Applications and Limitations of Vulkan-Layer-based Instrumentation

by

Deric Cheung

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

Master of Science

Department of Computing Science

University of Alberta

Abstract

The field of computer graphics is constantly evolving to meet the demands of modern applications. One such evolution is the introduction of low-level Application Programming Interfaces (APIs) such as Vulkan to give applications a greater amount of control over the hardware to facilitate efficient CPU-GPU interoperability and fine-grained scheduling and synchronization of work for the GPU. Despite the low-level nature of Vulkan, there is still a great deal of execution behaviors that are not well understood or are difficult to observe due to black-box hardware and driver implementations. To better understand these behaviors, the Vulkan layer system has been leveraged to profile and debug Vulkan applications by intercepting and modifying the behavior of Vulkan API function calls. The Vulkan layer system has been used to develop frameworks like Vulkan Vision, which employs shader instrumentation to capture shader execution traces and analyzes them to reveal shader execution behaviors such as control-flow divergence, Independent Thread Scheduling on NVIDIA GPUs, and execution hotspots. This thesis further explores the applications and limitations of the Vulkan layer system for analyzing execution behaviors of Vulkan applications to better understand and optimize their performance.

One of the contributions of this thesis is ReRay — an extensible toolchain and workflow for capturing, parsing, analyzing, and manipulating ray-tracing pipeline execution traces to better understand ray-tracing pipeline execution behaviors down to the hardware and driver level. Building on top of prior work with Vulkan Vision, ReRay provides insights into how ray-tracing pipelines in-

teract with the hardware and driver to facilitate the development of more efficient ray-tracing algorithms and applications. Black-box hardware and driver execution behaviors impacting ray-tracing performance such as warp execution order, shader execution divergence, and unnecessary intersection tests are revealed with the use of ReRay.

This thesis also explores the use of Vulkan layers to examine the management and usage of shader resources. Utilizing Vulkan's resource binding model efficiently can be challenging due to the complexity of managing many pipeline variants expecting varying types and numbers of resources for use by its shaders. A Vulkan layer is proposed to observe and evaluate the usage of resources in Vulkan applications, with an additional focus on uniform data — data that remains constant for all shader invocations of an executing pipeline. Uniform data is used for a variety of purposes, such as passing transformation matrices, lighting parameters, and material properties to shaders. Depending on the size, update frequency, and usage of uniform data, Vulkan features such as push constants, inline uniform blocks, and specialization constants can be used to optimize the management of uniform data to reduce the memory access and GPU state-change overheads associated with managing and using uniform data.

In spite of the power granted to Vulkan layers, the Vulkan API and the Vulkan Vision framework have limitations that prevent Vulkan layers from being used to examine cutting-edge hardware features such as Shader Execution Reordering on Ada Lovelace NVIDIA GPUs. Inefficiencies also arise when constructing a Vulkan layer to examine shader resource contents without application-specific context or knowledge. Further limitations in the Vulkan API prevent practical automatic optimization of shader-resource usage on behalf of an application by a application-agnostic Vulkan layer. A discussion of advancements in the Vulkan API, hardware features, and the design of graphics

renderers in game engines also highlights the changing landscape of shader resource management in Vulkan applications. These changes make some aspects of shader resource management and uniform data optimizations less relevant to modern Vulkan applications.

Preface

Chapter 3 of this thesis was once submitted, but not accepted, for publication as **D. Cheung**, T. Nowicki, J. N. Amaral, "ReRay: Deep Ray Tracing Performance Insights in Vulkan". The conceptualization of the tool was a collaborative brainstorming effort from all authors as well as D. Pankratz. I performed the prototyping, implementation, and evaluation of the tool. I also wrote the majority of the content of the chapter, with minor edits and revisions from T. Nowicki and J. N. Amaral. D. Pankratz did not wish to be included as an author in the submission, but nonetheless provided valuable feedback on the tool during the initial stages of development.

Chapter 4 of this thesis is an original work by myself, **D. Cheung**. No part of this chapter has been submitted for publication. I conceptualized and wrote the entirety of chapter, with J. N. Amaral reviewing the writing and making minor revisions along the way. J. N. Amaral also provided feedback on the content and structure of the chapter.

Acknowledgements

I would like to thank Tyler Nowicki for his wealth of knowledge regarding Vulkan, GPU architectures, and computer graphics. Tyler's advice has been invaluable to me throughout my research. I am grateful for the time he has spent listening to my ideas, monitoring my progress, and providing feedback.

I would also like to thank my supervisor at the University of Alberta, Dr. J. Nelson Amaral for his mentorship and guidance in my research. Nelson's experience and expertise in compilers and computer architecture have been instrumental in shaping my research, as well as his advice on how to present my work. I greatly appreciate the opportunities he has provided me with as his graduate student and the connections he has helped me make along the way.

Finally, I want to thank David Pankratz for his help in understanding the design and operations of Vulkan Vision and RayScope. David's work has been a great inspiration to me, and his help with brainstorming ideas for extending Vulkan Vision and RayScope has greatly influenced the development of my research.

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

Acronyms

AABB Axis-Aligned Bounding Box

API Application Programming Interface

AS Acceleration Structure

CPU Central Processing Unit

FPS Frames Per Second

GPU Graphics Processing Unit

PC Personal Computer

RRA Radeon Raytracing Analyzer

SER Shader Execution Reordering

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Thread

SM Streaming Multiprocessor

SPIR-V Standard Portable Intermediate Representation

VRAM Video Random Access Memory

Chapter 1

Introduction

The advent of low-level Application Programming Interfaces (APIs) for computer graphics, such as the Vulkan API [47], has revolutionized the way developers design graphics applications for real-time rendering. With a lower level of abstraction than previous APIs, such as OpenGL [24] and DirectX 11 [67], Vulkan provides developers with more control over the hardware, enabling fine-grained GPU work scheduling and efficient CPU-GPU interoperability. Performance is more predictable due to the thin layer of abstraction over Vulkan API functions, which allows developers to optimize their applications for specific hardware configurations as opposed to the one-size-fits-all approach of older APIs with higher levels of abstraction.

Despite the lower level of abstraction in Vulkan, there are still many execution behaviors that remain elusive — hidden behind the opaque nature of hardware and driver implementations. The Vulkan API abstracts underlying hardware- and driver-specific details to provide a consistent API across different platforms. This abstraction is especially evident in the execution of ray-tracing pipelines, which are a significant leap in complexity compared to the linear graphics pipelines used for rasterization in the vast majority of real-time rendering applications. Although the programming model of ray-tracing pipelines for a single thread is well-defined by the Vulkan specification [62], the details as to how the hardware maps the execution of the ray-tracing pipeline to multiple threads is intentionally left vague to allow hardware vendors to implement ray-tracing pipelines in a way that best suits their hardware mi-

croarchitecture. Even in the case of a single thread, the exact form of data structure used for acceleration structures to hold geometry data, and the algorithms for traversing the acceleration structure are left to the discretion of the hardware and driver implementations.

The need to peer into the behaviors of Vulkan pipeline execution has led to the development of tools such as Vulkan Vision [42], [43]. Vulkan Vision is a shader instrumentation framework that captures detailed shader execution traces from rendering pipelines to reveal execution behaviors such as control-flow divergence and execution hotspots, as well as hardware-specific behaviors like Independent Thread Scheduling on NVIDIA GPUs. The key enabler of Vulkan Vision is the Vulkan layer system it is built on top of. The Vulkan layer system allows software libraries, called Vulkan layers, to intercept and modify Vulkan API function calls in real time to alter the behavior of Vulkan applications. Vulkan layers, such as the Vulkan Validation Layers [63], are commonly used for profiling, debugging, and validation of Vulkan applications. Vulkan layers do not require modifications to the Vulkan application's source code, making them a powerful tool for analyzing Vulkan applications without requiring the developer to make changes to their application.

RayScope [42] is a tool built on top of the Vulkan-Vision framework for capturing, parsing, and analyzing ray-tracing pipeline execution traces to explore common pitfalls in the use of ray-tracing pipelines. RayScope focuses on examining application-level impacts on ray-tracing pipeline execution traces and performance, such as issues with ray parameterization and ray-geometry interactions that can heavily impact frame time — the time necessary to render a single image to be displayed on a display device — or cause visual artifacts. RayScope does not explore the hardware- and driver-level execution behaviors that are also present in ray-tracing pipeline execution traces. These behaviors include the acceleration-structure construction and traversal algorithms, and the mapping of ray-tracing pipeline execution to multiple hardware threads. Applications have limited control over these types of execution behaviors, and yet these behaviors can significantly impact ray-tracing performance. These behaviors should be accounted for when designing ray-tracing algorithms and

applications targeting a variety of hardware configurations.

The lower-level nature of Vulkan also introduces challenges for developers in determining how to best utilize the API to achieve optimal performance for all types of rendering tasks. One of these challenges is in the management of shader resources, such as buffers and textures, that are used by shaders to perform rendering tasks. Poor management of shader resources can lead to increased overheads due to increased memory usage, increased time spent preparing and processing GPU commands, and GPU state-change overheads such as stalls from waiting for pipelines to finish executing with older references to data. A Vulkan layer can feasibly be developed to observe the usage of shader resources in Vulkan applications to provide insights into how to optimize the management of these resources.

Despite the potential benefits to using Vulkan layers for profiling Vulkan applications and for revealing execution behavior, Vulkan has limitations that make observing and manipulating certain aspects of application execution behaviors inefficient. For instance, the Vulkan API is not at a low-enough level to allow a Vulkan layer to provide detailed information about certain execution behaviors such as hardware warp repacking in the ray-tracing pipeline.

Nonetheless, the Vulkan layer system is a powerful tool for profiling and debugging Vulkan applications, and the development of tools, such as Vulkan Vision and RayScope, have demonstrated the potential of the Vulkan layer system for revealing execution behavior in Vulkan applications. This thesis discusses applications, and limitations, of the Vulkan layer system for analyzing execution behavior of Vulkan applications to better understand their performance characteristics.

The contributions of this thesis are as follows:

1. An extensible toolchain, called ReRay, for capturing, parsing, and analyzing ray-tracing pipeline execution traces to reveal black-box hardware execution behaviors that impact ray-tracing performance, and enable experimentation with different hardware features and ray-tracing algorithms through the manipulation and re-analysis of the execution traces.

ReRay extends the work of Vulkan Vision and RayScope to give more insight into ray-tracing performance and extend the capabilities of existing tools.

2. A hardware-agnostic method for observing warp execution order in ray-tracing pipelines through the use of the ReRay toolchain, and an examination of NVIDIA's ray-tracing pipeline warp execution order for a variety of launch sizes.

3. A demonstration and evaluation of a tool in the ReRay toolchain, called the Replayer, for estimating overall SIMT efficiency of a ray-tracing pipeline execution trace after a simulated warp repacking is performed. The warp repacking is specified by a user-provided thread-to-warp assignment to enable experimentation. Performance results illustrate how warp repacking alters the shape of the distribution of ray-tracing pipeline execution times.

4. Out-of-the-box heat-map generation tools in the ReRay toolchain for examining SIMT efficiency, intersection shader activity, and unnecessary intersection tests throughout a frame rendered by a ray-tracing pipeline.

5. A characterization of the sources of unnecessary intersection tests in a ray-tracing pipeline execution. These sources of unnecessary intersection tests originate from acceleration structure construction and traversal.

6. A preliminary study of NVIDIA's explicit hardware warp repacking feature called Shader Execution Reordering (SER), including a discussion of issues regarding correctness when using the Vulkan Vision framework for examining ray-tracing pipeline execution traces from applications using hardware warp repacking. These issues prevent ReRay from being used to analyze such traces. An alternative method, using a proprietary Vulkan API extension, is employed instead to perform a preliminary study of SER performance and behavior on a sample Vulkan ray-tracing application.

7. A proposal discussing the construction of a Vulkan layer to observe the usage of shader resource descriptors and uniform buffers in Vulkan applications to leverage prior work on uniform buffer value specialization and to examine the relevance of prior work on descriptor-set optimization. General ideas for the design of the layer are presented alongside potential optimizations that take advantage of Vulkan-specific features not present in older APIs like OpenGL. Methods for measuring the benefits of these optimizations are also discussed.

8. An identification of several shortcomings in the Vulkan layer system that prevent the practical implementation of a Vulkan layer to perform automatic uniform buffer value specialization. These shortcomings stem from the design of the Vulkan API and the lack of application-specific information available to the Vulkan layer to seamlessly compile specialized pipelines, maintain correctness during specialization, and maximize the benefits of uniform buffer value specialization by eliminating writes to uniform buffers from the CPU.

9. A discussion of advances in the Vulkan API, hardware features, and the design of graphics renderers in game engines that have changed the way shader resources are managed in Vulkan applications. These advances have made prior work in descriptor-set optimization and uniform-buffer value specialization less relevant. These advances also made the design of a Vulkan layer to observe shader resource usage more challenging to implement.

Chapter 2

Background

2.1 GPU Execution Model

Rendering graphics is an inherently data-parallel computation: each pixel in a frame can be computed independently of other pixels by the same sequence of instructions. Graphics Processing Units (GPUs) are specialized hardware units designed to exploit this the inherent data-parallelism in graphics rendering and other workloads that also exhibit data parallelism. To facilitate efficient computation, modern GPU programming languages employ Single Instruction Multiple Data (SIMD) architectures that enable the execution of the same instruction on multiple data elements simultaneously. From the perspective of a GPU programmer, GPU programming languages have a Single Instruction, Multiple Thread (SIMT) execution model whereby groups of threads— called *warps* —execute the same instructions in lock-step. In the SIMT execution model inefficiencies can arise when threads in a warp diverge in their execution paths or memory accesses. Conditional code is executed my making some of the threads in a warp execute a No-Operation instruction (No-op) while the other threads executed the instruction in the code. When some of the threads in a warp execute No-op because of an earlier conditional, the computation is less efficient because of *control-flow divergence*. Threads that are executing a No-op are said to be inactive. In the worst case, the execution of each thread in a warp is serialized. *SIMT efficiency* is a measure of the average number of threads that are active per warp.

A warp of thread may execute an instruction that accesses memory. If all

the memory locations accessed by all the threads in the warp are in the same cache line – or in a small contiguous set of cache lines in some architectures – then the access is *coalesced*. If a memory access from the threads in a warp requires access to many different cache lines, then the computation has *data divergence*, which reduces the efficiency of the computation.

These sources of divergence are important to consider when optimizing GPU programs, as they can lead to significant performance bottlenecks, especially in graphics rendering, because the longest-running thread dictating the frame rate of an application.

2.2 Vulkan

Vulkan is a graphics and general-purpose compute Application Programming Interface (API) developed by the Khronos Group. The API is designed to provide developers with low-level access to the GPU. In this API the application is responsible for the handling of memory management, life-time management of resources, scheduling of tasks for the GPU, and synchronization between the CPU and the GPU.

The low-level design of Vulkan enables developers to optimize the performance of their applications by tailoring them to specific hardware configurations and application use cases. Although this was possible with older graphics APIs such as OpenGL, it was often difficult to achieve due to the high-level abstractions and the black-box nature of the graphics driver implementations that resulted in unpredictable performance.

A configuration of the GPU to perform work is represented in Vulkan by a *pipeline state object*, or simply a *pipeline* for short. A pipeline represents a logical sequence of stages through which data are processed. There are three types of pipelines in Vulkan: graphics, compute, and ray tracing. Graphics pipelines are used for rendering 3D graphics using a technique called *rasterization*. Compute pipelines are used for general-purpose computing on the GPU. Ray-tracing pipelines are used for rendering 3D graphics using a technique called *ray tracing*.

This thesis focuses on the instrumentation and optimization of programmable pipeline stages within each of the three types of pipelines that require the user to supply programs called *shaders*. Shaders are typically written in a high-level shading language such as the OpenGL Shading Language (GLSL) or High-Level Shader Language (HLSL) and compiled into SPIR-V bytecode for Vulkan to consume.

The compute pipeline is the simplest of the three pipeline types, consisting of a single programmable stage called a compute shader. It is used for general-purpose computing on the GPU, but is still used for graphics rendering in many applications to perform tasks that do not fit well into the graphics pipeline or ray-tracing pipeline. Such tasks include certain post-processing effects on images or physics calculations for physical interactions between objects in a scene.

The next two sections briefly describe the graphics and ray-tracing pipelines in Vulkan.

2.3 Graphics Pipeline

The graphics pipeline is a logical pipeline describing the process by which a GPU turns 3D objects into a 2D representation that is written to a frame buffer — a region of memory that holds the color values for each pixel in an image. The pipeline consists of several configurable or programmable stages that are executed in a fixed order. The inputs to the pipeline include vertex data, a depth buffer, textures, and other application-defined data to be used for rendering. An in-depth explanation of the graphics pipeline is beyond the scope of this thesis. Two mandatory programmable stages of the graphics pipeline — the vertex shader and fragment shader — are important to understand because they are the focus of the optimization work in this thesis.

The graphics pipeline begins with vertex processing, which involves reading vertex data from buffers and transforming the vertices into screen coordinates. The transformation of vertices into screen coordinates is handled by the *vertex shader* stage of the graphics pipeline. Applications may also use the vertex

shader to perform additional computations on the vertices, such as lighting calculations or texture coordinate transformations. Following vertex processing, the object is rasterized into *fragments* — pixels that may or may not be visible in the final frame buffer. The *fragment shader* stage is responsible for determining the color of each fragment in the frame buffer using textures, lighting conditions, and other information from application-defined input data provided to the pipeline. After fragment shading the fragments are blended with the existing values in the frame buffer, or discarded if they are occluded by other objects.

Applications may create multiple different graphics pipelines for different types of objects and visual effects. The collection of all objects to be rendered is typically stored in a graph data structure called a *scene graph*. A typical application may iterate over all objects in a scene graph and perform a draw call (i.e., an invocation of a graphics pipeline) to render each object. There are many optimizations that one could make to scene-graph iteration, such as view-frustum culling to omit draw calls for objects that are not within the camera's field of view, and occlusion culling to omit draw calls for objects that are completely occluded from the camera's point of view by other objects in the scene. Furthermore, the order in which objects are iterated over in the scene graph should aim to minimize the amount of state change on the GPU. State change refers to the process of switching between different graphics pipelines and switching of input data provided to the pipelines. State change can be costly due to the need for the GPU to reconfigure itself and introduce execution barriers to wait for the GPU to finish processing the previous pipeline with older data.

2.4 Ray-tracing Pipeline

Ray tracing is a rendering technique that simulates the paths of light throughout a scene with the resulting image formed by light rays that hit the camera. However, simulating all possible light rays emitted by light sources is infeasible. Instead, ray tracing is performed backwards: rays are traced out of the

camera and into the scene. Rays that hit geometry can generate additional rays, and rays that hit light sources will contribute to a pixel color for the rendered image.

The Vulkan API enables real-time ray tracing through a *ray-tracing pipeline*. While ray tracing is also possible using a feature called *ray queries*, this thesis focuses only on the ray-tracing pipeline, which is primarily used for fully ray-traced applications. Ray queries behave differently and are more often used in hybrid rendering techniques that combine traditional rasterization with ray-traced visual effects.

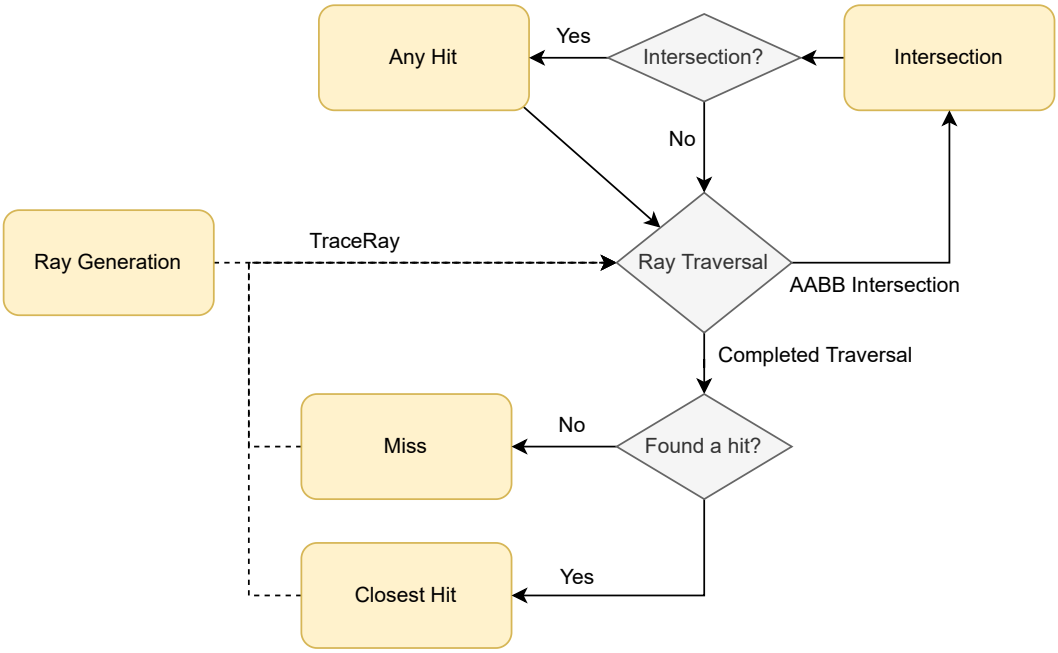

Figure 2.1: The ray-tracing pipeline

The Vulkan ray-tracing pipeline is shown in Figure 2.1. It consists of five types of programmable shaders shown in yellow rounded boxes: ray generation, intersection, any hit, closest hit, and miss. Each geometric primitive in the application can have a unique intersection, any hit, and closest-hit shader, but many primitives will often share the same set of shaders. There exists an additional callable shader that can be invoked by any shader in the pipeline, but it is not explored in this work.

The ray-tracing pipeline can be considered a GPU kernel with the number

of executing threads dictated by a *launch size* that is set by the application. In most cases, the launch size is the same as the dimensions of the window or canvas on which the application draws. Once the application sends the Vulkan API command `vkCmdTraceRaysKHR`, the ray-tracing pipeline is executed with the ray-generation shader as the entry point. The ray-generation shader calls the `OpTraceRayKHR` instruction to trace a ray and start the ray-traversal procedure.

Ray traversal finds all primitives that the ray hits and is responsible for invoking the other shaders in the pipeline. Application geometry is stored in a specialized data structure called an Acceleration Structure (AS) designed to efficiently find ray intersections with geometric primitives. Each geometric primitive in an acceleration structure is enclosed by an Axis-Aligned Bounding Box (AABB). When a primitive's AABB is intersected by a ray, its corresponding intersection shader is called to confirm whether or not the ray has intersected with the primitive contained in the AABB. Triangle primitives have their own implicit intersection shader that is handled by the hardware, while procedural (non-triangular) primitives must have an application-defined intersection shader to confirm whether or not a ray has intersected with the primitive. The intersection shader calls the `OpReportIntersectionKHR` function to report an intersection, where KHR indicates that this is a Khronos function.

If an intersection with a primitive is reported, the primitive's any-hit shader is invoked if available. Its purpose is to determine whether the intersection should be ignored. For example, the shader may perform an opacity test, which often involves expensive texture accesses, to check if the intersection is with a transparent part of the primitive. The any-hit shader calls the function `OpIgnoreIntersectionKHR` to cause the intersection to be ignored.

Once ray traversal is complete, the closest-hit shader of the closest primitive is invoked. If the ray did not intersect with anything, then the miss shader, specified by an argument of `OpTraceRayKHR`, is invoked instead. The miss shader and the closest-hit shader can also call `OpTraceRayKHR` to trace additional rays.

2.5 Ray-Tracing Performance

Unlike the graphics pipeline for rasterization, the ray-tracing pipeline is recursive. From each intersection additional rays may be traced with their origin at the intersection point. Together these rays are called a *ray path*. However, ray paths are unpredictable because small differences in the initial origin or direction can significantly change the subsequently intersected geometries. As a result, ray tracing often suffers from high divergence in SIMT/SIMD execution and memory accesses on GPUs. Clever techniques and simplifications or assumptions are often required to make ray-tracing performance adequate for real-time applications [15], [23], [29].

2.6 Vulkan Vision

To analyze the performance of Vulkan applications, Pankratz et al. developed the Vulkan Vision shader instrumentation framework. That framework is built on top of the Vulkan validation layer and provides several useful building blocks for creating new shader instrumentations such as unique identification of threads and warps, and for managing the reading, writing and transfer of instrumentation data between the host and the device. Moreover, Vulkan Vision works with any standards-conforming Vulkan application and does not require modification of the application source code to operate.

Using Vulkan Vision, Pankratz et al. show several insights into the performance of Vulkan ray-tracing applications, including a measure of SIMT efficiency, a characterization of causes for SIMT divergence, the behavior of NVIDIA-hardware-specific Independent Thread Scheduling, and execution hot-spot detection.

2.7 RayScope

Following the work on Vulkan Vision, RayScope [42] is a tool to collect and visualize the rays and geometry of a Vulkan ray-tracing application. By collecting data about every ray traced by an application, RayScope can identify

issues in ray-tracing applications such as visual defects and improperly set API parameters that result in poorer performance. RayScope instruments the ray-tracing shaders of an application using the Vulkan Vision framework to obtain the ray data necessary for its visualization tool.

Chapter 3

ReRay

Ray tracing is a technique widely-used to render photorealistic imagery for films and 3D visualization software. However, the use of ray tracing in real-time interactive rendering applications, such as video games, is relatively recent [52]. Despite hardware acceleration [35], [36], the introduction of a ray-tracing pipeline to graphics Application Programming Interfaces (APIs), and a strong interest by PC hardware vendors [53], the performance of real-time ray tracing still lags behind rasterization [45] — the dominant rendering technique used for most real-time graphics applications. One of the reasons for this disparity is the lack of tuning and optimizations to the ray-tracing pipeline in comparison to the more mature rasterization graphics pipeline. While there are many known performance issues in the ray-tracing pipeline, there are relatively few tools and methods to identify and overcome them.

This chapter addresses the following research question: *How can we provide insights into the performance of ray-tracing applications to aid in the design and optimization of ray-tracing algorithms and hardware-accelerator designs?* To answer this question, this chapter presents ReRay, a platform-agnostic and extensible toolchain and workflow for evaluating, debugging, and optimizing Vulkan ray-tracing applications by capturing and analyzing ray-trace execution data generated from automatic GPU shader code instrumentation.

The data obtained from GPU shader-code instrumentation can provide insights into hardware, driver, and application behavior to aid in ray-tracing algorithm design. These insights come in the form of performance metrics

and heat maps that reveal several aspects affecting ray-tracing performance such as hardware work assignment, and acceleration-structure construction and traversal.

ReRay can also be used to rapidly evaluate potential changes to the application's renderer before the developer carries out a costly implementation. To demonstrate this capability, we evaluate how SIMT efficiency and performance is influenced by the warp execution order and thread-to-warp assignment. ReRay is used to collect and analyze ray and warp execution data to visualize NVIDIA's warp execution order and thread-to-warp assignment without the use of proprietary NVIDIA extensions. We compare this assignment to other thread-to-warp assignments created by several warp repacking methods to improve SIMT efficiency and performance.

To aid in the rapid evaluation of changes to the rendering algorithm, ReRay implements a simulator to estimate the resulting SIMT efficiency of an application for any given thread-to-warp assignment. The simulator does not require re-running the application with the new thread-to-warp assignment to provide an estimate of SIMT efficiency. To measure real SIMT efficiency, we modify an application to use a custom thread-to-warp mapping by implementing *thread swizzling*, a technique that allows each thread to behave as another thread according to a specified one-to-one mapping. ReRay can be used to measure the real SIMT efficiency of the application from warp execution data for each thread-to-warp assignment. We then compare the performance of the various thread-to-warp assignments using NVIDIA Nsight Graphics [41] and NVIDIA Nsight Systems [38].

The rest of this chapter is organized as follows: Section 3.1 provides an overview of the data collection mechanism used by ReRay. Section 3.2 introduces the ReRay toolchain and its included modules. Section 3.3, discusses how ReRay's modules can be used to improve the performance of a single frame of a Vulkan ray-tracing application through warp repacking. Section 3.4 compares the SIMT efficiencies and performance of ReRay's repacking methods to NVIDIA's default thread-to-warp assignment. Finally, Section 3.5 outlines other modules included with ReRay for examining the performance of

ray-tracing applications from analysis of execution traces of the ray-tracing pipeline, such as wasteful shader executions during ray traversal.

3.1 Data-Capture Mechanism

ReRay's mechanism for data capture is a modification and extension of RayScope's ray-data collector which itself is built using the Vulkan Vision shader instrumentation framework. This section describes our collection of ray data as well as an improved method for measuring SIMT efficiency over that provided by Vulkan Vision.

3.1.1 Ray Data Collection

ReRay uses Vulkan Vision to instrument ray-tracing shaders at the beginning of shader calls and immediately before key ray-tracing function calls. The shaders are instrumented at the hardware-agnostic SPIR-V intermediate representation before they are sent to the Vulkan drivers.

The collected data is comprised of key *ray events* in ray-tracing shader programs. These events mark the execution of ray-tracing shaders and the values of key variables in those shaders, or calls to ray-tracing functions and their arguments inside ray-tracing shaders. More precisely, ReRay captures the following ray events:

1. **Begin:** A call to the function *OpTraceRayKHR*. The function arguments *Ray Origin*, *Ray Direction*, *Ray Tmin* and *Ray Tmax* are recorded along with this event. If the ray flag *SkipClosestHitShaderKHR* is set, then this is an **Occlusion Begin** event because this flag is most often used to trace shadow/occlusion rays that do not care what object is hit.

2. **Intersection:** Beginning of execution of an intersection shader. The values of the built-in variables *RayTmaxKHR*, *InstanceID* and *PrimitiveID* are written alongside this event. *RayTmaxKHR* in this event records the distance to the closest object so far in the ray traversal.

3. **Report Intersection:** A call to the function *OpReportIntersectionKHR*. The value of the *Hit* argument is recorded with this event, which contains the distance to the intersected object along the ray. This event may only occur in an intersection shader and therefore can only ever follow an **Intersection** event.

4. **Any Hit:** Beginning of execution of an any-hit shader. The values of the built-in variables *RayTmaxKHR*, *InstanceID* and *PrimitiveID* are written alongside this event. *RayTmaxKHR* in this event is the distance to the intersected object along the ray.

5. **Ignore Intersection:** A call to the function *OpIgnoreIntersectionKHR*. This event may only occur in an any-hit shader and therefore can only ever follow an **Any Hit** event.

6. **Miss:** Beginning of execution of a miss shader. No additional variables are recorded with this event, as no object is hit.

7. **Closest Hit:** Beginning of execution of a closest-hit shader. The values of the built-in variables *RayTmaxKHR*, *InstanceID* and *PrimitiveID* are written alongside this event. *RayTmaxKHR* in this event is the distance to the intersected object along the ray.

8. **Implicit Hit:** If a ray began with an **Occlusion Begin** event and did not end with a **Miss** event, then the ray is assumed to end with an **Implicit Hit** event.

In the **Any Hit**, **Intersection**, and **Closest Hit** ray events, ReRay captures the *InstanceID* and *PrimitiveID* to uniquely identify the object being processed by the shader.

Except for an implicit-hit event, which has no associated shader, the *shader ID* of each ray event is recorded to indicate which shader created the ray event. Applications may use more than one of the same shader type. Thus the shader ID can distinguish same-type ray events originated from different shaders.

To capture hardware behavior, ReRay uses Vulkan Vision's unique identification primitive to generate unique *thread IDs* and *warp IDs* for each thread and warp. Each ray event is recorded with the thread ID and warp ID that generated the event.

3.1.2 SIMT Shader Efficiency and Warp Data Collection

SIMT efficiency is a measure of the SIMD utilization of GPU hardware during execution. Pankratz et al. measured SIMT efficiency using Vulkan Vision by instrumenting shader programs to record the active thread bitmasks of warps at the entry of every basic block [43]. The SIMT efficiency is measured as the sum of the number of set bits (active threads) divided by the total number of bits from all bitmasks. However, this method of computing SIMT efficiency is not suitable for estimating the effectiveness of the warp repacking methods introduced later in the chapter. The reasons for this are two-fold: for one, Vulkan Vision does not track the origins of bitmasks from different warps and locations in shader source code; all bitmasks are instead aggregated together for the entire application. Second, the basic-block granularity of Vulkan Vision's method, in tandem with the lack of origin tracking, makes estimating changes to SIMT efficiency after warp repacking a complex task. As will be discussed in Section 3.2.1, we want a method of estimating SIMT efficiency that is closely tied to the ray-event data to make it easier to predict the resulting SIMT efficiency after warp repacking.

To measure SIMT efficiency, ReRay captures *warp events* from an application. Warp events record per-warp shader execution traces of an application's ray-tracing pipeline. Each warp event records the warp ID which generated it, the shader ID identifying the shader from which the event was generated, and the active thread bitmask of the warp at the time of the event. None of the shader variables or function arguments are recorded with a warp event because they are thread-specific.

In comparison to ray events, warp events consist only of **Begin**, **Occlusion Begin**, **Intersection**, **Any Hit**, **Miss**, and **Closest Hit**. **Report Intersec-**

tion and **Ignore Intersection** are not recorded for warp events because these result from relatively inexpensive function calls. Besides, an active thread bitmask for a **Report Intersection** warp event would be equivalent to the active thread bitmask for the subsequent **Any Hit** warp event, and an **Ignore Intersection** event does not affect SIMT efficiency because all threads in a warp will continue with the ray traversal procedure regardless of whether or not an intersection was ignored.

The active-thread bitmasks recorded by the warp events are used by ReRay to compute the SIMT efficiency of an application on a per-warp basis. For the **Begin** and **Occlusion Begin** events, the active-thread bitmask is recorded just before the call to *OpTraceRayKHR*. For the shader call events (**Intersection**, **Any Hit**, **Miss**, and **Closest Hit**), the active-thread bitmask is recorded at the entry point of the shader program. In effect, ReRay's SIMT efficiency metric is more a measure of shader execution divergence as opposed to all control-flow divergence that may be present during the execution of a ray-tracing pipeline.

A shortcoming in both ReRay and Vulkan Vision is that all bitmasks contribute equally to the SIMT efficiency computation. A more precise metric should take into account the amount of time spent executing with each active-thread bitmask. In Vulkan Vision every basic block contributes equally to SIMT efficiency; a basic block with few instructions would contribute just as much to SIMT efficiency as a basic block with many instructions. Likewise in ReRay's approach, every shader contributes equally to SIMT efficiency. A future study could consider weighting the bitmasks by an estimate of the expected execution time for the basic block or the shader corresponding to each bitmask. The methodology in ReRay assumes that *OpTraceRayKHR* calls and shader calls contribute more significantly to an application's overall SIMT efficiency than individual basic blocks within the shaders.

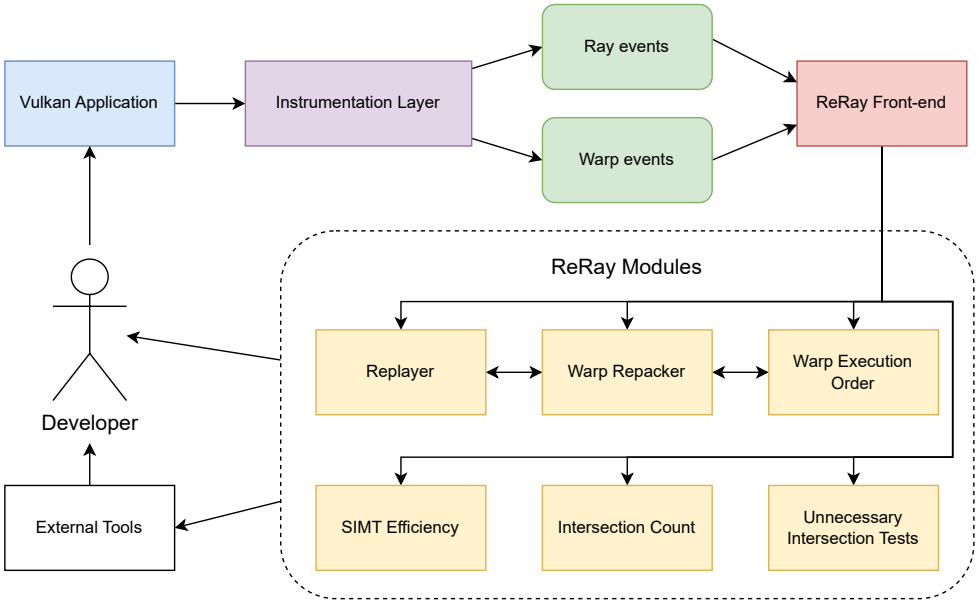

Figure 3.1: ReRay's architecture

3.2 ReRay Toolchain

This section presents ReRay's toolchain for reading and processing the ray and warp data described in Sections 3.1.1 and 3.1.2.

Figure 3.1 shows the complete ReRay architecture and pipeline. The ReRay toolchain architecture consists of a parser front end that is extended by various back-end modules. The front end contains procedures for reading, parsing, and writing ray and warp data. Back-end modules are components that extend the front end to perform additional processing of the data. These modules may be used to compute performance metrics, generate visualizations of certain ray behaviors, or evaluate different methods for sorting and packing threads into warps.

3.2.1 Estimating the Effects of Warp Repacking on SIMT Efficiency

ReRay was developed to enable the study of SIMT efficiency and other performance issues affecting ray tracing. Poor SIMT efficiency comes from divergent

```
// Per-thread ray events from threads in a warp of size 4.
begin, int1, miss
begin, int1, int2, repint, chit
begin, int2, repint, chit
begin, int2, repint, chit

// Synthetic warp-event trace generated by the Replayer
begin 1111, int1 1100, int2 0111, miss 1000, chit 0111
```

Listing 3.1: Simplified example of ray events per thread from four threads and a corresponding synthetic warp-event execution trace generated by the Replayer module. There is one ray generation shader (begin), one closest hit shader (chit), one miss shader, and two intersection shaders (int1, and int2). Only the names of ray events are shown. Ray origins, directions, and t-values are omitted for brevity.

execution within warps. However, many rays often exhibit similar execution traces. If one were to know the behavior of each ray before it is traced, then a way to increase SIMT efficiency is to have warps trace rays with similar behaviors to minimize divergent execution. ReRay attempts to examine a more granular approach to increasing SIMT efficiency by performing a reassignment of threads to warps — a process called *warp repacking*. Each thread in an application is responsible for tracing one or more rays in the ray-tracing pipeline. We hypothesize that applications have similar ray paths which can be executed together in warps to increase SIMT efficiency. An added benefit to this approach is that it does not require complex modification of an application's ray-tracing algorithm to implement.

ReRay's *Replayer* module estimates the changes in SIMT efficiency from warp repacking. When given ray data and a thread-to-warp assignment, the Replayer generates a synthetic version of warp data. The synthetic warp data is used to estimate SIMT efficiency after repacking. As shown in Listing 3.1, synthetic warp events are produced from an examination of **Begin** and shader call ray events from all threads in a warp. Threads that perform the same shader calls execute the shader together, forming the active thread bitmask for the corresponding synthetic warp event. The synthetic warp data reflects an idealized execution on hardware and tends to over-approximate SIMT efficiency as the length of execution traces increases. Nevertheless, synthetic

SIMT efficiency can be used as an estimate to improve packing before running the actual application.

3.2.2 Visualizing Warp Execution Order

The execution order of warps affects data access patterns and thus has a profound impact on performance when taking into account the memory hierarchy [66]. It is said that NVIDIA uses a Z-ordering to execute warps, but the exact properties of the ordering are unknown without using a proprietary NVIDIA Vulkan extension [48]. This ordering can be uncovered using Vulkan Vision without the need for a proprietary extension. As mentioned in Section 3.1.1, ReRay uses Vulkan Vision's unique identification primitive to generate unique thread IDs and warp IDs. Warp IDs are generated by an atomic increment of a value in an instrumentation buffer during runtime, whereas thread IDs are assigned in row-major order over the shader launch size. Therefore, warp IDs give the execution order of warps, and thread IDs are tied to the pixels computed by the threads.

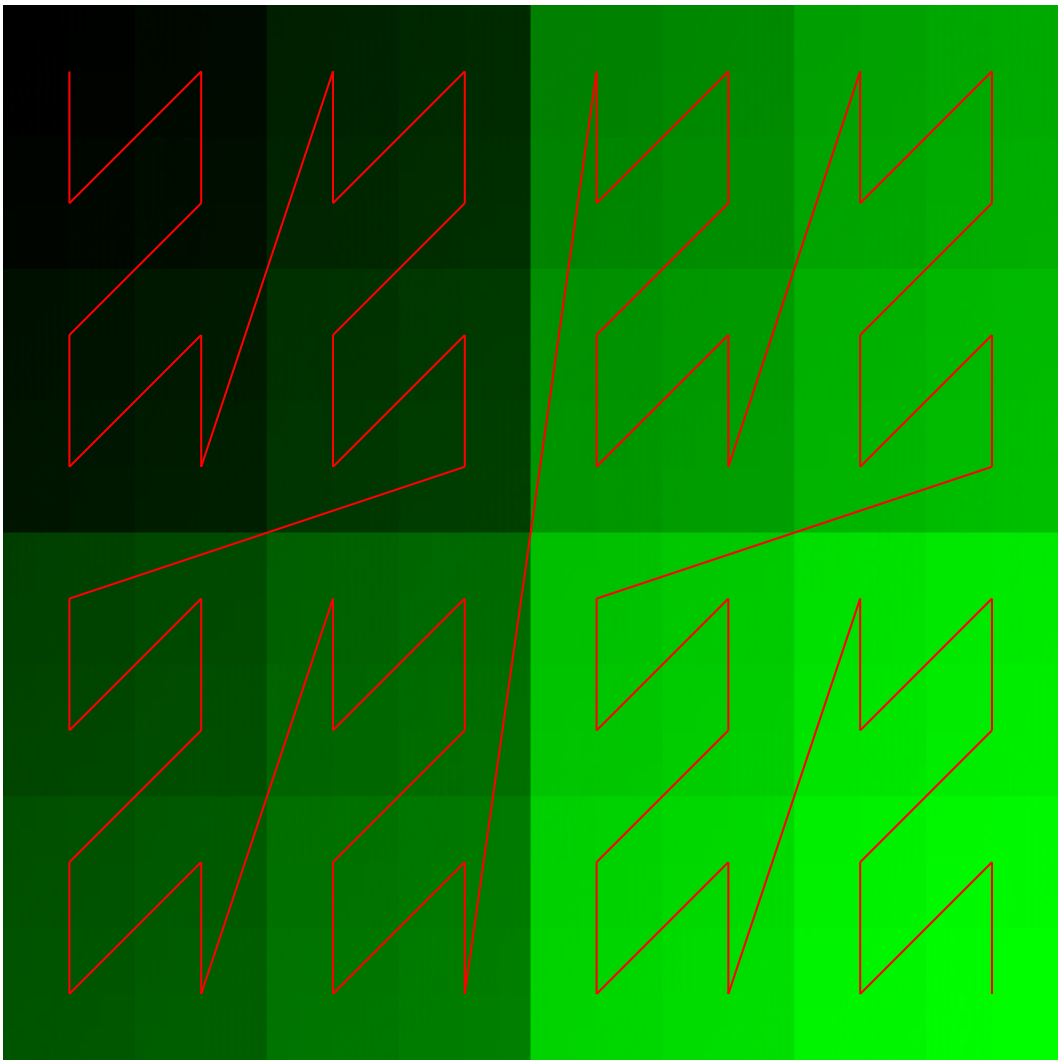

Figure 3.2: ReRay's visualization of warp execution order over a 2048x2048 pixel image. As warp ID increases, the color of pixels go from black to green. Red lines have been added to the image to help illustrate the Z-ordering.

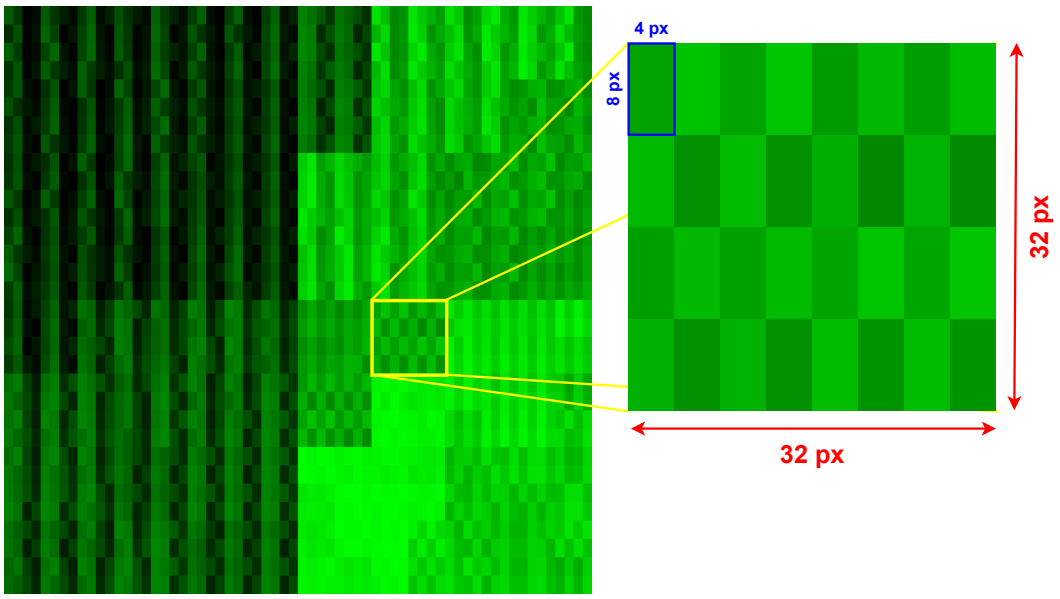

Figure 3.3: ReRay's visualization of warp execution order for a launch size of 256x256. A lighter shade of color indicates a higher warp ID (i.e., the warp executes later than darker colored warps). A 32x32 tile of pixels has been enlarged to show the warp shape.

Using ReRay's *Warp Execution Order* module, an image of the distribution of warp IDs over threads can be created from ray data to visualize the warp execution order. Figure 3.2 shows an image depicting a warp execution order for a launch size of 2048x2048, which was taken from a Vulkan application running on an RTX 3080 using driver version 512.15. Every Vulkan application we tested followed a similar warp execution order when using the same launch size for the ray-tracing pipeline. It is most often the case that the launch size is equal to the resolution at which the application is rendering. Warp IDs appear to follow a Z-ordering, recursively following a 2-3-1-4 pattern — second, third, first, then fourth quadrants over a square grid. However, at smaller scales and launch sizes the Z-ordering is not strict. As shown in Figure 3.3, the execution order of warps for a launch size of 256x256 appears to be chaotic. We suspect that the randomness is due to the state of the hardware warp scheduler at the time of execution and how the hardware resolves data races between concurrent atomic writes. Examining execution at this scale reveals that warps in the image consist of 8x4 rectangles of pixels, with each pixel

Launch Size	Tile Size
256x256	256x256
2048x2048	2048x2048
1024x500	256x512
1024x512	512x512
1024x520	32x32
1280x720	256x256
1600x900	32x32
1920x1080	256x256
2560x1440	512x512
3840x2160	512x512

Table 3.1: A table of some launch sizes and the corresponding tile size in pixels for NVIDIA's Z-ordering. For most applications, the launch size is equal to the resolution of the image being rendered.

corresponding to a thread within the warp.

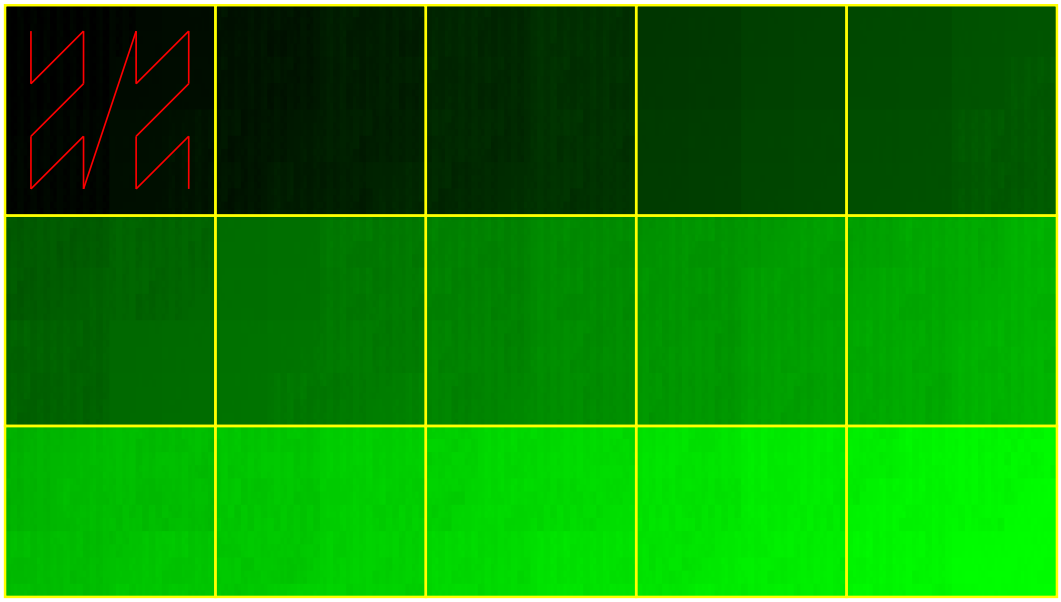

Figure 3.4: ReRay's visualization of warp execution order for a launch size of 1280x720. As warp ID increases, the color of pixels go from black to green. Yellow lines mark the borders of the tiles. Red lines have been added to the first tile to help illustrate the Z-ordering in each tile.

For non-power-of-two launch sizes, the warp execution follows a tiling approach to Z-ordering. Figure 3.4 shows an image depicting a warp execution order for a launch size of 1280x720. In this case, the image is divided into

square tiles of 256x256 pixels. The tiles are executed in row-major order, but within each tile, the warps execute loosely according to a Z-ordering. For launch sizes that have power-of-two dimensions, the tile size is the smaller of the width or height of the launch size. Table 3.1 shows a table of various launch sizes and the corresponding tile sizes. We have yet to discover a pattern that determines the tile size used for any non-power-of-two launch size, and we suspect that the tile sizes are manually set by the driver implementation for common launch sizes.

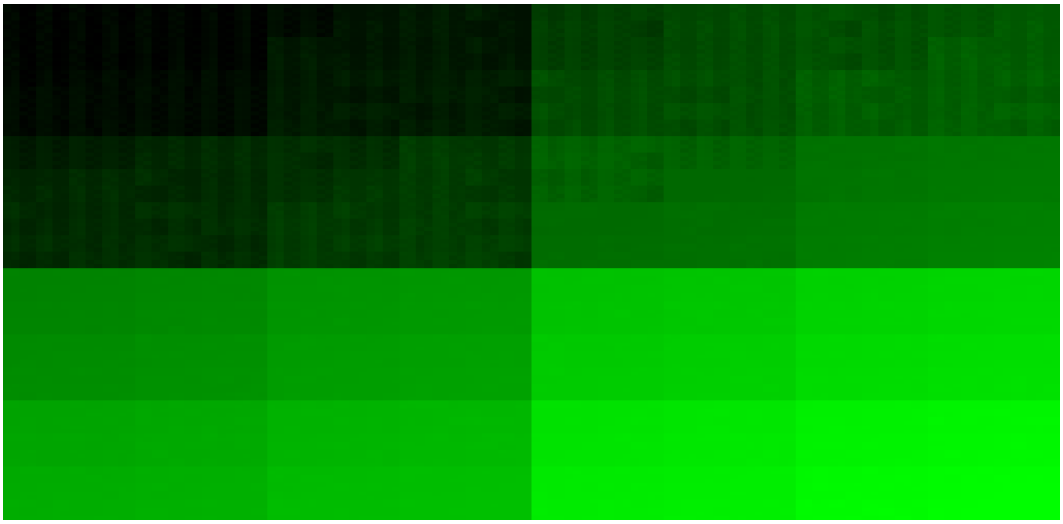

Figure 3.5: ReRay's visualization of warp execution order for a launch size of 1024x500.

Some odd behaviors occur at non-conventional launch sizes. For example, the shape of the warps has typically been 8x4 rectangles of pixels. However, this is not always the case. As can be seen in Figure 3.5 and Table 3.1, some non-conventional launch sizes (e.g., 1024x500) cause the warps to be arranged as 4x8 rectangles of threads with rectangular tiles. The Z-ordering has also changed to follow a Z pattern (2-1-3-4) instead of the 2-3-1-4 pattern seen previously.

The warp execution order, including the tile size and warp shape, depend only on the launch size of the ray-tracing pipeline. For common launch sizes, the tile size may be recorded and used to apply optimizations to algorithms.

For example, tile size could be an important factor in optimizing some algorithms such as creating spatial splits for geometry during acceleration structure construction [32]. Objects could be spatially split until all splits are expected to fall within a single tile. Another optimization could reverse the Z-ordering and tiling to obtain a linear execution order for warps, which can be used to follow a new execution order based on different tile sizes or space-filling curves. Voorhies formally defined a numerical measure of coherence for space-filling curves [66] which could be useful for choosing a curve with the appropriate level of coherence for a given algorithm because some algorithms may benefit from having lower coherence space-filling curves or larger tile sizes.

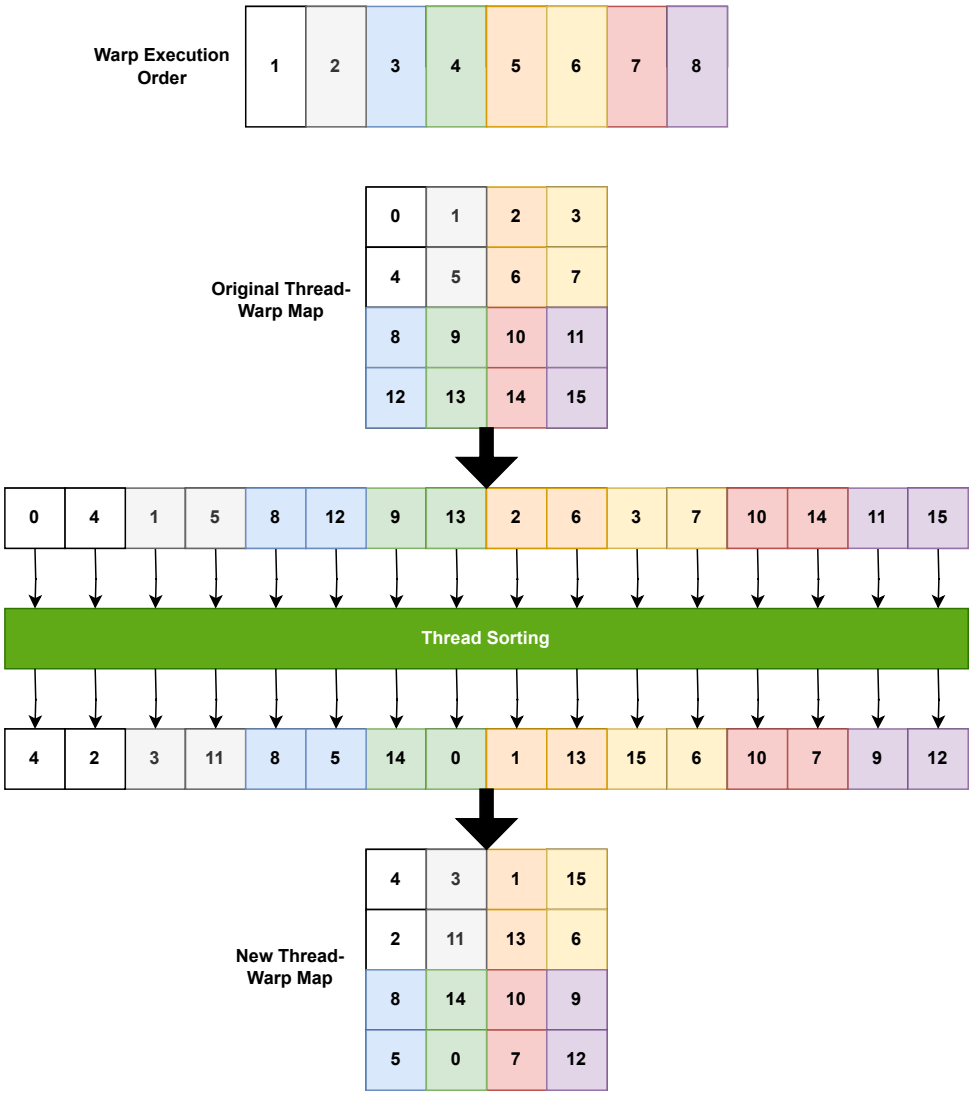

Figure 3.6: ReRay's thread sorting process illustrated with a warp size of 2.

3.3 Warp Repacking Methods

This Section presents a workflow that uses ReRay to improve the performance of a single frame of a Vulkan ray tracing application through warp repacking. Given that the assignment of threads to warps is determined by the drivers and by the hardware, and therefore it cannot be directly modified, this workflow uses a process called *thread swizzling* to repack warps. Thread swizzling

remaps thread IDs such that each thread behaves as if it were another thread. This technique is similar to thread-group ID swizzling [6], but it is applied to individual threads instead of warps. Thread swizzling replaces each shader's use of thread ID with an index into an 1D array, called the thread swizzle, containing a one-to-one mapping between thread IDs. For simplicity, this section describes the thread swizzling of a single unchanging frame of an application. This simplification enables all variables, except for the warp packing, to remain constant throughout the measurements.

The workflow begins with collecting ray and warp data from an application using the instrumentation layer from Section 3.1. The instrumentation layer produces two files: one, `.rays`, for the ray data and another, `.warps`, for warp data. These files are read by ReRay's toolchain front-end and organized into an internal format for use by back-end modules as described in Section 3.2.

Once the ray and warp data are loaded into ReRay's toolchain, the first step in performing warp repacking is to sort threads using an approximate measure of coherence. The sorting process is illustrated in Figure 3.6. The threads of a set of warps are unpacked into a 1D array A that is sorted to produce an array B. A thread swizzle is a one-to-one map from element A_i to B_i. The order in which warps are unpacked determines the resulting assignment of sorted threads to warps. The sorting may be performed globally over all warps, or in subsets or tiles of warps according to fixed tile sizes. For example, sorting may be performed within tiles of warps equivalent to the tiles in NVIDIA's Z-ordering. This chapter examines six sorting methods:

- **Identity:** This baseline represents NVIDIA's default thread-to-warp assignment. A thread swizzle that consists in an identity map is still applied in the sorting step to make the measurement procedure uniform for all cases.

- **Random:** Used as a benchmark, in this sorting each thread is randomly mapped to another thread.

- **Number of Rays (Numrays):** Sort all threads by the number of rays they trace throughout their lifetime. In the case of a tie, the threads are

sorted by their associated warp IDs to try to keep threads in the same warp if they were originally in the same warp. This simple heuristic attempts to maximize the amount of work that each warp processes concurrently. Warps are unpacked according to row-major order before sorting.

- **Tiled Numrays:** A tiled sorting method which uses the same heuristic as the Numrays sort. A tiled sorting method partitions the launch size of the ray-tracing pipeline into tiles. The sorting is then performed locally in each tile instead of globally across all threads. The tile size is chosen to be the tile size used in NVIDIA's Z-ordering for the current launch size. Warps in each tile are unpacked according to row-major order before sorting.

- **Objects:** Sorts all threads by the ordered list of objects that their ray paths intersect. An object encountered by a ray is uniquely determined by an instance-primitive ID, which is the concatenation of the instance ID and the primitive ID from the ray hit event (intersection, any-hit, or closest-hit). The instance-primitive ID of the first object that differs between two threads is used to compare using a less-than operator. This heuristic attempts to maximize the similarity of ray paths by grouping together threads that trace rays encountering the same sequences of objects. In the case that two threads have equivalent object encounters, the tie is broken by warp ID to try to keep threads in the same warp if they were originally in the same warp. If a thread T_i encounters more objects than another thread T_j, and the sequence of objects encountered by both threads is equivalent up to and including the point that T_j encounters its last object, then T_i is ordered before T_j in the sort. Warps are unpacked according to row-major order before sorting.

- **Tiled Objects:** A tiled sorting method that uses the same heuristic as the Objects sort. The tile size is chosen to be the tile size used in NVIDIA's Z-ordering for the current launch size. Warps in each tile are

unpacked according to row-major order before sorting.

To measure the benefits of a given warp repacking, one can manually implement thread swizzling into their application and apply the thread swizzles generated by the above sorting methods. Performance and SIMT efficiency can be measured by profiling an application running with thread swizzling. Alternatively, ReRay's Replayer module (described in Section 3.2.1) can be used to estimate the benefits of warp repacking without modifying or rerunning the application. The Replayer generates synthetic warp data from the application's ray data and the thread-to-warp assignments from each thread swizzle created by the sorting methods. The SIMT efficiency of the synthetic warp data, which we call *synthetic SIMT efficiency*, can be computed offline without the original application. Synthetic SIMT efficiency provides an estimate for the actual SIMT efficiency of an application after warp repacking.

While the main benefit of measuring synthetic SIMT efficiency is to avoid having to modify and rerun the application to support thread swizzling, there is a potential opportunity to automatically apply thread swizzling for an application through a Vulkan layer. However, automating thread swizzling comes with its own set of challenges with regards to determining how and where it should be applied in an application, the management of host and GPU memory for buffers to store the thread swizzles, and the modification of an application's shader resource binding procedures to appropriately update and bind resource descriptors for thread swizzling data. Solving these challenges is beyond the scope of this chapter.

3.4 Evaluation of Warp Repacking Methods

In this section, we compare the SIMT efficiency and performance of the warp repacking methods described in the previous section to NVIDIA's default thread-to-warp assignment.

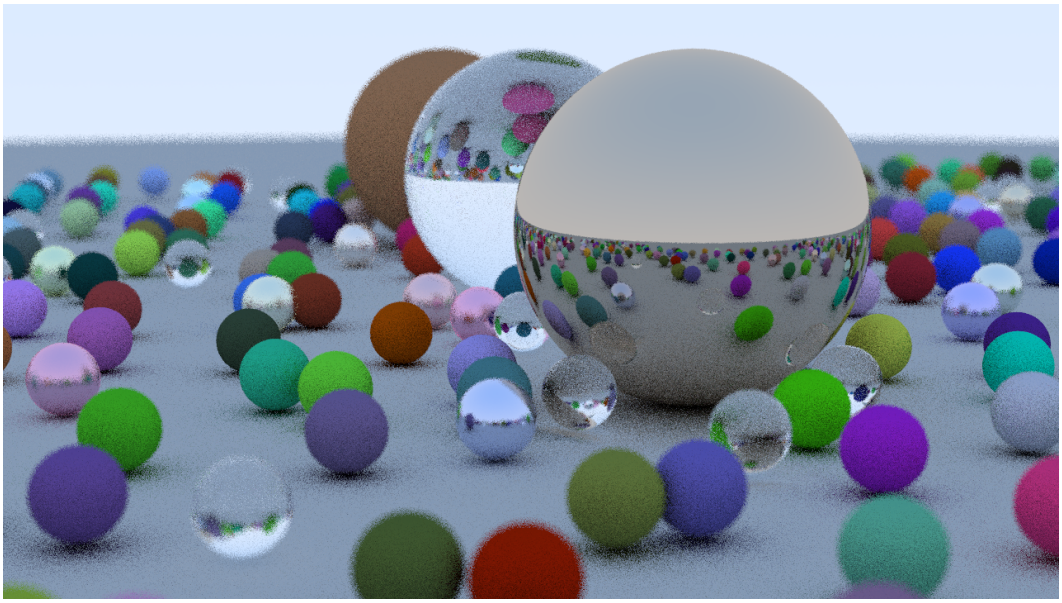

Figure 3.7: Our test application render (RayTracingInVulkan)

3.4.1 Experimental Setup

All the experiments were performed on a Windows 10 (21H2, build 19044) test machine using an NVIDIA RTX 3080 GPU with driver version 512.29. The NVIDIA Nsight Systems [38] software is used to measure the effects of warp repacking on GPU frame times. GPU clock rates are locked to their base frequencies using the NVIDIA System Management Interface [39] command line tool for all experiments.

The experiment uses a modified version of RayTracingInVulkan (Release 7) [13], a Vulkan implementation of Peter Shirley's Ray Tracing in One Weekend [49]. The only modification to the application is the implementation of thread swizzling to test the warp repacking methods. The underlying ray-tracing algorithms of the application remain untouched. The measurements use the scene named *Ray Tracing In One Weekend* that features a large variety of spherical objects with different material properties. All application settings are at their default value except for the number of samples per pixel that is restricted to six to reduce the memory requirements. Reducing the memory requirements is necessary because Vulkan Vision, in its current implementa-

tion, fails to produce instrumentation data when the memory consumption is too high. Ray accumulation is also turned off in the application to ensure an unchanging frame, which allows all variables and program behaviors to be kept constant except for the thread-to-warp assignment. The application's present mode is set to Immediate mode to allow it to render frames as quickly as possible without being stalled by the refresh rate of the display. The resolution (and ray-tracing pipeline launch size) of the application is also left at the default 1280x720. An image of the scene rendered by the application using these settings is shown in Figure 3.7.

Warp repacking is performed using the process outlined in Section 3.3 to obtain a thread swizzle for each sorting method used. For the tiled sorting methods, we use a tile size of 256x256 pixels which matches the tile sizes used for 1280x720 launch sizes in NVIDIA's Z-ordering, as shown in Figure 3.4.

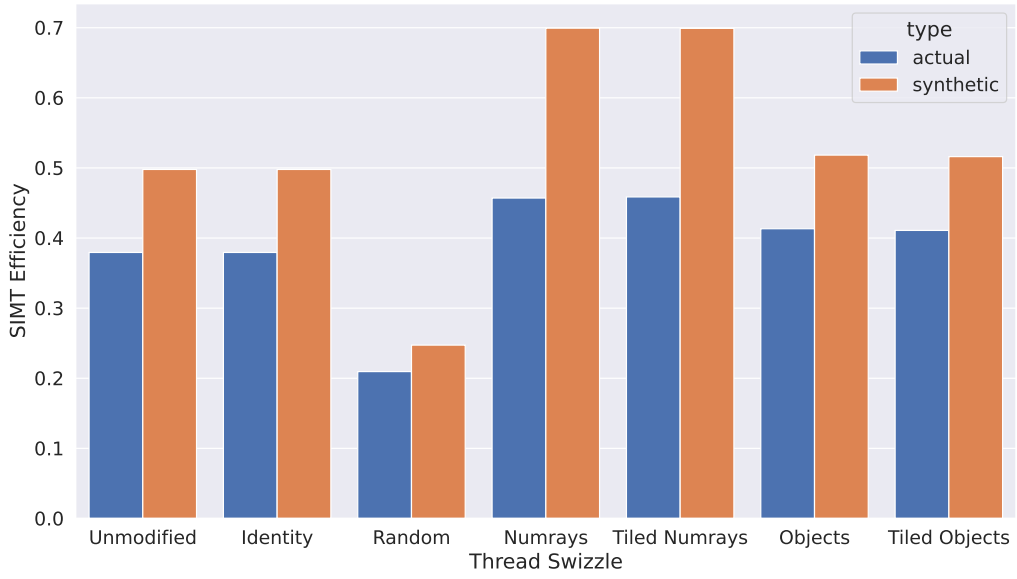

Figure 3.8: Actual and synthetic SIMT Efficiency of the application when using each thread swizzle. The "Unmodified" column represents the application before any modifications.

3.4.2 Analysis of SIMT Efficiency

Synthetic SIMT efficiency is measured for each thread swizzle to get an estimate of the improvement in relation to the actual SIMT efficiency of the application before running it. Afterwards, the application is run with each thread swizzle to collect ray and warp data for measuring the actual SIMT efficiencies after warp repacking. Figure 3.8 shows the SIMT efficiencies of the application with each thread swizzle. An "unmodified" column is included in the figure to indicate the SIMT efficiency of the original unmodified application.

Looking at synthetic SIMT efficiency in Figure 3.8, the Numrays and Tiled Numrays thread swizzles increased synthetic SIMT efficiency by around 40% over the Identity. These results are to be expected, as RayTracingInVulkan uses very few shader programs, which decreases the likelihood of thread divergence due to shader call divergence when sorting by the number of rays traced by each thread. Surprisingly, the Objects and Tiled Objects thread swizzles appear to only marginally affect synthetic SIMT efficiency. Both the Objects and Tiled Objects thread swizzles increase synthetic SIMT efficiency by 4%. The Objects and Tiled Objects thread swizzles likely have lower increases on SIMT efficiency due to coherence only being maximized towards the beginning of thread lifetimes; the longer a thread has been running, the more unique the thread's sequence of encountered objects becomes, causing it to differ from other threads that may have started with the same initial sequence of encountered objects. Objects and Tiled Objects sorting methods put together threads in the same warp if their sequences of encountered objects at the beginning is the same, and do not handle well the cases where some threads diverge from others in the same warp after some execution. The Object and Tiled Objects sorting methods may be more effective at improving SIMT efficiencies for ray-tracing applications that have specialized shaders for each material, or have shorter ray traces from lower numbers of samples per pixel and/or bounces per pixel.

Looking at the actual SIMT efficiency in Figure 3.8, it appears that the

increases in SIMT efficiency by Numrays and Tiled Numrays are not as significant as what was estimated from the synthetic SIMT efficiency. The Numrays and Tiled Numrays thread swizzles increased real SIMT efficiency by 20% and 21% respectively, versus the estimated 40% for synthetic SIMT efficiency. The Objects and Tiled Objects thread swizzles increased SIMT efficiency by 9% and 8% respectively, which are higher than the synthetic estimates of 4%. The differences between synthetic and actual SIMT efficiency may be due to the GPU's thread scheduling and warp execution behavior of shaders and how the Replayer decides warps should execute shaders. For example, NVIDIA's Independent Thread Scheduling tends to cause dips in SIMT efficiency due to some threads no longer executing in lockstep with other threads in the same warp [43]. Additionally, the GPU may also be performing hardware warp repacking during ray traversal; The Vulkan specification [62] states that certain ray tracing instructions may change the set of active threads during execution, including the assignment of threads to warps. However, such hardware warp repacking was not yet implemented when modern video game titles, such as Fortnite, were developing ray-traced visual effects [23]. We have yet to find documentation of ray sorting or warp repacking algorithms in recent driver and hardware releases.

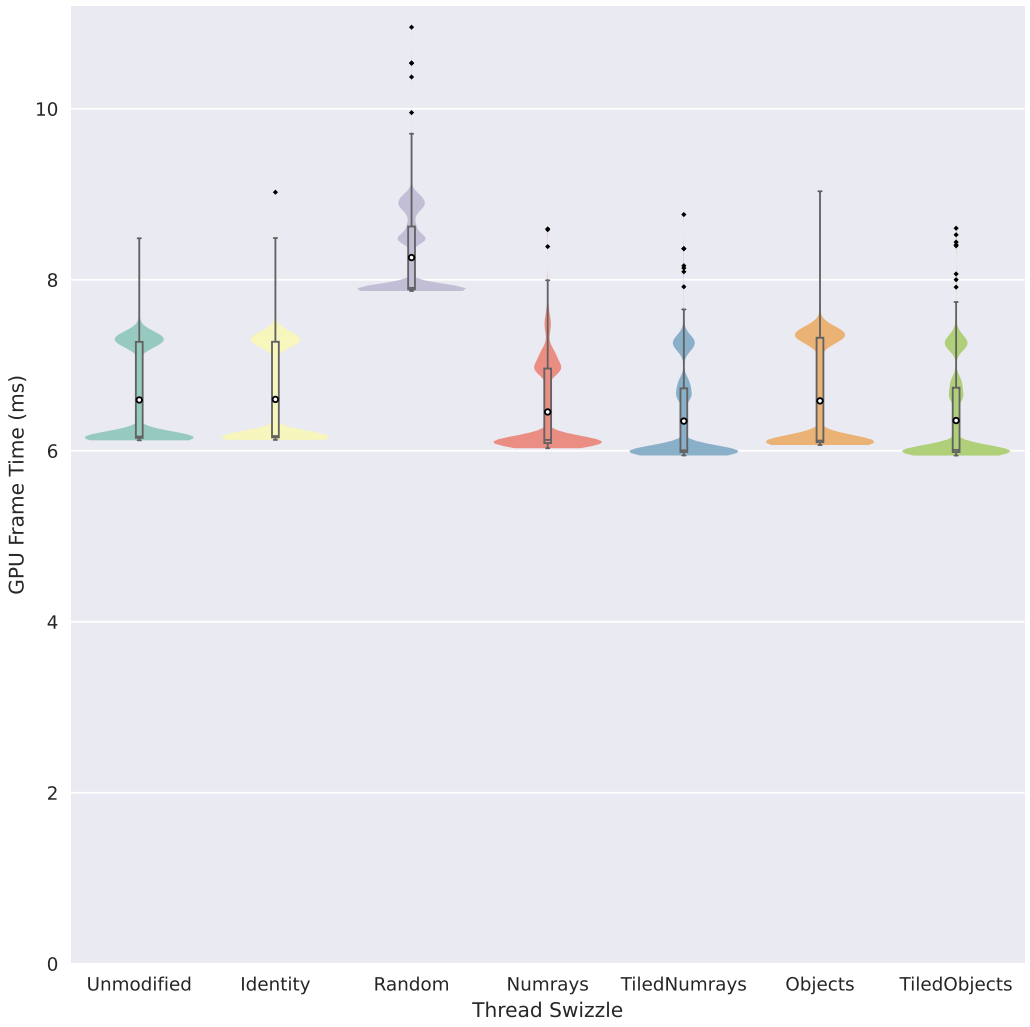

Figure 3.9: Violin plots with overlaid box plots of GPU frame times over 985 frames using each thread swizzle. Data points beyond 1.5 times the interquartile range are considered outliers and are displayed as diamonds for the inner box plots. A white dot in each plot indicates the average GPU frame time.

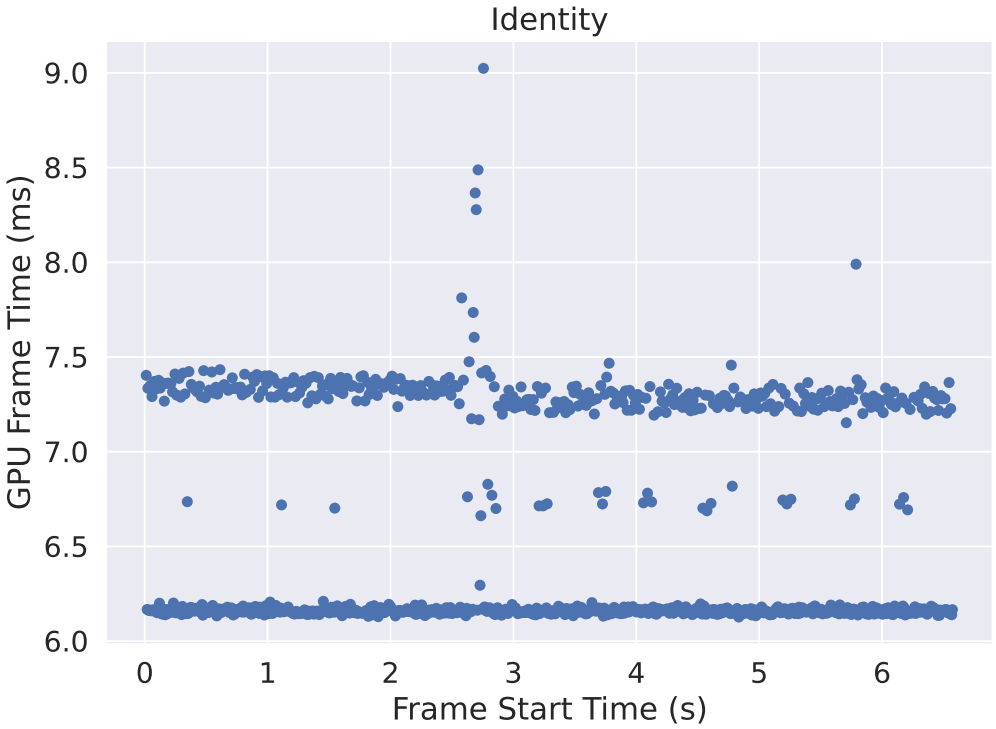

Figure 3.10: GPU frame times over application run time when rendering RayTracingInVulkan with the Identity swizzle. The interval displayed in the vertical axis does not start at 0.

Thread Swizzle	Change in Mean	Change in Median
Identity	+0.10%	+0.10%
Random	+25.26%	+28.20%
Numrays	-2.13%	-0.63%
Tiled Numrays	-3.75%	-2.61%
Objects	-0.17%	-0.76%
Tiled Objects	-3.65%	-2.53%

Table 3.2: Percent changes in mean and median GPU frame times for each thread swizzle.

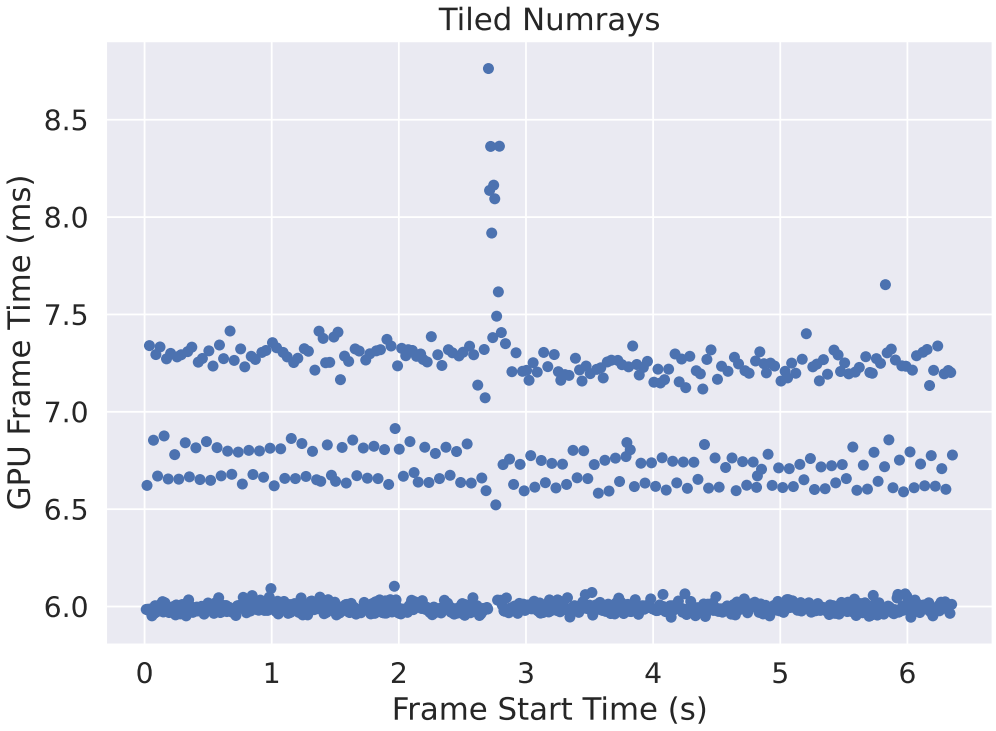

Figure 3.11: GPU frame times over application run time when rendering RayTracingInVulkan with the Tiled Numrays swizzle. The interval displayed in the vertical axis does not start at 0.

3.4.3 Analysis of GPU Frame Times

While SIMT efficiency is a significant factor in determining the performance of an application, it is not the only factor. To measure the performance of the application using each thread swizzle, we use NVIDIA Nsight Systems [38]

to measure GPU frame time. We set Nsight Systems to collect GPU metrics and Vulkan traces for 1000 frames of the application at a sampling rate of 10 kHz. The resulting profiling report measures the duration of 985 GPU frames from the application, a number which we found to be sufficient for consistently measuring the performance differences between the thread swizzles on repeated runs. When measuring GPU frame times, we lock the GPU clocks to the base frequency (1440 MHz) using NVIDIA's System Management Interface command line tool [39]. Figure 3.9 illustrates, using violin plots [20] overlaid with box plots, the distributions of GPU frame times when running the application with each thread swizzle. GPU frame times above or below a factor of 1.5 inter-quartile range are considered outliers and are shown as diamonds in the box plots. Circular markers in each box plot indicate the average GPU frame time achieved with the thread swizzle.

Looking at the violin plots in Figure 3.9, it appears that the distributions of GPU frame times are heavily right-skewed and multimodal. To observe how these multimodal distributions emerged from profiling data, Figure 3.10 plots GPU frame time over application run time using the Identity thread swizzle. To offer a better visualization of the differences, the vertical axis in Figure 3.10 and Figure 3.11 do not start at zero. GPU frame times take on values at distinct high and low "bands". While most GPU frame times take on values around the lower band of 6.16 ms, a significant portion of GPU frame times take on values around a high band of 7.30 ms. A few GPU frame times occur in between the two bands and above the high band. To show the effect of warp repacking on GPU frame times, Figure 3.11 plots GPU frame time over application run time using the Tiled Numrays thread swizzle. Again, there is a distinct high band and low band at which GPU frame times occur. Three observations can be made: 1. the lower band appears to have shifted down to around 5.98 ms; 2. there are noticeably fewer GPU frame times occurring around the high band; 3. more GPU frame times are occurring between the high and low bands. These observations are reflected in the shapes of the violin plots in Figure 3.9. We suspect that these behaviors are due to changes in microarchitectural state, but were unable to find any obvious strong

correlations between GPU frame times and statistics such as cache hit rates from NSight Systems profiling data. A more in-depth study may reveal how and why warp repacking is affecting the distribution of GPU frame times.

Regarding the overall performance achieved by the warp repacking methods, Table 3.2 lists the changes to mean and median GPU frame times achieved by applying the thread swizzles. One observation is that thread swizzling has negligible overhead because the Identity thread swizzle has increased mean and median GPU frame times by only 0.10% compared to the unmodified application. Another observation is that SIMT efficiency is not the primary factor that determines performance. Figure 3.8 showed that tiling did not significantly change the SIMT efficiencies of the resulting thread swizzles for the Numrays and Objects sorting methods. Yet, the tiled versions of the Numrays and Objects thread swizzles yield lower mean and median GPU frame times than their non-tiled counterparts. The violin plots in Figure 3.9 reveal that the Numrays and Objects swizzles also do not induce as many GPU frame times to fall between the high and low bands when compared to their tiled counterparts. The lower GPU frame times achieved when using thread swizzles created by tiled sorting methods may be attributed to better utilization of the memory hierarchy, as constraining the sorts within tiles ensures that the spatial locality of memory accesses is preserved. In essence, the tiling puts a limit on the maximum distance between two pixels a warp can be processing, as each thread is responsible for coloring one pixel in this application. Tiling ensures that the threads of each warp initiate ray tracing with similar primary ray origins and directions. The benefit of tiling is especially noticeable with the Objects thread swizzle, as it does not affect the mean GPU frame times as much as the Numrays, Tiled Numrays, and Tiled Objects sorts. The increase in variability of GPU frame times, illustrated by the inter-quartile ranges in the box plots, of the Objects thread swizzle compared to the Unmodified application and Identity thread swizzle may also be attributed to poorer utilization of the memory hierarchy.

Although GPU frame time results show that warp repacking can improve the performance of a ray-tracing application, the overall improvements to mean

and median GPU frame times are quite small. The largest reduction in the mean and median GPU frame times due to thread swizzles were of 3.75% and 2.61% respectively. This result is obtained with a warp repacking that overfits to a single frame by using information from an oracle and does not take into account the time spent creating the thread swizzles through thread sorting. This underwelming performance of warp repacking suggests that the default thread-warp mapping implemented by NVIDIA is efficient. However, Figure 3.9 suggests that there may be more benefits to warp repacking aside from overall performance: variability. Except for the non-tiled Objects sort, the thread swizzles significantly lower the variability of GPU frame times, as seen by comparing the sizes of the interquartile ranges of the box plots. The difference in GPU frame times between the high and low bands is 1 millisecond or more. Thus, exploring ways to reduce GPU frame time variability could significantly help real-time interactive applications with maintaining acceptable performance. Given that real-time interactive applications are expected to render at high frame rates, the allotted budget of time to render a frame is in the order of a few milliseconds. For example, Virtual-Reality devices are expected to hit target frame rates of at least 90 frames per second, which allows for a time budget of only 11.1 ms to render each frame. Shaving off even a fraction of a millisecond could allow for more headroom on the GPU to include additional visual fidelity and effects. However, further research is required to fully evaluate the effects of warp repacking on the distribution of GPU frame times to develop more concrete methods for reducing GPU frame time variability.

3.5 Efficiency and Performance Insights via Heat Maps

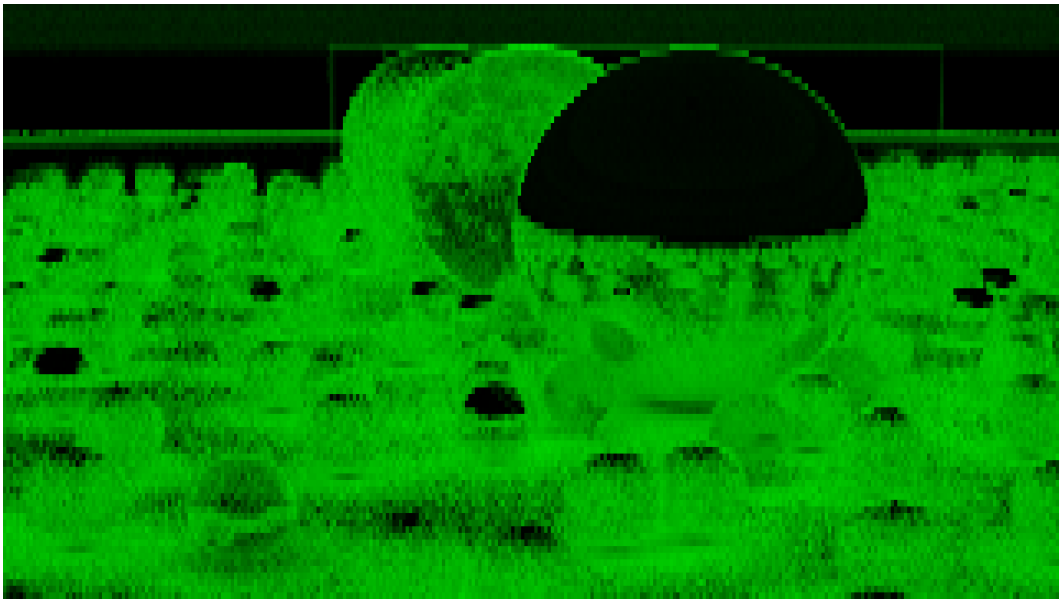

Figure 3.12: SIMT efficiency heatmap of RayTracingInVulkan. A lighter shade of color indicates higher thread divergence and therefore lower SIMT efficiency.

In addition to warp repacking, ReRay can be used to gain insights into the performance of ray-tracing applications through detailed heat map generation. A simple example is creating a heat map using the SIMT efficiency measure from Section 3.1.2. Such a SIMT efficiency heat map is illustrated in Figure 3.12 for the render in Figure 3.7. A heat map for SIMT efficiency may be used to assess the coherence of rays being traced across the screen for the given scene. If SIMT efficiency is a major concern for an application, the developers may choose to adopt a Wavefront ray tracer [26] in an attempt to maximize coherence at the cost of additional memory movement and consumption.

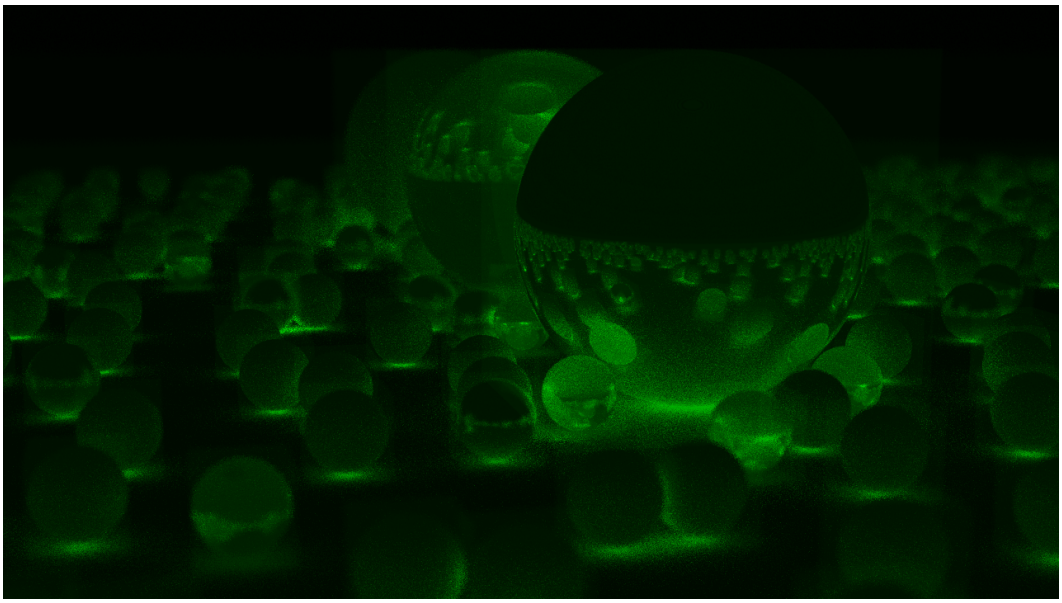

Figure 3.13: Heatmap of per-thread intersection shader call counts for RayTracingInVulkan. A lighter shade of color indicates a higher intersection shader call count.

The number of intersection shaders executed by a thread can be measured and visualized using ReRay via a heat map such as the one shown in Figure 3.13. A high number of intersection shader calls may be an indicator of *trapped rays* [42], where a thread traces a large number of rays that do not contribute much to the final result of the pixel's color While AMD's Radeon Raytracing Analyzer (RRA) [1] tool offers similar metrics via traversal counters, RRA is limited in that their counters are created via emulation of primary rays from the camera's viewpoint. ReRay, on the other hand, counts all intersection shader calls made by a thread, which includes primary rays, reflection rays, shadow rays, and all other rays traced by a thread. A shortcoming is that ReRay requires the application to enable intersection shaders for all its geometries to obtain accurate results.

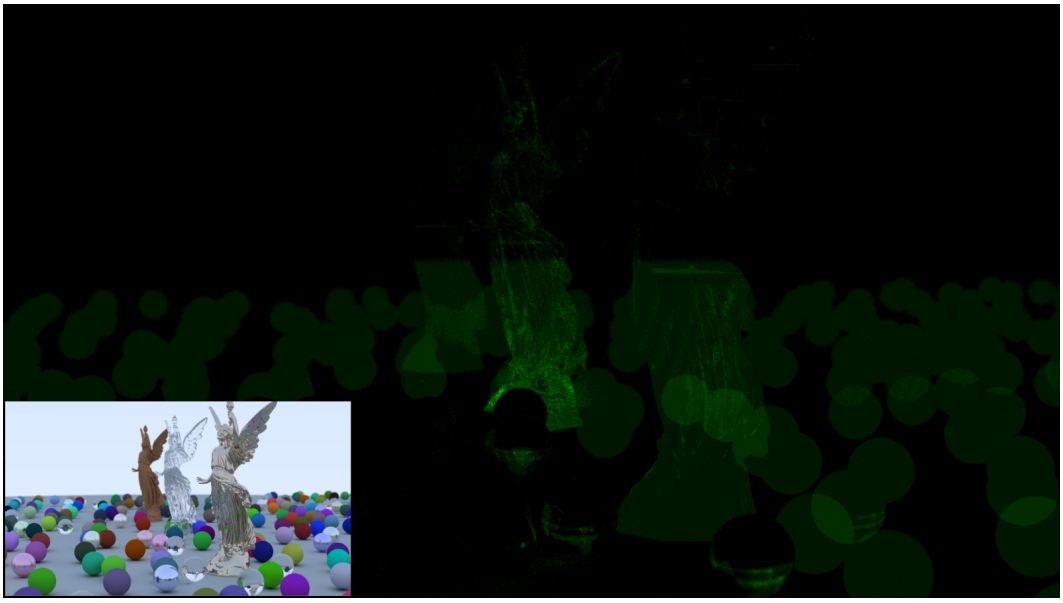

Figure 3.14: Heatmap of unnecessary intersection test counts for RayTracingInVulkan's Lucy In One Weekend scene. A lighter shade of color indicates a higher counts of unnecessary intersection tests. In the bottom left is the original image of the scene.

Another metric ReRay offers is the measure of *unnecessary intersection tests*. Unnecessary intersection tests are a consequence of the fact that the order in which intersections are found along a ray is unspecified [62]. To measure this metric, ReRay counts the number of any-hit shaders that would not need to be executed if ray traversal had processed intersections with objects in order of distance along the ray. While it would make more sense to count using the Report Intersection ray event mentioned in Section 3.1.1, the Vulkan API does not allow triangle meshes — geometry composed of only triangles — to have intersection shaders. Therefore, ReRay uses any-hit shaders as a proxy for intersection shaders to enable the counting of unnecessary intersection tests on triangle meshes. Figure 3.14 shows an example of a heat map of unnecessary intersection tests in the RayTracingInVulkan application after modifying the *Lucy In One Weekend* scene to have any-hit shaders on all geometry. This metric could be an indicator of a large number of overlapping AABBs in the acceleration structure, an indicator of poor subtree traversal ordering within an acceleration structure [32], or a consequence of the implementation of hardware

ray-traversal units in parallelizing ray-traversal work [27].

The metrics mentioned in this section illustrate the performance metrics that can be extracted from an application using the ray and warp data collected by ReRay's instrumentation layer and parsed by the ReRay toolchain front-end. Each of the metrics is implemented in its own module as shown in Figure 3.1. The ReRay toolchain is built to be simple and easy to extend — allowing the creation of new modules that can generate information to be used by developers or external tools.

3.6 Related Work

The tools most closely related to ReRay are Vulkan Vision [43] and RayScope [42], whose capabilities have been covered in Section 2.

When presenting Vulkan Vision, Pankratz et al. investigated the performance of Vulkan ray-tracing applications. However, they do not thoroughly explore the intricacies of ray behaviors in the ray-tracing pipeline; Vulkan Vision's insights are generalized over all rays traced by an application and does not concern itself with individual behaviors of threads and rays in the way that ReRay does.

In contrast to RayScope, ReRay is not a visualization tool but rather an extensible toolchain and workflow for capturing, parsing, and analysing application ray data, which allows for the study of ray-tracing performance at a deeper level than RayScope. ReRay collects additional ray-tracing pipeline events and geometry hit information that is not captured by RayScope. With minimal modification, RayScope can support ReRay's ray data format to create interactive ray visualizations, but going in the other direction is impossible due to missing information in RayScope's ray data that is required by some of ReRay's tools. This additional information is used by ReRay to provide insights into aspects affecting ray-tracing performance that do not necessarily manifest visually in the application or with a ray visualizer such as RayScope. For example, RayScope does not record intersection-shader call events, while ReRay does. RayScope's `Intersection` event is equivalent to ReRay's `Re-`

`portIntersection` event, which records calls to *OpReportIntersectionKHR*. However, by making an explicit distinction between an `Intersection` shader call and a `ReportIntersection` function call, one can examine the proportion of rays that hit versus miss an object contained within an AABB. This distinction can help determine whether an object's AABB is too large or examine the effectiveness of employing spatial splits [32] to subdivide the AABB. ReRay also records the *PrimitiveID* with Closest Hit, Any Hit, and Intersection events, while RayScope does not; this detailed recording makes ReRay more precise when examining objects hit by rays, as the *InstanceID* alone does not uniquely identify objects because more than one geometric primitive can share the same instance ID. Finally, ReRay optimizes the data collection process by writing binary files instead of storing the information in plain text, leading to an order of magnitude speedup in the capture of, reading, and processing of execution trace data.

The thread swizzling technique presented in this chapter is a similar to thread-group ID swizzling presented in the 2019 Game Developer's Conference and published in an NVIDIA blog post [6]. Thread-group ID swizzling is aimed at optimizing cache locality in compute shaders by remapping warp IDs. This is necessary because, unlike ray-tracing shaders, warps in compute shader dispatches are launched in row-major order instead of a Z-order like that described in Section 3.2.2. Therefore, developers need to implement their own tiling and space-filling curves to increase memory locality in compute shaders used for graphics. In contrast to thread-group ID swizzling, ReRay's thread swizzling works at thread ID granularity instead of warp ID granularity. This allows finer-grained control over how thread work is distributed over warps, which can be used to further optimize not only memory locality but also SIMT efficiency.

A proprietary tool for analyzing ray-tracing performance was recently announced by AMD, called the Radeon Raytracing Analyzer (RRA) [1]. RRA is primarily an acceleration structure visualization tool, but it can also generate heat maps of ray-traversal counters that are similar to intersection and any-hit shader count heat maps that can be generated using ReRay. These traversal

counters count the number of intersection tests and object hits along rays. RRA generates traversal counters through emulation by tracing rays from the camera to the closest hit. However, RRA's emulation is limited in that it does not account for the complete behavior of the application. An application can spawn more rays after the closest hit, which RRA does not account for. Furthermore, RRA does not give accurate traversal counters for procedural geometries, as it can not know the topology of procedural geometries from the acceleration structure alone. ReRay, on the other hand, does account for all rays traced by an application for a given frame, and also gives accurate counts for procedural geometries because ReRay does not perform emulation and retrieves its numbers from instrumentation of the running application. ReRay also does not require the acceleration structure to generate intersection and any-hit shader count heat maps. However, ReRay does require the application to enable intersection and any-hit shaders on all geometries to give accurate counts, but this process could be automated in future work.

Other tools exist for debugging and analyzing ray-tracing applications through the collection of ray data [14], [28], [50], [51]. However, these require extensive modifications to application source code to collect ray data and are designed for ray visualization or for the study of light propagation throughout a scene. Although a ray-visualization tool can be made to work with the ReRay toolchain, it is not the main focus of this chapter. Furthermore, the ray data collected by ReRay lacks key information required to fully study light propagation. ReRay instead focuses on capturing and studying interactions within the Vulkan ray-tracing pipeline, as well as the distribution of work across threads and warps on the hardware. ReRay uses this information to produce performance metrics and heat maps that reveal specifics about hardware execution and driver implementations of data structures and algorithms used for ray tracing in Vulkan.

While there are a number of existing debuggers and profilers for applications using Vulkan or other graphics APIs [22], [33], [41], they mainly operate by measuring the timing and frequency of graphics API calls. These tools also monitor hardware performance counters to aid in reasoning about performance

bottlenecks. ReRay is complimentary to these types of tools and focuses on providing a method to deeply analyze the performance of `traceRay` calls in Vulkan, which invoke the ray-tracing pipeline.

Finally, while this chapter shows that SIMT efficiency and ray-tracing performance can be improved through reassignment of threads to warps, other more sophisticated and effective methods have been devised for this goal. For instance, algorithmic changes can exploit the types and frequency of materials and textures in specific games and scenes [15], [23], alternative formulations of the ray-tracing algorithm for better hardware utilization at the cost of additional memory usage [26], [65], ray-sorting algorithms [23], [31], the addition of ray-intersection prediction hardware [29], and hardware shader execution reordering [40]. However, ReRay could be extended and used to evaluate the effectiveness of these methods in a similar way to how it evaluates thread swizzling. In particular, shader execution reordering is a promising avenue for further study with ReRay and is further discussed in the next section.

3.7 Limitations and Future Work

ReRay is a powerful tool for analyzing and optimizing Vulkan ray-tracing applications, but it has limitations that should be addressed in future work. Furthermore, there are avenues for exploring new features and improvements to the toolchain that would enhance its utility for developers and researchers.

3.7.1 Benchmark Selection and Availability

The evaluation of ReRay's warp repacking methods used a single benchmark — RayTracingInVulkan [13] — to measure the performance for different sorting methods. The benchmark was chosen due to its simplicity and ease of modification to implement thread swizzling. However, this benchmark is not representative of the workloads that developers are interested in optimizing. In comparison with other ray-tracing applications, *Ray Tracing In One Weekend* uses a handful of simple shaders. The scenes from RayTracingInVulkan used for the evaluation are composed of a few hundred textureless objects,

which are simple in comparison to more complex scenes found in video games and other applications which may consist of thousands of objects with more materials and textures that put more strain on the memory hierarchy of the GPU.

More extensive studies with ReRay should use a wider variety of ray-tracing benchmarks to evaluate the performance of any proposed optimization method. At the time of ReRay's inception, the number of ray-tracing applications available was limited, and even more limited when constrained to considering applications specifically utilizing the Vulkan ray-tracing extension. Although the number of available ray-tracing applications has steadily been increasing as ray tracing becomes more popular in the gaming industry and hardware support becomes more widespread, the vast majority of ray-tracing applications are built using the competing DirectX 12 graphics API. To circumvent this issue, the Vulkan Vision framework could be extended to work with the VKD3D-Proton [17] Vulkan implementation of DirectX 12 to enable the analysis of ray-tracing workloads from a wider variety of applications built using the competing DirectX 12 API.

3.7.2 Unnecessary Intersection Test Classification

The current implementation of ReRay reports unnecessary intersection tests as intersection tests that would not have occurred if intersections with objects along a ray had been processed in order of increasing distance along the ray. These unnecessary intersection tests may be further classified as occurring due to three reasons: 1. **(R1)** the acceleration structure, as defined by the application, contains geometry whose AABBs overlap; 2. **(R2)** quantization [32] and other techniques performed by the device or driver to optimize acceleration structures may cause rays to intersect with AABBs that would not have intersected with the geometry's original AABB definitions [62]; or 3. **(R3)** hardware ray-traversal units may not process ray intersections in order of distance along the ray due to the construction of the acceleration structure and implementations of the ray-traversal algorithm to take advantage of parallel processing [27], [32].

The current implementation of ReRay does not provide methods to determine the cause of the unnecessary intersection tests. Future work could extend ReRay to provide a way to classify the causes of unnecessary intersection tests.

One obstacle to the classification of unnecessary intersection tests is that the information about the acceleration structure produced by the device or driver is opaque to the application. A possible way to overcome this obstacle is to create a reference acceleration structure using the application's original definitions of AABBs and simulate execution of the ray-traversal process using this reference acceleration structure. The simulated ray-traversal process must be an ideal ray-traversal process that processes ray intersections in order of distance along the ray. Differences between the simulated ray-traversal process and the actual ray-traversal that occurred during the ray-tracing pipeline's execution could be used to infer information about the opaque acceleration structure and partially classify causes of unnecessary intersection tests:

1. Any unnecessary intersection tests caused by overlapping AABBs from the original application's AABB definitions **(R1)** will be present in both the simulated ray-traversal process and the actual ray-traversal process.

2. Any unnecessary intersection tests caused by quantization and other techniques performed by the device or driver to optimize acceleration structures **(R2)** would only be present in the actual ray-traversal process and not in the simulated ray-traversal process. This is because the AABBs in the opaque acceleration structure must be greater or equal to the original size of the AABBs [62].

3. Any unnecessary intersection tests caused by hardware ray-traversal units not processing ray intersections in order of distance along the ray **(R3)** would only be present in the actual ray-traversal process and not in the simulated ray-traversal process.

As described, this technique would only be able to classify unnecessary intersection tests as belonging to **R1** or to the set {**R2**,**R3**}. Another limitation of this proposed technique is that it only works for procedural geometries.

The AABBs of triangle meshes are not defined by the application but are instead generated by the driver during acceleration-structure construction. This technique may not be useful to most ray-tracing applications because such applications use triangle meshes.

3.7.3 Hardware Warp Repacking Support

The current implementation of ReRay does not support hardware warp repacking during ray-tracing-pipeline execution. This is a limitation of the current implementation of Vulkan Vision; Vulkan Vision determines the thread-to-warp assignment of a ray-tracing pipeline just once at the beginning of execution of the ray-generation shader through the use of subgroup (i.e., warp-wide) operations and an atomic counter [43]. This method has two problems: 1. **(P1)** it does not consider the possibility of the assignment of threads-to-warps changing during the execution of the rest of the ray-tracing pipeline; and 2. **(P2)** it relies upon the assumption that all threads in a warp are active when beginning execution of the ray-generation shader, as there is only one such shader in a ray-tracing pipeline.

The Vulkan specification [62] defines *invocation repack instructions* as instructions that may alter the set of threads that are executing and change the current thread-to-warp assignment. The `OpTraceRayKHR` instruction is one such invocation repack instruction that is executed in the ray-generation shader to begin ray tracing [62]. Hence, if the hardware performs warp repacking during ray tracing, the thread-to-warp assignment may change during the execution of the ray-tracing pipeline, and the thread-to-warp assignment recorded by Vulkan Vision would no longer be correct due to **P1**.

To support hardware warp repacking, Vulkan Vision must be extended to capture the new thread-to-warp assignment at each invocation repack instruction and to update the thread-to-warp assignment accordingly. This is another complex problem to solve because the thread-to-warp assignment may change during the execution of `OpTraceRayKHR`, which implies that the thread-to-warp assignment may change when executing other shaders in the ray-tracing pipeline (Intersection, Any-Hit, Closest-Hit, and Miss). Due to

P2, the method of determining thread-to-warp assignments using Vulkan subgroup operations and an atomic counter can no longer be applied because the assumption is broken when considering other shader types in the ray-tracing pipeline, where only a subset of threads in a warp may be active due to each thread tracing a different ray and therefore intersecting different objects requiring different shaders to be executed. Moreover, due to hardware features such as NVIDIA's Independent Thread Scheduling, not all threads in the same warp may return from execution of the `OpTraceRayKHR` instruction at the same time, yet again breaking the assumption in **P2**. This creates a problem where the exact thread-to-warp assignment during execution is known only at the beginning of the ray-generation shader and nowhere else in the ray-tracing pipeline.

The most straight-forward solution to this problem is to use a proprietary extension to capture the thread-to-warp assignment information directly from the driver. Another solution is to propose a modification to the Vulkan specification to allow usage of the existing built-in variable `SubgroupId` within ray-tracing shaders, which would enable determining of the thread-to-warp assignment without the need for a proprietary extension.

3.7.4 Shader Execution Reordering

A recent technology introduced by NVIDIA, called Shader Execution Reordering (SER) [40], enables developers to perform explicit hardware warp repacking on NVIDIA GPUs. At a high level, SER simply replaces the `OpTraceRayKHR` instruction with three new instructions: `OpHitObjectTraceRayNV`, `OpReorderThreadWithHitObjectNV`, and `OpHitObjectExecuteShaderNV`. `OpHitObjectTraceRayNV` traces a ray, executing Any-Hit and Intersection shaders to find the closest hit, and records the hit information into an object called a *HitObject*. `OpReorderThreadWithHitObjectNV` performs hardware warp repacking based on the information contained within the HitObject and an additional optional bit vector argument containing hint bits that the developer is free to define. `OpHitObjectExecuteShaderNV` executes the Closest-Hit and/or Miss shader for the corresponding HitObject.

The `OpReorderThreadWithHitObjectNV` instruction is the most interesting of the three instructions because it allows developers to provide *coherence hints* in the form of a bit vector to guide the hardware into creating more *coherent warps* during hardware warp repacking. A *coherent warp* in this context means that all the threads in the warp execute similar shaders and access similar locations in memory. The order of priority when repacking threads into warps is stated in the SER white paper [40] to be:

1. Shader ID of the Closest-Hit or Miss shader to be executed for the HitObject
2. Coherence hint bits, from the most significant to least significant bits
3. Spatial location of the intersection with the HitObject

ReRay provides opportunities to study the effects that SER could have on an application's ray-tracing-pipeline-execution trace. Notably, these studies could be performed without having access to a GPU that supports SER. Since SER is currently only available on NVIDIA GPUs with the Ada Lovelace architecture released late in 2022, SER is still a novel feature that not all developers may have access to.

One application for ReRay with regards to SER is guiding the creation of coherence hints for the reorder-thread function. Shader-code instrumentation for ReRay could be extended to capture information such as branch patterns and memory access patterns in the execution of Closest-Hit and Miss shaders in the ray-tracing pipeline. Correlation between the properties of objects that are hit, and the branch patterns and memory access patterns of the shaders that are executed could be used to generate coherence hints based on the properties of the objects.

Another application for ReRay with regards to SER is to examine the distribution of Closest-Hit and Miss shader calls from the execution trace of a ray-tracing pipeline and then to estimate the maximum benefits to SIMT efficiency that is possible by using SER to perform warp repacking. To obtain this estimate, a new module similar to ReRay's Replayer module from

```
// Warp 0
0. begin, int1, miss,          begin, int1, repint, chit
1. begin, int1, repint, chit,  begin, int1, miss
2. begin, int2, repint, chit,  begin, int2, repint, chit
3. begin, int2, repint, chit,  begin, int2, repint, chit

// Warp 1
4. begin, int1, repint, chit,  begin, int1, miss
5. begin, int1, miss,          begin, int1, repint, chit
6. begin, int2, miss,          begin, int2, miss
7. begin, int2, miss,          begin, int2, miss
```

Listing 3.2: Example of ray events per thread from two warps each consisting of four threads that each trace two rays. There is one ray generation shader (begin), one closest hit shader (chit), one miss shader, and two intersection shaders (int1, and int2).

Section 3.2.1 could be created to simulate the operation of SER. Such a module would break down existing per-thread-ray-event traces into multiple traces such that each trace contains ray events for one ray from each thread up to, but not including, Closest-Hit and Miss shader events. The Closest-Hit and Miss shader events are instead processed at the beginning of the next trace with warp repacking performed to make coherent warps for Closest-Hit and Miss shader execution. To illustrate, consider a per-thread trace of ray events shown in Listing 3.2 consisting of two warps, with each warp composed of four threads and each thread tracing two rays. A module for simulating SER could produce a set of three new ray-event traces as shown in Listing 3.3. Trace A shows the initial per-thread ray events for generating the primary rays, up to, but not including, Closest-Hit and Miss shader events. The Closest-Hit and Miss shader events are enclosed in round brackets to represent that they are queued for execution after warp repacking, and are not actually a part of the execution trace. Trace B is created by performing warp repacking on trace A using a simulation of SER. In this case, the warps can be repacked such that all Closest-Hit shader executions are packed into warp 0 and all Miss shader executions are packed into warp 1. In more complex traces, involving larger numbers of warps, the warp repacking would use developer-provided coherence hints (if available) and the spatial locations of the ray events (not shown in the trace for brevity) to create more coherent warps. Analogously to trace

```
========== PER-THREAD-RAY-EVENT TRACE A ==========

// Warp 0
0. begin, int1, (miss)
1. begin, int1, repint, (chit)
2. begin, int2, repint, (chit)
3. begin, int2, repint, (chit)

// Warp 1
4. begin, int1, repint, (chit)
5. begin, int1, (miss)
6. begin, int2, (miss)
7. begin, int2, (miss)

========== PER-THREAD-RAY-EVENT TRACE B ==========

// Warp 0
0. miss,  begin, int1, repint, (chit)
5. miss,  begin, int1, repint, (chit)
6. miss,  begin, int2, (miss)
7. miss,  begin, int2, (miss)

// Warp 1
4. chit,  begin, int1, (miss)
1. chit,  begin, int1, (miss)
2. chit,  begin, int2, repint, (chit)
3. chit,  begin, int2, repint, (chit)

========== PER-THREAD-RAY-EVENT TRACE C ==========

// Warp 0
4. miss,  begin
1. miss,  begin
6. miss,  begin
7. miss,  begin

// Warp 1
0. chit,  begin
5. chit,  begin
2. chit,  begin
3. chit,  begin
```

Listing 3.3: Per-thread-ray-event traces from Listing 3.2 broken into three new per-thread-ray-tracing-event traces to simulate the operation of SER. Ray events in enclosed in round brackets signify that the event is queued for execution after SER performs warp repacking, and is executed at the beginning of the next ray-tracing-event trace.

B, trace C is created by performing warp repacking on trace B, again using a simulation of SER. The per-thread-ray events in Trace C all end with a `begin` event to represent execution returning to the Ray-Generation shader after the Closest-Hit and Miss shaders have completed execution.

Some limitations and considerations when implementing a module for simulating SER are:

1. There is no guarantee that SER performs warp repacking across all threads in such a synchronized manner as shown in the example traces from Listing 3.3. Due to Independent Thread Scheduling, threads may return from ray traversal (`OpHitObjectTraceRayNV`) at different times, and the GPU may perform warp repacking with whatever set of threads are currently available at the time. The repacked warps could then begin Closest-Hit and Miss shader execution before other threads have returned from ray traversal;

2. The details of how spatial location is factored into warp repacking are not provided in the SER white paper [40]. There are multiple different ways to produce sorting keys from spatial locations [31], and the method employed by SER for warp repacking is unknown.

3. The costs of warp repacking, such as the time taken to produce sorting keys, cluster threads into coherent warps, and migrate live registers of threads between warps and across Streaming Multiprocessors, are unknown. These costs may be significant and may outweigh the benefits of improved warp coherence, especially if the shaders executed by the threads are trivial to execute [40]

4. Providing developer-defined coherence hints to the ReRay module for simulating [40] would require an easy-to-use mechanism. Such a mechanism may come in the form of an intrinsic instruction that developers may insert into their Closest-Hit and Miss shaders to provide coherence hints for ReRay's Vulkan Vision shader-code instrumentation to capture and record alongside the ray-tracing-pipeline-execution trace. These in-

trinsic instructions would have no effect on the actual execution of the shader and would only be used by ReRay to record the coherence hints

In addition to estimating the effects of SER on an application's ray-tracing-pipeline-execution trace, ReRay could also be a powerful tool for examine the thread-to-warp assignments created by SER after `OpReorderThreadWithHitObjectNV`. The current implementation of ReRay does not support capturing ray-tracing-pipeline-execution traces when hardware warp repacking is performed. To support SER, ReRay's Vulkan Vision shader-code instrumentation could be extended to capture the new thread-to-warp assignments after each `OpReorderThreadWithHitObjectNV` instruction. However, there is still the issue of Independent Thread Scheduling that may cause threads in the same warp to return from `OpHitObjectTraceRayNV`, and consequently `OpReorderThreadWithHitObjectNV`, at different times. Therefore, Vulkan Vision's method of determining the thread-to-warp assignment cannot be used. A proprietary extension may be used instead to determine the thread-to-warp assignment directly from the driver within each shader. However, as will be seen in the next section, such an extension comes with its own set of problems in addition to no longer being platform-agnostic.

3.7.5 Shader Execution Reordering Preliminary Performance Study

With Shader Execution Reordering (SER) being a novel technology, there is a lack of information about how SER affects the performance of ray-tracing applications. Although ReRay can be used to simulate the operation of SER and estimate the benefits of improved warp coherence, it is also important to understand how SER affects the performance of ray-tracing applications in practice.

As a preliminary study, the RayTracingInVulkan [13] application is modified to use SER. The modification replaces the `OpTraceRayKHR` instruction with the three new instructions provided by SER. No coherence hints were provided to the `OpReorderThreadWithHitObjectNV` instruction. All applica-

tion settings are at their default values, except that ray accumulation is turned off to ensure that the rays being traced per frame are the same to enable a performance comparison of the application with and without SER. The application's present mode is also set to Immediate mode to allow it to render frames as quickly as possible without being stalled by the refresh rate of the display. This preliminary study is performed on an NVIDIA GeForce RTX 4060 Laptop GPU.

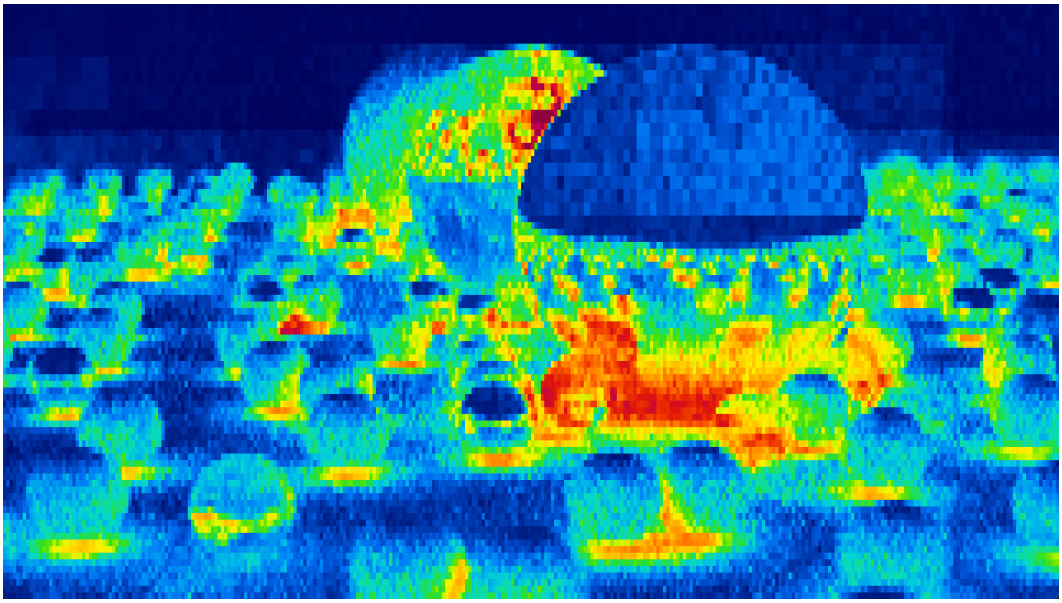

Figure 3.15: Shader-timing heat map for the RayTracingInVulkan application on the *Ray Tracing In One Weekend* scene.

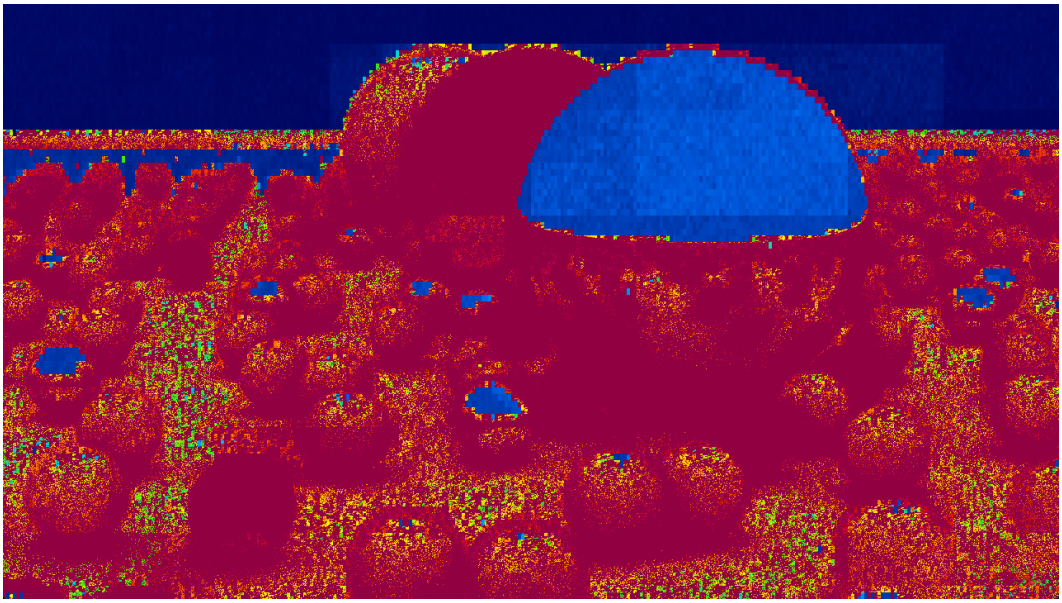

Figure 3.16: Shader-timing heatmap for the RayTracingInVulkan application on the *Ray Tracing In One Weekend* scene with Shader Execution Reordering enabled.

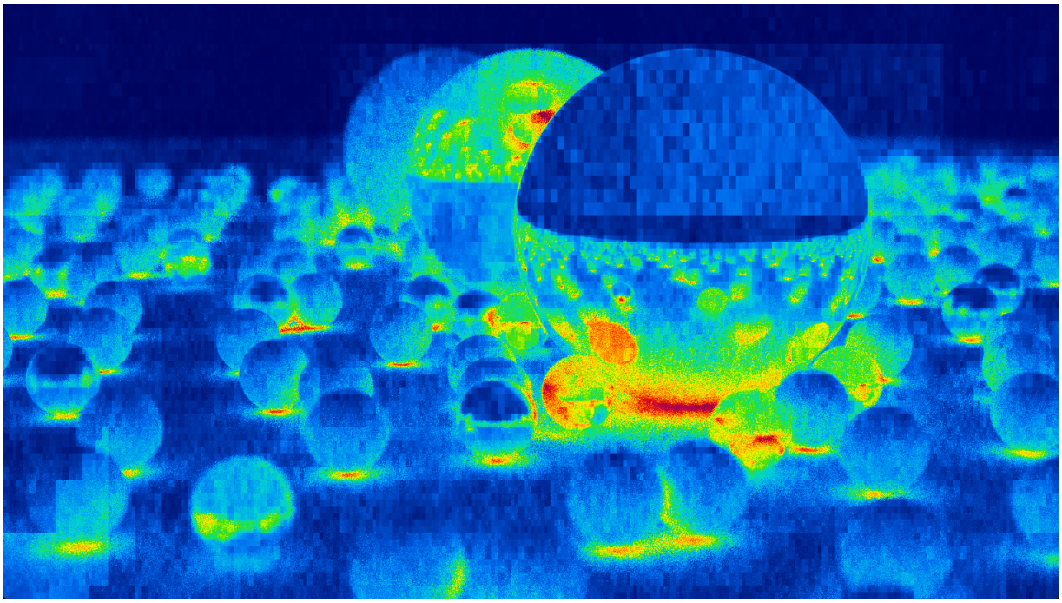

Figure 3.17: Shader-timing heat map for the RayTracingInVulkan application on the *Ray Tracing In One Weekend* scene with Shader Execution Reordering but with the `OpReorderThreadWithHitObjectNV` instruction omitted to disable warp repacking.

RayTracingInVulkan has a built-in shader-timing heat map that enables the visualization of the time it takes for each pixel to be computed. The effect of SER in the rendering of the *Ray Tracing In One Weekend* scene from Figure 3.7 can me observed using this heat map. Figure 3.15 shows a screenshot of the shader-timing heat map of the unmodified application (**Baseline**). Figure 3.16 shows a screenshot of the shader-timing heat map of the application with the described modification to enable SER (**SER-Enabled**). Figure 3.17 shows a screenshot of the shader-timing heat map of the application with SER enabled, but the `OpReorderThreadWithHitObjectNV` instruction is omitted to disable warp repacking (**SER-NoRepack**). The same default heat map scale value of 1.5 is used for all the figures to allow for comparison.

In addition to shader-timing heat maps, the application includes a benchmark mode that measures the average frame rate, in Frames Per Second (FPS), over a period of one minute. According to the benchmark, **Baseline** has an average frame rate of 85.9 FPS. **SER-Enabled** has an average frame rate of 60.7 FPS. **SER-NoRepack** has an average frame rate of 74.8 FPS.

SER-Enabled has a 29.3% decrease in average frame rate compared to **Baseline**. This difference is also reflected in the shader-timing heat maps, where much of the shader-timing heat map of **SER-Enabled** (Figure 3.16) is hot where **Baseline** (Figure 3.15) was cold. The performance decrease may be due to the simplicity of the shaders in the application and the costs associated with hardware warp repacking. The shaders used by RayTracingInVulkan are not very costly to execute and the scene is lacking in object count, variety of materials, and use of textures as found in more complex ray-tracing applications. A more complex ray-tracing application with more costly shaders and complex scenes may take more advantage of SER to obtain a performance increase. For instance, the Unreal Engine 5 game engine reportedly obtained a 20-50% increase in performance due to a simple modification to enable SER [37] — one that is analagous to the modification described for the RayTracingInVulkan application in this preliminary study.

While it is clear that SER does not improve performance for the RayTracingInVulkan application, it is not clear whether the performance decrease

is entirely due to the overheads of warp repacking and lack of shader and scene complexity. As reported, **SER-NoRepack** has an average frame rate of 74.8 FPS, which is still an 18.9% decrease in average frame rate compared to **Baseline**. The shader-timing heat map for **SER-NoRepack** (Figure 3.17) also does not appear to have increased the amount of hot pixels compared to **Baseline** (Figure 3.15). Peculiarly, **SER-NoRepack**'s shader-timing heat map appears to have less hot pixels overall, indicating that many of the pixels in the image are taking less time to compute than in the **Baseline**. Perhaps the most plausible explanation for the performance decrease from **SER-NoRepack** is due to Independent Thread Scheduling, where threads in the same warp are no longer executing in lock-step. Evidence for Independent Thread Scheduling is visible when comparing the shader-timing heat maps at a per-warp level. Warps are composed of threads computing pixels in 8x4 rectangles (see Section 3.2.2) across the frame. In **Baseline**, all threads in the same warp are colored the same, as shown by all pixels in 8x4 rectangles receiving the same color. This coloring indicates that all the threads in the warp are taking the same amount of time to compute, and therefore suggests execution is occurring in lock-step. Meanwhile, the shader-timing heat map for **SER-NoRepack** shows individual pixels colored as opposed to entire warps, indicating that threads in the same warp are taking different amounts of time to complete. These heat maps suggest that the threads in the warp are not executing in lock-step—a distinct characteristic of Independent Thread Scheduling. Independent Thread Scheduling may cause a performance decrease because the frame rate is determined by the pixel that takes the longest time to compute. When Independent Thread Scheduling is enabled, individual threads may take longer to compute than when threads are executing in lock-step because a warp needs to be scheduled to execute for multiple Program Counters (one for each thread) as opposed to a warp executing with a single Program Counter for all threads in lock-step.

Another area of interest with SER is the thread-to-warp assignments created after warp repacking, and the distribution of warps to Streaming Multiprocessors (SMs) across the NVIDIA GPU. To examine these areas of interest,

a proprietary extension, `VK_NV_shader_sm_built_ins`, provides volatile built-in variables to shaders that enable querying the current thread's warp ID and SM ID. Unlike Vulkan Vision's warp IDs, `VK_NV_shader_sm_built_ins`'s warp IDs are not unique across the entire GPU but are unique within an SM. The NVIDIA GeForce RTX 4060 Laptop GPU used in this study has 24 SMs.

Some questions that could be answered with this information are:

1. **(Q1)** Is SER actually performing warp repacking?
2. **(Q2)** Does SER repack warps across SMs?

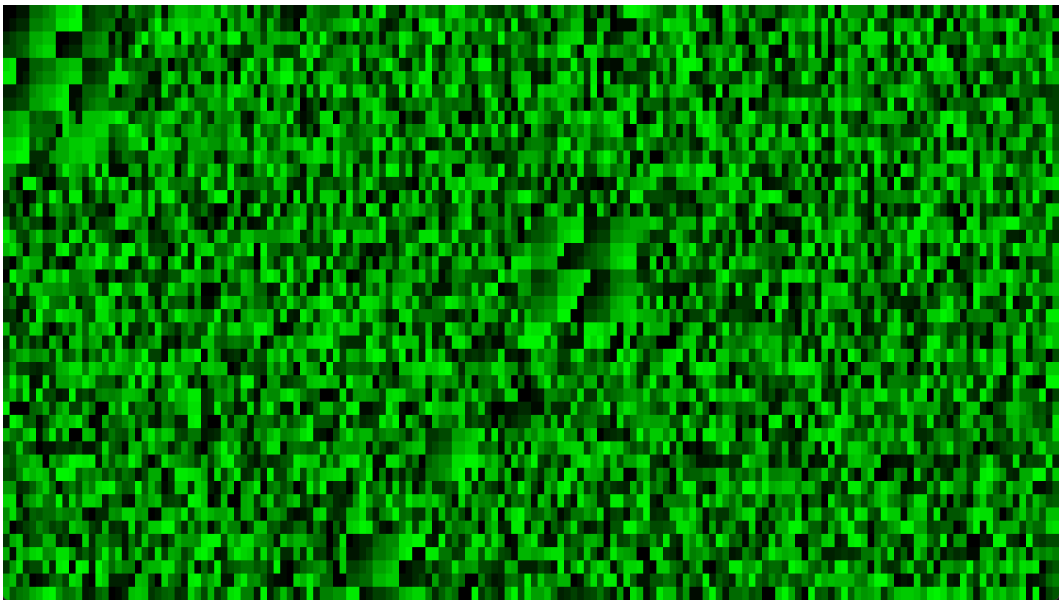

Figure 3.18: Visualization of SM ID assignment for each pixel in the RayTracingInVulkan application. Each pixel color is one of 24 shades of green corresponding to an SM ID, from black (SM ID 0) to bright green (SM ID 23).

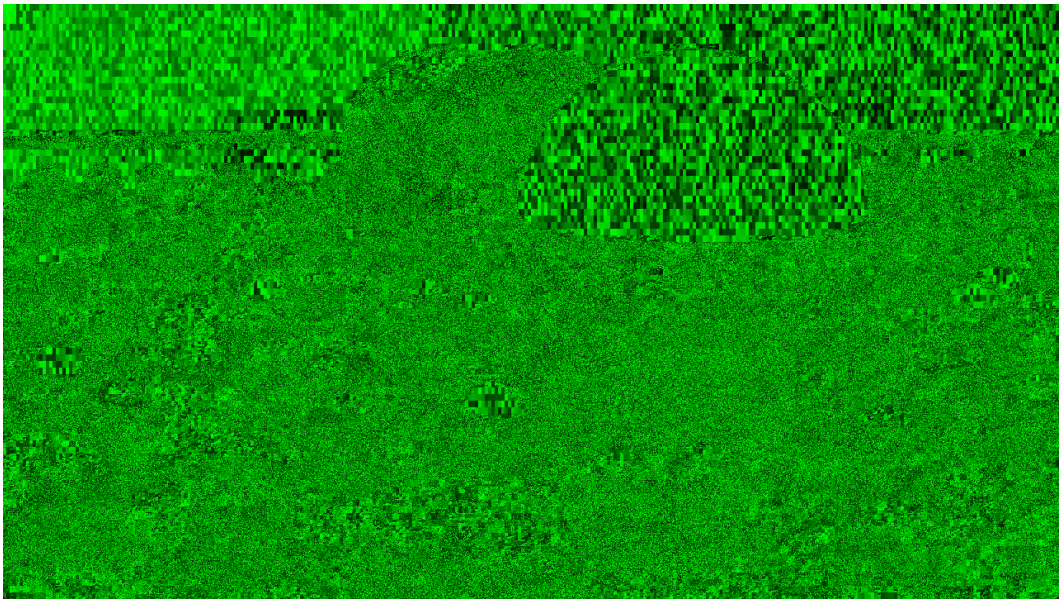

Figure 3.19: Visualization of SM ID assignment for each pixel in the RayTracingInVulkan application with SER enabled. Each pixel color is one of 24 shades of green corresponding to an SM ID, from black (SM ID 0) to bright green (SM ID 23).

Answering Q1 is important to confirm that the modification to enable SER in RayTracingInVulkan is actually working, and that SER indeed does what NVIDIA claims that it does. To confirm that SER is performing warp repacking, the Ray-Generation shader of RayTracingInVulkan is modified to color pixels corresponding to the SM ID reported by each thread. The color of each pixel is one of 24 shades of green, each shade corresponding to an SM ID, with brighter shades corresponding to a higher SM ID number. The modification is performed after ray tracing has completed, and the code overwriting the pixel color is enclosed by a conditional statement to ensure that the rest of the Ray-Generation shader is not eliminated by the shader compilers' dead-code elimination passes. Figure 3.18 shows the SM ID assignment for each pixel in the RayTracingInVulkan application without SER (**Baseline**). Figure 3.19 shows the SM ID assignment for each pixel in the RayTracingInVulkan application with SER enabled (**SER-Enabled**). If SER is enabled but the `OpReorderThreadWithHitObjectNV` instruction is omitted to disable warp repacking (**SER-NoRepack**), the SM ID assignment appears to be the same

as Figure 3.18, which confirms that warp repacking was indeed not performed when examining **SER-NoRepack** from the performance test, and that the performance decrease was due to Independent Thread Scheduling.

As can be seen in Figure 3.18, each SM operates on 8x4 rectangles of pixels scattered across different regions of the frame, which confirms that warps are not being repacked and simply retain their thread-to-warp assignments from the beginning of the Ray-Generation shader to the end. In contrast, Figure 3.19 shows that SMs are no longer constrained to operating on warps composed of pixels in 8x4 rectangles, which is evidence that warp repacking is being performed to create more coherent warps that no longer conform to the original thread-to-warp assignments.

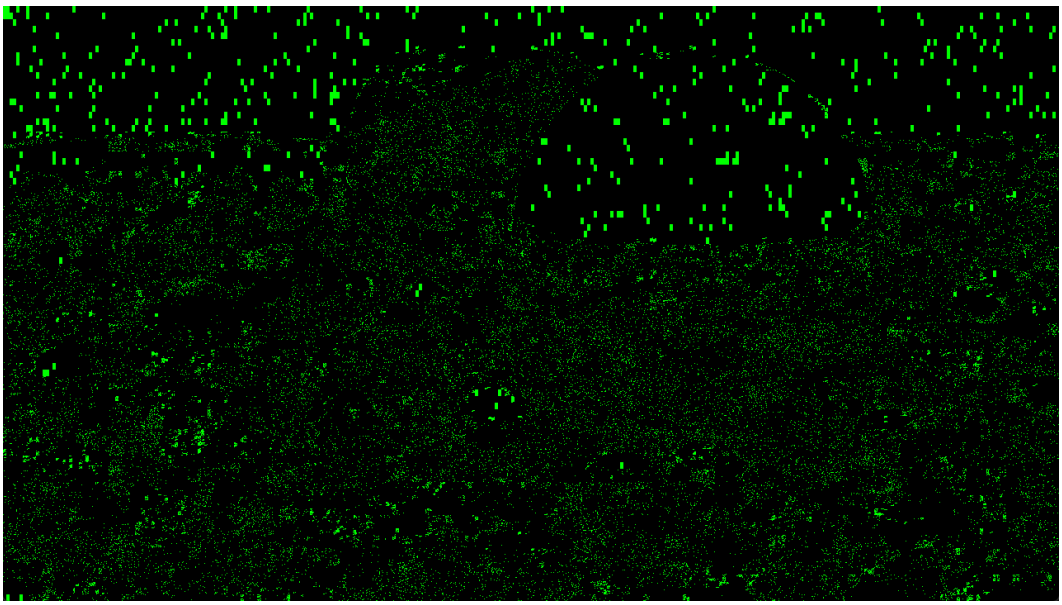

Figure 3.20: Visualization of the workload for SM ID 23 in the RayTracingInVulkan application with SER enabled.

Although it is not visually obvious in Figure 3.19, SMs 0 and 1 are not processing any pixels in the **SER-Enabled** configuration. Figure 3.20 highlights all pixels that are processed by the SM 23 when SER is enabled, with all other pixels being colored black. Creating similar figures for SMs 0 and 1 would show that no pixels are being processed by these SMs when SER is enabled. Furthermore, if SER is enabled but with the `OpReorderThread-`

`WithHitObjectNV` instruction omitted to disable warp repacking, SMs 0 and 1 continue processing pixels as would be expected with SER disabled. These empirical observations suggests that SMs 0 and 1 may be responsible for performing warp repacking. Therefore, when SER is enabled, not only are there the overheads of warp repacking to consider, but also two SMs are no longer processing pixels, which may further contribute to the performance decrease observed in the performance test.

To answer Q2, the Ray-Generation shader of RayTracingInVulkan with SER enabled is modified to record the SM ID of the SM each thread is assigned to, and output a green color if the SM ID has changed at the end of the Ray-Generation shader. Otherwise, the pixel is colored black. Unexpectedly, this modification appears to disable SER. Simply the act of reading and storing SM ID into a local variable at the beginning of the Ray-Generation shader is enough to disable SER, even when the local variable holding the SM ID is unused. While it appears that SMs 0 and 1 still do not process any pixels, the rest of the SMs process pixels in roughly the same manner as the application without SER enabled. This unexpected interaction between the `VK_NV_shader_sm_built_ins` extension and SER makes it impossible to answer Q2 or other questions regarding SER behavior.

3.8 Conclusion

ReRay is a toolchain and a workflow for assessing, debugging, and optimizing the performance of Vulkan ray-tracing applications using ray-trace execution data automatically captured through instrumentation. ReRay does not require hand modification of application source code, making it a readily available and easily applicable tool for any standard-conforming Vulkan ray-tracing application. Using ReRay, we have shown that black-box hardware- and driver-level implementation behaviors such as warp execution order and thread-to-warp assignment can be revealed and assessed by examining and manipulating thread and warp IDs. Furthermore, metrics such as SIMT efficiency, intersection-shader call counts, and unnecessary intersection test counts can be generated

by ReRay to reveal performance bottlenecks such as areas of high thread divergence, trapped rays, and wasteful shader execution.

ReRay is an open-source project and it is extensible to allow users to develop new performance metrics, simulators, and visualization tools for Vulkan ray-tracing applications. ReRay is a great addition to any Vulkan developer's toolkit because it provides a greater insight into ray-tracing performance by capturing and examining all ray-tracing pipeline events that occur in any frame of an application.

Chapter 4

Shader Resource Analysis and Optimization

Applications must manage the allocation, binding, and updating of resources used by shaders in their pipelines. The Vulkan API provides a low-level interface for the application to manage resources. The flexibility of this low-level interface enables developers to optimize resource management to suit the needs of their applications to an extent that was not possible with higher-level APIs such as OpenGL. However, this optimization requires developers to have a deeper understanding of the life cycle of resources and how they are used in a graphics pipeline.

This chapter proposes a method for capturing and observing the usage of resources in Vulkan applications, with a focus on uniform buffers. Uniform buffers are used to store read-only data that remain constant for all shader invocations within a particular draw call. This data is typically used to pass transformation matrices, lighting parameters, material properties, and graphics settings to shaders in a pipeline. Individual values within a uniform buffer may be updated at varying frequencies because there is a wide variety of data that may reside in uniform buffers. Developers can optimize their applications by analyzing the usage of individual values within uniform buffers to make informed decisions about how to manage and restructure uniform buffers to reduce memory access and GPU state-change overheads [11], [19].

The proposed method involves the creation of a Vulkan layer that intercepts Vulkan API calls to observe and capture the usage of descriptor sets

and uniform buffers in standards-conforming Vulkan applications. Descriptor sets containing bindings with varying update frequencies can be split into multiple descriptor sets to reduce memory management overheads and GPU state-change overheads [19]. The layer is also capable of capturing statistics regarding the usage of individual values within uniform buffers by each pipeline in an application to give recommendations to developers such as using push constants or inline uniform blocks for small amounts of frequently updated sets of uniform values. If the values in an uniform buffer remain constant throughout the observed lifetime of an application, then the buffer may be an eligible target for value specializations that eliminate memory accesses [11].

4.1 Device Memory Types and Resources in Vulkan

Vulkan has several types of *device memory* from which resources may be allocated. A device memory is accessible by the GPU and it can be classified by its properties such as: host visible, host coherent, host cached, and device local. The CPU can directly access host-visible memory. Device-local memory reside on the device itself (i.e., GPU VRAM). Host-cached memory is cached by the CPU. A memory is host coherent if writes to the memory by the CPU are visible to the GPU — and vice-versa — without the need for explicit flushing of caches and invalidation of mapped memory ranges. A memory type may have multiple properties. Applications are responsible for choosing an appropriate available memory type with properties that best suit the use-cases of a given resource to be allocated.

In Vulkan, a resource is a view of a region of memory with associated formatting information that describes how the data in the memory region should be interpreted. There are two primary resource types in Vulkan: buffers and images. Buffers are linear arrays of bytes whose contents are up to the application to interpret. Images are multi-dimensional arrays of data used for textures, frame buffers, and other image-related purposes. Buffers and images are further classified by their usage, such as uniform buffers, storage

buffers, vertex buffers, index buffers, storage images, and sampled images. This chapter focuses on uniform buffers used to store read-only data that remain constant for all shader invocations within a draw call.

4.2 Shader Resource Binding

Programmable shader stages in pipelines require access to resources. A *pipeline layout* describes the set of resources that are accessible by the various stages of a pipeline. Conceptually, a pipeline layout is composed of a sequence of *descriptor-set layouts*. Each descriptor-set layout is a blueprint for a collection of references to specific resource types. A descriptor-set layout is further organized into *descriptor bindings* which are fixed-size arrays of references to resources of the same type. A descriptor binding containing more than one reference to a resource is also called a *descriptor array*. A *descriptor set* is an instantiation of a descriptor-set layout with concretized references to resources. An individual reference to a resource is called a resource descriptor, or just *descriptor*. In shader code, a resource is denoted by a *set number*, a *binding number*, and an array index if the descriptor binding contains more than one descriptor. The set number corresponds to the index of a descriptor set in the pipeline layout. The binding number corresponds to the index of a descriptor binding in the descriptor set. The array index corresponds to the index of the descriptor in a descriptor binding, which may be omitted if the descriptor binding contains only one descriptor. Figure 4.1 illustrates the relationships between a pipeline, pipeline layout, descriptor set, and descriptor set layout using Unified Modelling Language (UML)-style connections.

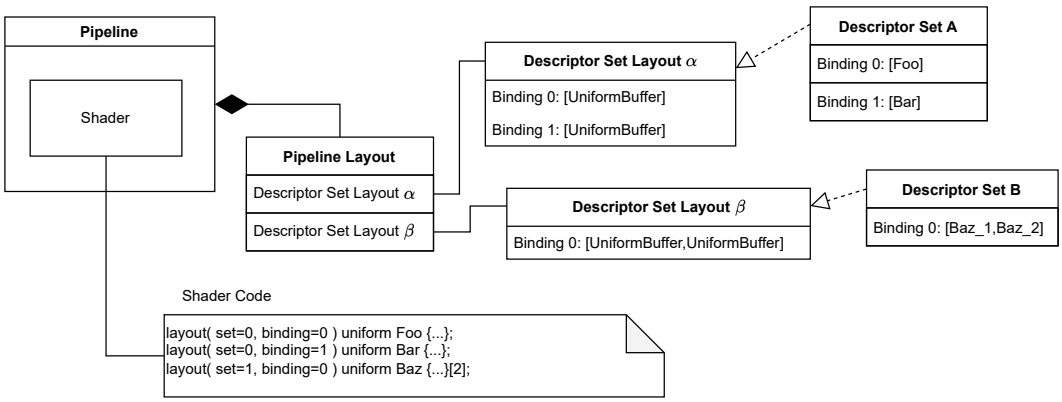

Figure 4.1: An illustration of the relationships between a Pipeline, Pipeline Layout, Descriptor Set, and Descriptor-Set Layout. Relationships between entities are shown with UML-style connections.

In addition to descriptor-set layouts, a pipeline layout may also contain *push-constant ranges* [56] and *inline uniform blocks* [60]. Inline uniform blocks are a way to store uniform data directly in descriptor sets without the need for a separate buffer. A push-constant range is a sequence of bytes provided by a command before a pipeline invocation. Due to the way they are stored, both push-constant ranges and inline uniform blocks have restrictions on the amount of data that they can hold, but the benefit is that they reduce the amount of indirection required to read the data. Inline uniform blocks have the advantage of being reusable across multiple pipelines, while push constants are specific to a single pipeline.

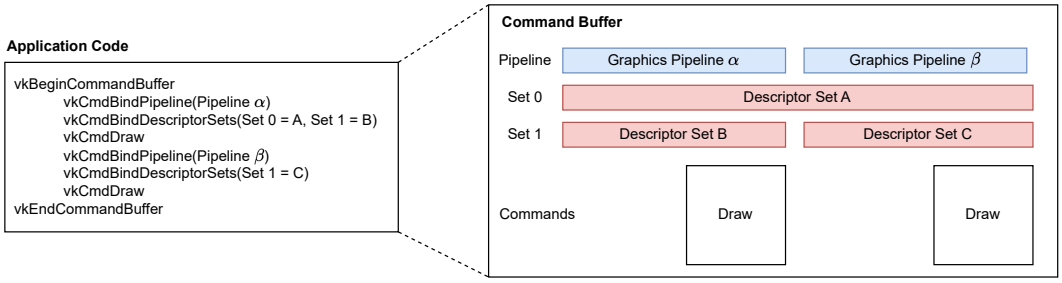

Figure 4.2: Simplified illustration of the process by which application code records a command buffer for submission to a queue for execution.

In order to invoke a pipeline for execution, an application must record

commands into a *command buffer*. Command buffers are a region of memory containing commands to be executed by the GPU. A command buffer may contain multiple invocations of the same or different pipelines with differing descriptor sets and push constants. In the example shown in Figure 4.2 two different graphics pipelines α and β are recorded into a single command buffer for execution. Names of the commands in Vulkan use the term *bind* to describe the act of setting the state of the GPU. For each pipeline, the application must bind the pipeline to be executed (*vkCmdBindPipeline*), bind the descriptor sets to be used by the pipeline according to its pipeline layout (*vkCmdBindDescriptorSets*), set push constants (if any), and then invoke the pipeline with a draw command (*vkCmdDraw*). Inline uniform blocks behave the same as descriptor sets and are bound in the same fashion. After a command buffer has been recorded, it is submitted to a GPU queue for execution.

Pipeline layout compatibility enables pipelines to share descriptor sets during command-buffer recording. In Figure 4.2 both pipelines α and β use descriptor set A in descriptor-set index 0, but descriptor set A was not explicitly given in the *vkCmdBindDescriptorSets* command before the *vkCmdDraw* command for pipeline β. This is because pipeline layout compatibility allows descriptor sets to remain bound across multiple *vkCmdBindPipeline* commands so long as the pipeline layouts of each pipeline expect equivalent descriptor-set layouts from set number zero up to and including the set number where the descriptor sets of interest are bound.

An additional type of descriptor that applications may use to allow flexibility in resource binding is called *dynamic descriptors*. Descriptor sets can contain a mix of both regular (static) and dynamic descriptors. However, descriptor sets containing dynamic descriptors require an additional parameter specifying an array of dynamic offsets — one for each dynamic descriptor — when bound to a pipeline with the *vkCmdBindDescriptorSets* command. Dynamic offsets are used to specify the offset in the buffer referenced by each dynamic descriptor. Although the buffer referenced by a descriptor is fixed, the dynamic offset for each dynamic descriptor can be changed between pipeline invocations by binding the same descriptor set and changing the dynamic

offsets. Therefore, dynamic descriptors enable more opportunities to reuse descriptor sets across multiple pipelines because applications can place multiple resources into a large buffer and reuse the same descriptor set across multiple pipeline invocations by changing the dynamic offsets to different regions of the buffer.

4.3 Related Work

Since the introduction of the Vulkan API, developers have been exploring ways to optimize their applications to take advantage of the low-level control it provides over older APIs such as OpenGL. One particular area of optimization is in the management and use of resources by shaders in pipelines.

In OpenGL, the primary means of supplying uniform data to shaders is via uniform variables. Uniform variables are global variables declared in a shader program that contain uniform data passed to it from the application through the OpenGL API. In GLSL, uniform variables are declared with the keyword `uniform` and have a type, a name, and an optional array size. For example, the statement `uniform vec4 mvpMatrix` declares a uniform variable of type mat4 (a 4x4 matrix of floats) with the name `mvpMatrix`. The application sets the values of uniform variables using OpenGL functions — requiring only the name and type of the uniform variable, and the value or values to be assigned to the variable. For example, to set the value the uniform variable `mvpMatrix` one would call the function `glUniformMatrix4fv(loc, 1, GL_FALSE, matrix)`. `loc` is the location of the uniform variable obtained by `glGetUniformLocation(program , "mvpMatrix")` where `program` is an object representing the OpenGL shader program and `"mvpMatrix"` is the name of the uniform variable to be located. Uniform variables may be arrays of elements or matrices, thus the constant `1` specifies that there is only one matrix. `GL_FALSE` states that the matrix should not be transposed by the driver before assigning it to the uniform variable. `matrix` is an array of 16 floats representing the matrix elements in column-major order. The benefit of uniform variables is that memory management and the transfer of data between the CPU and GPU is handled by the graphics driver,

which is frequently updated with heuristics to maintain good performance for a wide variety of applications.

Crawford et al. show that a significant portion of uniform variables in a variety of OpenGL games contains unused data or remains constant during the runtime of replayed application execution traces [10], [11]. Furthermore, Crawford et al.'s experiments indicate significant performance improvements to individual graphics pipelines and replays of application execution traces when modified to eliminate unused and runtime-constant uniform data. These performance improvements may be attributed to two factors: (1) a reduction in the number of OpenGL calls to update uniform data because runtime-constant uniform data is specialized into shaders instead of being supplied by the CPU; and (2) the reduced amount of data transferred to the GPU by the remaining OpenGL calls due to a truncation of large uniform data arrays containing unused data elements.

However, the performance evaluation method employed by Crawford et al. is not representative of real-world applications. Crawford et al. created a specialized testing harness to run vertex and fragment shaders extracted from application traces in an isolated environment. The testing harness assigned arbitrary default values to all uniform variables in the shaders. These values were not representative of typical values used by the application. Crawford acknowledged the issue, but claimed that the experiments would take far longer to run if they were to extract and use real-world values from the original applications [10].

Unlike OpenGL, Vulkan does not have the concept of uniform variables. Instead, all uniform data must be organized into uniform buffers. Developers have control over the layout of the uniform buffers that they provide to shaders and can share the same uniform buffers between multiple shader stages and graphics pipelines. Furthermore, applications have the additional complexity of managing and binding to pipelines the descriptor sets that reference the uniform buffers. As a result, the design of graphics rendering engines in games that traditionally used OpenGL have evolved over time with the transition to Vulkan.

Yong et al. address an issue regarding pipeline layouts in Vulkan applications previously developed for the OpenGL or other high-level APIs [19]. They claim that due to lack of shading-language features to organize shader resources, developers are inclined to use a single "monolithic" descriptor set to hold all resources used by a pipeline. Performance issues arise when a descriptor set contains descriptors that are updated at wildly differing frequencies because a descriptor set is immutable from the time it is bound to a command buffer to the time every command buffer binding the descriptor set has completed execution on the GPU. If any descriptors in a descriptor set will differ between two pipeline invocations in the same command buffer or in coexisting command buffers, then each pipeline invocation will require its own dedicated descriptor set to be allocated and bound even if many of the descriptors in the descriptor set are the same between pipeline invocations. By partitioning descriptors into multiple descriptor sets based on their rates of change, the number of large descriptor sets that must be allocated and bound can be drastically reduced. Instead, smaller descriptor sets can be allocated and bound to pipelines to reduce the amount of memory that must be allocated to descriptor sets, the amount of CPU time spent updating and writing commands to bind descriptor sets, and the amount of data to be sent to the GPU. Yong et al. demonstrated that significant performance improvements can be achieved when descriptors are partitioned as such. To facilitate these changes, Yong et al. created a shading-language library to allow developers to more easily organize resources in shader code. However, a question remains as to whether monolithic descriptor sets are still a problem in modern Vulkan applications and, if they are, how developers can be informed about the usage of resources in their pipelines to make informed decisions about how to better organize resources in a graphics pipeline. Even if an application does not use monolithic descriptor sets, the partitioning of descriptors into descriptor sets may still not be optimal.

This chapter proposes a method that extends the work of Crawford et al. and Yong et al. by providing developers with insights into the usage of descriptor sets and uniform buffers in Vulkan applications. These insights can

inform developers about ways to optimize applications through the restructuring of descriptor sets and uniform buffers to reduce memory access and GPU state-change overheads. The presentation contrasts with the work by Crawford et al. highlighting differences in the available tools, profiling strategies, optimization techniques, and performance evaluation methodologies between OpenGL and Vulkan.

4.4 Shader Resource Usage Profiling Layer

This section describes a Vulkan instrumentation layer that captures and observes descriptor sets and uniform buffers in Vulkan applications. The layer operates by intercepting key Vulkan functions responsible for creating, managing, and using descriptor sets and uniform buffers in an application. Figure 4.3 shows a block diagram of several key steps that an application takes to prepare pipelines for execution on the GPU. Arrows between blocks indicate dependencies between steps, with the name of the Vulkan object passed between the steps. Steps that are not dependent on each other may be executed in parallel by multiple threads. Applications take additional steps to prepare pipelines for execution on the GPU, which are not shown in the diagram because the instrumentation layer does not need to intercept these steps to observe descriptor set usage and uniform buffer contents.

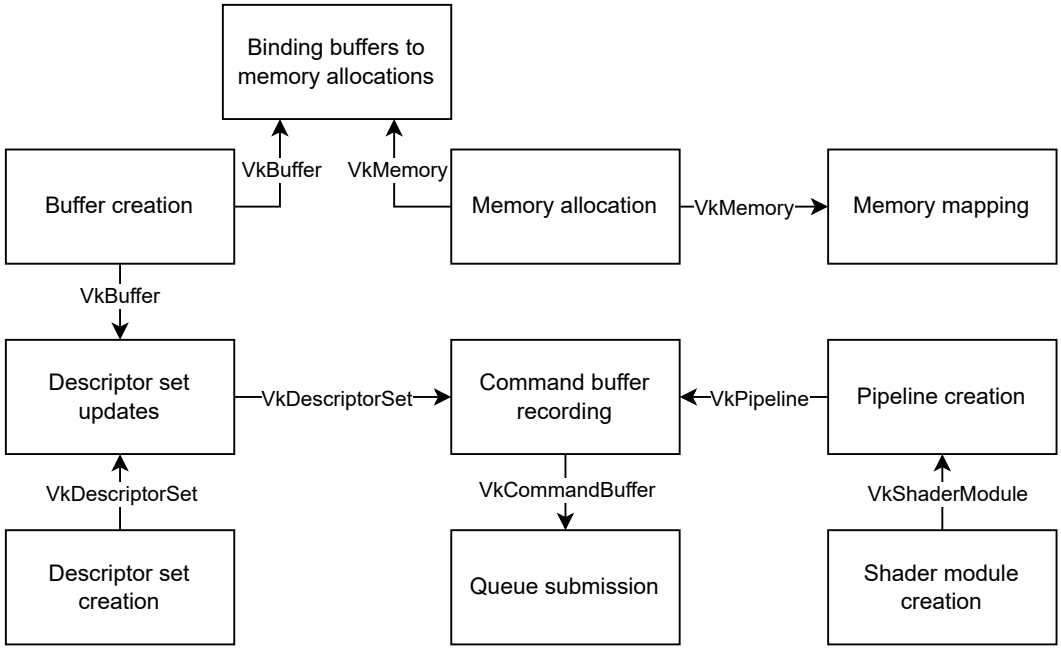

Figure 4.3: A block diagram of various steps Vulkan applications take to prepare pipelines for execution on the GPU. Arrows between blocks indicate dependencies between steps, with the name of the Vulkan object passed between them.

The operation of the layer can be broken down into two facilities: descriptor-set usage tracking and uniform-buffer value profiling. Uniform-buffer value profiling is an extension to the descriptor-set usage tracking.

This layer is designed for the version 1.0 core Vulkan specification and does not support later versions or extensions. Some features from extensions and later versions of Vulkan that affect the operation of the layer, or provide additional optimization opportunities from profiling data, are discussed in subsequent chapters.

4.4.1 Descriptor Set Usage Tracking

Observing the usage of descriptor sets in Vulkan applications involves tracking the creation, updating, and binding of descriptor sets in command buffers before each pipeline invocation.

Descriptor set creation. The process to create descriptor sets applications follows these steps: create a descriptor memory pool; define descriptor-set layouts; and then allocate descriptor sets from the pool using the layouts. The layer intercepts the function *vkAllocateDescriptorSets* function to keep a record of all descriptor sets allocated by the application.

Descriptor set updates. Applications update the contents of descriptor sets with the *vkUpdateDescriptorSets* function. Descriptor sets may be updated at any time before they are bound to a command buffer. The *vkUpdateDescriptorSets* function receives as a parameter an array of *VkWriteDescriptorSet* structs, which describes operations to write descriptors into descriptor sets. The operation description includes: 1. the type of descriptor — corresponding to the type of resource that it will reference — to be written; 2. the destination descriptor set; 3. the destination descriptor binding number; 4. the destination descriptor array index; 5. the number of descriptors to write starting from the destination descriptor array index; 6. a pointer to the resources to be referenced by the descriptors; and 7. in the case of a buffer resource, an offset into the buffer. *vkUpdateDescriptorSets* also takes an array of *VkCopyDescriptorSet* structs that describes operations to copy descriptors from one descriptor set to another. The Vulkan specification states that the operations are processed in the order in which they appear in each array, and that the write operations are performed first, followed by the copy operations. The layer interprets the write and copy operations to update the contents of all descriptor sets after each *vkUpdateDescriptorSets* call.

Command buffer recording. During command-buffer recording, the layer keeps a list of bound descriptor sets before each pipeline invocation. Keeping this list requires the layer to interpret the *vkCmdBindPipeline* and *vkCmdBindDescriptorSets* commands to determine which pipeline was last bound and what descriptor sets were bound before each *vkCmdDraw*. An example of command-buffer recording is shown in Figure 4.2. The order and number of *vkCmdBindPipeline* and *vkCmdBindDescriptorSets* commands before a

vkCmdDraw can vary, but having more than one of each of these commands is wasteful and is not a typical behavior in Vulkan applications. All compatible descriptor sets must be bound for all set numbers for each pipeline. Leaving a descriptor set unbound leads to undefined behavior. A separate Vulkan validation layer catches these kinds of errors during application development and thus they should not appear in production applications.

Command buffers are considered invalid if any descriptor set updates are made to a descriptor set that is currently bound by them. Therefore, the layer can perform an analysis of descriptor-set usage within each command buffer during or after command-buffer recording because the descriptor set contents will not change until the command buffer has been submitted and completed execution on the GPU. Furthermore, this analysis can be carried out independently for each command buffer because the state of bound descriptor sets and pipelines do not persist across command-buffer boundaries even when multiple command buffers are consecutively submitted to a GPU queue for execution.

Queue submission. A layer can combine per-command-buffer analysis results to build a profile for the overall descriptor set usage by an application. This profile can be built just before command buffers are submitted to a GPU queue for execution using the *vkQueueSubmit* command. Details of the analysis are discussed in Section 4.5.1 along with suggestions for how the information can be used to optimize applications.

4.4.2 Uniform Buffer Value Profiling

Uniform buffer value profiling extends the descriptor-set usage tracking facility by reading the contents of uniform buffers before they are submitted to a GPU queue for execution. All memory allocations, buffers, and mapped memory addresses, created by the application are tracked by the layer to identify and keep information necessary to read uniform buffer data. GPU commands for updating and copying data to uniform buffers are also tracked to handle uniform buffers residing in non-host-visible memory.

Buffer creation and memory allocation. The *vkCreateBuffer* function is intercepted to make a record of all buffers created by an application. Buffers created with the usage flag `VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT` are uniform buffers. Allocation of device memory is done using the *vkAllocateMemory* function which is intercepted by the layer to keep a record of all such memory allocations created by the application. The records include the memory type, properties, and size of memory allocations, as well as the Vulkan object *VkMemory* that represents the memory allocation. The association of buffers to memory allocations is recorded by intercepting the *vkBindBufferMemory* function, which associates a buffer to a memory allocation with an offset starting address.

The size of a memory allocation is not necessarily the same as the size of a buffer. Applications are encouraged to allocate a large memory space and then to sub-allocate resources from that space because there are operating-system-imposed limits on the maximum number of simultaneously active memory allocations in use by an application. Also, memory allocation/de-allocation can be costly due to interaction with the operating system and drivers [55]. Even if a dedicated allocation is created for a single buffer, the size of this allocation may differ due to hardware-specific allocation size and alignment requirements queried using the function *vkGetBufferMemoryRequirements*.

Uniform buffers may reside in different memory types leading to either direct or indirect reading of their contents by the layer. If the memory type has the host-visible property, then the layer can read the contents of the uniform buffer through a void pointer obtained by mapping the memory to the application's virtual address space. However, reads may be slow if the memory type does not also have the host-cached property. If the memory type does not have the host-visible property, then the application will be writing data to the uniform buffer using GPU commands which may involve the use of an intermediate host-visible transfer buffer that is copied to the uniform buffer residing in non-host-visible memory. The layer must track these GPU commands to determine the contents of uniform buffers residing in non-host-visible memory.

Memory mapping. Applications use the functions *vkMapMemory* and *vkUnmapMemory* to map and un-map host-visible device memory (represented by a *VkMemory* object) into the application's virtual address space. Thus, the application uses a void pointer to read and write data to the memory region. The layer intercepts these function to keep a record of all mapped memory regions, the resulting void pointers, and the offsets into each memory region where the void pointers point to. Applications may un-map memory regions when they are done reading or writing data to them, but may also leave them mapped for the lifetime of the application. Memory that stays mapped for the lifetime of an application is referred to as *persistently-mapped memory*.

Shader module creation. To determine the format of data residing in uniform buffers, the layer intercepts the *vkCreateShaderModule* function responsible for creating shader modules in programmable shader stages of pipelines. The SPIR-V byte code passed to the function contains annotations describing the location and memory layout of all uniform buffers used by the shaders. The location consists of a set number, binding number, count of descriptors in the descriptor binding, and composite data type of a uniform buffer (uniform data structure). The memory layout consists of annotations describing the data types, offsets, and array element strides of members of the uniform data structure.

Additionally, an analysis of the SPIR-V byte code can be performed to find all uses of uniform buffer data in the shader code. As found by Crawford et al., significant portions of uniform data provided to graphics pipelines are not used by their shaders [10], [11]. Application developers can be informed about unused uniform buffer data that could potentially be removed from uniform buffers to improve GPU cache utilization and reduce memory transfer overheads between the CPU and GPU.

Pipeline creation. The application creates graphics pipelines using the function *vkCreateGraphicsPipelines*. The function takes an array of *VkGraphicsPipelineCreateInfo* structs that describes the configuration of graphics pipelines

to be created, including the shaders used by each pipeline. The association of shader modules to pipelines is recorded by the layer to determine which pipelines use which shaders and determine the format of data residing in uniform buffers bound to and used by the pipeline by referring to the annotations in the shader modules' SPIR-V byte code.

Command buffer recording. When descriptor sets are bound in a command buffer with *vkCmdBindDescriptorSets*, the layer records the dynamic offsets for any descriptor sets containing dynamic descriptors. The associations of bound pipelines to descriptor sets before pipeline invocations are already handled by the descriptor set usage tracking facility. It is at this point that the layer can determine the uniform buffers used by each pipeline invocation. However, the contents of uniform buffers may yet change before the pipeline is executed on the GPU.

As mentioned earlier, if a uniform buffer does not reside in host-visible memory, then the uniform buffer data is provided from GPU commands. The commands for updating and copying data to buffers are *vkCmdUpdateBuffer* and *vkCmdCopyBuffer* respectively. The buffer update operation, *vkCmdUpdateBuffer*, is used to update the contents of a buffer with a small amount of data provided by the application. The buffer copy operation, *vkCmdCopyBuffer*, is used to copy the contents of one buffer to another buffer. In the case of a buffer update to a uniform buffer, the layer can read the data directly passed to the *vkCmdUpdateBuffer* function to determine the contents of the uniform buffer. In the case of a buffer copy to a uniform buffer, the layer will need to make a record of the association between the source buffer and the destination uniform buffer. The layer can later read the contents of the source buffer to determine the contents of the destination uniform buffer. The source buffer is likely a host-visible transfer buffer used by the application to copy data to the destination uniform buffer residing in non-host-visible memory. The layer will need to separately determine how to read the contents of the source buffer.

Queue submission. Just before command buffers are submitted to a GPU command queue for execution using the *vkQueueSubmit* function, the layer reads the contents of uniform buffers (or the source transfer buffers for buffer copy operations) bound to pipelines in the submitted command buffers. The layer reads uniform buffer contents before queue submission because this is the last opportunity to read the contents of uniform buffers before they are executed on the GPU. There may be edge cases where the contents of uniform buffers are modified after queue submission on the GPU by aliasing uniform buffer memory with a read-and-write shader-storage buffer and then modifying the data in shader code, but this is not common practice in Vulkan applications and may result in undefined behavior.

Reading all uniform buffer contents just before queue submission may significantly delay the submission of command buffers to the GPU. If this delay is unacceptable, then the layer may need to read the contents of uniform buffers as the application modifies them. This is a difficult task because the layer must know when the application modifies data through the void pointer, and what bytes were modified without reading and comparing to a copy. An approach to achieve this task is to create shadow memory allocations for each memory allocation associated with a uniform buffer. The layer can then replace the void pointer returned by *vkMapMemory* with a pointer to the shadow memory allocation that the application has no access to. When the application attempts to read or write data to the shadow memory, a custom memory access violation handler can be used to forward the read or write operation to the actual memory allocation. The use of this handler enables the layer to read the contents of uniform buffers as the application modifies them page-by-page. This approach is used by the gfxreconstruct Vulkan frame capture tool to capture and replay Vulkan applications [30]. Another benefit to shadow memory is that the frequency at which uniform buffer data is updated can be determined. The layer can suggest binding less-frequently-updated uniform buffers to device-local memory to minimize memory access latency during shader execution at the expense of increased overhead for updating the data.

4.5 Benefits and Evaluation

Using a Vulkan layer to analyze the usage of descriptor sets and uniform buffers in Vulkan applications can provide insight into opportunities for performance improvements through more efficient use of these resources to reduce the number of GPU state changes and the number of memory accesses. This section describes how to analyze descriptor-set and uniform-buffer usage per pipeline. It then discusses how the results of the analyses may be used to enable Vulkan features such as dynamic descriptors and inline uniform blocks with the goal of improving descriptor-set and uniform-buffer usage. Finally it discussed methods for evaluating the effectiveness of these improvements.

4.5.1 Descriptor Set Usage Analysis

The primary goal of improving descriptor-set usage is to reduce the amount of descriptor data that needs to be allocated, updated, and bound in each pipeline invocation. There are two conditions that require the allocation, updating, and bounding of a new descriptor set: (1) at least one of the descriptors in a descriptor set must be changed before the next pipeline invocation in the command buffer; or (2) the descriptor set is incompatible with the pipeline layout of the next pipeline to be bound in the command buffer. The binding of new descriptor sets because of differences in pipeline layouts may be unavoidable due to differences in inputs required by the newly-bound pipeline. However, for consecutive invocations of the same pipeline, it is possible to either eliminate or lower the cost of the binding of new descriptor sets by (1) repartitioning the descriptors into different descriptor sets; and (2) modifying the pipeline's layout to match the new partitioning.

Consecutive pipeline invocations of the same pipeline occur within a command buffer because hardware vendors recommend pipeline invocations to be sorted within command buffers to minimize the number of times that a new pipeline needs to be bound [3], [9]. Applications are also encouraged to use large command buffers to minimize the total number of command buffers in use [3]. Therefore, command buffers contain a large number of pipeline invo-

cations, and pipeline invocations occur consecutively in the command buffer.

Consider the command buffer in Figure 4.4. This command buffer binds a pipeline that uses a monolithic descriptor set to reference all resources used by the pipeline. The data used by the pipeline differs between pipeline invocations as seen in the timeline of bound descriptor sets for each pipeline invocation, shown beneath the command-buffer illustration. Once a descriptor set is bound in a command buffer it is immutable. Thus, each pipeline invocation requires its own dedicated descriptor set because each of them differs from the descriptor set bound to the previous pipeline invocation. However, many of the descriptors in the descriptor sets do not change between pipeline invocations, resulting in redundant descriptors. Descriptor sets must be allocated and written to before being bound to the command buffer. Redundant descriptors cause wasting of memory and CPU time for the writing of redundant data to similar descriptor sets and require extra data to be sent to the GPU when the command buffer is submitted for execution. A repartitioning of descriptors into different descriptor sets should aim to reduce the amount of redundancy between descriptor sets.

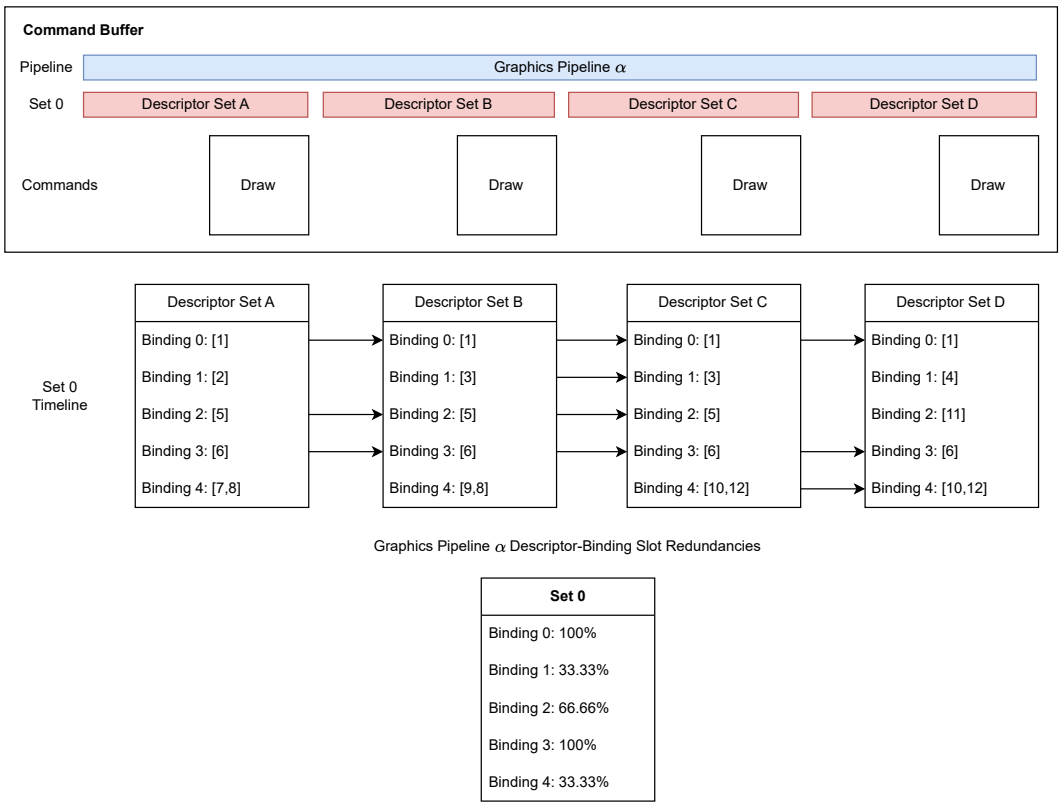

Figure 4.4: An example of a command buffer that performs four consecutive draw calls (pipeline invocations) using a pipeline with a monolithic descriptor set pipeline layout. Beneath the illustration of the command buffer is a timeline showing the descriptor sets bound to set number 0 (Set 0) before every pipeline invocation. Each descriptor set is illustrated by a list of its descriptor bindings in descriptor-binding slots numbered from 0 to 4. Each descriptor binding contains an array of descriptors denoted by an ordered comma-separated list of positive integers enclosed in square brackets. Arrows between descriptor-binding slots on the timeline indicate redundancy at the descriptor-binding slot due to the descriptor bindings being equivalent across descriptor sets. Beneath the timeline is the redundancy measurement result for all descriptor-binding slots.

Repartitioning by Descriptor-Binding Slot Redundancy

As described in Section 4.2, a pipeline layout is defined by a sequence of descriptor-set layouts. During command-buffer recording, descriptor sets are bound to pipelines at specific slots denoted by set numbers corresponding to the index of a descriptor-set layout in the pipeline layout. Each set-numbered slot may only hold a descriptor set that has a format matching that described

by the corresponding descriptor-set layout. Descriptor sets are composed of a sequence of descriptor bindings, where a descriptor binding is a fixed-size array of descriptors of the same type. Each descriptor binding belongs to a slot in the descriptor set indexed by a binding number. Shaders access descriptors by specifying a set number, a binding number and, if the descriptor binding contains more than one descriptor, an array index.

A repartitioning of descriptors into descriptor sets to reduce the number of redundant descriptors between descriptor sets involves modifying the pipeline's layout. For simplicity, repartitioning is performed at the granularity of *descriptor-binding slots* rather than individual descriptors. A descriptor-binding slot is a tuple (s, b) where s is a set number and b is a binding number. A repartitioning of descriptor-binding slots creates a mapping of $(s, b) \mapsto (s', b')$ where (s', b') is the set number and binding number of the same descriptor-binding slot in the new pipeline layout. All descriptor bindings that were bound to slot (s, b) in the previous pipeline layout are bound to (s', b') in the new pipeline layout. This repartitioning facilitates the modification of shader code to conform to new pipeline layouts by changing the set and binding numbers where descriptors are accessed.

The following definitions apply to a single pipeline:

- N is the number of descriptor-set layouts in the pipeline layout.
- $S = \{s \in \mathbb{N}^0 : s \in [0, N)\}$ is the set of set numbers in the pipeline layout.

We need the following additional definitions:

- K_s is the number of descriptor bindings in any descriptor set bound to set number s.
- $B_s = \{b \in \mathbb{N}^0 : b \in [0, K_s)\}$ is the set of binding numbers in any descriptor set bound to set number s. According to the definitions of descriptor-set layout, pipeline layout, and descriptor set, all descriptor sets bound to the same set number must have the same number of descriptor bindings.

- $D = \{(s, b) : s \in S, b \in B_s\}$ is the set of descriptor-binding slots where descriptor bindings may reside.

- T is the number of times the pipeline is invoked consecutively in a command buffer.

- $v^t_{(s,b)}$ is the content of a descriptor binding residing in descriptor-binding slot (s, b) immediately before the t-th pipeline invocation in the command buffer, where $t \in \mathbb{N}$ and $t \leq T$. Two descriptor bindings $v^{t_1}_{(s,b)}$ and $v^{t_2}_{(s,b)}$ are equal if and only if they contain the same descriptors and, in the case of multiple descriptors, the descriptors appear in the same order.

A repartitioning of descriptor-binding slots can be informed by measuring the amount of redundancy in descriptor bindings that occurs in each descriptor-binding slot. Descriptor bindings $v^t_{(s,b)}$ and $v^{t+1}_{(s,b)}$ are considered redundant in descriptor-binding slot (s, b) if the descriptor bindings are equal between the consecutive pipeline invocations t and $t + 1$. The overall redundancy occurring in descriptor-binding slot (s, b) is defined as follows:

$$R_{(s,b)} = \frac{\sum_{t=1}^{T-1} 1_{v^t_{(s,b)} = v^{t+1}_{(s,b)}}(t)}{T - 1}$$

where $1_{v^t_{(s,b)} = v^{t+1}_{(s,b)}}(t)$ is an indicator function that returns 1 if $v^t_{(s,b)}$ and $v^{t+1}_{(s,b)}$ are equal, and 0 otherwise.

Descriptor-binding slots with similar amounts of redundancy may be grouped together into descriptor sets for the new pipeline layout to minimize the total number of descriptors that need to be allocated, updated, and bound in the command buffer. To illustrate, consider the monolithic descriptor set pipeline layout example in Figure 4.4. Redundancy occurring at a descriptor-binding slot is illustrated visually by arrows connecting descriptor-binding slots across descriptor sets on the timeline of bound descriptor sets. The bottom of Figure 4.4 lists the overall redundancy of each descriptor-binding slot. A reparti-

tioning of descriptor-binding slots in Figure 4.4 is as follows:

$$(0,0) \mapsto (0,0),$$
$$(0,3) \mapsto (0,1),$$
$$(0,2) \mapsto (1,0),$$
$$(0,1) \mapsto (2,0),$$
$$(0,4) \mapsto (2,1)$$

The result of this repartitioning is shown in Figure 4.5. The repartitioning by redundancy in descriptor-binding slots takes advantage of pipeline layout compatibility to reuse descriptor sets across pipeline invocations. Thereby, this repartitioning reduces the number of descriptor sets that need to be allocated, updated, and bound in the command buffer. This reuse of descriptor sets is illustrated in the timeline of bound descriptor-sets in Figure 4.5 by dashed arrows between descriptor-binding slots of the same descriptor set across pipeline invocations. Although the dashed arrows indicate that the descriptor-binding slot is occupied by a descriptor binding from a reused descriptor set, the dashed arrows do still count towards the redundancy of a descriptor-binding slot to maintain the property that redundancy in a descriptor-binding slot remains the same after a repartitioning of descriptor-binding slots.

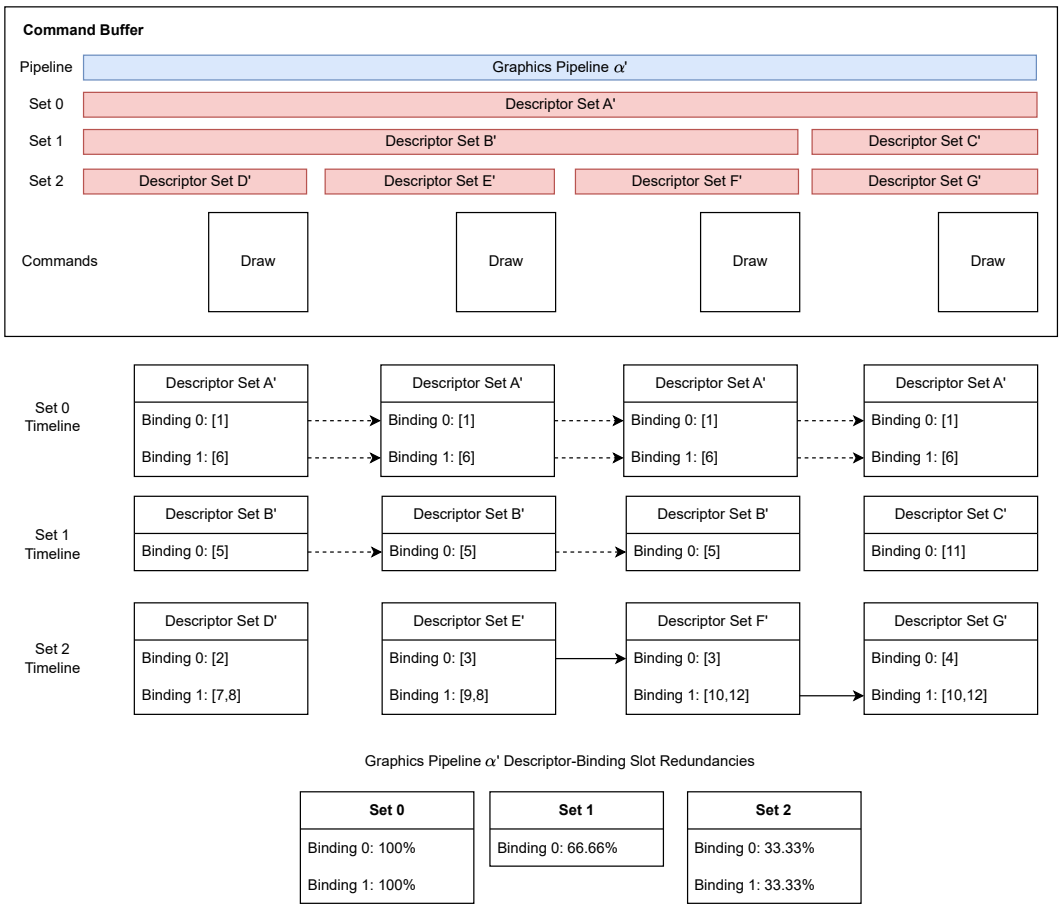

Figure 4.5: Illustration of a command buffer that performs four consecutive draw calls (pipeline invocations). Beneath the illustration of the command buffer is a timeline showing the descriptor sets bound at each set number before every pipeline invocation. Each descriptor set is illustrated by a list of its descriptor bindings in numbered descriptor-binding slots. Each descriptor binding contains an array of descriptors denoted by an ordered comma-separated list of positive integers enclosed in square brackets. Arrows between descriptor-binding slots on the timeline indicate redundancy at the descriptor-binding slot due to the descriptor bindings being equivalent across descriptor sets. Beneath the timeline is the redundancy measurement result for all descriptor-binding slots.

Redundancy in a descriptor-binding slot is closely related to the rate of change of descriptor data with respect to the number of pipeline invocations. A high rate of change in a descriptor-binding slot results in few pipeline invocations between changes, which implies a low redundancy in the descriptor-binding slot — and vice-versa. Grouping descriptor-binding slots with similar

redundancy into descriptor sets is equivalent to grouping descriptor-binding slots with similar rates of change in descriptor data into the same descriptor set, which is the suggested approach to creating descriptor sets [3], [19].

Evaluating Descriptor-Binding Slot Repartitioning

Recall the primary goal of improving descriptor-set usage is to reduce the amount of descriptor data that needs to be allocated, updated, and bound in each pipeline invocation. The repartitioning of descriptor-binding slots by redundancy achieves this goal by reducing the amount of redundant descriptors between descriptor sets bound across consecutive pipeline invocations. Figure 4.4 displays redundancies present in four consecutive pipeline invocations of a pipeline α using a monolithic descriptor set to hold all of its resource descriptors. Figure 4.5 displays equivalent invocations of pipeline α' whose layout is a partitioning of the monolithic descriptor set used by pipeline α. To quantitatively evaluate the effectiveness of the repartitioning, the number of descriptors used over the same sequence of pipeline invocations is compared before and after repartitioning. Figure 4.6 depicts all descriptor sets bound before each pipeline invocation for pipelines α and α. Pipeline α binds a total of 4 unique descriptor sets over the course of four pipeline invocations, and each descriptor set contains 6 descriptors. Therefore, pipeline α uses a total of 24 descriptors. Pipeline α' binds a total of 7 unique descriptor sets over the course of the same four pipeline invocations:

- There is 1 unique descriptor set associated with set number 0, and it contains 2 descriptors
- There are 2 unique descriptor sets associated with set number 1, and they each contain 1 descriptor
- There are 4 unique descriptor sets associated with set number 2, and they each contain 3 descriptors

Therefore, pipeline α' uses a total of 16 descriptors. The repartitioning has resulted in an overall 33.33% reduction in the number of descriptors used over the same four pipeline invocations.

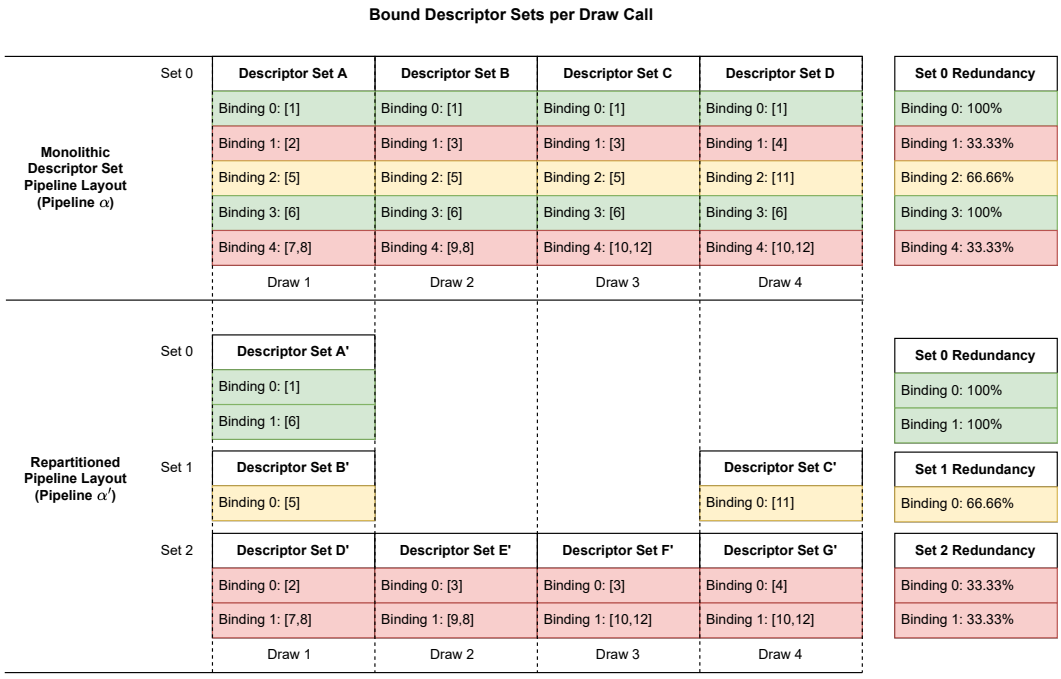

Figure 4.6: A depiction of all descriptor sets bound just before each draw call (pipeline invocation) for two different pipelines α and α'. Pipeline α uses a monolithic descriptor set pipeline layout. Pipeline α' a pipeline layout where descriptor sets are created to group descriptor-binding slots with similar redundancy. The redundancy in each descriptor-binding slot is listed on the right-most column. Descriptor-binding slots are also colored based on their redundancy: A red color indicates low redundancy (33.33%); a yellow color indicates medium redundancy (66.66%); a green color indicates high redundancy (100%).

While the example repartitioning in Figure 4.5 uses three descriptor sets, it is possible to use four descriptor sets to further reduce the number of redundant descriptors between descriptor sets. To scale up to pipeline layouts that use more descriptor bindings with varying rates of change, it may be possible to dedicate a descriptor set to each descriptor binding. Due to hardware limitations, however, there is a maximum number of descriptor sets that can be bound to a pipeline at any given time. This limit can be obtained by querying the *vkGetPhysicalDeviceProperties* function and reading the integer property *maxBoundDescriptorSets*. Furthermore, it is a balancing act to determine what is the optimal number of descriptor sets to use and how many descriptor

bindings to place in each descriptor set. Nuno et al. recommend, on NVIDIA's technical blog regarding best practices for Vulkan API usage, that applications should achieve the conflicting goals of keeping the number of descriptor sets in pipeline layouts to a minimum, while also minimizing the number of descriptors in descriptor sets [34]. However, it is not clear which of these two goals is more important. A study of the trade-offs between the number of descriptor sets and the number of descriptors in descriptor sets is needed to determine a good partitioning of descriptors into descriptor sets.

4.5.2 Uniform Buffer Value Analysis

Opportunities to improve the management of uniform buffers can be identified through analysis of the data contained within the uniform buffers for each pipeline invocation. For OpenGL programs, Crawford et al. finds that significant portions of the data in uniform buffers is often unused or remain constant at runtime. They use value specialization and truncation of arrays in shader code to improve performance [10], [11]. In Vulkan, further opportunities for optimization are available through the use of features that combine uniform data with shader-resource binding mechanisms such as push constants and inline uniform blocks.

Just before a command buffer is submitted to the GPU for execution, the uniform buffer value profiling facility of the Vulkan layer (Section 4.4.2) can read all the data contained within every uniform buffer used by each pipeline invocation in the command buffer. The formats of the data contained within the uniform buffers are also known to the Vulkan layer because the SPIR-V byte code of the shaders comprising each pipeline are read by the layer to obtain the layout of the uniform data structure. With this knowledge, analyses can be performed to determine information about each field of a uniform data structure such as the frequency of changes in value, the number of unique values observed, and top-N values observed [8].

The goal of uniform buffer value analysis is to provide information about the usage of uniform data structures by each pipeline of the application. Pipelines can be analyzed independently of each other, and the results of

the analysis may be used to optimize the usage of uniform buffers for each pipeline. Shader SPIR-V byte code analysis can reveal unused fields or array elements of uniform data structures may be eliminated from uniform buffers to reduce memory usage, memory transfer times, and improve GPU cache utilization. Likewise, fields of uniform data structures that appear to remain constant at runtime may be eligible for value specialization in shader code. Value specialization replaces the field of the uniform data structure with a compile-time constant value in the shader code, eliminating the need for the field to be present in the uniform buffer and enabling compiler transformations to further optimize the shader code such as constant folding and dead code elimination. This also eliminates memory accesses to uniform buffers and can enable the elimination of the associated data from the uniform buffer entirely. In Vulkan, value specialization can be achieved using specialization constants. Specialization constants are a feature of the Vulkan API that can be used to specialize the values of variables in shader code at pipeline creation time. The SPIR-V byte code of shaders can be modified to replace references to the runtime-constant uniform data with specialization constants. The advantage of using specialization constants over directly performing value specialization in shader code is that the shader code does not need to be duplicated by the application or an optimization layer for each possible specialization; instead, the Vulkan driver compiles and transforms the shader code for each specialization at pipeline creation time.

Similar work in value specialization has been done by Crawford et al. for OpenGL programs with respect to uniform variables instead of buffers [10], [11]. While OpenGL does support uniform buffers, the work by Crawford et al. focuses on the use of uniform variables which are easier to eliminate from shader code. In Vulkan, it is not easy to make changes to uniform buffers since the same uniform buffer may be used by multiple pipelines; some of the data contained within a uniform buffer may be used by some pipelines and not others. It may be more costly for an application to manage multiple uniform buffers than to manage a single uniform buffer with potentially unused data for some of the pipelines that use it. Managing multiple specialized uniform

buffers for each pipeline could result in redundant data between uniform buffers — leading to increased memory usage. Additional overheads in managing resource descriptors for each uniform buffer is also an issue, as each uniform buffer requires its own descriptor for a pipeline to have access to it.

A more practical type of value specialization called zero-value specialization [44] is applicable when fields of uniform data structures often take on the value of zero. Zero-value specialization is a form of value specialization that inserts specialized fast-paths into code that replaces references to zero-originating variables with a compile-time constant value of zero, greatly reducing and simplifying many arithmetic operations in the fast path. Zero-value specialization does not only apply to uniform buffer data; it can apply to all sources of zeros, from textures to arithmetic and comparison operations in shader code.

Optimizing Uniform Buffer Usage with Descriptor Sets

Other approaches exist in Vulkan to optimize uniform buffer usage when considering their usage in tandem with descriptor sets. For instance, it may be the case that there are duplicate uniform buffers. This would be observed when a uniform buffer descriptor has changed between two pipeline invocations, but the data read from the uniform buffer remains the same. It may be possible to eliminate one of the two uniform buffers and its associated descriptor. More opportunities exist when considering Vulkan features such as push constants and inline uniform blocks that can be used to improve the efficiency of both uniform buffer and descriptor set usage simultaneously.

Push constants are a way to provide a small amount of data to a pipeline at the time of a pipeline invocation [56]. Push constants are stored in a region of memory that is part of the command buffer and therefore are not backed by a uniform buffer. The amount of data that can be stored in push constants is hardware-dependent, and can be queried by the application using the *vkGetPhysicalDeviceProperties* function and reading the integer property *maxPushConstantsSize*. In addition to not requiring a backing uniform buffer, push constants do not require a descriptor set to be modified or bound to be

used by a pipeline. These properties make push constants useful for small data that changes frequently, due to less overheads required in updating the data and fewer indirections required to access the data. An example of data that may be suitable for push constants is a transformation matrix that is different for each draw call in a command buffer. A transformation matrix can be set in push constants before each pipeline invocation in the command buffer. The pipeline can then read the transformation matrix directly from push constants without needing to read from a uniform buffer through a resource descriptor.

To determine the suitability of using push constants for particular fields of uniform data structures, the Vulkan layer can analyze the frequency of changes in the values of suitably-small fields. Furthermore, since uniform buffers may be shared between multiple pipelines, such analyses should combine results from multiple pipelines that use the same uniform buffer. The Vulkan layer can then provide recommendations to the application on which fields of uniform data structures may be suitable for push constants, and list all pipelines that should be modified to accommodate the use of push constants for those fields and the change in uniform buffer format.

Inline uniform blocks are a feature available in a Vulkan extension *VK_EXT_inline_uniform_block* or in Vulkan 1.3 as a way to store uniform data directly in descriptor sets without the need for a uniform buffer [60]. Like push constants, inline uniform blocks reduce the amount of indirection required to read uniform data. The amount of data that can be stored in an inline uniform block is also hardware-dependent, and can be queried by the application using the *vkGetPhysicalDeviceProperties2* function. Furthermore, there is a maximum number of inline uniform blocks can be included in pipeline layouts and accessed by shader stages, queried with the same function. Unlike push constants, inline uniform blocks can be shared between multiple disjoint pipeline invocations, since push constant data is shared globally for all pipeline invocations and may be overwritten between pipeline invocations. Eligibility of uniform buffers to be replaced with inline uniform blocks can be determined by a Vulkan layer by examining the sizes of uniform buffers. Deciding which uniform buffers should be replaced with inline uniform blocks can be a topic for

future research, utilizing static and dynamic analysis of the usage of uniform buffers by shaders across pipelines.

Evaluating Uniform Buffer Optimizations

Several aspects are considered when evaluating the effectiveness of optimizations to uniform buffer usage on pipeline execution times in Vulkan applications. A performance-measurement layer separate from the profiling layer described in Section 4.4 can be made to provide fine-grained measurements of application performance on both the CPU and GPU sides.

On the CPU side, there is the time spent writing data into uniform buffers. This time may be measurable, but would incur significant overheads due to requiring the use of shadow memory to be able to observe writes to uniform buffer memory, since the application writes to uniform buffers directly through a void pointer as opposed to a Vulkan API function that can be intercepted by a Vulkan layer.

Since uniform buffer optimizations such as the use of push constants and inline uniform blocks can affect descriptor sets, another CPU-side aspect of performance is the time taken to allocate descriptor sets, write data into them, writing commands to bind them in command buffers, and submit the command buffers for execution. The CPU time taken by these steps can be measured by computing the delta in time before and after their respective Vulkan commands outlined in Section 4.4.

On the GPU side, the running time of pipelines before and after optimizations is of primary interest. One method to measure the running time of pipelines is to insert a GPU timestamp query [57] before and after pipeline invocations in command buffers. Timestamp queries require the allocation of a memory pool — called a query pool — for the timestamp queries to store results to and for the CPU to read results from. Query pools are created with the *vkCreateQueryPool* function. During command buffer recording, a Vulkan layer can insert timestamp query commands with the function *vkCmdWriteTimestamp* to write GPU timestamps into a query pool. The results of timestamp queries may be polled and read by the CPU using the *vkGet-*

QueryPoolResults function. The metric of time recorded by GPU timestamp queries is not defined in the Vulkan API, but is guaranteed to monotonically increase over time. The accuracy of GPU timestamp also varies between different hardware.

The *vkCmdWriteTimestamp* command introduces an execution dependency on all commands executed before it in the command buffer before the timestamp is written. Due to the execution dependency, timestamp queries will prevent the GPU from possibly overlapping the execution of commands across the boundary of the timestamp query. On one hand, the execution dependency incurs significant overhead on application performance, which prevents measuring performance in real-time during application execution if there are many timestamp queries. On the other hand, the execution dependency ensures that pipelines execute in isolation, as they will not overlap with other pipeline executions in the command buffer. Due to the overheads, using many GPU timestamps to measure pipeline execution performance is best suited for measuring performance in frames captured using a frame-capture tool such as NVIDIA NSight Graphics' C++ Capture [41] or gfxreconstruct [30]. These frame captures are reproducible snapshots of a program's execution trace to render a single frame or a sequence of frames. Vulkan layers can be applied to a frame capture in the same way as the original application, and their reproducibility allows repeated performance measurements on the same frame. Using GPU timestamps on frame captures enables the performance measurement of each pipeline in a reproducible but realistic setting that uses the same inputs as the original application — avoiding the issue Crawford et al. had of lacking representative shader inputs when measuring the performance of individual pipelines in an isolated environment [10].

While CPU time measurements and GPU timestamp queries on frame captures may provide insights into how optimizations affect the performance of pipelines in a single frame, they may not provide a complete picture of the performance of the application as a whole. Performance of multiple frame captures is required to determine the average performance of the application across multiple scenarios. The question of what frames to capture that are represen-

tative of the application's performance in all scenarios is an open question. Video games are interactive applications that respond to user input and also have non-deterministic behavior due to the use of random number generators, physics simulations, and other sources of unpredictability. It may not be feasible to generalize performance results from a small number of frame captures to the entire application.

Instead of fine-grained performance measurements, coarse-grained performance measurements are more suitable for evaluating the effectiveness of all optimizations. Frame-time measurements are the primary metric used by game developers to evaluate the performance of their applications [64]. Frame time is a measure of the time taken to render a single frame of the application, in milliseconds, and is inversely proportional to the frame rate or FPS of the application. Many tools exist to measure the frame time of an application in real time, and often come built into the applications themselves. The frame time of the application should be monitored during prolonged execution over a wide variety of scenarios, such as during play-testing and, if applicable, the benchmark modes of the application. Frame captures are made when the frame time drops below a certain threshold set by the target frame rate of the application by the developers (e.g., 16.67 milliseconds for a 60 frames-per-second target frame rate). Optimizations to uniform buffer and descriptor usage can be made to frame captures to determine if the frame time improves compared to the original frame captures. If so, the optimizations may be applied to the original application, and further testing is required to determine if the improvements are consistent across all observed scenarios during play-testing and benchmarking.

Frame profiling tools such as NVIDIA NSight Graphics [41] and AMD's Radeon GPU Profiler [2] may also be used to provide insights into the performance of application frame captures. These tools provide CPU time measurements and GPU time measurements for various Vulkan API functions and GPU commands like that achieved with CPU time and GPU timestamp queries using a Vulkan layer. Unlike the Vulkan layer, the frame-profiling tools measure GPU execution time of pipelines in milliseconds to allow a developer to

more easily identify pipeline executions that dominate frame time. Hardware performance counters such cache throughput and hit rates are also provided by these tools. Since uniform buffer and descriptor optimizations may lower the number of memory accesses, cache throughput and hit rates may be a useful metric to determine if the optimizations are effective. However, these profiling tools do not have an API that can be used to automate the collection and analysis of performance data, and therefore are not suitable for use in automated performance testing. To minimize overheads and enable real-time fine-grained performance measurement in an application, a Vulkan layer may be adapted to collect performance data for specific pipelines of interest as opposed to all pipelines in the application.

4.6 Technological Advancements and Considerations

4.6.1 Vulkan Extensions to Resource Descriptors

The Vulkan API is constantly evolving, and new features and extensions are being added to the API to enable more ways to improve performance and flexibility. Some of these features and extensions affect the usage of descriptor sets in Vulkan applications. Some of the new features and extensions that are relevant to the analysis of descriptor sets include: descriptor buffers, descriptor indexing, and push descriptors

Descriptor buffers. The Vulkan API extension *VK_EXT_descriptor_buffer* [58] introduces a new way to create, update, and bind resource descriptors in Vulkan applications through the use of *resource descriptor buffers*, or just *descriptor buffers* for short. Descriptor buffers do away with the abstract descriptor pool and descriptor set model used in the core Vulkan API and instead allow applications to directly manage resource descriptors by using buffers as the backing storage for them. No longer do applications use the functions *vkCreateDescriptorPool*, *vkAllocateDescriptorSets*, *vkUpdateDescriptorSets*, and *vkCmdBindDescriptorSets* to allocate, update, and bind descrip-

tor sets. A different set of functions are provided to facilitate the creation, update, and binding of descriptor buffers.

Supporting descriptor buffers in the layer would require drastic changes to the Descriptor-Set Usage Tracking facility to accommodate the new descriptor buffer model. Instead of intercepting *vkAllocateDescriptorSets* to track descriptor-set allocations, the layer would track the creation of buffers from *vkCreateBuffer* that have usage flags of the form *VK_BUFFER_USAGE_*_DESCRIPTOR_BUFFER_BIT_EXT* indicating that they are descriptor buffers. The contents of descriptor buffers are updated using the *vkGetDescriptorEXT* function which takes in a device address of a resource (as opposed to the Vulkan object representing the resource) and writes a corresponding descriptor into a descriptor buffer. For buffers, the device address is queried by the application using the function *vkGetBufferDeviceAddressEXT*. Similar functions exist to query other types of resources for their device addresses. The layer would need to intercept the *vkGetDescriptorEXT* and *vkGet*DeviceAddressEXT* functions to keep track of the contents of descriptor buffers and associate the device addresses of resources in the descriptors with their corresponding Vulkan objects. The binding of descriptor buffers to command buffers is accomplished using the functions *vkCmdBindDescriptorBuffersEXT* and *vkCmdSetDescriptorBufferOffsetsEXT*, which the layer intercepts to track what descriptor buffers are bound to each command buffer and what offsets within the descriptor buffers are used before each pipeline invocation.

Descriptor indexing. Descriptor indexing [59] was a Vulkan extension that has now become a core feature in Vulkan 1.2. Descriptor indexing enables two features to Vulkan applications, but only one of them is relevant to the analysis of descriptor sets. This feature is called Update-After-Bind. It allows applications to update individual descriptors in a descriptor set without invalidating existing command buffers that have bound the descriptor set. To enable this feature, individual descriptor bindings can be flagged as update-after-bind when creating descriptor-set layouts. With Update-After-Bind there is no need to re-record command buffers when descriptor sets are modified. When a de-

scriptor set has update-after-bind descriptor bindings, the layer does not need to perform an analysis of descriptor-set usage within each command buffer during or after command-buffer recording because command buffers are not necessarily invalidated when descriptor-set updates are made to descriptor sets that are bound to them. The analysis of descriptor-set usage within command buffers are instead deferred to just before command-buffer submission time, as this is the latest point in time when the application may update descriptor sets before they are used by the GPU.

In addition to Update-After-Bind, an additional flag called *VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT* can be set. Like Update-After-Bind, this flag allows applications to update individual descriptors in a descriptor set without invalidating existing command buffers that have bound the descriptor set. The difference is that descriptors may be updated even while a command buffer is currently pending execution on the GPU, as long as the descriptors being updated are not used by the command buffer in pipeline shader stages. The *VK_DESCRIPTOR_BINDING_UPDATE_UNUSED_WHILE_PENDING_BIT* can be paired with the flag *VK_DESCRIPTOR_BINDING_PARTIALLY_BOUND_BIT* to allow the of updating of descriptors even while a command buffer is currently executing, as long as the descriptors being updated are not dynamically used by any active shader invocations. These flags complicate descriptor-set profiling and evaluation because descriptors in descriptor sets are no longer static after command-buffer recording or submission. However, the greater flexibility of being able to modify descriptor sets at more points in time may make the purpose of a descriptor-set usage profiling layer less relevant because applications need far fewer descriptor sets to supply resources to pipelines.

Push descriptors. The Vulkan API extension *VK_KHR_push_descriptor* [61] adds a way to supply descriptors to pipelines inline within command buffers without the need for descriptor sets. Push descriptors are similar to push constants in that they are provided by a command before a pipeline invocation. Their memory is managed internally by the command buffer and are

therefore limited in size. The layer can track the usage of push descriptors in command buffers by intercepting the *vkCmdPushDescriptorSetKHR* function. The *vkCmdPushDescriptorSetKHR* function takes in a pipeline layout to be programmed to, a set number for the set in the pipeline layout to be updated, and an array of *VkWriteDescriptorSet* structures describing the descriptors to be pushed. Push descriptors can not contain dynamic descriptors.

4.6.2 Automatic Uniform Value Specialization

Vulkan layers have the potential to automatically employ profile-guided optimization by modifying the behavior of Vulkan API function calls and making new Vulkan API function calls on behalf of the application. One possible optimization is to automatically specialize uniform data values in pipelines' shader stages based on the data values that are most frequently used or are runtime-constant. However, there are several challenges to implementing automatic uniform data value specialization in a Vulkan layer.

Pipeline compilation. A well-known problem in video games is the long compilation times for pipelines [7], [12]. Games may have thousands of pipelines for rendering different objects with varying materials and visual effects. Failing to compile a pipeline (or several pipelines) in time for when it is needed to render a frame results in a noticeable delay described as "stuttering" or a sudden drop in application frame rate that negatively impacts the user experience. Game engines and the Vulkan API have several techniques to mitigate this problem, such as pre-compiling pipelines during game loading screens, using multi-threading to preemptively compile pipelines in the background [46], using pipeline caches to save compiled pipelines for later reuse [4], and using pipeline derivatives to compile similar pipelines more quickly [4]. Despite these techniques, the problem of pipeline compilation time remains a significant challenge because different combinations of, for example, material properties may require a different pipeline to be compiled. The total number of possible variations of pipelines may also be infeasible to pre-compile or to store in a pipeline cache.

A Vulkan layer that automatically specializes uniform data values must be able to compile pipelines in a timely manner to avoid stuttering. Integrating the compilation of specialized pipelines is a non-trivial task that requires the layer to have knowledge of the application's pipeline compilation process and the ability to modify the pipeline compilation process to compile specialized pipelines in addition to the original pipelines. Depending on the number of values that need to be specialized, a large number of specialized pipelines may be compiled — scaling exponentially with the number of values that needs to be specialized per variable in uniform buffers.

Maintaining correctness. Automatic uniform value specialization can lead to incorrectly-rendered frames if an incorrect specialized pipeline is chosen to replace the original pipeline in a pipeline invocation. Maintaining correctness means that the layer must be able to determine when it is safe to specialize a pipeline and when it is not, while also ensuring that the specialized pipeline is correct for the uniform data values that are known at the time a pipeline is to be executed. One area where an automatic uniform value specialization layer may fail to maintain correctness is during command-buffer recording when the layer is unable to determine the contents of uniform buffers that will be used by a pipeline.

Command buffers are immutable once they have been recorded. Descriptor sets with update-after-bind descriptor bindings may be updated after command-buffer recording and before command-buffer submission. The contents of the uniform buffers referenced by descriptors are also mutable before the uniform buffers are used by an executing pipeline. These factors make it impossible to replace pipeline invocations with invocations of specialized pipelines during the typical time an application performs command-buffer recording.

An approach to tackling the command-buffer recording problem is for the layer to make blueprints of command buffers that bind specializable pipelines. Just before the original command buffer is submitted, the layer can record a new command buffer using the blueprint while substituting the original

pipelines with correctly-chosen specialized pipelines based on values in the uniform buffers. The original command buffer to be submitted is replaced with the new command buffer that binds the specialized pipelines.

Major performance issues arise from this approach. Reading the contents of uniform buffers (or intermediate transfer buffers in the case of uniform buffers in non-host-visible memory) to determine the correct specialized pipeline to use is costly, and further exacerbated when the buffers reside in memory that does not have the host-cached property to enable fast reads. Applications may also reuse command buffers multiple times to avoid having to re-record them, which makes this approach negate the benefits of having reusable command buffers.

Eliminating memory accesses. Substituting pipeline invocations with invocations of specialized pipelines may enable the layer to eliminate writes to uniform buffer memory for the uniform data values that are specialized. However, the layer has no knowledge of which uniform buffers created by the application are used for which pipelines until command buffer recording. The layer also has no knowledge of the values that applications intend to write to uniform buffer memory until the write actually occurs (if using shadow memory to track writes) or until the layer reads the contents of the uniform buffer memory to check the contents. This lack of knowledge about the application prevents the layer from being able to discard writes to uniform buffer memory that are not needed by the specialized pipelines. Thus, performance improvements, if any, would be from minimizing uniform buffer memory accesses from shader execution on the GPU as opposed to eliminating writes to uniform buffer memory from the CPU.

All the issues mentioned with regards to pipeline compilation, maintaining correctness, and eliminating memory accesses make automatic uniform value specialization a challenging and impractical optimization to implement in a Vulkan layer designed to be application-agnostic. Uniform value specialization is better suited for application-specific optimizations that are implemented by the application itself, as the application has the necessary knowledge of its own

pipeline compilation process, uniform buffer usage, and uniform data values to make informed decisions about when and how to specialize pipelines and reduce uniform buffer memory accesses.

4.6.3 Other Pipeline Types

Uniform buffers are not exclusive to graphics pipelines and can be used by other types of pipelines in Vulkan such as compute pipelines, ray-tracing pipelines, and mesh-shading pipelines.

Compute pipelines. Compute pipelines are used for general-purpose computation on the GPU. Compute pipelines, unlike graphics pipelines, do not have a 1-to-1 mapping of threads to work items such as vertices or fragments. Compute pipelines are composed of a single compute-shader stage that is executed by multiple threads organized into *workgroups*. A workgroup is a group of threads that execute the compute shader and have access to a shared workgroup-local memory to enable communication between threads in the workgroup. When a compute pipeline is invoked using the *vkCmdDispatch* function, the application specifies the number of workgroups to spawn to execute the compute shader. Within compute shaders, each thread can query its own ID within a workgroup and the ID of the workgroup it belongs to. The inputs and outputs of compute pipelines are defined entirely by the application based on the compute pipeline's layout and the implementation of the compute shader itself.

To support descriptor set and uniform buffer profiling on compute pipelines, the layer would need to intercept the *vkCmdDispatch* function to identify compute pipeline invocations in command buffers. The rest of the layer would function the same as for graphics pipelines. Compute pipelines exhibit different resource usage patterns compared to graphics pipelines due to their general-purpose nature, and may therefore have more or less opportunities for optimization with respect to descriptor set and uniform buffer usage depending on their use case.

Mesh-shading pipelines. The mesh-shading pipeline is a relatively recent pipeline type introduced in the Vulkan API that is an alternative to the graphics pipeline. In addition to the vertex and fragment shader stages of a graphics pipeline, a graphics pipeline also has the optional tessellation control, tessellation evaluation, and geometry shader stages. These optional stages grant applications the ability to modify the geometry of objects to be rendered after vertex shading and before rasterization. Modifications possible with the optional stages include subdividing the geometry to produce more vertices (i.e., tessellation) and transforming geometric primitives into different primitives, possibly with more, or fewer, vertices.

The mesh-shading pipeline replaces the vertex, tessellation control, tessellation evaluation, and geometry shader stages with a task shader stage and a mesh shader stage. Unlike the shader stages that they replace in the graphics pipeline, task and mesh shaders function much like compute shaders in that they are executed in terms of workgroups and have access to shared workgroup-local memory. The task shader stage emits zero or more mesh-shader workgroups, while the mesh shader stage processes the workgroups emitted by the task shader stage and produces triangles for the rasterization stage of the pipeline. The compute-shader-like nature of task and mesh shader stages allow for more predictable performance across different hardware due to the execution model more closely resembling how modern GPUs operate [18]. The flexibility offered by the mesh-shading pipeline enables applications to perform the same tasks as the vertex, tessellation, and geometry shader stages of the graphics pipeline if desired. The mesh-shading pipeline also enables new techniques that were not possible with the graphics pipeline such as meshlet rendering, where a complex object (mesh) is divided into small groups of triangles called meshlets that are processed together. The task shader would be responsible for generating mesh-shader workgroups to process each meshlet, and the mesh shader would be responsible for producing triangles from each meshlet for rasterization. Notably, the task shader can be used to cull meshlets that are not visible to the camera by simply omitting mesh-shader workgroups for those meshlets, thereby reducing the amount of work that the

rest of the pipeline needs to do [25]. The meshlets themselves are pre-generated as opposed to being generated on-the-fly by the GPU as is done in a graphics pipeline before vertex shading, which allows an application to employ different meshlet generation strategies to optimize aspects such as vertex cache usage on the GPU or enable more opportunities for culling in the task shader [21].

To support descriptor set and uniform buffer profiling on mesh-shading pipelines, the layer would need to intercept the *vkCmdDrawMeshTasksEXT* function to identify mesh-shading pipeline invocations in command buffers. The rest of the layer would function the same as for graphics pipelines. Mesh shading pipelines, like compute shader pipelines, may exhibit different resource usage patterns compared to graphics pipelines. An application may create mesh-shading pipelines that mirror the functionality of graphics pipelines, or to use mesh shading pipelines to implement alternative rendering techniques.

Ray-tracing pipelines. Ray-tracing pipelines are a type of pipeline in the Vulkan API that enables applications to perform hardware-accelerated ray-tracing on the GPU. Ray-tracing pipelines are also seldomly used to render single objects or visual effects like graphics pipelines typically do. Instead, a single ray-tracing pipeline invocation is often enough to render entire scenes with complex lighting and shadowing effects. To accomplish this feat, ray-tracing pipelines reference a *shader-binding table* that maps objects in the scene to shaders that are executed when rays intersect the objects. Due to the wide variety of shaders, uniform buffers are less used in ray-tracing pipelines compared to the graphics pipeline since there is less opportunity for reuse of uniform data across a wide variety of different shaders. Instead of uniform buffers, constant data specific to each object may be stored in shader-binding table records alongside their shaders.

To support descriptor-set and uniform-buffer profiling on ray-tracing pipelines, the layer would need to intercept the *vkCmdTraceRays* function to identify ray-tracing pipeline invocations in command buffers. The rest of the layer would function the same as for graphics pipelines.

4.6.4 GPU-Driven Rendering

A typical application iterates over all objects in a scene and performs a draw call for each object individually, while binding appropriate descriptors and pipelines prior to each draw call according to the object's properties. This procedure is quite inefficient due to the overheads of recording multiple commands on the CPU to be sent to the GPU, and the frequent altering of GPU state to render each object using different descriptors and pipelines.

As GPUs became more general purpose, game developers began to devise ways to offload as much of the rendering process to the GPU as possible. The general idea is to maximize the amount of work that the GPU can do independently of the CPU by loading the entire scene into GPU memory and letting the GPU decide how to render the scene with minimal CPU intervention. This shift towards offloading the rendering process to the GPU is called GPU-driven rendering. GPU-driven rendering was first popularized by the game Assassin's Creed Unity from Ubisoft and has since been gaining more traction in the video-game industry [5], [16].

The key enabler of GPU-driven rendering is the introduction of indirect draw commands in Vulkan 1.2. Draw commands typically take a number of parameters such as the offset into the vertex buffer to start processing vertices, and the number of vertices to process. An indirect draw command instead takes in a buffer that contains the parameters which the GPU reads from to execute the draw command (or multiple draw commands). The key aspect of this buffer is that it may be modified by the GPU within shaders such as a compute shader from a compute pipeline — enabling use of compute shaders to generate draw calls and perform tasks such as frustum and occlusion culling to optimize scene rendering.

To support descriptor set and uniform buffer profiling on applications using indirect draw commands, the layer would need to intercept the *vkCmdDrawIndirect* function. The rest of the layer would function the same as for graphics pipelines. However, a consequence of indirect draw commands is that although the GPU can now generate its own draw commands, the graphics

pipeline used for the draw commands remains fixed. Therefore, the graphics pipeline must be generalized to handle a wide variety of different types of objects. To handle this generality, game engines have adopted a *bindless* model for storing and using resource descriptors used in graphics pipelines [54]. No longer are there uniform buffers or push constants associated with each object. All per-object resources are aggregated into large shader-storage buffers that are indexed into by the GPU to access the resources needed to render specific objects. The bindless model allows the GPU to access resources without the need for the CPU to bind new descriptor sets for each object or set of objects to be rendered. In a typical bindless model, descriptors are consolidated into a single monolithic descriptor set that is bound once at the beginning of rendering. Modifications to descriptors within the descriptor set are enabled with the use of the descriptor indexing feature of Vulkan 1.2 to update descriptors on-the-fly even during command buffer execution. As a consequence of the bindless model, there are fewer opportunities for the optimization of descriptor set and uniform buffer usage in applications using a GPU-driven renderer.

4.7 Conclusion

Vulkan's layer system enables the creation of a powerful tool for developers to observe and capture the usage of resources in Vulkan applications. The Vulkan layer proposed in this chapter provides insights into the usage of descriptor sets and uniform buffers in Vulkan applications' command buffers and pipelines.

By taking advantage of the low-level control provided by Vulkan, developers are able to optimize their applications in ways previously not possible with older APIs like OpenGL, such as the restructuring of pipeline layouts and descriptor sets to minimize memory management and GPU state-change overheads. Optimizations through value specialization of OpenGL's uniform variables is a technique that can be extended to Vulkan's uniform buffers to eliminate memory accesses and reduce the amount of data transferred to the GPU. Further optimizations tying together uniform buffers and descriptors can

be achieved by utilizing Vulkan-specific features such as push constants and inline uniform blocks to reduce the amount of indirection required to access uniform buffer data.

However, with the rapidly-evolving landscape of computer graphics, the importance of descriptor set and uniform buffer optimizations diminishes as new hardware and API features are implemented to make shader resource management more flexible and efficient. The industry is shifting back to the use of monolithic descriptor sets due to hardware and API features enabling "bindless", dynamic and flexible updates to individual descriptors in a descriptor set. Much of the data that used to reside in per-object uniform buffers are now aggregated into large shader storage buffers to facilitate GPU-driven rendering and bindless descriptor designs. These factors render some of the optimizations proposed in this chapter less applicable due to changes in the way shader resources are managed in modern Vulkan applications.

Chapter 5

Conclusion

This thesis expands upon the list of applications and limitations of the Vulkan layer system for profiling Vulkan applications to better understand execution behaviors affecting their performance.

Chapter 3 introduced the ReRay toolchain and workflow that builds on top of the existing work of Vulkan Vision and RayScope. ReRay is used for evaluating, debugging, and optimizing Vulkan ray-tracing applications by capturing and analyzing ray-trace execution data generated from automatic GPU shader-code instrumentation. Using ReRay, hardware execution behavior such as warp execution order and unnecessary intersection tests are revealed. The chapter also demonstrates and discusses how the ReRay toolchain can be used to rapidly evaluate potential changes to the application's renderer before the developer carries out a costly implementation, such as the implementation of warp repacking and how it impacts SIMT efficiency estimates in ray-tracing pipeline execution traces. Limitations of ReRay are also discussed, such as the inability to analyze ray-tracing pipeline execution traces from applications using NVIDIA's Shader Execution Reordering (SER) feature due to issues stemming from assumptions made in the Vulkan-Vision framework for obtaining unique identifiers for warps without the use of a proprietary Vulkan extension. Even with the use of a proprietary extension, SER remains elusive to study due to unexpected behavior of the extension when used to query warp identifiers before warp repacking.

Chapter 4 proposed a Vulkan profiling layer to observe the usage of shader

resource descriptors and uniform buffers in Vulkan applications to leverage prior work on uniform-buffer value specialization and to examine the relevance of prior work on descriptor set optimization. The proposed layer provides insights into the resource management procedures of Vulkan applications, and can inform developers about ways to optimize their applications through restructuring of descriptor sets and uniform buffers to reduce memory access and GPU state-change overheads. The chapter mentions inefficiencies of the Vulkan layer when reading uniform buffer data from Vulkan applications, as buffer data is written to through void pointers and not through the Vulkan API. This thesis also discussed the practical limitations of the implementation of an application-agnostic Vulkan layer to automatically perform shader resource optimizations on behalf of the application. Despite the power of application-agnostic Vulkan layers to modify application behavior, these layers cannot alter the shader resource management system used by an application. Finally, the chapter also discusses advancements in the Vulkan API, hardware features, and the design of graphics renderers that complicate the design of the Vulkan profiling layer, and makes prior work in descriptor set optimization and uniform buffer value specialization less relevant for modern Vulkan applications.

There are several directions for future work that may further explore the capabilities and utilities of the Vulkan layer system. As mentioned in Chapter 3, new methods may be explored for obtaining unique warp identifiers for applications that use SER or other hardware warp repacking techniques. Such a method would allow further study of how SER performs warp repacking and affects the execution traces of ray-tracing pipelines. Other aspects of the Vulkan API can also be examined and evaluated with the Vulkan layer system. For instance, synchronization is a critical aspect of the Vulkan API to ensure that work (e.g., buffer copies, pipeline execution) starts and completes execution in the correct order. Synchronization is necessary to ensure that work items do not read or write to the same memory locations at the same time, or execute before dependent work items have completed. A Vulkan layer may be developed to examine existing dependencies between work items and

identify potential synchronization bottlenecks from the overuse or misuse of Vulkan synchronization primitives. Performance optimizations may be possible by reducing the number of synchronization primitives used, reordering work items to reduce the number of synchronization primitives required, or using alternative synchronization primitives to reduce stalling. With the flexibility of the Vulkan layer system, many aspects of Vulkan API usage can be examined to analyze the performance of Vulkan applications and identify potential optimizations.

References

[1] AMD. "Amd radeon raytracing analyzer." (Mar. 2022), [Online]. Available: https://www.youtube.com/watch?v=qae_skv1GsA.

[2] AMD. "Radeon gpu profiler," AMD. (2024), [Online]. Available: https://gpuopen.com/rgp/.

[3] *Amd rdna performance guide*, en-GB. [Online]. Available: https://gpuopen.com/learn/rdna-performance-guide/.

[4] D. Archard, *Pipeline state object caching*, Presentation at SIGGRAPH 2016, 2016. [Online]. Available: https://www.khronos.org/assets/uploads/developers/library/2016-siggraph/3D-BOF-SIGGRAPH_Jul16.pdf.

[5] W. Bahnassi, *Advancing gpu-driven rendering with work graphs in direct3d*, en-US, Mar. 2024. [Online]. Available: https://developer.nvidia.com/blog/advancing-gpu-driven-rendering-with-work-graphs-in-direct3d-12/.

[6] L. Bavoil. "Optimizing dx12/dxr gpu workloads using nsight graphics: Gpu trace and the peak-performance-percentage (p3) method (presented by nvidia)." (2019), [Online]. Available: https://www.gdcvault.com/play/1026202/Optimizing-DX12-DXR-GPU-Workloads.

[7] S. Butler, *What is shader compilation and why does it make pc games stutter?* en, Nov. 2022. [Online]. Available: https://www.howtogeek.com/846514/what-is-shader-compilation-and-why-does-it-make-pc-games-stutter/.

[8] B. Calder, P. Feller, and A. Eustace, "Value profiling," in *Proceedings of 30th Annual International Symposium on Microarchitecture*, IEEE, 1997, pp. 259–269.

[9] T. Cheblokov. "Advanced api performance: Pipeline state objects." (Oct. 2023), [Online]. Available: https://developer.nvidia.com/blog/advanced-api-performance-pipeline-state-objects/.

[10] L. Crawford, "Shader optimization and specialization," Ph.D. dissertation, University of Edinburgh, 2022.

[11] L. Crawford and M. O'Boyle, "Specialization opportunities in graphical workloads," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, IEEE, 2019, pp. 272–283.

[12] N. Evanson, *Shader compilation and why it causes stuttering, explained*, en-US, Feb. 2023. [Online]. Available: https://www.techspot.com/article/2629-shader-compilation-explained/.

[13] T. Fautre. "Ray tracing in vulkan." (2024), [Online]. Available: https://github.com/GPSnoopy/RayTracingInVulkan.

[14] 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, ISBN: 9781450311946. DOI: 10.1145/2159616.2159628. [Online]. Available: https://doi.org/10.1145/2159616.2159628.

[15] H. Gruen, C. Benthin, and S. Woop, "Sub-triangle opacity masks for faster ray tracing of transparent objects," *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, vol. 3, no. 2, pp. 1–12, 2020.

[16] U. Haar and S. Aaltonen, *Gpu-driven rendering pipelines*, 2015. [Online]. Available: https://www.advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf.

[17] K. Hans. "Vkd3d-proton." en. (2020), [Online]. Available: https://github.com/HansKristian-Work/vkd3d-proton.

[18] S. Hargreaves. "Reinventing the geometry pipeline: Mesh shaders in directx 12." en. (Mar. 2020), [Online]. Available: https://www.youtube.com/watch?v=CFXKTXtil34.

[19] Y. He, T. Foley, T. Hofstee, H. Long, and K. Fatahalian, "Shader components: Modular and high performance shader development," en, *ACM Transactions on Graphics*, vol. 36, no. 4, pp. 1–11, Aug. 2017, ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3072959.3073648. [Online]. Available: https://dl.acm.org/doi/10.1145/3072959.3073648.

[20] J. L. Hintze and R. D. Nelson, "Violin plots: A box plot-density trace synergism," *The American Statistician*, vol. 52, no. 2, pp. 181–184, 1998.

[21] M. B. Jensen, J. R. Frisvad, and J. A. Bærentzen, "Performance comparison of meshlet generation strategies," *Journal of Computer Graphics Techniques (JCGT)*, vol. 12, no. 2, pp. 1–27, Dec. 2023, ISSN: 2331-7418. [Online]. Available: http://jcgt.org/published/0012/02/01/.

[22] B. Karlsson. "Renderdoc." (2024), [Online]. Available: https://renderdoc.org/.

[23] P. Kelly, Y. O'Donnell, K. Elst, J. Cañada, and E. Hart, "Ray tracing in fortnite," in Aug. 2021, pp. 791–821, ISBN: 978-1-4842-7184-1. DOI: 10.1007/978-1-4842-7185-8_48.

[24] J. Kessenich, G. Sellers, and D. Shreiner, *OpenGL Programming Guide: The official guide to learning OpenGL, version 4.5 with SPIR-V*. Addison-Wesley Professional, 2016.

[25] C. Kubisch. "Introduction to turing mesh shaders." en. (Sep. 2018), [Online]. Available: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/.

[26] 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, ISBN: 9781450321358. DOI: 10.1145/2492045.2492060. [Online]. Available: https://doi.org/10.1145/2492045.2492060.

[27] S. M. Laine, T. O. Aila, and T. T. KARRAS, *Relative encoding for a block-based bounding volume hierarchy*, en, Jul. 2018. [Online]. Available: https://patents.google.com/patent/US10032289B2/en.

[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, 15 Jul. 2014. DOI: https://doi.org/10.2478/cait-2014-0018. [Online]. Available: https://content.sciendo.com/view/journals/cait/14/2/article-p38.xml.

[29] L. Liu, W. Chang, F. Demoullin, *et al.*, "Intersection prediction for accelerated gpu ray tracing," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21, Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 709–723, ISBN: 9781450385572. DOI: 10.1145/3466752.3480097. [Online]. Available: https://doi-org.login.ezproxy.library.ualberta.ca/10.1145/3466752.3480097.

[30] LunarG, Inc. "Gfxreconstruct api capture and replay - vulkan." (2024), [Online]. Available: https://vulkan.lunarg.com/doc/sdk/1.3.280.0/windows/capture_tools.html.

[31] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, "On ray reordering techniques for faster gpu ray tracing," in *Symposium on Interactive 3D Graphics and Games*, 2020, pp. 1–9.

[32] D. Meister, S. Ogaki, C. Benthin, M. J. Doyle, M. Guthe, and J. Bittner, "A survey on bounding volume hierarchies for ray tracing," in *Computer Graphics Forum*, Wiley Online Library, vol. 40, 2021, pp. 683–712.

[33] Microsoft. "Pix on windows." (2021), [Online]. Available: https://devblogs.microsoft.com/pix/.

[34] M. R. Nuno Subtil and I. Fedorov. "Tips and tricks: Vulkan dos and don'ts." en. (Jun. 2019), [Online]. Available: https://developer.nvidia.com/blog/vulkan-dos-donts/.

[35] NVIDIA. "Accelerate applications on nvidia ampere." en. (2018), [Online]. Available: https://developer.nvidia.com/nvidia-ampere.

[36] NVIDIA. "Nvidia turing." en. (May 2021), [Online]. Available: https://www.nvidia.com/en-in/geforce/turing/.

[37] NVIDIA. "Improve shader performance and in-game frame rates with shader execution reordering." en. (2022), [Online]. Available: https://developer.nvidia.com/blog/improve-shader-performance-and-in-game-frame-rates-with-shader-execution-reordering/.

[38] NVIDIA. "Nvidia nsight systems," NVIDIA. (2022), [Online]. Available: https://developer.nvidia.com/nsight-systems.

[39] NVIDIA. "Nvidia system management interface," NVIDIA. (2022), [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface.

[40] NVIDIA. "Shader execution reordering." en. (2022), [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/gameworks/ser-whitepaper.pdf.

[41] NVIDIA. "Nvidia nsight graphics," NVIDIA. (2024), [Online]. Available: https://developer.nvidia.com/nsight-graphics.

[42] D. Pankratz, "Data-driven analysis and design of vulkan ray-tracing applications using automatic instrumentation," 2021.

[43] 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. DOI: 10.1109/CGO51591.2021.9370320.

[44] R. Rangan, M. W. Stephenson, A. Ukarande, S. Murthy, V. Agarwal, and M. Blackstein, "Zeroploit: Exploiting zero valued operands in interactive gaming applications," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 3, pp. 1–26, 2020.

[45] T. Schiesser. "Ray tracing & dlss with the geforce rtx 3080." en. (Oct. 2020), [Online]. Available: https://www.techspot.com/article/2109-nvidia-rtx-3080-ray-tracing-dlss/.

[46] M. Schott, *Vulkan multi-threading*, en, Apr. 2016. [Online]. Available: https://developer.nvidia.com/sites/default/files/akamai/gameworks/blog/munich/mschott_vulkan_multi_threading.pdf.

[47] G. Sellers and J. Kessenich, *Vulkan programming guide: The official guide to learning vulkan*. Addison-Wesley Professional, 2016.

[48] J. Shepard. "Visualizing gl_nv_shader_sm_builtins." en. (2020), [Online]. Available: https://wunkolo.github.io/post/2020/02/visualizing-gl_nv_shader_sm_builtins/.

[49] P. Shirley, *Ray Tracing in One Weekend*, S. Hollasch and T. David Black, Eds. 2020. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[50] 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, ISBN: 978-3-03868-025-3. DOI: 10.2312/vmv.20161356.

[51] G. Simons, S. Herholz, V. Petitjean, *et al.*, "Applying visual analytics to physically based rendering," in *Computer Graphics Forum*, Wiley Online Library, vol. 38, 2019, pp. 197–208. DOI: 10.1111/cgf.13452.

[52] P. Slusallek, P. Shirley, W. Mark, G. Stoll, and I. Wald, "Introduction to real-time ray tracing," in *ACM SIGGRAPH 2005 Courses*, ser. SIGGRAPH '05, Los Angeles, California: Association for Computing Machinery, 2005, 1–es, ISBN: 9781450378338. DOI: 10.1145/1198555.1198740. [Online]. Available: https://doi.org/10.1145/1198555.1198740.

[53] The Khronos Group, Inc. "Vulkan sdk, tools and drivers are ray tracing ready." (Feb. 2020), [Online]. Available: https://www.khronos.org/news/press/vulkan-sdk-tools-and-drivers-are-ray-tracing-ready.

[54] The Khronos Group, Inc. "Descriptor indexing." (2024), [Online]. Available: https://docs.vulkan.org/samples/latest/samples/extensions/descriptor_indexing/README.html.

[55] The Khronos Group, Inc. "Memory allocation." (2024), [Online]. Available: https://docs.vulkan.org/guide/latest/memory_allocation.html.

[56] The Khronos Group, Inc. "Push constants." (2024), [Online]. Available: https://docs.vulkan.org/guide/latest/push_constants.html.

[57] The Khronos Group, Inc. "Timestamp queries." (2024), [Online]. Available: https://docs.vulkan.org/samples/latest/samples/api/timestamp_queries/README.html.

[58] The Khronos Group, Inc. "Vk_ext_descriptor_buffer." (2024), [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_descriptor_buffer.html.

[59] The Khronos Group, Inc. "Vk_ext_descriptor_indexing." (2024), [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_EXT_descriptor_indexing.html.

[60] The Khronos Group, Inc. "Vk_ext_inline_uniform_block." (2024), [Online]. Available: https://docs.vulkan.org/guide/latest/extensions/VK_EXT_inline_uniform_block.html.

[61] The Khronos Group, Inc. "Vk_khr_push_descriptor." (2024), [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_push_descriptor.html.

[62] The Khronos Group, Inc. "Vulkan specification." (2024), [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html.

[63] The Khronos Group, Inc. "Vulkan validation layers." (2024), [Online]. Available: https://github.com/KhronosGroup/Vulkan-ValidationLayers.

[64] Unity Technologies. "Performance profiling tips for game developers." en. (), [Online]. Available: https://unity.com/how-to/best-practices-for-profiling-game-performance.

[65] D. G. Van Antwerpen, "Unbiased physically based rendering on the gpu," 2011. [Online]. Available: http://resolver.tudelft.nl/uuid:4a5be464-dc52-4bd0-9ede-faefdaff8be6.

[66] D. Voorhies, "I.8 - space-filling curves and a measure of coherence," in *Graphics Gems II*, J. ARVO, Ed., San Diego: Morgan Kaufmann, 1991, pp. 26–30, ISBN: 978-0-08-050754-5. DOI: https://doi.org/10.1016/B978-0-08-050754-5.50018-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780080507545500189.

[67] J. Zink, M. Pettineo, and J. Hoxley, *Practical rendering and computation with Direct3D 11*. AK Peters/CRC Press, 2016.