Thread Scheduling (ITS) whereby threads no longer execute a GPU kernel in lockstep. V-Vision's visualizations reveal that divergent `traceRay` calls, as shown in Figure 3.9, trigger ITS.

Inactive threads in line 2 do not wait at the join point in line 4. ITS causes warp execution to split whereby multiple PCs are executing concurrently. The entrypoint of the graphics pipeline must be executed exactly once by each warp. Under the effects of ITS, the exit of the graphics pipeline may be executed multiple times. ITS can be quantified by comparing how many times the entrypoint and exit were executed. Figure 3.10 shows the exit execution count in gray, and entrypoint execution count in black. The exit count is normalized relative to the entry execution count to quantify how split the warp execution is. Control-flow divergence inherent in random path tracing triggers ITS, as evidenced by Cornell Box, Reflective Ball, Hairball, Sponza, Window (Indirect), and Sky (Indirect).

3.3.4 Application Insights

SIMT Efficiency

SIMT efficiency is the percentage of active threads across all basic-block executions. It measures the utilization of a SIMD GPU hardware. The low SIMT efficiency in ray tracing is due to unpredictable control flow, such as rays executing different shaders, and poor work assignment due to variation in ray bounces [10]. Existing GPU instrumentation frameworks capture SIMT efficiency of GPGPU ray tracers. However, capturing this metric in a graphics pipeline is challenging because of restrictions between shader stages. V-Vision's primitives overcome these challenges and are able to track the execution across indirect function calls and shader modules.

Figure 3.11 reports the SIMT Efficiency of each pipeline invocation. The frames from RayTracingInVulkan, Cornell Box and Reflective Ball exhibit similar SIMT Efficiency despite many differences in the frames themselves. In the Cornell Box scene, rays bounce multiple times because they are trapped in the box. Each bounce triggers control-flow divergence. Reflective Ball has fewer bounces but many divergent intersections based on material (reflection, refraction, opacity) lowering the SIMT Efficiency. The impact of geometry on SIMT efficiency is observed in Hairball and Sponza, both from ChameleonRT.

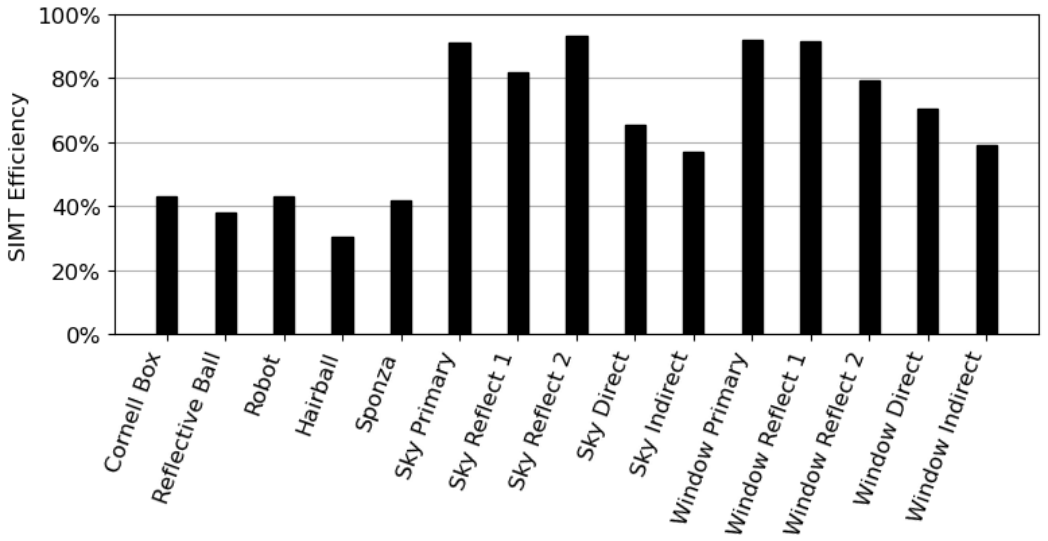

Figure 3.11: SIMT efficiency of profiled frames.

With many thin hairs, Hairball triggers many incoherent ray bounces, thus lowering SIMT Efficiency when compared to Sponza. While the first three pipeline invocations of Window and Sky have high SIMT Efficiency because their effects are deterministic, the subsequent pipeline invocations, *Direct* and *Indirect*, perform random path tracing reducing SIMT Efficiency. Random path tracing in Quake II RTX has higher SIMT Efficiency than the other path tracers because it only performs 1 ray bounce per thread, compared to 3 in VkRaytrace, 16 in RayTracingInVulkan and 5 in ChameleonRT, limiting the divergence.

Comparing Figure 3.10 to Figure 3.11 indicates that in general low SIMT efficiency is related to ITS. However, Robot, Sky Direct and Window Direct do not conform to this pattern. In Robot, for some warps, all the rays hit a skybox while, in other warps, the rays hit geometry. For the ones that hit the skybox, there is little work to do. In warps processing rays that hit geometry there is significant divergence that leads to ITS, resulting in the difference between entry and exit executions in Figure 3.10. The frame execution has low SIMT Efficiency, as shown in Figure 3.11, because the warps that do most of the work have high divergence. ITS occurs in both the Direct and the Indirect pipelines of Sky and Window because of a branch that tests if a surface is facing away

from the Sun. The effect is more pronounced in the Indirect pipeline because of a random ray bounce before the branch.

Divergence Characterization

There are three sources of divergence: i) threads that complete the ray-tracing pipeline and remain idle while the rest of the warp continues to execute; ii) divergent indirect function calls when rays hit different objects; and iii) control-flow divergence caused by branch instructions.

```
1  void chit() {
2    // Shader Entrypoint
3    WarpUpdate(subgroupBallot(true));
4  }
5
6  void main() {
7    if (cond){
8      // Early Return
9      WarpUpdate(subgroupBallot(true));
10     return;
11   }
12   // Pre-traceRay
13   WarpUpdate(subgroupBallot(true));
14   traceRay(...);
15   // Post-traceRay
16   WarpUpdate(subgroupBallot(true));
17 }
```

Figure 3.12: Simplified GLSL representation of instrumentation to characterization factors contributing to SIMT divergence.

When a warp executes an instruction, each inactive thread accounts for an inactive instruction-execution slot. V-Vision's Divergence Characterization analysis reports the total number of such slots for each divergence factor. To do so, it uses the program basic-block data to count the number of inactive instruction slots for each inactive thread. In the example of instrumentation in Figure 3.12 the `chit` function is the closest hit shader. Each of the instrumentation calls in lines 3, 9, 13, and 16 record a bit mask indicating the active threads at that execution point. Threads active in line 9 cause early-return divergence. Instrumentation calls, such as the one in line 3, inserted in every

shader trace the individual execution of every thread. Whenever a thread becomes inactive at the entry point of a given shader executed by the warp, there is indirect-function-call divergence. The instrumentation calls in lines 13 and 16 capture the start and end of the `traceRay` execution trace respectively. An inactive thread that has not been captured as either an early-return divergence or as indirect-function-call divergence must be inactive due to control-flow divergence.

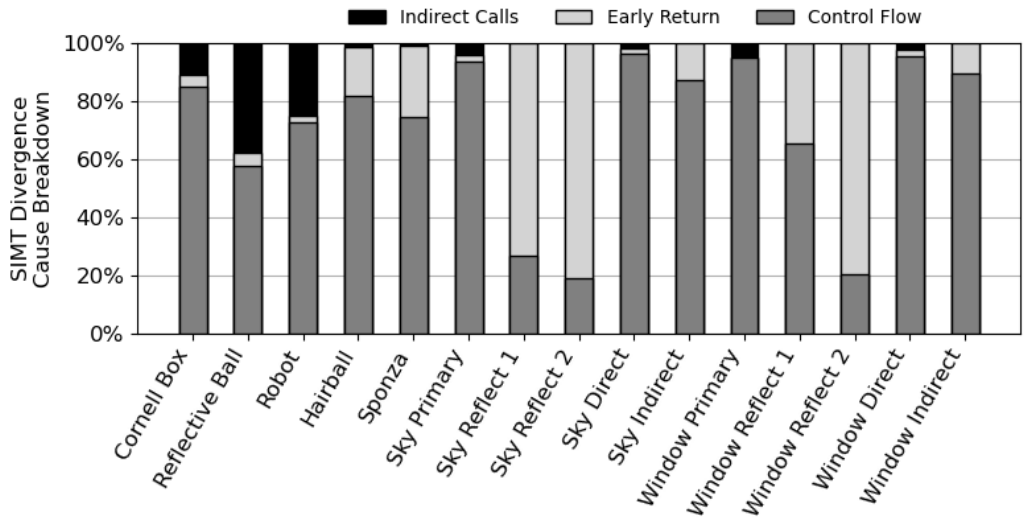

Figure 3.13: Characterization of factors contributing to SIMT Divergence.

Figure 3.13 presents the divergence characterization. In Cornell Box, Reflective Ball, and Robot the control-flow divergence is caused by the varying ray path lengths in the tracing loops and by variations in material types that lead to different actions upon collision. Invocations of the intersection and any-hit shaders lead to indirect-call divergence. For instance, when rays collide with spheres in Reflective Ball, the intersection shader invocations lead to many divergent indirect function calls. Miss and closest-hit shaders cause much less divergence because they are invoked at most once per `traceRay` call. Thus, indirect-function-call divergence is less significant for other frames. Variable number of ray bounces account for the early-return divergence in Hairball and Sponza. Sky and Window Reflect 1 and 2 have high early-return divergence because of checks for collisions with reflective materials. The effect

is less pronounced in Window Reflect 1 because part of the frame is reflective material.

Ray-Tracing Hotspot Detection

The behaviour of the ray-tracing pipeline is complex, involving cycles and recursion, and difficult to reason about. The same visual effects in the ray-tracing pipeline may be implemented in different shader stages. Architecture-specific complexities, such as ITS, further complicates writing shaders. Visualizing hotspots that may be the result of geometry or other runtime factors, allows the developer to focus their refactoring efforts.

V-Vision's instrumentation utility, SIMT Efficiency, records a static mapping from instrumentation callsite id to the PCs of all instructions in the basic block. The utility also captures the number of active threads in each dynamic basic-block execution. The number of runtime threads is added to the totals of each instruction in the basic block to create the dynamic instruction count.

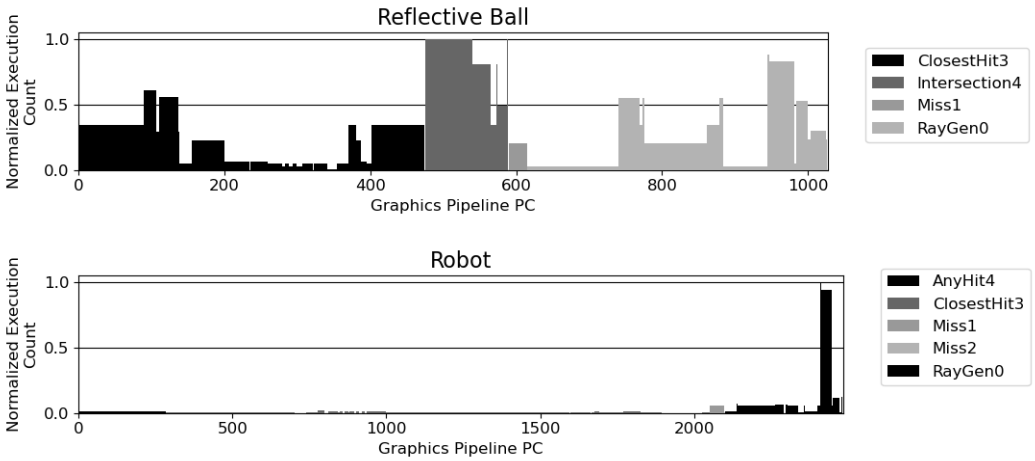

Figure 3.14: Hotspots for Reflective Ball and Robot frames, normalized to maximum dynamic instruction execution count.

Figure 3.14 presents the dynamic instruction execution counts of each PC normalized to the maximum for Reflective Ball and Robot. Each color in the figure represents a shader in the ray-tracing pipeline. The intersection shader of Reflective Ball dominates the execution of the pipeline because all objects

in the scene are spheres. The closest-hit shader is more complicated here than the closest-hits of other frames due to reflection and refraction effects. A pseudorandom number generator dominates the dynamic instruction count of Robot. The pseudorandom number generator executes 16 iterations of a loop to create a random direction for a ray. As most rays in Robot miss geometry, the random direction must be recomputed for nearly every ray. If the geometry becomes larger, then more rays would collide with it and perform more bounces before requiring a new pseudorandom number. This hotspot is a clear example of a bottleneck that occurs in the presence of a specific scene configuration.

Warp and Thread Lifetimes

Imbalanced work allocation degrades GPU performance because threads that complete their work first become inactive. The same principle applies to warps that are scheduled in groups. Imbalanced thread assignment is also a problem in path-tracing on GPGPUs [10]. The GPGPU solution of fixing work assignment with work-coarsening does not translate to graphics where frame latency is a key measure. Work assignment is also unpredictable because it is impacted by the geometry of the scene.

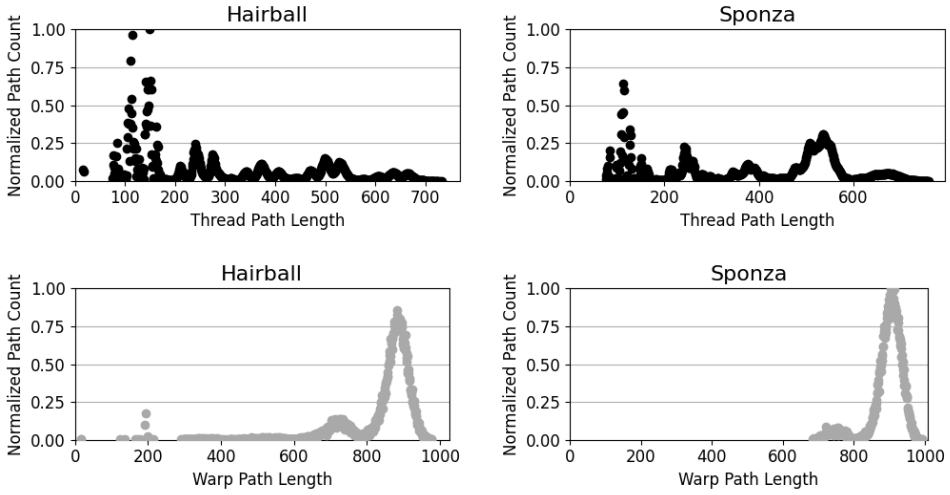

Figure 3.15: Thread and Warp lifetimes for Hairball and Sponza, normalized to maximum path count between both scenes. Other frames omitted for brevity.

Execution Trace is a V-Vision's instrumentation utility that provides com-

plete per-thread and per-warp execution traces. The lifetime of a thread or warp is defined as the number of basic blocks executed and can be derived from the execution traces. Figure 3.15 shows the thread and warp lifetimes for both the Hairball and Sponza frames. A large bump centered around path length of 530 is present in Sponza but not in Hairball. This bump is caused by Sponza being an enclosed space, trapping rays into a higher number of consecutive bounces. Rays that bounce off of Hairball's geometry and then miss, account for the higher incidence of path lengths in the range from 100 to 200 basic blocks. The wider range of thread path lengths causes a wider range of warp path lengths in Hairball. In comparison, Sponza has a very compressed range of warp paths due to being an enclosed space. Both scenes have a bump where rays are traced up to the maximum bounce depth. These results show that geometry has a strong effect on work assignment at the thread and warp level.

Ray Generation Thread Paths

The ray-generation shader generates rays and invokes `traceRay` to perform traversal and intersection, known to be expensive and to benefit from hardware acceleration [7]. Variable number of bounces and conditional calls based on material type influence the behaviour of the ray-tracing pipeline [10]. Understanding the effect of material type and geometry offers opportunities for optimizations, such as value specialization.

The thread paths generated by the Thread-Compaction utility offer insights into application design. Each thread receives a thread path, a bitmask representing the runtime invocations of `traceRay`. The number of unique bitmasks indicates the potential for divergence in executing expensive `traceRay` calls. Accumulating the frequencies of thread paths reveals threads' proclivities for number of bounces or visual effects over their complete execution.

Table 3.3 shows the total number of unique thread-paths and the top 3 frequencies of individual paths. The thread paths that miss geometry are very significant when present. The frequencies 49.11% in Cornell Box, 16.1% in Reflective Ball, and 89.79% in Robot are due to rays missing geometry. Reflective Ball has many different material types and thus executes many

Frame	**Path Count**	**1st Highest**	**2nd Highest**	**3rd Highest**
Cornell Box	126	49.11%	2.93%	0.99%
Reflective Ball	129	26.49%	16.1%	5.27%
Robot	51	89.79%	4.93%	2.61%
Hairball	10	27.64%	17.34%	16.16%
Sponza	9	45.74%	14.91%	12.57%
Sky Primary	1	100.0%	0	0
Sky Reflect 1	2	98.67%	1.33%	0
Sky Reflect 2	2	99.99%	0.01%	0
Sky Direct	4	47.09%	43.89%	5.39%
Sky Indirect	4	53.72%	39.21%	3.63%
Window Primary	1	100.0%	0	0
Window Reflect 1	2	73.8%	26.2%	0
Window Reflect 2	2	99.31%	0.69%	0
Window Direct	4	71.07%	26.43%	1.57%
Window Indirect	4	55.29%	41.41%	1.73%

Table 3.3: Unique path count and top 3 frequencies of individual paths generated by Thread Compaction utility.

thread paths. For instance, reflection requires a different ray than refraction while opacity does not need any rays. Excluding the rays that miss, variable ray bounce lengths cause a high path count of 126 for Cornell Box, each path having a low frequency. The Reflect 1 pipeline invocation in Window differs from Reflect 1 in Sky due to a conditional `Return` statement that quits when the object collided with is not reflective. The Window frame contains a section of floor and wall that accounts for the 26% of threads taking a different path. Sky has no reflective material so all threads take the `Return` statement. ChameleonRT has fewer unique paths due to not using different material types and tracing fewer rays per thread than RayTracingInVulkan.

The paths leading to `traceRay` calls are influenced by application and scene design. Implementing complicated visual effects and geometry increases the variation of runtime behaviour. Separating ray tracing into different pipeline invocations, as in the case of Quake II RTX, reduces variance. When present, the skybox causes threads to take the same short path and creates an opportunity to introduce better work balancing.

3.4 The Cost of Instrumentation

Architecture-specific data, such as register allocation, cannot be captured using SPIR-V instrumentation because SPIR-V is not bound to an architecture. Conversely, SPIR-V instrumentation has the advantage that it is supported across architectures and vendors. SPIR-V instrumentation may impact performance due to the intangible effect it has on downstream compilers. However, manual high-level instrumentation is successful in improving performance in production games, so there is value in timing and performance data [34].

Figure 3.16 presents the frame latency and memory overheads of the respective utilities to produce the results in Section 3.3. Each utility may generate multiple outputs. For example, SIMT Efficiency creates the dynamic instruction count, graphics pipeline hotspots visualization, and full SIMT Efficiency analysis. The overheads range from 14× to 1502× increased latency over no instrumentation. In the worst case of SIMT efficiency for Reflective Ball,

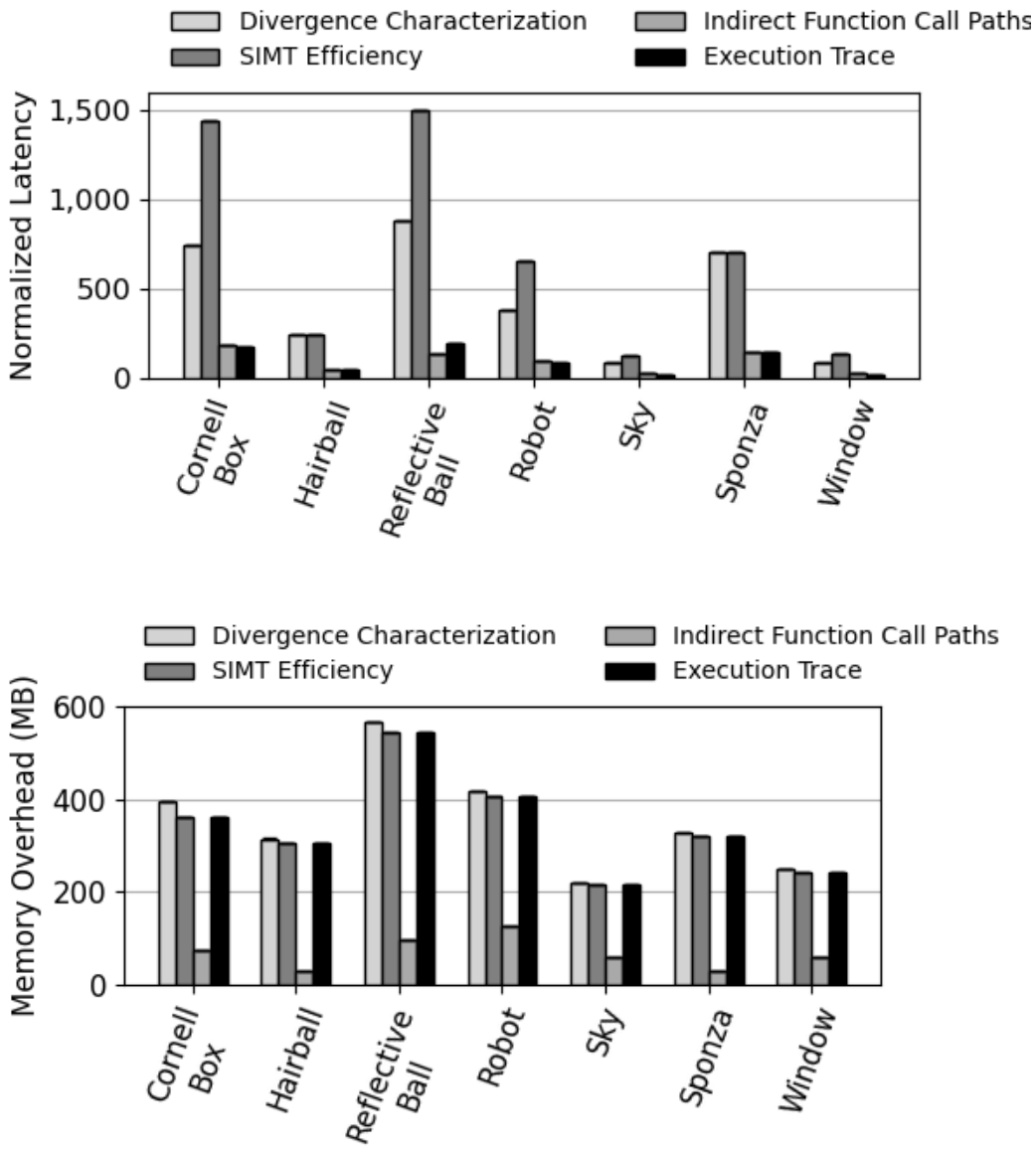

Figure 3.16: **Top:** Overhead of data collection and analysis for each mode. **Bottom:** Device data overhead for each mode.

the complete execution time is 100 seconds. This performance degradation is reasonable because program behaviour cannot otherwise be captured. Instrumentation and analysis happen offline during design of architecture or code generation solutions, therefore this execution time is manageable. Figure 3.16 also illustrates that analysis contributes significantly more overhead than the instrumentation itself. Execution Trace and SIMT Efficiency collect the same amount of data but have vastly different overheads. The most data produced by the instrumentation is 566 MB which is 9.4% of the available memory of the 1660Ti.

3.5 Conclusion

This chapter presented V-Vision, a framework for performing graphics applications profiling through automatic instrumentation without requiring the application source. V-Vision contributes SPIR-V instrumentation primitives that are used to construct instrumentation utilities. The instrumentation utilities produce static and dynamic data which are both consumed by analyses that return meaningful application performance data. V-Vision's execution trace assisted in revealing architecture specific behaviour, such as NVIDIA's independent thread scheduling, when executing applications implementing the recent ray-tracing extension to Vulkan. V-Vision is also capable of estimating the upper-bound benefit of hardware changes, such as performing thread compaction, for ray-tracing applications. In this chapter, we focus on control-flow divergence issues that plague ray tracing. However, V-Vision is not limited to studying control-flow as the instruction primitives readily map to applications such as value and memory-access divergence. V-Vision's compatibility with any Vulkan application will allow developers to glean new insights, and create their own utilities and analyses using the framework presented. As we demonstrate in the next chapter, the V-Vision framework is flexible and can form the basis for gathering high-level insights into ray-tracing application design in addition to the low-level insights outlined in this chapter.

Chapter 4

RayScope: Interactive Visualizations of Vulkan Ray-Tracing Applications Using Automatic and Application-Agnostic Instrumentation

Real-time ray tracing is experiencing significant momentum in several markets, including gaming consoles, high-end desktops, and mobile devices [18]. This momentum is driven by hardware acceleration and first-class support in graphics APIs [19], [36], and manifests in the ever increasing adoption of real-time ray tracing in the gaming industry [18].

Despite recent hardware acceleration and Application Programming Interface (API) standardization, real-time ray tracing still imposes more challenges in functionality, performance, and energy efficiency than traditional rendering using rasterization. Developers missing important flags that are available in the Vulkan API is a common issue stemming from the API's complexity. However, aside from API complexity, implementing ray-tracing applications is, in itself, a sophisticated process with a large surface area for bugs and performance pitfalls [15]. Ray-tracing applications must interact with geometry, often generated by a third-party. Therefore, the interaction of the runtime ray-tracing implementation and scene geometry is often unpredictable. For

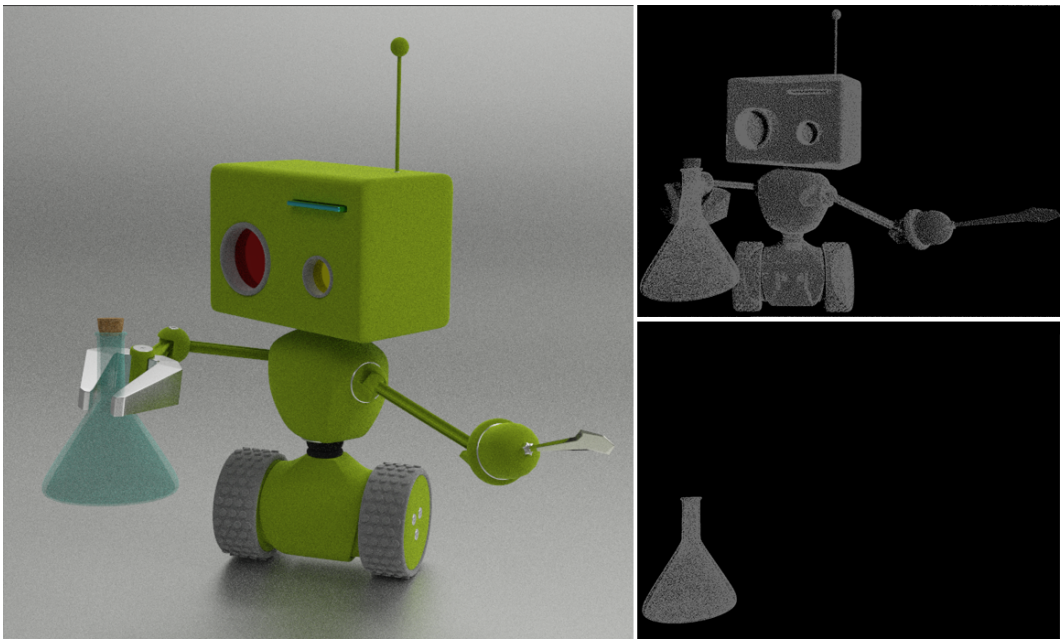

Figure 4.1: **Left:** The rendering of an opaque robot holding a transparent vial by VkRaytrace, a Vulkan ray-tracing application. **Top Right:** A point-cloud visualization of all any-hit shaders executed by the application, enabled by RayScope, reveals that all parts of the opaque robot are being tested for transparency—this is an implementation oversight. Any-hit shaders selectively ignore ray intersection and are used to implement transparency in VkRaytrace. **Bottom Right:** After applying RayScope's performance recommendation to mark parts of the robot as opaque, the RayScope's point-cloud visualization reveals that executions of any-hit shader are reduced by 96.8% when they are only executed on the transparent vial.

example, geometry may have imperceptible holes that allow rays to escape and incorrectly affect the lighting [9]. Geometry may also trap rays in small dark crevices where ray bounces do not improve the final rendered image. Vulkan ray-tracing applications must also specify the minimum allowable distance for a ray intersection. There is little guidance on how to choose this value but improperly setting it leads to unnecessary computations. Section 4.3 shows that these issues can be detected in Vulkan ray-tracing applications and visualized using RayScope.

Figure 4.1 shows an example of a hidden ray-tracing performance pitfall. The application renders a scene containing an opaque robot holding a transparent test tube. The visually correct scene is rendered using Vulkan ray tracing [19]. However, the picture in the top right of Figure 4.1 shows that rays must test every pixel of the robot's opaque body for transparency, leading to unnecessary computations. To avoid these computations the Vulkan ray-tracing specification provides a flag to label opaque objects. The picture on the bottom right of Figure 4.1 shows that 96.8% of the run-time transparency tests can be eliminated by adding the opaqueness flag to the robot's body.

Many tools, such as RenderDoc [21], NSight Graphics [43], Intel GPA [20], Microsoft Pix [37], and Vulkan Vision [49], are available for visualizing the incremental rendering process in modern rasterization-centric graphics APIs . These tools typically help in isolating each draw call within a frame and in isolating the effect of vertex and fragment stages using the transform feedback process [21]. Some of these tools have added recent features that help in visualizing the impact of scene geometry in modern ray-tracing APIs [37], [43]. However, to the best of our knowledge, there is no support for visualizing ray paths in modern ray-tracing APIs, a non-trivial process as shown in this chapter.

This chapter addresses the following problem: *Is there useful application-agnostic ray-tracing information available in Vulkan, how should the information be captured, and how should the data be represented to assist developers in debugging, profiling and design tasks?*

This chapter presents RayScope; the first tool that uses the Vulkan ray-

tracing API for automatic ray visualization in ray-tracing applications. RayScope is an open-source platform-independent tool with a workflow that automatically captures detailed ray-tracing information and features an interactive visualizer to display this information. RayScope's visualizer accurately represents procedural geometry using runtime execution information that prior state-of-the-art tools only approximate.

RayScope combines compiler instrumentation of the shader code with instrumentation of Vulkan API calls to automatically and transparently collect ray-tracing information from a Vulkan application. This instrumentation is performed using a Vulkan validation layer that intercepts and modifies Vulkan API calls from the application before they reach the driver.

We characterize the application-independent geometry information, ray information, and ray-geometry interaction information in the Vulkan API. RayScope automatically collects this ray-tracing information using Vulkan API and shader instrumentation. RayScope emits the ray and geometry data as human-readable files to encourage use with other visualization tools.

RayScope produces a human-readable ray-tracing execution file containing each ray's `start` and `end` location in the scene as well as intermediate shader executions. The geometry and execution information is fed into RayScope's interactive visualizer, implemented with the Unity game engine [56]. The visualizer also analyzes the ray information to present features of interest, such as the ray that bounced most, to developers. RayScope's visualizer, shader instrumentation, and Vulkan API instrumentation are open source and the ray-tracing execution file is human readable to encourage custom integrations with games or game engines.

In addition to visualization, RayScope's instrumentation provides sufficient data to detect common issues in Vulkan ray-tracing applications. Some issues stem from a poor understanding of the sophisticated Vulkan API and others are pathological to ray tracing, such as rays becoming trapped in geometry configurations.

Through case studies, we demonstrate that RayScope is effective in assisting debugging and supporting application-design tasks. These case studies

demonstrate that RayScope produces visualizations that allow long-standing bugs to be quickly identified; provides effective performance recommendations resulting in tangible performance benefits; and allows emergent issues with well-studied models, such as Sponza, to be identified.

The rest of this Chapter is organized as follows. Section 4.1 discusses our characterization of application-independent ray-tracing information and the modified Vulkan Validation Layer used by RayScope to automatically capture it. Section 4.2 presents RayScope's operation and interactive visualizer. Section 4.3 demonstrates how RayScope can be applied for debugging and application design tasks through case studies of Vulkan ray-tracing applications. Conclusions are presented in section 4.4.

4.1 Ray-Tracing Information Capture in RayScope

This section presents a characterization of application-agnostic ray-tracing information in Vulkan and discusses RayScope's mechanisms to overcome challenges in collecting the data outlined in the characterization. RayScope's data collection is automated and requires no changes to the application's source code or recompilation.

Figure 4.2 shows our characterization of Vulkan's ray-tracing specification. The blue box on the left contains the geometry data that the application must initialize before ray-tracing can execute. The geometry information is derived from the specification of the Vulkan acceleration structure. Section 4.1.1 discusses the difficulties of capturing fine-grained geometry information and RayScope's solutions to overcome them. The orange box on the right contains the ray-tracing-execution information that is implicitly generated when the GPU executes the ray tracing binary. This information is derived from the Vulkan ray-tracing pipeline and can be further characterized into conditional geometry and ray behaviour events. Conditional geometry events determine the validity of ray intersections with geometry, based on a runtime condition. Ray behaviour events receive the result of the ray traversal and determine whether the ray

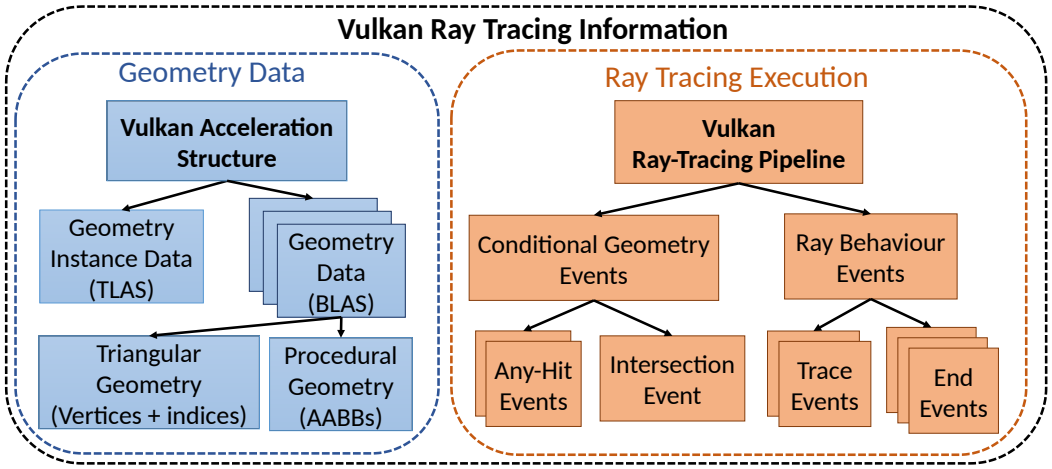

Figure 4.2: Characterization of application-agnostic information available implicitly in the Vulkan ray-tracing API. We divide the ray-tracing information into two main categories: raw geometry data (blue) and runtime execution data (orange). To be able to effectively visualize the behaviour of arbitrary Vulkan applications, RayScope must gather all the data in both trees.

should end or be traced again. Section 4.1.2 specifies RayScope's mechanism to identify and capture the events at application runtime. By gathering all the information shown in Figure 4.2, RayScope captures a complete view of the application's rays, geometries, and ray-geometry interactions.

4.1.1 Geometry-Data Capture

To collect scene geometry RayScope must follow the flow of geometry data from creation to usage through Vulkan API calls. RayScope identifies the geometry buffers used to build the AS and infers the type of data that they contain, either triangular or axis-aligned bounding box (AABB). Vulkan applications copy geometry data between buffers and consequently RayScope must track the application's geometry data through copy operations.

RayScope captures all AS information because the AS defines the geometry side of ray-geometry interactions. RayScope's instrumentation outputs each BLAS as a human-readable OBJ file [29]. The TLAS is output as a text file containing the transform matrices and BLAS file name of each geometry instance.

Capture Mechanism: Figure 4.3 shows the instrumentation needed to

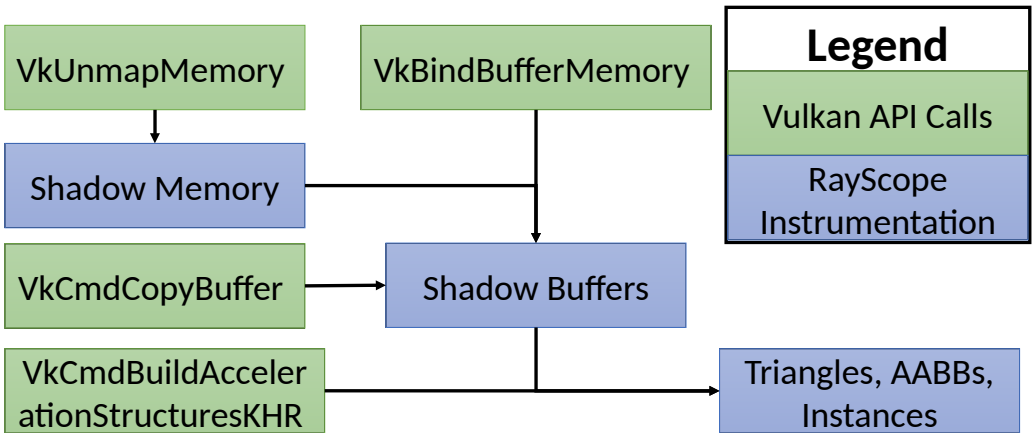

Figure 4.3: RayScope's graphics instrumentation for capturing geometry instance data and geometry data.

capture geometry data. The following instrumentation is also performed by RayScope, but omitted from the figure for clarity: tracking the creation and destruction of buffers; and tracking device addresses for buffers, memory allocations, and acceleration structures. The Vulkan API calls referenced in this section are defined in greater detail in Table 2.1.

The device memory is populated with host data using `vkMapMemory` that generates a host-accessible pointer. The application may call `vkUnmapMemory` to release the host-accessible pointer. Shadow copies of device memory are created immediately before the application calls `vkUnmapMemory` because the memory must be populated.

When `vkBindBufferMemory` is called, RayScope creates a shadow copy of the buffer as a subset of the shadow memory region. If `vkCmdCopyBuffer` is called, RayScope creates another shadow buffer entry for the destination buffer. RayScope uses the contents of the shadow buffers to reconstruct the geometry and the geometry instances when `vkCmdBuildAccelerationStructuresKHR` is called.

4.1.2 Ray-Tracing Execution Capture

To reconstruct a detailed view of ray-tracing in an application-agnostic manner, RayScope must leverage information present in the runtime execution of the ray-tracing pipeline. To provide developers with recommendations of hardware

Event Name	**Built-in Variable**	**Built-in Function**	**Shader Stage**
Trace		✓	
Trace Miss-Only		✓	
Miss			✓
Closest Hit	✓		✓
Implicit Hit			✓
Any Hit	✓		✓
Intersection		✓	
Ignore Intersection		✓	

Table 4.1: Characterization of ray events that are captured from the Vulkan ray-tracing pipeline and the application-independent information used to generate them. The column built-in variable, built-in function, and shader stage receive a checkmark if they are used to generate the event.

utilization, RayScope must also capture hardware-level information in the execution-trace. To accomplish both of these tasks, RayScope inserts shader instrumentation to record the values of key variables and the executions of key shaders.

The shaders in the Vulkan ray-tracing pipeline define how rays and geometry behave. The shader stages in the ray-tracing pipeline receive ray-tracing information through builtin variables. For example, the any-hit and closest-hit shaders receive the distance of the hit from the ray origin in the `glHitTEXT` builtin variable. The shader stages themselves encode information about the execution. For example, the miss shader is executed if a ray misses geometry. The properties of shader stages and builtins are application independent because they are specified in the Vulkan API.

Table 4.1 shows the ray-tracing execution events that RayScope captures. The table also shows the information, given by the Vulkan API, that is used to generate the event. The events can be classified into two types: conditional geometry and ray behaviour, as shown in Figure 4.2. The Trace, Trace-Miss-Only, Miss, Closest-Hit, and Implicit-Hit events completely specify ray start

points and ray end points. The Any-Hit event records all hits that were considered for validity. The Intersection event records all hits with application-specified bounding boxes. The Ignore-Intersection event records all hits that were ignored. These events allow RayScope to reconstruct the geometry and ray behaviour in an application-agnostic manner.

RayScope's shader instrumentation records the assignment of threads to subgroups using builtin subgroup functions. Each ray-tracing execution event records the thread that generated that execution. Using this information, RayScope can reconstruct thread and subgroup behaviour. RayScope uses the thread and subgroup behaviour information for visualization and automatic recommendations.

```
0:410:trace -0.199989 -0.500232 5.499992,
chit -6.022733 3.282307 -1.041899,
trace -6.022733 3.282307 -1.041899,
chit -9.976090 -1.805820 -1.237896
```

Figure 4.4: Formatted ray-path from RayScope's ray-tracing execution file. The first two values 0 and 410 are the thread id and subgroup id respectively. Thereafter RayScope reports ray events and their positions. This example contains 2 rays' origin and closest-hit positions.

Figure 4.4 shows an excerpt of RayScope's ray-tracing execution information file. The complete file contains the runtime information for all threads and all ray-events from the pipeline execution. The file is designed to be human-readable to aid in debugging and to encourage custom integrations with other visualization tools.

Capture Mechanism: RayScope creates an instrumentation pass using the framework Vulkan Vision [49] to capture detailed thread-level execution data. RayScope's shader instrumentation pass uses the following mechanisms for generating the ray-tracing pipeline events:

1. **Trace:** This event records the values of the *ray origin*, *ray direction*, and *Tmax* arguments of `traceRayEXT`. The data is recorded immediately before `traceRayEXT` executes.

2. **Trace-Miss-Only:** This event is the same as **Trace** but only records data for `traceRayEXT` calls where the flag `glRayFlagsSkipClosestHitShaderEXT` is present. This flag indicates that the closest-hit shader should not be executed for the ray. Therefore, a call to `traceRayEXT` either executes the miss shader or executes nothing. RayScope classifies this type of ray as a miss-only ray because it tests if there was a miss.

3. **Miss:** This event records that the miss shader executed.

4. **Closest Hit:** This event records the value of `gl_HitTEXT` and `gl_InstanceID` when the closest-hit shader is executed.

5. **Implicit Hit:** This event is recorded when there was no **Miss** event for a **Trace-Miss-Only** event.

6. **Any Hit:** This event records `gl_HitTEXT` and `gl_InstanceID` when the any-hit shader is executed.

7. **Intersection:** This event records `gl_HitTEXT` when `reportIntersectionEXT` is executed.

8. **Ignore Intersection:** This event records that `ignoreIntersectionEXT` executed.

4.1.3 Validation Layer

Vulkan Vision provides a shader instrumentation framework for Vulkan through a validation layer [49]. RayScope implements a new instrumentation pass and a new analysis using Vulkan Vision to capture ray-tracing execution events. Moreover, as discussed, RayScope instruments the Vulkan API functions that are responsible for building ray-tracing geometry.

4.2 RayScope Operation

This section details the operation and the user interface of RayScope's interactive visualizer shown in Figure 4.5. RayScope instruments the API calls

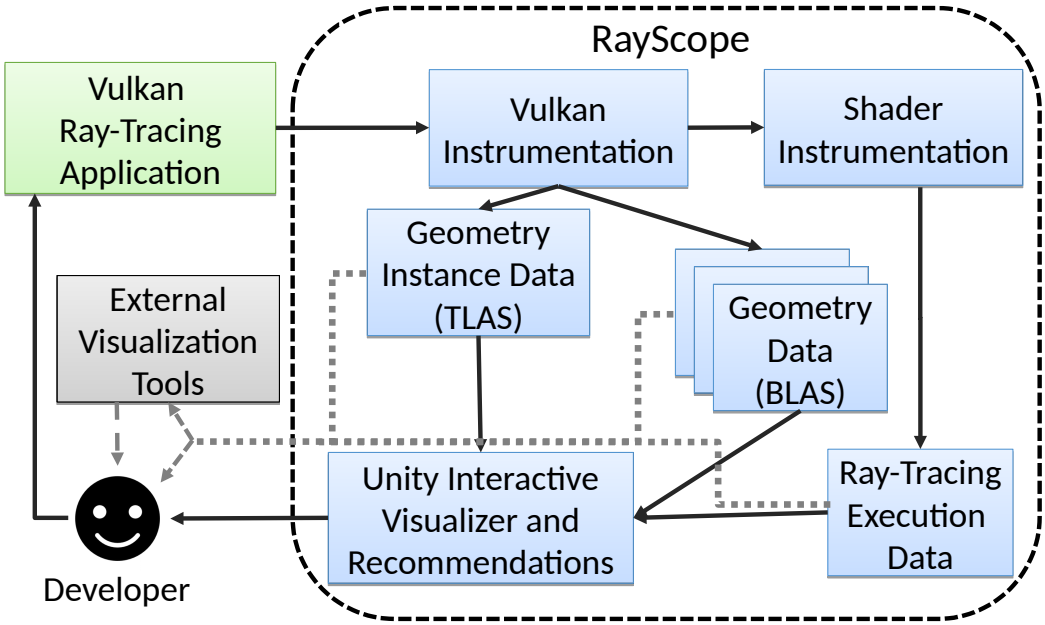

Figure 4.5: Operation of RayScope. RayScope uses instrumentation to capture geometry and ray-tracing execution data. RayScope's interactive visualizer analyzes the data to produce visualization and recommendations. The dotted lines indicate that RayScope's instrumentation data can be integrated with other visualization tools or interpreted by the developer directly.

and shaders to intercepts API calls generated by a Vulkan application. This instrumentation produces the geometry instance data and the geometry data. The shader instrumentation produces the runtime ray-tracing execution data. RayScope's visualizer automatically analyzes the raw geometry and execution data and converts it to a visual form. A developer may interact with RayScope's visualizer to understand and improve their ray-tracing implementation. A developer may also directly inspect RayScope's instrumentation data or interact with another visualization using RayScope's data.

RayScope's visualizer is implemented in Unity [56] because Unity offers high-level constructs for visualization and interactivity. This implementation leverages the builtin rendering and UI features with moderate customization to create RayScope's visualizer. Another benefit of Unity is the editor view. The editor view allows geometry and rays to be recolored, selectively enabled, disabled, or serialized at runtime. This level of control allows new features to be quickly prototyped and one-off issues to be investigated without writing any

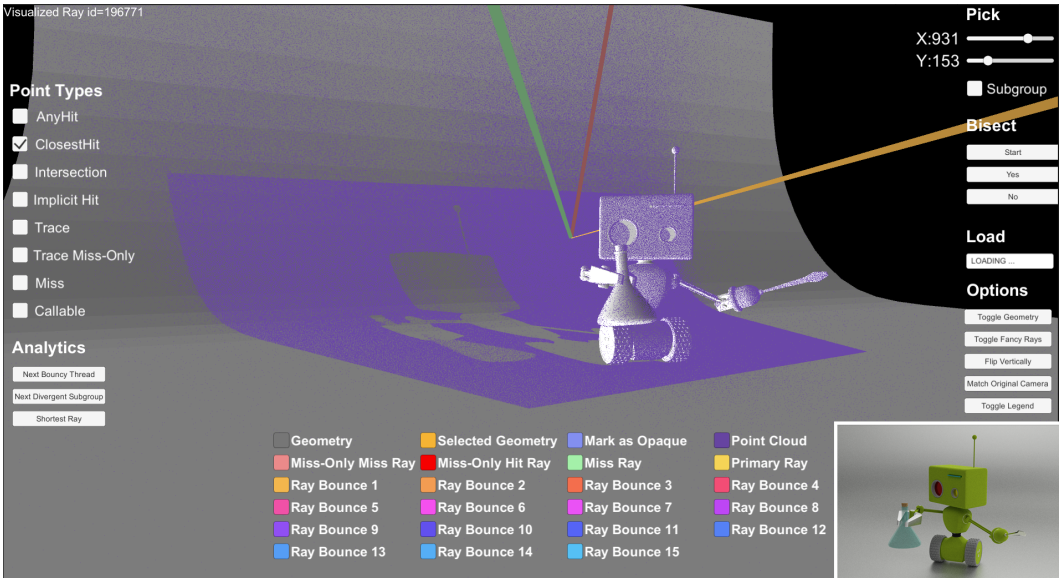

Figure 4.6: Screenshot of RayScope's visualizer with VkRaytrace's renderered image overlaid in the bottom right. The image shows the VkRaytrace scene with geometry (gray), closest-hit point cloud (purple), and ray-pick visualization enabled. The yellow ray is a primary ray from the camera. The red ray is a miss-only ray that misses. The green ray is a secondary ray that misses. The purple rectangle projected onto the Robot and its backdrop are the points in the point cloud corresponding to primary rays from the camera. The UI controls for interacting with the visualizer are around the borders of the screen. The legend, shown at the bottom, is dynamically generated based on user-chosen colours.

code. Unity has first-class support for loading and viewing 3D objects. We used this feature to validate RayScope's geometry reconstruction.

4.2.1 Graphical User Interface

Graphics applications are inherently visual and lend themselves to being analyzed visually [15]. RayScope's visualizer shows information that was previously unavailable to developers, such as ray paths or shader executions. RayScope's visualizer offers high-level features based on synthesizing the raw geometry and the ray data, such as allowing a developer to pick a ray path by thread id, in addition to visualizing geometry and ray paths. A developer may interact with these features through the UI of RayScope's visualizer.

Figure 4.6 shows a screenshot of RayScope's visualizer. The image in the bottom right of Figure 4.6 shows the rendered image of the application, VkRaytrace [11] that is being visualized. The visualizer is presenting the geometry of a Vulkan ray-tracing application overlaid with ray and point cloud information. The geometry is a robot holding a vial and a rounded backdrop coloured as shades of gray. The point cloud, showing the locations of all closest-hit shader execution locations, is colored purple. The point cloud shows a rectangle projected on the robot and its backdrop. This rectangle corresponds to the area hit by primary rays from the camera. The rectangle can be understood as showing a light source at the position of the camera analogous to shadow mapping [16]. The rest of the point cloud shows the other closest-hit executions resulting from bouncing rays. The visualized rays are computing the color for the pixel at $x = 931$ and $y = 153$ as selected using the pick feature. The yellow ray begins on the right and passes behind the robot before colliding with the backdrop. This collision spawns a secondary ray (green) and miss-only ray (red).

Geometry

Visualizing the geometry on its own is insufficient for understanding the runtime behaviour of the ray-tracing pipeline. However, showing both rays and geometry encapsulates the runtime behaviour. RayScope's visualizer offers the ability to

render the geometry data from a Vulkan application. This visualization lets a developer clearly see the geometry interaction with rays.

RayScope's visualizer uses a multithreaded implementation to load the scene geometry. The visualizer launches a thread to load the TLAS file containing the geometry instances. Each geometry instance has a transform matrix and a reference to a TLAS file. After determining all geometry instances, the visualizer launches a thread for each unique TLAS file. A thread loading a TLAS file constructs a mesh representing the triangular geometry and the bounding boxes from the original Vulkan application. After all meshes are created, Unity GameObjects are instantiated for each TLAS instance. Finally, the GameObjects are positioned according to the transform matrix from the geometry-instance information.

A benefit of maintaining the TLAS and BLAS information in the visualizer is that instances referencing the same TLAS can share the same mesh and thus reduce the amount of data required by the GPU. Another benefit is that RayScope can graphically represent information about the TLAS as shown in Section 4.3. Finally, since RayScope analyses the OBJ file format [29] for the TLAS file, it is easy to examine each TLAS individually. Through this analysis, we found several instances of applications using copies of the same geometry that could be deduplicated using instancing.

Effectively rendering the geometry instances requires the values of the surface normals. However, the normals of the geometry cannot be automatically deduced from the Vulkan specification. Some applications define custom vertex normals and mix triangle winding directions within a single BLAS. Therefore, there is no reliable way to determine the surface normal using instrumentation.

RayScope generates new normals for the triangles in the geometry based on a fixed winding direction. This normal generation results in normals that may be flipped causing issues with Unity's culling and lighting. RayScope uses a custom shader that disables back-face culling to ensure that no triangles are incorrectly culled. The shader also performs flat shading because lighting based on a flipped normal results in poor visualizations. Flat shading is acceptable because the goal of the visualizer is to present data and not visual attractiveness.

Disabling back-face culling may impair performance but in our experience there was no noticeable degradation.

Ray Paths

Vulkan ray-tracing applications define how rays behave in shaders. For example, shaders define the number of times that a ray bounces and the direction of the ray for each bounce. These shaders are executed hundreds of thousands of times per pipeline execution. Vulkan developers have little recourse other than trial and error to implement ray interactions correctly. To solve this problem, RayScope's visualizer generates a comprehensive set of ray paths for all rays traced for a single thread.

RayScope's visualizer loads the ray-event data asynchronously to avoid stalling the main thread. The visualizer constructs a mapping from thread id to a list of ray events. The visualizer also constructs a mapping from subgroup id to a list of thread ids. This information forms the basis for several visualizations and recommendations.

To render ray paths, RayScope's visualizer has two modes for visualizing rays: constructing a mesh with line topology, or using Unity's LineRenderer. The LineRenderer component has a very significant overhead when scaled to millions of rays but has more advanced visualization capabilities, such as a narrowing shape indicating ray direction. Therefore, we use the LineRenderer when the number of rays to visualize is low and the mesh when the ray count is high. RayScope's visualizer can show arbitrary rays, allowing visualizations such as all primary rays or all miss-only rays.

Another benefit of constructing a 3D mesh of the rays is that the ray visualization intuitively composes with geometry visualization. For example, if instead the rays were rendered in screen space, rays would occlude geometry despite being behind it in the original Vulkan application.

Point Cloud

Vulkan geometry can be composed either of triangles or procedural objects. RayScope visualizes triangle-based geometry using meshes as described above.

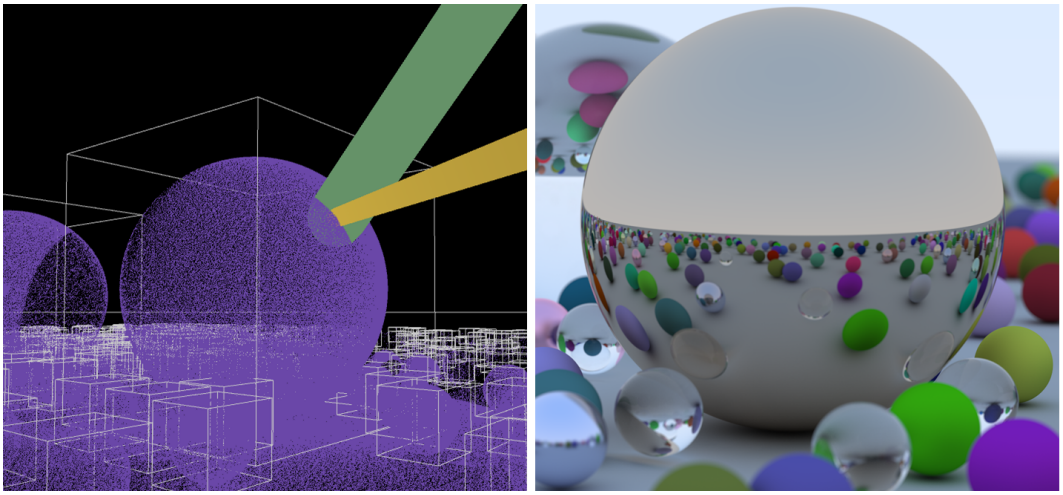

Figure 4.7: **Left:** RayScope's visualization of procedural objects from application RayTracingInVulkan. The visualization combines bounding box visualization (gray boxes), closest-hit point cloud (purple), and ray paths (green + yellow ray). The boxes visualize the application-provided bounding box for each procedural object in the scene. The point cloud visualization is crucial to observe the true shape of the object. Otherwise the wireframe boxes would be the only visualization of procedural objects. **Right:** Screenshot of application's rendered image.

This approach does not work for procedural objects that are defined at runtime by executing intersection shaders. To visualize procedural objects, RayScope offers a point-cloud visualization. Figure 4.7 shows RayScope's visualization of procedurally defined spheres and their application-defined bounding boxes. The point-cloud visualization shows the spherical shape of the object which is only known at application runtime. Visualizing the bounding boxes also allows developers to check if they are sized correctly to minimize wasteful intersection shader executions.

For reference, Figure 4.8 shows the prior state-of-the-art visualizations of procedural objects in NSight Graphics [43] and PIX [37]. The images were captured from a DirectX12 [36] application because NSight Graphics crashes when used with Vulkan applications developed on the up-to-date Khronos ray-tracing extension and PIX does not support Vulkan. RayScope could be applied to DirectX ray-tracing applications once the DXVK [51] compatibility project supports DirectX12 and ray-tracing. In both NSight Graphics and PIX, the visualization renders a rectangular prism with the same dimensions and

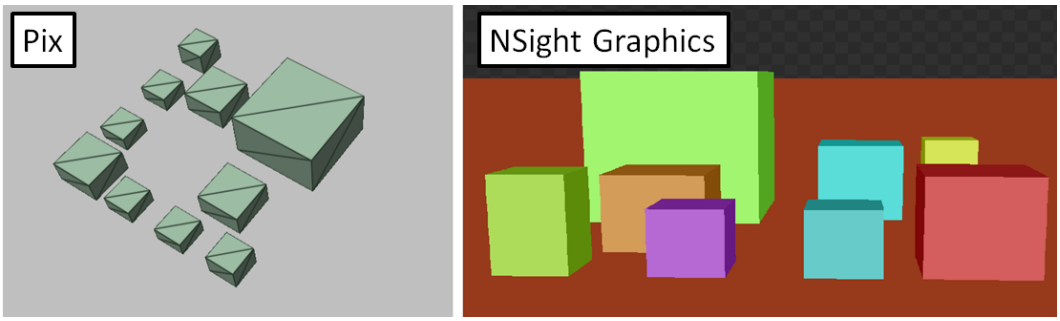

Figure 4.8: PIX and NSight Graphics visualization of procedural objects. The objects' true shapes are hidden in the visualization.

position as the AABBs of the application's object. However, this visualization conceals the true shape of the objects and obfuscates how rays would interact with them. Therefore, neither visualization allows the developer to check if the AABB was sized correctly, however such a check is possible in RayScope. In contrast, RayScope shows the AABB as a wireframe and the shape of procedural objects hit by rays using the point-cloud visualization.

The point-cloud visualizer is useful for more than just visualizing procedural objects. RayScope's visualizer provides control to the user to select the types of points to be included in the point cloud. The example in Figure 4.1 illustrates that the point cloud of any-hit executions can be used to evaluate if objects are correctly being marked as opaque. Generally, RayScope's point-cloud visualization shows implicit aspects of the ray-tracing pipeline, such as shader executions, in 3D.

RayScope's point-cloud visualization also shows information that is not visualized in a ray path. For example, the traceRay operation requires that all candidate hits be established to determine the closest hit. If bounding boxes of procedural objects overlap, multiple intersection shader executions may be required to determine all candidate hits. Comparing RayScope's point cloud visualization of intersection events to closest-hit events visually shows how many wasted intersection shader executions there were. In general, RayScope's point cloud visualization is key to visualizing dynamic geometry interactions.

RayScope generates a point cloud by scanning the ray-tracing information file for all events in a user-defined set of ray-event types. A mesh with point

topology is generated with a point for every ray event. If the user changes the set of ray-event types, a new mesh is generated according to the new set. Both Figure 4.6 and Figure 4.7 show that the point-cloud visualization, geometry visualization, and ray visualization can be combined.

Pick Coordinate

Rendering issues may manifest as visual artifacts in a subset of pixels. Currently, debugging these issues requires invasive changes to the application or a process of trial and error. In these cases, it is extremely helpful to understand the precise behaviour of the threads computing the results for those pixels.

RayScope's visualizer allows a user to select a thread coordinate using sliders for the x and y value. The visualizer shows the ray paths of the user-selected thread. Typically, the thread coordinate corresponds directly to the pixel coordinate. This feature is designed to enable developers to debug pixels with incorrect colour values. Additionally, the feature may be used by users to better understand how the ray-tracing algorithm proceeds for certain threads and pixels.

Bisecting Rays

Rendering issues may not manifest as visual artifacts. This is emphasized in ray tracing due to the widespread adoption of denoising [54]. Therefore, to ensure that rays are being utilized effectively, it is necessary to have a visualization other than the final image. RayScope offers the ability to view all ray paths and then bisect them to detect suboptimal ray behaviour.

A user may display all ray paths for a Vulkan ray-tracing pipeline invocation by pressing `Start`. RayScope's visualizer allows a user to choose `Yes` if they see the ray of interest or `No` otherwise. A user can identify an issue by viewing all ray paths and then narrow down the issue to a single ray path in seconds. The id of the offending thread is reported allowing a more detailed inspection of the thread in the ray file.

4.2.2 Recommendations

RayScope's ray-tracing execution data file contains all rays, often millions, generated for a single pipeline invocation. This amount of data is difficult to sift through manually but contains rich execution information. RayScope's visualizer analyzes this data and allows developers to view recommended features of interests through the UI.

Opaque Flags: Any-hit shaders are not executed for ray intersections with opaque objects. RayScope automatically recommends that the developer should consider marking geometry as opaque if RayScope detects that the any-hit never ignored a collision. In that case, marking the object as opaque would not change the closest hit. The example in Figure 4.1 demonstrates the effect of implementing this recommendation. Section 4.3 discusses this feature of RayScope in more detail.

Work Hotspots: RayScope's visualizer recommends the threads with the most ray-tracing work to the user. This feature is designed to assist in understanding the ray-tracing algorithm's dynamic interaction with scene geometry. For example, rays can become trapped and bounce many times in a small space with little overall colour contribution. RayScope's visualizer offers a button to quickly cycle through rays with maximum work.

Poor Hardware Utilization: RayScope's visualizer highlights the subgroup that has the lowest average of active threads for each ray-tracing operation. This feature highlights aspects of the scene and ray-tracing pipeline that result in poor hardware utilization of the GPU. In this case, the developer could consider switching to a wavefront ray tracer [27] or limit the bounce count to improve the hardware utilization. RayScope's visualizer also has a button to cycle between divergent subgroups.

4.3 Case Studies

This section illustrates, through the study of four Vulkan ray-tracing applications shown in Table 4.2, how RayScope effectively assists both debugging and application design tasks. Unless otherwise specified, the data was collected on

Application	3D Obj	Issue Types Discovered
ChameleonRT [57]	Sponza [35]	2
RayTracingInVulkan [14]		2
VkRaytrace [11]		1
PBRVulkan [62]		2

Table 4.2: Applications studied with RayScope. Additional 3D models were only required for ChameleonRT. RayScope aided in discovering different types of issues in all applications.

an NVIDIA RTX 3080 with driver version 461.40.

Bugs in ray-tracing applications may arise from the complexities of the ray-tracing or of the Vulkan specification. RayScope's visualizations allow a developer to observe the execution of the ray-tracing pipeline visually and can be of great assistance in discovering unknown bugs or investigating known ones. The automatic instrumentation in RayScope enables developers with little or no application expertise to find and investigate bugs.

The geometry in a scene has a large impact on the runtime behaviour of the ray-tracing implementation [9], [49]. It is difficult to detect poor ray-geometry interactions, such as trapped rays, without an execution trace. RayScope's visualizations can aid in designing scenes that are ray-tracing friendly by eliding wasteful rays. For example, the visualizations can highlight sections of the geometry allowing rays to leak out, waste computation, and potentially cause visual artifacts. The visualizations of the patched geometry also confirm that the fix was successful. In addition to detecting poor ray behaviour, RayScope also detects poor hardware utilization in the application. RayScope's recommendations highlight issues, such as ray incoherence, that were previously challenging for developers to detect and investigate. The remainder of this section details specific examples of RayScope assisting in finding bugs, poor ray-geometry interactions, and application design issues.

4.3.1 Path-Tracer Bug

Bugs in renderers that do not manifest visually degrade performance and can remain undiscovered for a long time. Gribble *et al.* found an unknown bug causing performance degradation in a CPU path tracer [15]. RayScope's visualizations facilitated the discovery of an analogous bug in a Vulkan path-tracing application PBRVulkan [62]. The following discussion of this bug highlights that ray tracing is difficult to implement correctly independent of the tracer.

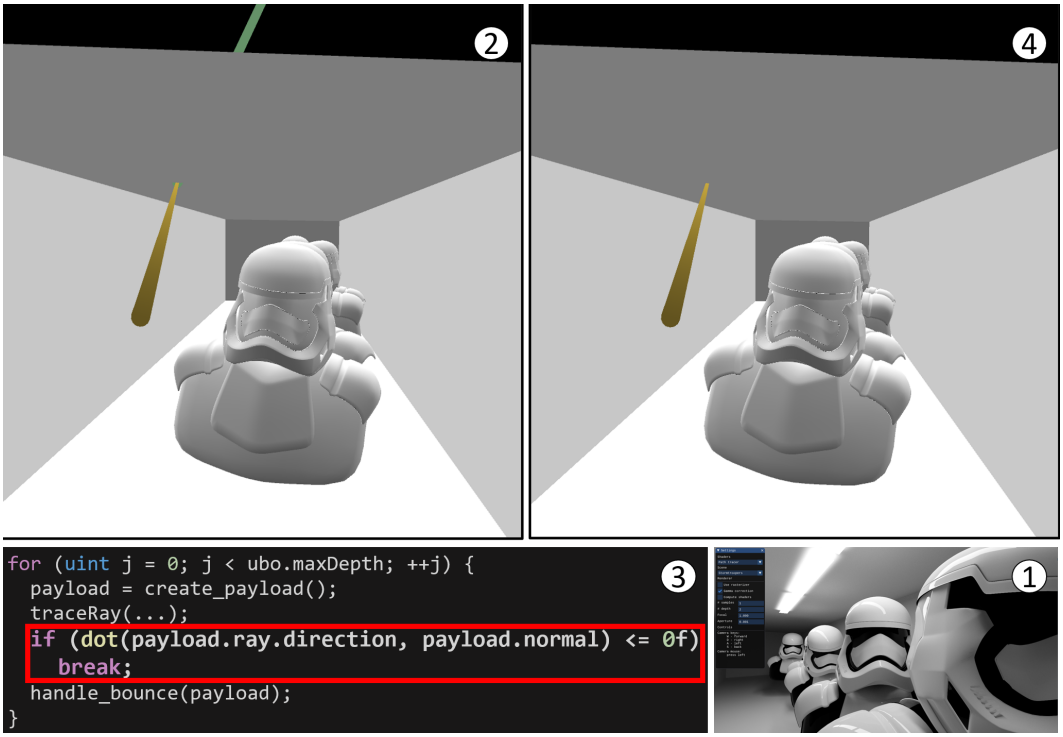

Figure 4.9: **Top Left:** RayScope's visualization of PBRVulkan's default path-tracing implementation. The yellow ray is generated from the camera. The green ray is the secondary ray incorrectly generated from the back-face of the box. **Top Right:** RayScope's visualization of PBRVulkan after fixing a bug causing rays to be generated from the back faces of triangles. RayScope's visualization shows that the green ray is no longer traced. **Bottom Left:** Pseudocode for our patch where our addition is outlined with a red box. The if statement terminates rays before they can be generated from the back faces of triangles. **Bottom Right:** Rendered image of PBRVulkan.

Figure 4.9 shows RayScope's visualization of PBRVulkan's path-tracing implementation before and after fixing a bug that caused rays to be generated

from the back faces of triangles. Conceptually, this bug is equivalent to light passing through opaque objects. This bug did not result in any visual artifacts, afflicted all scenes, and has been present in PBRVulkan since its inception. It is likely that this bug would have remained undiscovered without RayScope's visualization.

Figure 4.9 shows the visualizations that led to the discovery and fixing of the bug in PBRVulkan. The scene is enclosed by 5 faces of an opaque box ①. Visualizing all rays reveals that rays escaped through the solid faces of the box. This is counterintuitive because, if implemented correctly, all rays would bounce within the box and only escape through the open face. Rays generated from the back face of the box are clearly incorrect in this scene as they can never hit another object or light source. Bisecting the rays shows a single ray that exhibits the issue ②. This ray hit one of the faces of the box, and the next bounce is issued from the back-face of the triangle. The issue manifests as an incorrect direction of secondary ray bounces. Adding a check ③—highlighted with a red box—that breaks out of the ray-bounce loop when the dot product of ray direction and the normal is negative fixes the issue. RayScope's visualization of the same ray shows the issue no longer occurs after the successful fix ④. 3.3% of all rays traced for the stormtrooper scene were generated from the back-faces of triangles. Thus, fixing this bug by breaking out of the bounce loop reduces the number of rays by 3.3%.

4.3.2 Ambient-Occlusion Bug

Shader bugs often manifest as visual artifacts in the final image of rendering applications. When the issue only influences a subset of pixels, the developer must determine the responsible effect or material. Even knowing the affected pixels, it is time consuming to find the cause of the issue. RayScope improves the situation by collecting a complete trace of each thread obviating the need for tedious *printf* debugging by the developer. RayScope's visualizer also provides the pick feature to visualize all the rays traced for a given thread and its pixel.

Figure 4.10 shows PBRVulkan's Ambient Occlusion (AO) implementation

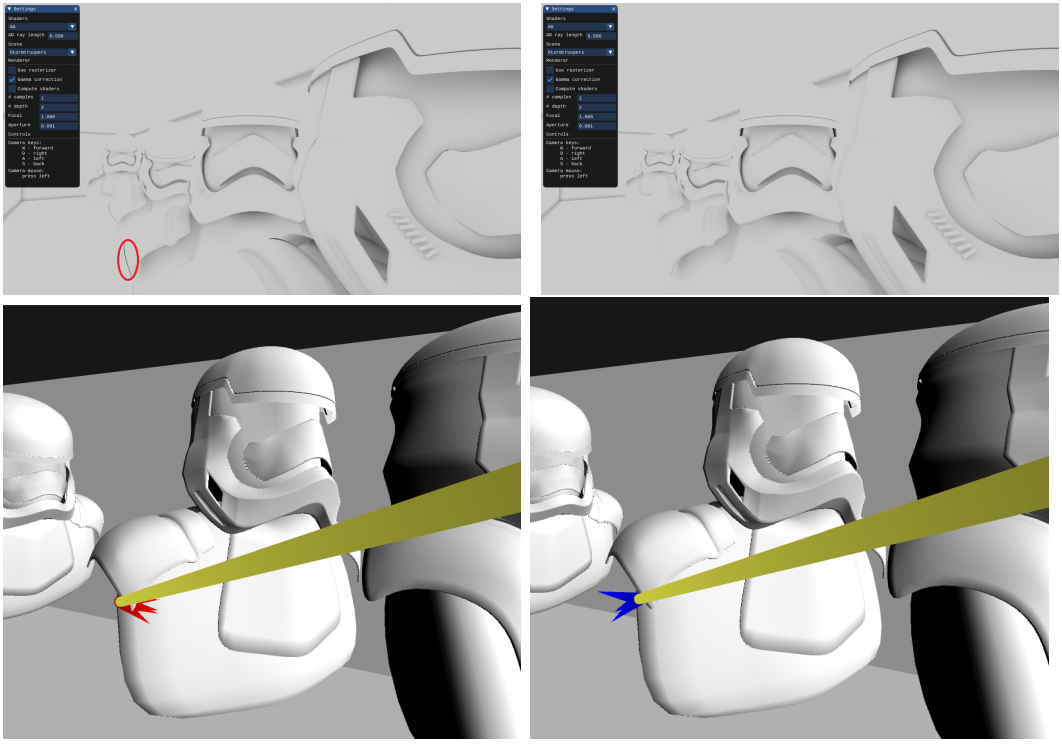

Figure 4.10: **Top Left:** PBRVulkan's AO image with dark visual artifact in area that is not occluded. The incorrectly coloured pixels are highlighed with a red oval. **Top Right:** PBRVulkan's AO image after bug is fixed and visual artifacts are no longer present. **Bottom Left:** RayScope's geometry and visualization of ray path generated by the incorrect AO implementation. The yellow ray is the primary ray generated from the camera. The red rays are occlusion rays being generated pointing into geometry. All occlusion rays hit geometry giving the pixel a very dark color. **Bottom Right:** RayScope's geometry and visualization of ray path generated by patched AO implementation. The blue rays are occlusion rays that are now pointing away from geometry.

before and after fixing a bug causing visual artifacts. The figure also shows RayScope's visualization that guided the fixing of the problem, and RayScope's visualization of the corrected implementation.

PBRVulkan's final rendered image, top left image of Figure 4.10, in AO mode shows visual artifacts. The bottom left image of Figure 4.10 shows the visualization generated by entering the coordinate of a problematic pixel using RayScope's pick feature. The resultant visualization shows that miss-only rays receive a direction pointing into the geometry rather than pointing away from it. The type of the ray (miss-only) narrows down the traceRay call sites that could be responsible to a single callsite. The direction for the miss-only ray direction is not calculated using the normal. Instead, the direction is calculated using the face-forward normal—if the dot product of the normal and the incident ray direction is less than 0, then then normal is inverted. Using the face-forward normal results in some miss-only rays using the negative surface normal. These rays trivially collide with the surface they are generated from. Therefore, the rays receive a 100% occlusion rate and causes a very dark artifact, circled in red in the top left image of Figure 4.10.

Switching from the face-forward normal to the surface normal fixes the ray direction. The image in the top right of Figure 4.10 shows the final rendered image after this change. The image in the bottom right of Figure 4.10 shows the impact of this fix on the visualization.

4.3.3 Opaque-Flag Recommendation

Vulkan is a verbose API that provides the driver with information to optimize the application [19]. For example, specifying that a buffer must be *host coherent* introduces additional overhead for communication between the host and the device and thus this specification must be used carefully. Such flags have non-trivial performance implications. However, there are a large number of such flags and it is difficult for a developer to correctly set all of them. RayScope automatically detects when the opaqueness flag is missing from geometry and reports it in the visualization.

The opaqueness flag dictates how the Vulkan driver interacts with instances

of a BLAS. This feature allows different instances of the same geometry to be transparent and opaque. The any-hit shader does not need to be executed against opaque geometry instances. Ray-tracing best practices recommend marking as many geometries opaque as possible [54].

Figure 4.1 shows the image produced by the application VkRaytrace [11] and RayScope's visualizations of the any-hit shader point cloud. The picture in the top right of Figure 4.1 shows the any-hit point cloud. Points being present for the robot indicates that the robot is being tested for transparency. The backdrop that can be observed in Figure 4.6 is omitted from Figure 4.1 for clarity.

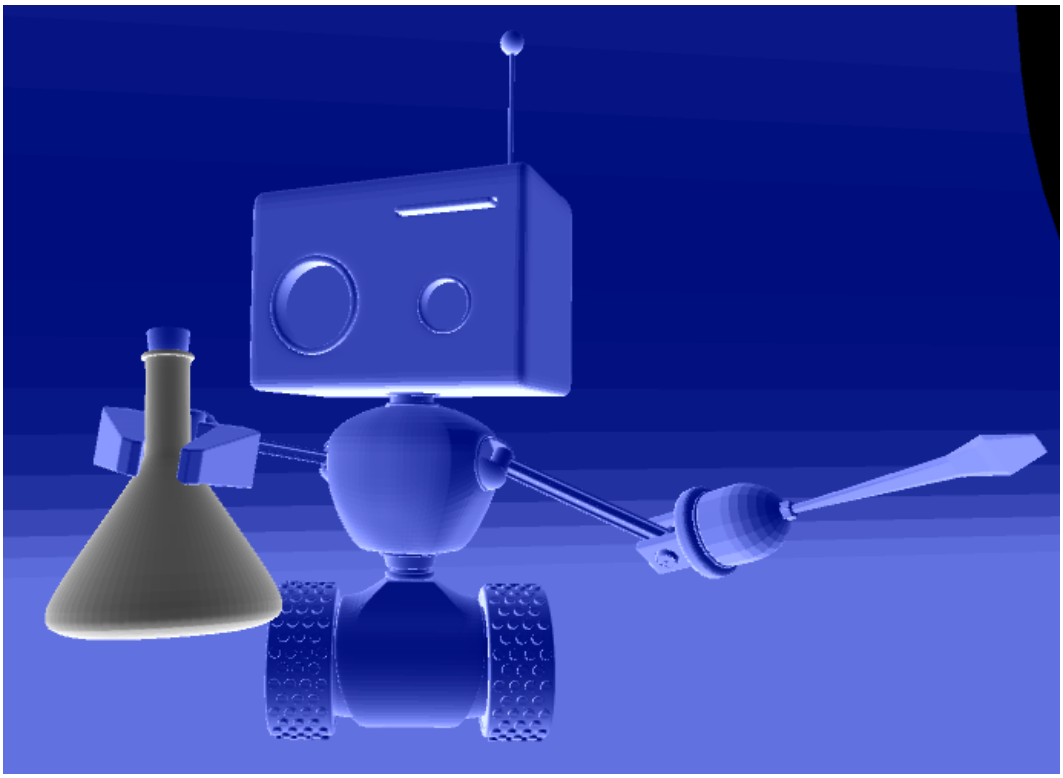

Figure 4.11: RayScope's visualizer showing the BLAS instances to investigate. All geometry drawn in blue did not leverage the transparency feature despite executing the any-hit. The gray geometry does require the any-hit.

Figure 4.11 shows RayScope's recommendation for the geometry instances that should be marked as opaque. Geometry instance are assigned the colour blue to indicate that they should be investigated by the developer. The transparent vial is gray because RayScope recognizes it is properly utilizing

transparency. All other instances are correctly detected as candidates for marking as opaque. The picture in the bottom right of Figure 4.1 shows the effect of implementing RayScope's recommendation on the any-hit point cloud.

```
set<int> any_hit_ids = {}
set<int> ignored_ids = {}

void OnEvent(EventType type, Vec3 pos, int instance_id){
  if(type == AnyHit){
    any_hit_ids.insert(instance_id);
  } else if (type == IgnoreIntersection){
    ignored_ids.insert(instance_id);
  }
}

void OnDone(){
  set<int> ids_to_investigate = set_difference(any_hit_ids,
      ignored_ids);
}
```

Figure 4.12: Pseudocode for RayScope's algorithm that recommends instance ids that should be marked as opaque.

Figure 4.12 shows RayScope's algorithm for deciding the set of instance ids that should be investigated. RayScope maintains two sets. The any-hit shader executes on the instances in the set *any_hit_ids* while the `ignoreIntersection` instruction executes on the instances in the set *ignored_ids*. The sets are updated whenever an **Any Hit** or **IgnoreIntersection** event is processed. After all events are processed, RayScope outputs the set difference of $any\_hit\_ids - ignored\_ids$. An alternative, and simpler, strategy would be to output the complement of the set of ignored ids. However, such a set would erroneously identify instances that are transparent but had no execution information.

Vulkan Vision's shader trace utility can measure the number of shader executions before and after the change [49]. The number of any-hit shaders was reduced by 96.8%.

An evaluation of the end-to-end performance impact of this change starts with annotating the application to record frame latencies before and after the

change. Given that VkRaytrace does not begin ray-tracing until the model is loaded, the evaluation starts after the model is loaded. The evaluation is based on a sample of 100 frames for the 1660ti and 1000 frames for the 3080. The baseline measurement is collected before the optimized measurement to ensure that any throttling that may occur does not inflate the speedup. The measurement was repeated 5 times per GPU revealing an FPS increase of 3.15% on an RTX 3080 and an increase of 15.71% on GTX 1660Ti with negligible variance between measurement repetitions.

Correctly marking geometry as opaque has a more significant impact on the 1660Ti because it does not have hardware-accelerated ray-box and ray-triangle intersection tests. Therefore, the 1660ti must execute a software implementation of the BVH traversal algorithm. An examination of the output of Vulkan Vision's execution-trace utility reveals that the software implementation eagerly evaluates shaders during the BVH traversal. In contrast, the hardware implementation batches the shader executions after the BVH traversal. Therefore, avoiding any-hit shader executions is more beneficial to the software implementation because it reduces register pressure of the software BVH traversal and because the eager any-hit execution has lower SIMT efficiency.

4.3.4 Leaky Geometry

Dynamic ray behaviour depends greatly on the models present in the scene. When faces of geometry are not contiguous, rays can escape through the resultant holes. Holes in geometry can degrade visual correctness in production games [9]. Holes can result in wasteful rays that never collide or receive light information. Some holes may be too small to see by inspecting the geometry on its own and therefore it is important to view the geometry in the context of the ray paths. Approaches exist that attempt to statically prove that geometry has no leaks, but this is not applicable to modern real-time ray tracing applications that generate dynamic geometry, such as BattleField V [22]. RayScope helps to identify leaky geometry in two ways. The first way is the point-cloud visualization that shows events in places that should be impossible to reach,

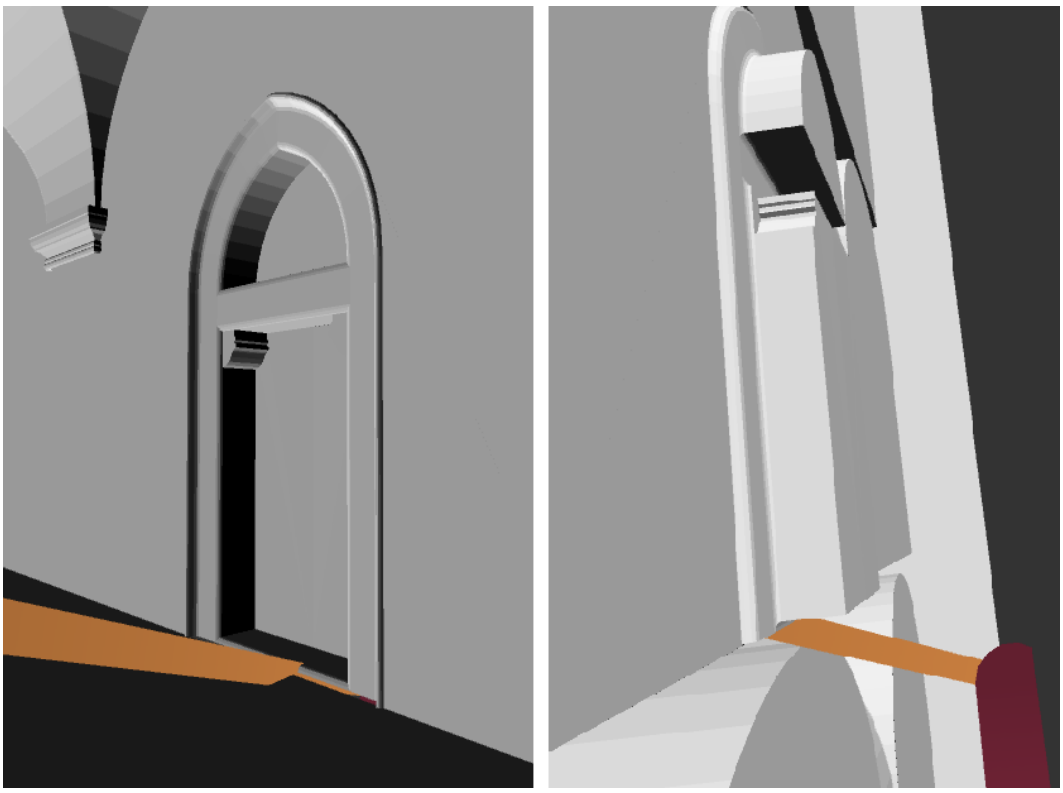

Figure 4.13: **Left:** RayScope visualizing a ray leaking through small gap below a door on the second floor of Sponza in ChameleonRT. **Right:** The same ray visualized from behind the door. The ray hits the inside of the Sponza model and produces another secondary ray that cannot reach a light source.

such as inside a sealed box. The second way is the all-ray visualization that shows the ray paths that escape through imperceptible holes.

The picture at left in Figure 4.13 shows a ray leaking through a hole in the second floor of the Sponza model [35]. The ray proceeds to bounce three more times in an area that it should never reach. In ChameleonRT, each bounce also generates a shadow ray, omitted from Figure 4.13 for clarity. This needless computation also affects the visual correctness of the final image.

The Cornell Box in PBRVulkan is defined as a set of faces that are aligned using a floating-point offset. RayScope's visualization in Figure 4.14 shows a ray that escapes through an imperceptible gap between the faces of the box.

Figure 4.13 and Figure 4.14 highlight the problem of rays escaping through holes. Both PBRVulkan and ChameleonRT had many more rays that leaked through gaps. RayScope's visualization of all rays highlights rays that escape.

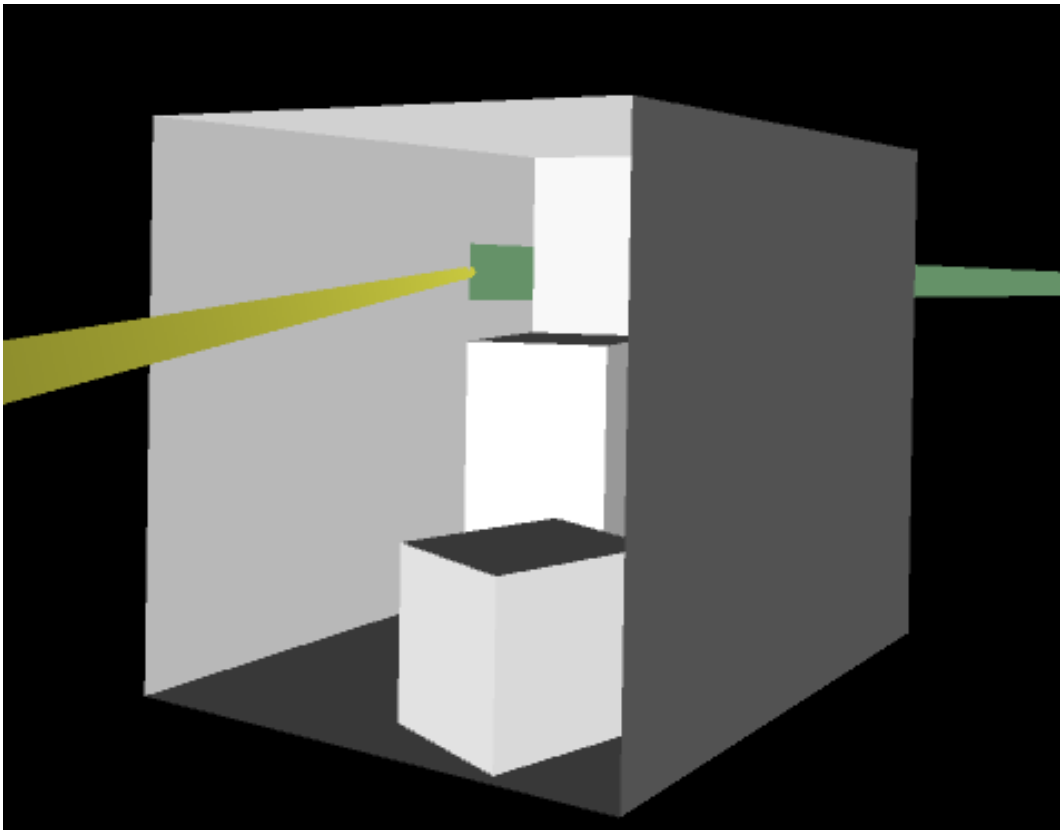

Figure 4.14: A ray leaking through imperceptible gap in PBRVulkan's Cornell Box due to improper geometry construction. The yellow ray is the primary ray generated from the camera. The green ray escapes between the faces of the box.

The bisect feature can then be used to determine the problematic section of geometry that let a ray through. This section of geometry may be fixed, and the process repeated to construct ray-tracing-friendly geometry.

4.3.5 Properly Setting Tmin

All Vulkan ray-tracing developers must provide a *Tmin* value to the traceRay operation. *Tmin* specifies the minimum distance from the ray origin to valid intersections. Setting this value is complicated by geometries being arbitrarily scaled. Little guidance is given on how to choose *Tmin* and often it is set to an arbitrary power of ten. RayScope aids in detecting instances where *Tmin* was set improperly and provides a constructive mechanism for choosing *Tmin*. RayScope's visualizer provides the minimum distance of all rays traced at runtime. The *Tmin* value can be based on this analysis instead of an arbitrary value.

Setting *Tmin* to be too high, results in rays leaking through objects and producing incorrect results. Figure 4.15 shows rays escaping the Cornell Box in RayTracingInVulkan. Unlike PBRVulkan, RayTracingInVulkan specified Cornell Box to share the vertices at the corners. Therefore, rays are not escaping through minute gaps in the geometry because there are no gaps. In this case, rays escape because *Tmin* is set too high and omits the ray intersection. It is unlikely that the developer is aware of this problem because fixing the problem only requires changing one value in the Ray Generation shader.

Another problem comes from setting the value of *Tmin* to be too low. In this case, rays may have intersections that should have been omitted. Figure 4.16 shows a ray escaping through the floor of Sponza due to the value of *Tmin* being too low. The orange ray collides with the floor of the model. Then a new ray is generated that has a direction that is nearly parallel with the floor. The new ray has an intersection with a distance value of 0.00033. The next bounce is generated based on the direction of the previous ray. The direction of the previous ray was almost parallel with the floor causing the next ray to generate on the other side of the floor. The developer likely did not consider the possibility of rays intersecting the floor at a nearly parallel direction resulting

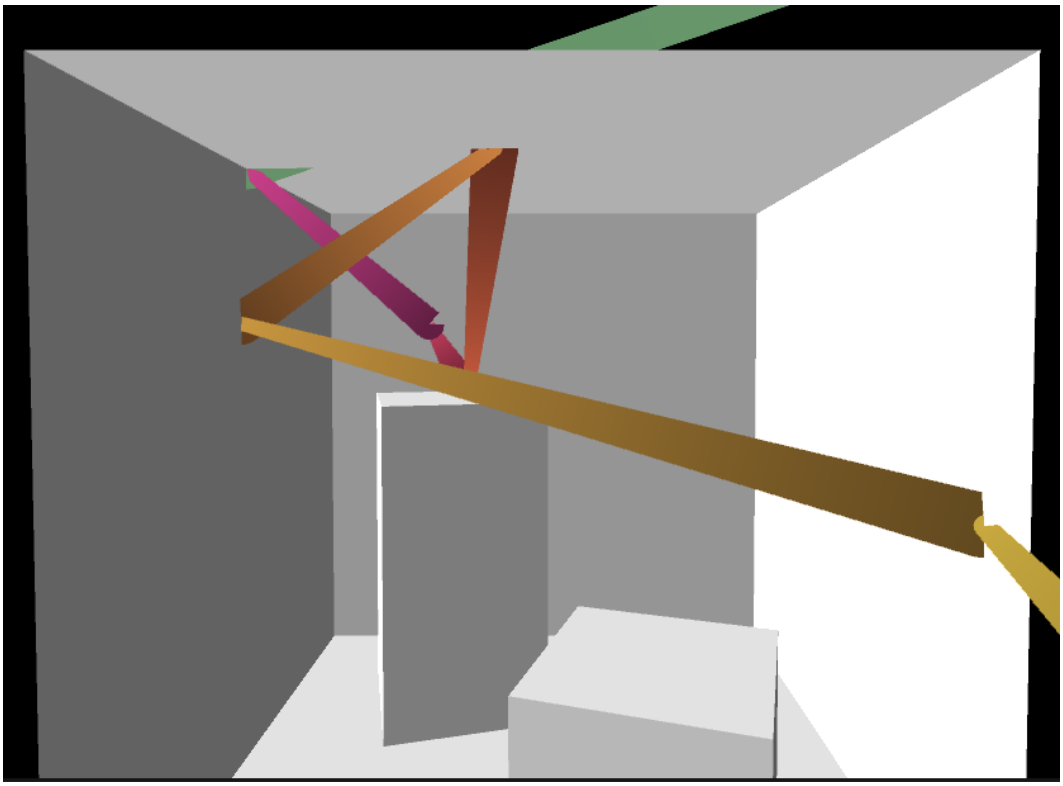

Figure 4.15: A ray leaking through corner of Cornell Box in RayTracingInVulkan due to a too-high value of Tmin. The ray bounces 5 times before striking the precise point where 2 faces of the Cornell Box meet. The green ray is generated inside the box but does not collide with the top of the box due to a too-high value of Tmin.

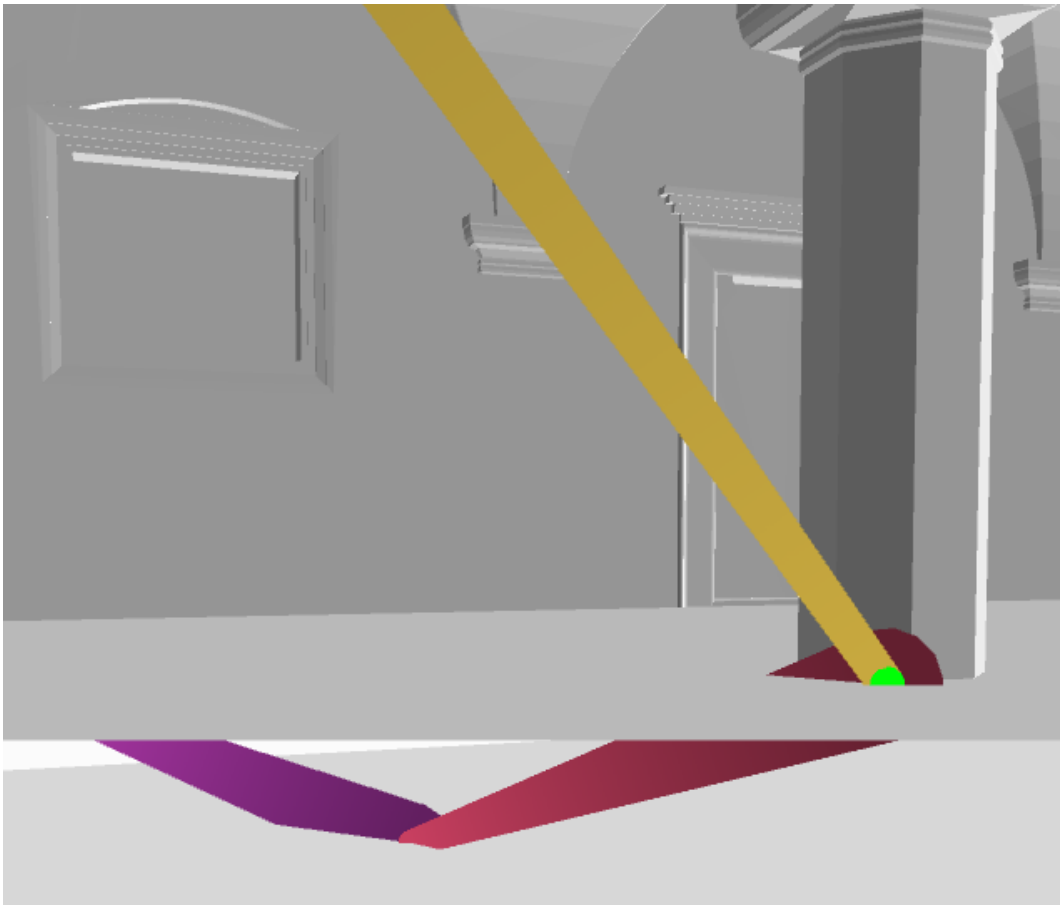

Figure 4.16: A ray leaking through the floor of Sponza in ChameleonRT due to a too-low value of Tmin. The yellow ray hits the floor of Sponza. A ray with length 0.00033 is generated (shown as a green circle) and collides with the floor immediately. The purple ray is generated through the floor and continues to bounce around the inside of the Sponza model.

in the incorrect direction of the red ray.

RayScope's visualization can be used to determine an appropriate value for *Tmin* for different combinations of scenes and applications. Given this feedback, it is easy to refine the value of *Tmin*.

4.3.6 Poor Ray-Geometry Interactions

Most Divergent: Ray-tracing is known to be troublesome for GPUs due to incoherent bounces [27]. This information is difficult to act on because ray bounces are dependent on the ray-tracing algorithm and on the geometry. Additionally, the thread-to-subgroup assignment has a large impact on spatial locality and ray coherence. RayScope's instrumentation captures the warp and thread responsible for each ray-event. This information is encoded into the ray file so that the visualizer can account for the hardware-level execution when making performance recommendations. A user can quickly cycle between subgroup with poor hardware utilization using RayScope's visualizer's UI.

RayScope records the subgroups with the worst divergence—measured as the subgroup with the greatest total of threads that are inactive when a ray is traced. RayScope's visualizer offers the option to view the most divergent subgroups.

Figure 4.17 shows a visualization of the subgroup with the lowest average of active threads in RayTracingInVulkan [14]. Primary rays enter from the left of the image and all rays but one miss geometry. The final ray bounces fifteen times before finally missing and concluding the subgroup execution. The SIMT efficiency of traceRay calls would be 9% for the visualized subgroup.

Based on this recommendation developers may choose to switch from a monolithic ray-tracing implementation to a wavefront implementation [27]. Based on RayScope's visualizations, a limit could be placed on the number of consecutive bounces that can occur due to internal reflection.

Trapped Rays: Rays bouncing many times in small areas are difficult to detect. These rays are unlikely to improve the image on consecutive bounces because the light information remains constant in that small area. More generally, choosing the maximum bounce limit is a tradeoff between visual

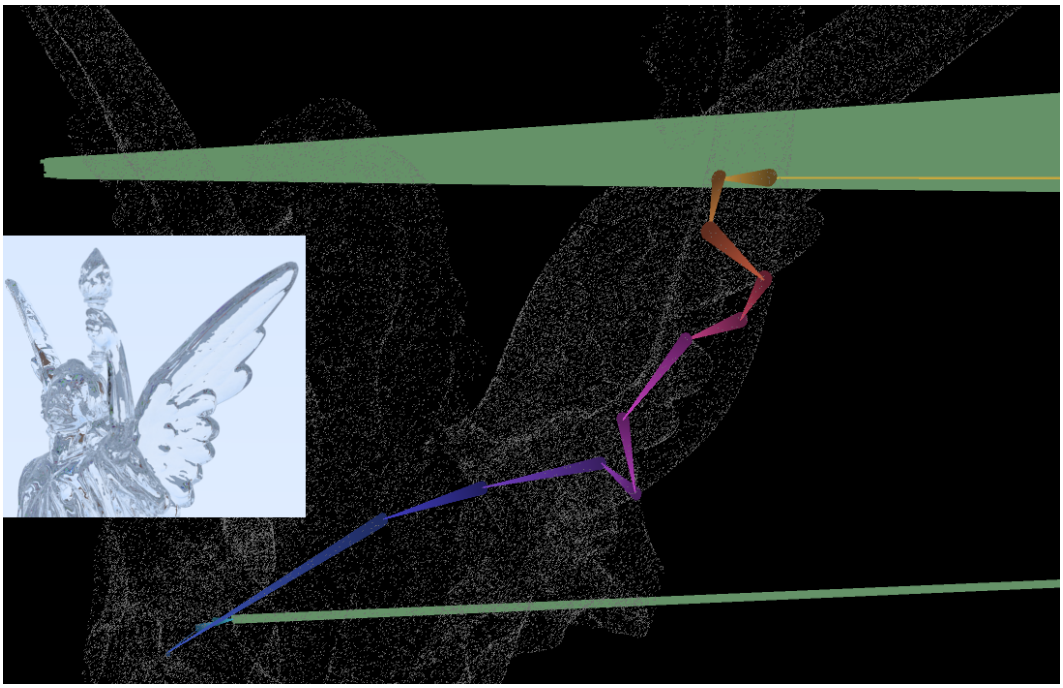

Figure 4.17: RayScope's visualization of rays that most poorly utilize SIMD hardware for rendering of transparent Lucy model (screenshot of final image overlaid left). The Lucy model is shown as a point cloud of closest-hit events. 32 primary rays enter from the top-right of the image. 31 of the primary rays miss geometry and are coloured green. The threads tracing these rays have no more work. 1 primary ray hits Lucy's arm and 15 more rays are traced down the arm and through the torso for internal reflection. Each of those 15 bounces would have 1 thread active for each traceRay operation. Disproportionate work assignment between threads is the worst-case for GPU-based ray tracing.

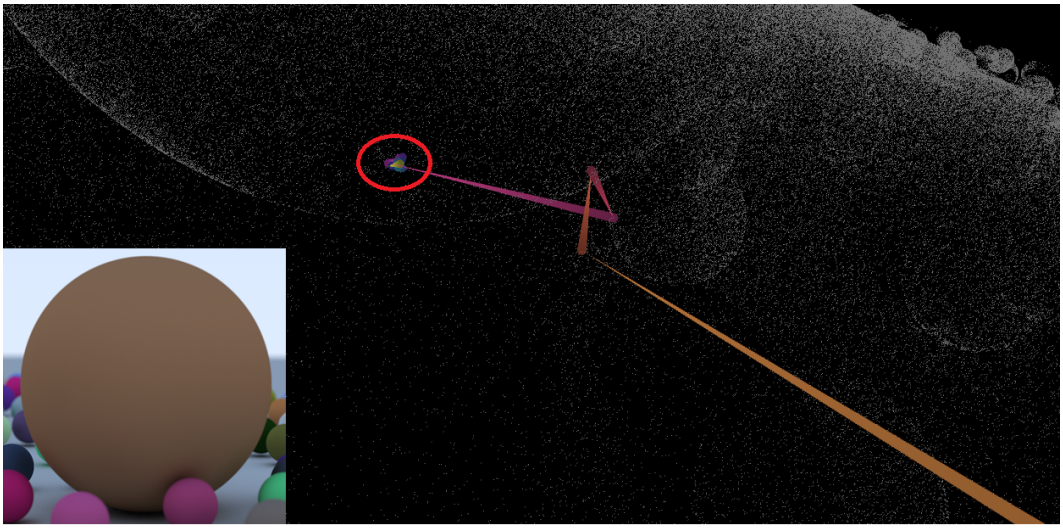

Figure 4.18: RayScope's recommended and visualized high-work ray-path. The scene geometry, composed of spheres, is shown as a point cloud of intersection events. The final rendered image is overlaid in the bottom left for clarity. The yellow ray is the primary ray generated from the camera. The ray bounces 3 more times before becoming trapped underneath the sphere. The red circled region contains 12 ray-bounce operations as the ray becomes trapped.

quality and computation cost. RayScope's visualizer allows a developer to cycle through the threads that traced the most rays at runtime. Developers may choose to close off the areas that trap rays or limit the maximum bounce count to reduce these cases.

Figure 4.18 shows an example of RayScope's recommended and visualized high-work ray path from RayTracingInVulkan. The application RayTracingInVulkan does not accumulate any colour unless a ray eventually hits a light source. Despite performing 16 bounces, the ray in Figure 4.18 receives no color. 12 of the bounces are in a tight area between the sphere and the floor. This is one of many examples in the application where a ray becomes trapped. These poor dynamic ray-geometry interactions induce greater computation for no discernible increase in visual quality.

To remedy this problem, the floor could be raised, or the spheres lowered to reduce the occurrence of trapped rays. Additionally, shadow effects could be implemented using occlusion rays to reduce the number of rays required.

4.4 Conclusions and Future Work

RayScope's automatic instrumentation gathers application-agnostic ray-tracing information available in Vulkan ray-tracing applications. RayScope's visualizer reconstructs the ray paths and geometry and allows the user to interact with them. Additionally, RayScope's visualizer creates point clouds to represent implicit, and otherwise invisible, ray-tracing events. These visualizations aid in understanding, profiling, debugging, and designing Vulkan ray-tracing applications. The outputs of RayScope's instrumentation are readily understandable and should allow other visualization tools to easily integrate the instrumentation data. Tools such as NSight Graphics and Pix can readily benefit because they already visualize the acceleration structure. Additionally, integration with game engines would allow game developers to observe how the engine-provided ray-tracing implementation interacts with their geometry.

RayScope's shader instrumentation can effectively detect that objects are incorrectly marked as transparent. RayScope shall also be able to recommend using a cull mask if hidden geometry is implemented as an any-hit shader that always ignored collisions. RayScope's visualizer also has the potential to recommend the shortest rays to the user by detecting when rays perform trivial bounces.

RayScope's point-cloud feature allows the visualization of procedural geometry. The point-cloud data could be converted to a triangular mesh to embed within the scene and thus allow it to be treated similarly to a triangular BLAS instance. For example, the developer could experiment with moving geometry around in the scene using the Unity editor view. A natural extension of converting point clouds to geometry is to create optimized representations of triangular meshes. For example, certain parts of models may be unreachable by rays and therefore should not be represented as detailed triangles.

RayScope is a highly capable addition to a Vulkan developer's toolkit. RayScope's unprecedented view into the runtime execution of the monolithic ray-tracing pipeline enables deeper understanding leading to patched bugs, more efficient execution, and fixes to leaky geometry. RayScope demonstrates

the power of leveraging information already available in the Vulkan specification. Analogous tools for compute and rasterization workloads could be developed using the V-Vision framework.

Chapter 5

Related Work

The first tool presented in this thesis, V-Vision, is a SPIR-V instrumentation framework with applications across the GPU software-hardware stack. The second tool presented, RayScope, extends V-Vision to include automatic graphics API call instrumentation to visualize the Vulkan ray-tracing process. Both tools are instrumentation-based, aid in debugging and profiling tasks, and generate visualizations. This chapter is organized as follows. Section 5.1 discusses proprietary instrumentation tools, open-source instrumentation tools, and manual instrumentation. Section 5.2 covers event-based debuggers and profilers, runtime debuggers, and static debugging and profiling. Section 5.3 explores visualization tools that aid in graphics application development.

5.1 Instrumentation Tools

Proprietary: NSight Graphics collects samples of the graphics pipeline binary being executed and accumulates hardware performance counters [23]. Similar to NSight Graphics, Intel GPA uses vendor-specific hardware performance counters to characterize application performance on Intel Graphics Hardware [17]. Microsoft Pix is a graphics profiling tool that achieves cross-vendor support by leveraging the debugging capabilities of a single graphics API [37]. In comparison to these tools, V-Vision provides fine-grained data from the code executing on the GPU with a commensurately higher overhead. Thus, V-Vision may complement the proprietary tools in situations where performance bottlenecks occur in the graphics pipeline itself. V-Vision sets itself apart as an open-source

framework for developing studies, such as Thread Compaction, which require precise execution data.

NVBit [59] uses library injection to create wrapper functions for CUDA driver calls to perform on-the-fly instrumentation. NVBit instruments SASS allowing it perform analyses, such as register allocation, that are impossible for V-Vision at the SPIR-V level. V-Vision provides instruction primitives to overcome graphics instrumentation challenges, similar to NVBit's abstraction of SASS instructions that overcome binary instrumentation challenges.

Zeroploit implements value-specialization for DirectX using IR instrumentation [50]. Zeroploit observes, with manual instrumentation, that many common shader operations produce a value of 0. V-Vision's instruction primitives can collect frequencies of a given value to automatically detect such opportunities in existing games using Vulkan.

Open-source: Strengert *et al.* developed a debugging tool for OpenGL applications that instruments shaders based on user guidance [55]. Their tool uses library injection which allows for modification of a shader before it is executed with a custom source-to-source transpilation. Their approach is geared towards providing values from variables indicated by the user. This limits the tool, in its current form, from capturing general information such as the SIMT efficiency. V-Vision leverages existing compiler infrastructure for SPIR-V instrumentation, so implementing auto-instrumentation is more productive than in a custom transpiler. V-Vision is effective for aiding in debugging by generating execution traces for user-specified warps or threads.

Manual: Runtime feedback includes the developer manually placing `printf` calls within a shader to test if that call is reached and recording a set of runtime values on invocation [25], [30]. Debugging rendering pipelines using `printf` is challenging due to the number of threads. If all threads call `printf` then a prohibitive number of messages are created. If few threads call `printf` then issues that occur only in certain threads may be missed. Since `printf` is often insufficient, developers create their own solutions to inspect specific values. For example, developers may choose to include an extra texture in debug builds of their application. The developer may then manually

instrument certain values by writing them into the texture and inspecting them after execution concludes. Inspecting the output values is time consuming, and this approach increases the amount of code to maintain. A flaw with both approaches is that developers must interpret the data, such as position values, as text instead of viewing them in the context of the scene. RayScope reconstructs the ray-tracing scene through visualization greatly simplifying the task of interpreting the data.

5.2 Debugging and Profiling Tools

Event Based: Many hardware vendors provide proprietary tools for logging host-side rendering events and for profiling their execution [3], [20], [21], [37], [42], [43]. Such tools can be used to measure the timing and frequency of draw calls and other graphics API events. Developers use these tools to identify shaders that take longer than expected to complete and rendering algorithms that generate an unexpected number of graphics API events. Many GPUs provide hardware counters that are monitored by these tools and report wall-clock execution time and various other runtime statistics, such as the number of memory accesses or the number of expensive instruction invocations made during each draw call. To debug pixel color issues, NSight Graphics also includes a method for collating all the draw calls that contributed to a given pixel's color. Several tools for event profiling of GPU compute workloads, such as those programmed with CUDA, have similar capabilities [44]. V-Vision and RayScope allow the sophisticated behaviour of graphics pipeline executions to be understood and both complement coarse-grained event-based tools that identify problematic pipeline executions.

Runtime: Runtime debuggers provide the ability to break execution at a given line of code to examine the values stored in register and memory. CPU/host debuggers, such as GDB, are common and widely used. Debugging program execution in a GPU to examine performance issues at a pixel level is considerably more challenging. This is partly because the GPU executes millions of threads across thousands of cores often arranged into SIMT units

(warps) and has a distinct memory and register file from the host CPU. When debugging execution on the GPU is not possible, device emulation is an option. Alternatively, instrumentation can record an execution trace to be reviewed offline. Warp occupancy and instruction latency can then be reported with the execution trace to give the developer insights into the shader's performance [2]. On NVIDIA GPUs, a runtime debugger for CUDA allows developers to step through execution of an entire SIMT group (warp), to inspect values stored both in the host and in the GPU memory as well as in PTX or SASS registers [39], [40], and to identify memory access violations [41]. V-Vision is implemented in a validation layer that operates above the Vulkan driver and therefore cannot leverage hardware-specific features unless they are offered as a Vulkan extension. This limits V-Vision's ability to inspect hardware register and memory values, but if a value exists at the SPIR-V level, V-Vision can be used to inspect it. As demonstrated with RayScope, there is significant information known at the SPIR-V level and much value in presenting it.

Static: Offline static analysis includes compiler-generated errors, warnings and recommendations. Offline shader compilation is carried out without full knowledge of the rendering pipeline. Static errors and warnings check for conformance to the language specification and common programming mistakes. Other features of static analysis tools include cycle-count estimates, disassembly, and examining the effects of compiler passes [1], [4]. These tools allow developers to optimize their shaders for a specific GPU architecture. Static performance estimates are less applicable to workloads with highly variable work distribution such as ray tracing. Performance estimates of programmable shader stages do not take into account the fixed-function logic of a rendering pipeline. V-Vision and RayScope are orthogonal and complementary to static analysis.

5.3 Visualization Tools

For offline visualization, Duca *et al* developed a tool that performs shader and OpenGL API call instrumentation to develop a database of rasterization information [12]. This database can be searched using a query language and

generates results that are visualized in a GUI. RayScope presents automatic recommendations and visualizations of the ray-tracing data without requiring the user to learn a query language.

Several tools have been created to visualize ray tracing [15], [28], [53]. rtVTK employs a plug-in model that requires developers to manually instrument their renderer code. The visualization of ray-tracing behaviour produced by this tool was used to find longstanding issues. Lesev *et al.* [28] focus on bulk data collection of ray information that is streamed to a user. They propose shipping the instrumented ray-tracing application to end users. They also allow arbitrary data to be associated with rays, requiring the developer to also create an analysis to understand the final data stream. Similar to rtVTK, this approach requires manual instrumentation, adjusting application code, and is limited to CPU ray tracers. Simons *et al.* [53] allow developers to optimize scene parameters to converge faster in physically-based offline rendering. Features, including light intensity and paths, can be better understood using their visualization. Their tool can help developers evaluate the impact of geometry positions. For instance, they use manual instrumentation to analyze radiance while optimizing positions. Their results indicate that offline optimization of geometry positions can have a significant impact. RayScope is more broadly applicable than these works because it performs automatic instrumentation of a major graphics API. RayScope also requires no application expertise to insert instrumentation because its automatic instrumentation is based on the Vulkan specification. However, RayScope is currently limited by the information available in Vulkan. In the future, RayScope and V-Vision could be extended to allow manual annotation of the shader source with values such as light-source position and types.

NSight Graphics [43] and Pix [37] provide a tool that uses rasterization to visualize the acceleration structure allowing developers to identify redundant BLAS nodes. As demonstrated, RayScope improves the state-of-the-art AS visualization by showing procedural objects using a point cloud of their intersection shader executions.

Chapter 6

Conclusion

This thesis contributes a significant improvement to the developer tools available in the Vulkan ecosystem. Prior to this contribution, basic questions such as *how efficiently is the hardware being utilized?* or *is my ray-tracing implementation correct?* were time-consuming to answer and required invasive application changes. The answers to such questions are necessary for informed design decisions. In answering such questions, this work identified systematic issues across Vulkan ray-tracing implementations causing wasteful computation and affecting visual correctness.

Chapter 3 presented V-Vision. V-Vision is a framework for generating, analyzing, and presenting execution data to developers. The chapter explored challenges in understanding runtime behaviour of the sophisticated Vulkan ray-tracing pipeline. The instrumentation primitives available in V-Vision allow warp execution to be tracked through the fixed-function logic inserted by downstream compilers. This unprecedented view into the runtime execution yields many interesting insights, such as observing the differences between the black-box software and hardware implementations of *traceRayEXT* in the NVIDIA ecosystem. The chapter then explored challenges in using the existing but inefficient instrumentation infrastructure. To remedy this, V-Vision introduces the concept of instrumentation callsite ids and a method for transferring statically known data at compile-time. Finally, the chapter presented, through case studies, the potential for V-Vision to be applied for application, driver, compiler, and hardware studies.

Chapter 4 presented RayScope, a tool that exploits V-Vision's potential for application-level studies. RayScope automatically captures application-agnostic geometry and ray-tracing execution traces from Vulkan applications. The geometry and ray-tracing data is stored in an accessible format to encourage future research and visualizations. RayScope offers an interactive visualizer to present and analyze the data captured from the Vulkan application. RayScope's visualizer provides novel visualization methods of Vulkan ray-tracing data. These visualizations allow a developer to see implicit or invisible aspects of their application's execution such as shader executions. In case studies using RayScope, we identify systemic issues that afflict Vulkan ray-tracing applications. Many of the recommendations and bug fixes motivated by RayScope are trivial changes. This reinforces that Vulkan developers lacked proper support for creating Vulkan ray-tracing applications.

There are several dimensions that this work can be extended in. Through interoperability efforts, V-Vision and RayScope could be applied to Direct3D. The shadow allocation methodology used in RayScope could be extended more parts of Vulkan to provide more automatic recommendations. Other debugging and visualization tools could be developed using V-Vision for rasterization, mesh, and compute pipelines. In sum, the work presented in this thesis lays the groundwork for gaining a deeper understanding of the dynamic behaviour of graphics applications. The work further demonstrates the insights that can be gained by extending the groundwork into a highly specialized study of ray tracing.

References

[1] AMD. (2012). GPU ShaderAnalyzer (ARCHIVED CONTENT). Accessed: 2021-03-11, [Online]. Available: https://gpuopen.com/archived/gpu-shaderanalyzer/. 82

[2] ——, (2021). AMD Radeon GPU Profiler: Analyze. Adjust. Accelerate. Accessed: 2021-03-11, [Online]. Available: https://gpuopen.com/rgp/. 82

[3] Apple. (2021). Metal: Render advanced 3D graphics and perform data-parallel computations using graphics processors. Accessed: 2021-03-11, [Online]. Available: https://developer.apple.com/documentation/metal/. 10, 81

[4] ARM. (2021). Mali Offline Compiler: Purpose built for Mali. Accessed: 2021-03-11, [Online]. Available: https://developer.arm.com/tools-and-software/graphics-and-gaming/arm-mobile-studio/components/mali-offline-compiler. 82

[5] B. Bargen and P. Donnelly, *Inside DirectX, Microsoft Programming Series*. Microsoft Press, Redmond, Washington, 1998. 1, 7

[6] C. Bouville and K. Bouatouch, "Developments in ray-tracing," in *Advances in Computer Graphics IV*, W. T. Hewitt, M. Grave, and M. Roch, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 154–212. 7

[7] J. Burgess, "RTX on the NVIDIA Turing GPU," *IEEE Micro*, vol. 40, no. 2, pp. 36–44, 2020. 37

[8] A. Burnes. (2020). NVIDIA DLSS 2.0: A Big Leap In AI Rendering. Accessed: 2021-03-11, [Online]. Available: https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/. 7

[9] J. Choi, J. Kjellin, P. Willbo, and D. Zhdan. (2020). Ray Traced Reflections in 'Wolfenstein: Youngblood' (Presented by NVIDIA). Accessed: 2021-03-11, [Online]. Available: https://www.gdcvault.com/play/1026723/Ray-Traced-Reflections-in-Wolfenstein. 5, 44, 61, 68

[10] S. Damani, D. R. Johnson, M. Stephenson, S. W. Keckler, E. Yan, M. McKeown, and O. Giroux, "Speculative Reconvergence for Improved SIMT Efficiency," in *Intern. Symp. on Code Generation and Optimization (CGO)*, San Diego, CA, USA, 2020, pp. 121–132. 31, 36, 37

[11] N. DesignWorks. (2020). VK_RAYTRACE. Accessed: 2021-03-11, [Online]. Available: https://github.com/nvpro-samples/vk_raytrace. 27, 54, 61, 66

[12] N. Duca, K. Niski, J. Bilodeau, M. Bolitho, Y. Chen, and J. Cohen, "A Relational Debugging Engine for the Graphics Pipeline," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 453–463, Jul. 2005. 82

[13] W. W. L. Fung and T. M. Aamodt, "Thread Block Compaction for Efficient SIMT Control Flow," in *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*, ser. HPCA '11, USA: IEEE Computer Society, 2011, pp. 25–36. 19, 22, 28

[14] GPSnoopy. (2021). Ray Tracing In Vulkan. Accessed: 2021-03-11, [Online]. Available: https://github.com/GPSnoopy/RayTracingInVulkan. 27, 61, 74

[15] C. Gribble, J. Fisher, D. Eby, E. Quigley, and G. Ludwig, "Ray Tracing Visualization Toolkit," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '12, Costa Mesa, California: Association for Computing Machinery, 2012, pp. 71–78. 2, 8, 42, 54, 62, 83

[16] G. Guennebaud, L. Barthe, and M. Paulin, "Real-time Soft Shadow Mapping by Backprojection," in *Eurographics conference on Rendering Techniques*, Nicosia, Cyprus: Eurographics Association, 2006, pp. 227–234. 54

[17] S. Guo, P. Gerasimov, and B. Aona, "Practical game performance analysis using Intel graphics performance analyzers," *Intel Corporation White Paper*, 2011. 18, 79

[18] T. K. G. Inc. (Feb. 2020). Vulkan SDK, Tools and Drivers are Ray Tracing Ready. Accessed: 2021-03-11, [Online]. Available: https://www.khronos.org/news/press/vulkan-sdk-tools-and-drivers-are-ray-tracing-ready. 42

[19] ——, (2021). Vulkan Overview. Accessed: 2021-03-11, [Online]. Available: https://www.khronos.org/vulkan/. 7, 42, 44, 65

[20] Intel. (2020). Intel Graphics Performance Analyzers. Accessed: 2021-03-11, [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/graphics-performance-analyzers.html. 44, 81

[21] B. Karlsson. (2018). RenderDoc. Accessed: 2021-03-11, [Online]. Available: https://renderdoc.org/. 44, 81

[22] A. Keller, T. Viitanen, C. Barré-Brisebois, C. Schied, and M. McGuire, "Are we done with ray tracing?" In *ACM SIGGRAPH 2019 Courses*, ser. Proc. Int'l Conf. on Computer Graphics and Interactive Techniques (SIGGRAPH), 2019. 68

[23] R. Kerschner and J. Klei, *Speed of Light DXR Ray Tracing with NVIDIA Nsight Graphics (Presented by NVIDIA)*, https://www.gdcvault.com/play/1026187/Speed-of-Light-DXR-Ray, Accessed: 2021-03-11. 79

[24] J. Kessenich, B. Ouriel, and R. Krisch, "SPIR-V Specification," *Khronos Group*, vol. 3, 2018. 17

[25] Khronos. (2011). printf Function. Accessed: 2021-03-11, [Online]. Available: https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/printfFunction.html. 80

[26] ——, *Architecture of the Vulkan Loader Interfaces*, https://github.com/KhronosGroup/Vulkan-Loader/blob/master/loader/LoaderAndLayerInterface.md, Accessed: 2021-03-11. 19

[27] S. Laine, T. Karras, and T. Aila, "Megakernels Considered Harmful: Wavefront Path Tracing on GPUs," in *Proceedings of the 5th High-Performance Graphics Conference*, ser. HPG '13, Anaheim, California: Association for Computing Machinery, 2013, pp. 137–143. 60, 74

[28] H. Lesev and A. Penev, "A framework for visual dynamic analysis of ray tracing algorithms," *Cybernetics and Information Technologies*, vol. 14, no. 2, pp. 38–49, 2014. 2, 8, 83

[29] Library of Congress. (2020). Wavefront OBJ File Format. Accessed: 2021-03-11, [Online]. Available: https://www.loc.gov/preservation/digital/formats/fdd/fdd000507.shtml. 47, 55

[30] LunarG. (2021). Debug Printf. Accessed: 2021-03-11, [Online]. Available: https://vulkan.lunarg.com/doc/sdk/1.2.162.1/linux/debug_printf.html. 8, 80

[31] ——, (2021). Getting Started with the Windows Vulkan SDK. Accessed: 2021-03-11, [Online]. Available: https://vulkan.lunarg.com/doc/sdk/1.2.162.1/windows/getting_started.html. 10

[32] ——, *GFXReconstruct*, https://github.com/LunarG/gfxreconstruct/blob/dev/README.md, Accessed: 2021-03-11. 19

[33] ——, *GPU-Assisted Validation*, https://vulkan.lunarg.com/doc/view/1.1.114.0/windows/gpu_validation.html, Accessed: 2021-03-11. 20

[34] J. MacArthur and M. Stich, *Profiling DXR Shaders with Timer Instrumentation*, Accessed: 2021-03-11. 8, 39

[35] M. McGuire, *Computer Graphics Archive*, Accessed: 2021-03-11, 2017. [Online]. Available: https://casual-effects.com/data. 27, 61, 69

[36] Microsoft. (2021). Direct3D 12 Graphics. Accessed: 2021-03-11, [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics. 10, 42, 57

[37] ——, (2021). Pix on Windows. Accessed: 2021-03-11, [Online]. Available: https://devblogs.microsoft.com/pix/. 18, 44, 57, 79, 81, 83

[38] A. Munshi, B. Gaster, T. G. Mattson, and D. Ginsburg, *OpenCL programming guide*. Pearson Education, 2011. 2

[39] NVIDIA. (2020). NVIDIA Nsight Visual Studio Edition. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/nsight-visual-studio-edition. 82

[40] ——, (2021). CUDA-GDB. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/cuda-gdb. 82

[41] ——, (2021). CUDA-MEMCHECK. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/cuda-memcheck. 82

[42] ——, (2021). Linux Graphics Debugger. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/designworks/linux-graphics-debugger. 81

[43] ——, (Feb. 2021). NVIDIA Nsight Graphics. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/nsight-graphics. 18, 44, 57, 81, 83

[44] ——, (2021). NVIDIA Nsight Systems. Accessed: 2021-03-11, [Online]. Available: https://developer.nvidia.com/nsight-systems. 81

[45] ——, *CUDA C++ Programming Guide*, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, Accessed: 2021-03-11. 1, 2

[46] ——, *CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler*, https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/, Accessed: 2021-03-11. 1

[47] ——, *NVIDIA Turing GPU Architecture: Graphics reinvented*, https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, Accessed: 2021-03-11. 10

[48] ——, *Quake II RTX*, https://github.com/NVIDIA/Q2RTX, Accessed: 2021-03-11. 27

[49] D. Pankratz, T. Nowicki, A. Eltantawy, and J. N. Amaral, "Vulkan Vision: Ray Tracing Workload Characterization using Automatic Graphics Instrumentation," in *Code Generation and Optimization (CGO)*, Virtual Conference: ACM/IEEE, 2021, pp. 137–149. 44, 50, 51, 61, 67

[50] R. Rangan, M. W. Stephenson, A. Ukarande, S. Murthy, V. Agarwal, and M. Blackstein, "Zeroploit: Exploiting Zero Valued Operands in Interactive Gaming Applications," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, Aug. 2020. 80

[51] P. Rebohle. (2021). DXVK. Accessed: 2021-03-11, [Online]. Available: https://github.com/doitsujin/dxvk. 57

[52] D. Shen, S. L. Song, A. Li, and X. Liu, "CUDAAdvisor: LLVM-Based Runtime Profiling for Modern GPUs," in *Intern. Symp. on Code Generation and Optimization (CGO)*, Vienna, Austria, 2018, pp. 214–227. [Online]. Available: https://doi.org/10.1145/3168831. 1, 8

[53] G. Simons, M. Ament, S. Herholz, C. Dachsbacher, M. Eisemann, and E. Eisemann, "An Interactive Information Visualization Approach to Physically-Based Rendering," in *Conference on Vision, Modeling & Visualization*, M. Hullin, M. Stamminger, and T. Weinkauf, Eds., Bayreuth, Germany: The Eurographics Association, Oct. 2016, pp. 145–152. 2, 7, 8, 83

[54] J. Sjoholm. (2020). Best Practices: Using NVIDIA RTX Ray Tracing. Accessed: 2021-03-11, [Online]. Available: `https://developer.nvidia.com/blog/best-practices-using-nvidia-rtx-ray-tracing/`. 7, 59, 66

[55] M. Strengert, T. Klein, and T. Ertl, "A Hardware-Aware Debugger for the OpenGL Shading Language," in *Graphics Hardware (GH)*, San Diego, CA, USA, 2007. 8, 18, 80

[56] U. Technologies. (2021). Unity. Accessed: 2021-03-11, [Online]. Available: `https://unity.com/`. 45, 52

[57] W. Usher. (2019). ChameleonRT. Accessed: 2021-03-11, [Online]. Available: `https://github.com/Twinklebear/ChameleonRT`. 27, 61

[58] Valve, *Steam Overlay*, `https://partner.steamgames.com/doc/features/overlay`, Accessed: 2021-03-11. 19

[59] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *International Symposium on Microarchitecture (MICRO)*, Columbus, OH, USA, 2019, pp. 372–383. 1, 8, 80

[60] I. Wald, "Active Thread Compaction for GPU Path Tracing," in *High Performance Graphics (HPG)*, Vancouver, BC, Canada, 2011, pp. 51–58. 22

[61] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL programming guide: the official guide to learning OpenGL, version 1.2*. Addison-Wesley Longman Publishing Co., Inc., 1999. 1

[62] Zielon. (2020). Ray tracer sandbox in Vulkan. Accessed: 2021-03-11, [Online]. Available: `https://github.com/Zielon/PBRVulkan`. 61, 62