Vulkan Vision: Ray Tracing Workload Characterization using Automatic Graphics Instrumentation

David Pankratz
University of Alberta, CA
pankratz@ualberta.ca

Tyler Nowicki
Huawei Technologies Canada
tyler.bryce.nowicki@huawei.com

Ahmed Eltantawy
Huawei Technologies Canada
ahmed.eltantawy@huawei.com

José Nelson Amaral
University of Alberta, CA
jamaral@ualberta.ca

***Abstract*—While there are mature performance monitoring, profiling and instrumentation tools to help understanding the dynamic behaviour of general-purpose GPU applications, the abstract programming models of graphics applications have limited the development of such tools for graphics. This paper introduces Vulkan Vision (V-Vision), a framework for collecting detailed GPU execution data from Vulkan applications to guide hardware-informed improvements. A core contribution of V-Vision is providing out-of-the-box data collection for capturing complete dynamic warp and thread execution traces. V-Vision also provides analyses for the follow purposes: identifying and visualizing application hotspots to guide optimization, characterizing application behaviour and estimating the effect of architectural modifications. This paper demonstrates the potential for these analyses in applications that utilize the recent ray-tracing extension in Vulkan and describes new insights about the applications and the underlying hardware.**

***Index Terms*—Vulkan, Ray Tracing, Profiling, Instrumentation**

I. Introduction

Performance and quality of real-time graphics applications — such as gaming, augmented and virtual reality — have a direct impact on end-user experience and thus are considered as a key differentiator in several markets, including gaming consoles, mobile devices, and high-end desktops. To achieve high performance, the process of translating a scene description into an image on a screen has been long standardized in graphics Application Programming Interfaces (APIs) — such as Direct3D, and OpenGL [1], [2] — as a monolithic GPU kernel, referred to as a graphics pipeline. Characterizing the performance bottlenecks of a graphics pipeline is challenging because the graphics programming model is hardware agnostic making it more obscure than a low-level programming model such as Compute Unified Device Architecture (CUDA) [3]. CUDA application developers benefit from numerous program characterization tools using profiling and instrumentation [4]–[6]. The infrastructure for graphics applications is less developed but the need to create high quality and performant GPU code remains.

Recently developed, low-level graphics APIs, such as Vulkan [7] and Metal [8] enable efficient graphics pipeline implementations. A graphics pipeline consists of multiple programmable stages, known as shaders. Data movement and manipulation across these stages is defined abstractly by the API and the flow varies according to the implementations. Code and code generation quality for programmable shaders are a key contributor to the performance of a graphics pipeline.

TABLE I: Landscape of graphics profiling.

Tool	Open Source	Shader Instrumentation	Automatic Profiling	Target Specific	API
NSight Graphics [10]	No	No	Yes	Yes	DirectX OpenGL Vulkan
Intel GPA [11]	No	No	Yes	Yes	DirectX OpenGL Vulkan Metal
PIX [12]	No	No	Yes	No	DirectX
Strengert *et al.* [13]	Yes	Yes	No	No	OpenGL
V-Vision	Yes	Yes	Yes	No	Vulkan

In comparison to traditional software, where call graph traces reveal the flow of execution, the implicit data and control flow in a graphics pipelines is obscure to application and system developers. Inscrutability is compounded by recent extensions, such as ray tracing, with more complex pipelines. Thus, system developers have less support to improve performance in a graphics pipeline. The programmable shaders significantly influence performance, yet the run-time behaviour (e.g. execution trace, control flow, hotspots) of these shaders is not provided. There is a need for instrumentation tools to analyze graphics pipelines. The efficacy of instrumentation has been recognized in the compute domain of GPUs that use APIs such as CUDA [3], and OpenCL [9]. No existing framework facilitates shader instrumentation for graphics applications profiling, as shown in Table I. This paper introduces V-Vision, an open-source framework to capture fine-grained GPU execution data, independently of vendor-specific compilers.

The Vulkan API is a widely adopted graphics and compute API that minimizes driver overhead. This API elides default

error checking and instead offers an optional validation layer to provide debugging capabilities. Validation layers support analysis and instrumentation of Standard Portable Intermediate Representation (SPIR-V) shader code. SPIR-V is a vendor-independent pre-compiled intermediate representation for device code required by Vulkan [14]. The existing support for SPIR-V instrumentation provides only error checking and the printf debugging extension. V-Vision extends the instrumentation infrastructure to support generic application profiling.

V-Vision extends the instrumentation framework with a set of ready-to-use instrumentation primitives, instrumentation utilities, and instrumentation analytics.

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

The output of instrumentation utilities are consumed by the analytics in V-Vision to characterize runtime behaviour and identify performance bottlenecks, such as serialized indirect function calls. The analytics include simulating improvements to performance bottlenecks, such as thread compaction to reduce the impact of serialization [15]. Lastly, the analyses produce visualizations within the shader code to streamline refactoring to solve performance issues.

The contributions of this paper are:

1) V-Vision, a framework for graphics shader instrumentation in Vulkan applications. It extends Vulkan validation layer instrumentation capabilities to enable meaningful performance and behaviour-centric instrumentation. V-Vision is open source under MindInsight at https://gitee.com/mindspore/mindinsight.
2) A set of instrumentation utilities enabling automatic out-of-the-box instrumentation.
3) An analysis of raytracing applications using V-Vision.

The rest of this paper is organized as follows: in Section II provides necessary background. Section III explains the architecture of V-Vision. In Section IV, we discuss the instrumentation primitives provided in V-Vision. In Section V, we characterize ray-tracing Vulkan applications using instrumentation utilities. Section VI evaluates V-Vision's overhead and compares binary and Intermediate Representation (IR) instrumentation. Section VII discusses related work. Finally, we conclude in Section VIII.

II. Background

GPU Execution Model: Graphics applications are naturally parallel because each stage in the graphics pipeline is executed over a multi-dimensional buffer of independent elements allowing each element to be evaluated by its own thread. For example, each pixel in the frame is independent allowing them to be evaluated by separate threads running in parallel. Threads are grouped into *w*arps, where threads in each warp execute in lockstep on different data on Single Instruction, Multiple Data (SIMD) hardware units. This execution paradigm reduces instruction fetching, decoding, and scheduling overheads and enables coalesced memory accesses. Control flow divergence reduces the efficiency of Single Instruction, Multiple Thread (SIMT) execution due to threads becoming inactive in portions of the execution path. In the worst case of divergence, the execution of threads in a warp is completely serialized. The utilization of SIMD hardware units can be assessed using SIMT efficiency, the average percentage of active threads per warp.

OpenGL Shading Language (GLSL) and SPIR-V Vulkan requires SPIR-V for each programmable shader stage and the driver is responsible for just-in-time compiling SPIR-V to machine code. SPIR-V is generated by compiling OpenGL Shading Language (GLSL) shaders, although other shader languages can also be compiled to SPIR-V. GLSL specifies the builtin variables and functions available within each shader stage. Vulkan validation layers must account for the builtins that are unique to each shader stage.

Ray Tracing in GPUs: Ray tracing is a computationally expensive workload that improves the visual quality of graphics applications. Only recently, real-time ray tracing became possible on modern GPUs [16], [17]. In anticipation of GPU vendors supporting real-time ray tracing, ray-tracing interfaces have been standardized in modern graphics APIs [1], [7].

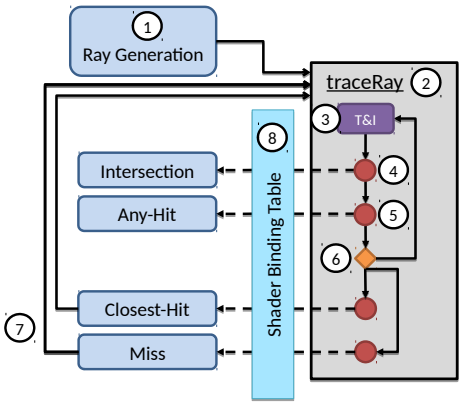

Fig. 1: Ray-tracing pipeline in Vulkan ray-tracing extension.

The Vulkan ray-tracing pipeline, shown in Figure 1, is executed by all threads in a warp. It is distinct from the raster graphics pipeline that consists of vertex, tessellation, geometry, and fragment shaders. The ray-tracing pipeline is recursive and nonlinear — unlike the linear execution of the raster graphics pipeline. Other details have been omitted to simplify the discussion.

Execution of the ray-tracing pipeline begins at the ray generation shader (1). The ray generation shader creates new rays and traces them through a scene of 3D objects by invoking

`traceRay` (2). A `traceRay` call translates by a mixture of compiler transformations and hardware acceleration into the flow described next.

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

In ray tracing, usually each thread traces a different ray. When executing `traceRay`, rays allocated to threads within the same warp may hit different objects and execute different shaders. The result is divergent indirect function calls that implement the flow of execution. Fully divergent indirect function calls, where each thread in a warp executes a different shader, serializes the execution of these functions. If recursion occurs within the called shader, SIMT utilization can be further impacted.

III. V-Vision Architecture

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

Vulkan Validation Layers: Vulkan performs error checking and other debugging capabilities in validation layers. Validation layers are intermediaries between Vulkan applications and the driver and require no code modification or recompilation [18].

Validation layers support use cases such as in-game overlays [19], and game traces captured with the *gfxreconstruct* layer [20]. These examples are possible because validation layers transparently compose with any, standards-conforming, Vulkan application across major operating systems and GPU architectures. Validation layers therefore support profiling any Vulkan application as no changes are required from the application developer.

Tool Architecture: V-Vision extends the standard Vulkan validation layer that provides error checking and printf. Figure 2 depicts V-Vision's architecture, interface and execution-flow. V-Vision's contributions are identified with stars. Vulkan validation layers intercept and modify the host-code APIs (1) and device-code SPIR-V kernels (2). Transformations of the device code performs dynamic error checking and supports debugging with the `debugPrintfEXT` builtin [21] (3). By extending the instrumentation framework, V-Vision supports generic application profiling.

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

V-Vision provides ready-to-use utilities (5) built using the instrumentation primitives: i) SIMT Efficiency: measure active threads at every basic block in graphics pipeline; ii) Divergence Characterization: measure respective impacts of control-flow, indirect function call and early-return divergence. iii) Indirect Function Call Paths: reveal all execution traces leading to indirect function calls; and iv) Execution Trace: complete warp and thread execution traces throughout the graphics pipeline; These utilities are exposed as flags (6). The utilities can be used by application developers and hardware designers to expose bottlenecks, such as poor SIMT Efficiency due to serialized indirect function call execution, and opportunities to improve them, such as thread compaction or shader refactoring.

Execution Flow: The workflow of V-Vision is shown in Figure 2 through the black arrows that represent the order of events. The dotted boxes represent the components of a Vulkan validation layer performing instrumentation and the boxes marked with a star are novel contributions of V-Vision. Removing all the dotted boxes results in the operation mode of a Vulkan application without validation. The flow begins with the execution of Vulkan API calls (7). Next, the validation layer intercepts the application's device-buffer creation to add a *StorageBuffer* — a readable and writable type of storage that is visible anywhere in the graphics pipeline — where the instrumentation data will reside (8). The validation layer receives the SPIR-V source for every shader module that the Vulkan application creates (9). The validation layer triggers V-Vision's SPIR-V automatic-instrumentation pass. This pass adds the desired instrumentation, according to the instrumentation utility, to each SPIR-V module source, thus creating new writes to the storage buffer (10). Thereafter, events outside of the validation layer occur: a graphics pipeline is created as a sequence of shaders (11), the pipeline is compiled into a GPU-specific binary (12), and the pipeline is run on the GPU (13). The last event is providing the instrumentation data to the analyses in V-Vision (14).

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

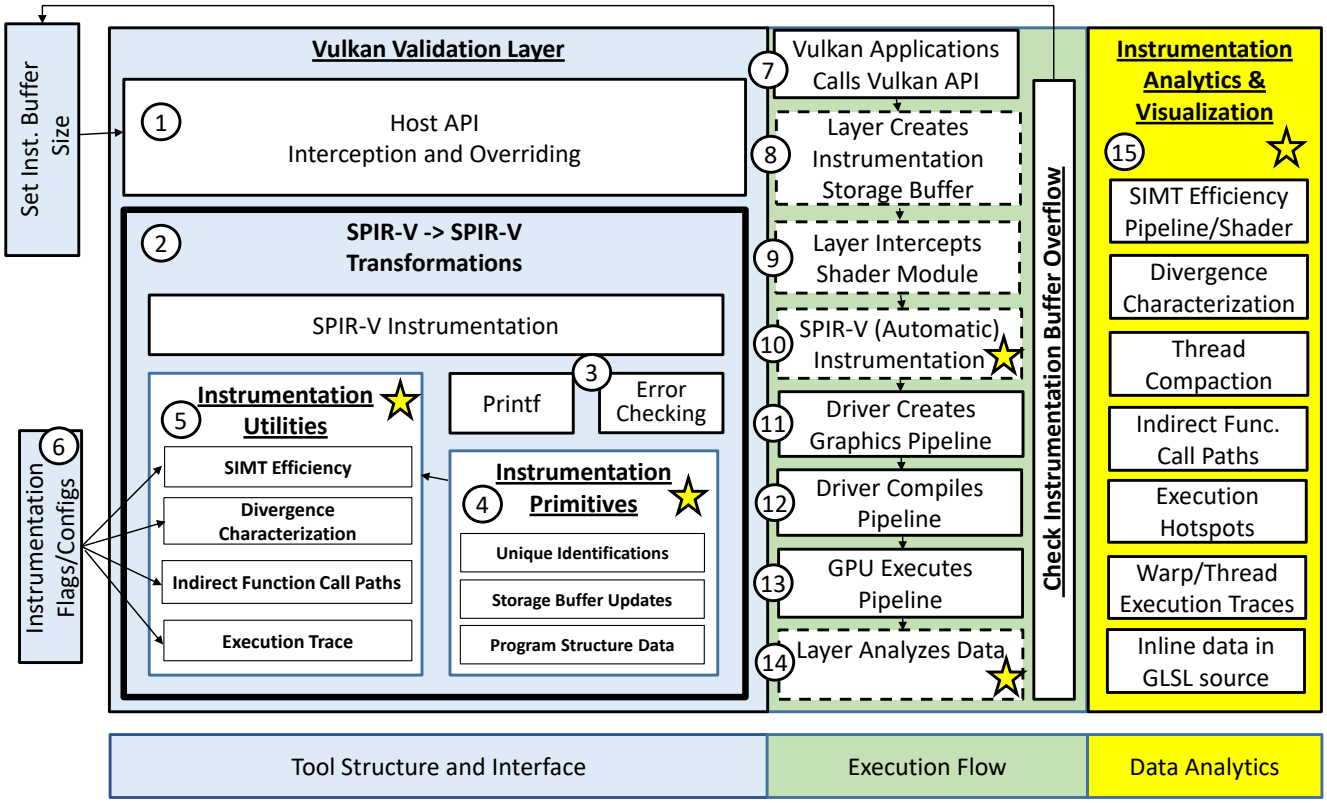

Fig. 2: V-Vision Architecture, Interface, and Execution Flow. The stars in the figure mark which modules are a novel contribution. The dotted boxes in Execution Flow represent the steps when using a validation layer for instrumentation.

function calls based on an oracle's knowledge [15], [22]; iv) Indirect-Function-Call Paths: enables the study of the SIMT efficiency of ray generation `traceRay` by reconstructing runtime thread paths; v) Execution Hotspots: visualize graphics pipeline hotspots based on dynamic instruction execution counts using program source code information; vi) Warp and Thread Execution Traces: study work allocation on thread and warp granularities by tracing execution; and vii) Inline Data Representation: present data inline at the point it was captured from in GLSL source. This paper demonstrates the insights that are revealed by applying V-Vision to applications implementing Vulkan ray-tracing pipeline in Section V.

IV. Instrumentation Primitives

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

Buffer Updates Primitives: The core of V-Vision's auto-instrumentation is `ThreadUpdate`, shown in Figure 3 using GLSL. `ThreadUpdate` safely writes an entry to the StorageBuffer, denoted as `buf` in the figure. Line 2 atomically adds the number of words to write, stored in `entry_size`, to a special location in the StorageBuffer, `buf[1]`.

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

Fig. 3: GLSL representation of ThreadUpdate primitive in V-Vision. The ThreadUpdate primitive appends an entry to the StorageBuffer, `buf`, for each thread that invokes it.

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

of which may vary. This primitive is called `ThreadUpdate` because every thread that executes it will append a new entry in the StorageBuffer.

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

Fig. 4: GLSL representation of WarpUpdate primitive. The WarpUpate primitive appends an entry to the storage buffer for each warp that invokes it.

The next primitive provided by V-Vision, `WarpUpdate`, writes an entry in the StorageBuffer for each warp that executes it. A GLSL representation of `WarpWrite` is shown in Figure 4. The `if` statement on line 2 differentiates `WarpUpdate` from `ThreadUpdate` using the GLSL builtin `subgroupElect`. The builtin will return true for the lowest-numbered thread in a warp, and false for all other threads. Thus, calling `ThreadUpdate` only if `subgroupElect` is true, results in one entry being written to the buffer. A special case of `WarpUpdate` is to compose it with the GLSL builtin `subgroupBallot(true)` to measure how many threads are active. `subgroupBallot(true)` evaluates the predicate `true` for all active threads in the warp. Thus, a bitmask is recorded that has the property: bit_i is set iff $thread_i$ is active.

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

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

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

The GLSL representation of the primitive `CreateWarpId` is shown in Figure 5. This primitive is executed by every thread in a warp. In line 2 each thread creates a copy of `warp_id`. The `subgroupElect` call in line 3 returns true to the lowest-numbered thread in the warp and returns false to all others. The only thread that executes line 4 receives the current next available id, stored in position zero of the StorageBuffer, and increments it atomically. Thus, each warp receives a unique id. The GLSL builtin `subgroupBroadcastFirst` on line 5 is a synchronization point with two distinct functions.

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

Fig. 5: GLSL representation of V-Vision's instrumentation primitive CreateWarpId. The primitive creates a warp id and every thread in the warp writes it to the buffer. `buf[0]` is a dedicated location in the StorageBuffer for creating warp ids.

When invoked by the lowest-numbered thread's value of `warp_id` it broadcasts this value to all other threads. When invoked with zero by all other threads, it returns the unique `warp_id` broadcasted by the the lowest-numbered thread. Thus, after line 5 all the threads in the warp have the same value in their local `warp_id`. In line 6 each thread calls `ThreadUpdate` to write the value of `warp_id` to the StorageBuffer. `ThreadUpdate` also writes the *thread-work id* of each thread, and thus enables the creation of a complete mapping from *thread-work id* to *warp id*. An alternative use of this instrumentation is determining how the driver assigns *thread-work ids* to warps. In a ray-tracing application, it revealed an 4×8 zig-zag assignment rather than a linear assignment.

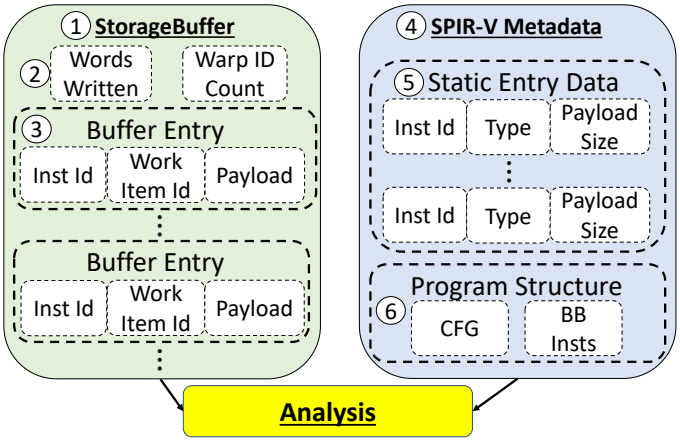

Fig. 6: V-Vision's layout of StorageBuffer containing runtime instrumentation data and SPIR-V metadata used to complement the runtime data.

Unique SPIR-V Operation ID: Figure 6 shows the organization of the data created by V-Vision. In the StorageBuffer (1), *Words Written* (2) is the number of words that the instrumentation primitives write when `ThreadUpdate` or `WarpUpdate` are called. V-Vision reports this value to the user if it exceeds the capacity of the StorageBuffer. The user may then rerun the application with a larger buffer. *Warp ID Count* is atomically incremented by `CreateWarpId` to assign each warp a unique id. Buffer entries (3) are appended to the buffer by `ThreadUpdate` and `WarpUpdate`. The *inst id* value identifies which instrumentation call site produced the

data. *thread-work id* identifies each thread and *payload* contains the customizable entry data.

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

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

Fig. 7: Data captured from V-Vision's SIMT Efficiency instrumentation utility. Presented as inline comments in the GLSL representation of the shader.

The instrumentation callsite ids support a visualization of analyses in V-Vision. `OpLine` is an SPIR-V debug instruction designed to encode file information throughout the SPIR-V module. V-Vision repurposes `OpLine` to present data throughout the GLSL representation of each shader by transforming line directives into GLSL comments. The result, shown in Figure 7, is warp-execution trace information presented as inline comments. Lines 1 and 4 show the total number of warps that executed each instrumentation call along with histograms of the number of active threads in each warp. This inline presentation leverages code understanding when examining the profile data.

V. Ray-Tracing Insights

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

A. Methodology

The data for each case study was collected on an NVIDIA Turing 1660Ti with beta driver version 451.79 that supports the `VK_RAY_TRACING_KHR` extension. Table II shows the frames captured in each application studied and, where applicable, the 3D object from Casual Effects [24]. Except for ChameleonRT, frames are captured using NSight Graphics 2020.3 C++ capture. For ChameleonRT, NSight Graphics failed to create a capture. Instead, the first frame of running ChameleonRT on Hairball and Sponza models was instrumented. For all measurements, each frame is executed 5 times. All experiments are combined in a list and executed once, the list is scrambled before the next execution. This method ensures that temporary variations in the execution environment that are not under control will increase the variability of the measurements but not insert biases. The frames captured from Quake II RTX, Sky and Window, each execute the ray-tracing pipeline 5 times: Primary Rays, Direct Rays, Reflection Refraction 1, Reflection Refraction 2, Indirect Rays.

TABLE II: Frames studied and the application they were captured from. 3D Obj files were only required for ChameleonRT.

Frame	Application	3D Obj
Hairball	ChameleonRT [23]	Hairball [24]
Sponza	ChameleonRT [23]	Sponza [24]
Sky	Quake II RTX [25]	
Window	Quake II RTX [25]	
Cornell Box	RayTracingInVulkan [26]	
Reflective Ball	RayTracingInVulkan [26]	
Robot	VK_RAYTRACE [27]	

B. Hardware Insights

Thread Compaction: Thread compaction is a proposed hardware modification to increase SIMT efficiency by repacking threads [15]. Thread compaction may create warps composed only of inactive threads that do not need to execute. A relevant analysis question is how much thread compaction can be used to improve ray tracing for each frame.

Figure 9 presents an example of the instrumentation for Thread Compaction, V-Vision's instrumentation utility that tracks the dynamic thread paths executing `traceRay`. Using the instrumentation on line 5 to capture the threads executing the `traceRay` call on line 6, the analysis counts how many times each thread executes that inner-loop call. The instrumentation on line 11 captures threads exiting the inner loop. The analysis builds a *thread path* — a bit vector with a bit for each execution of `traceRay` — for each thread. A one in the thread path indicates that the thread is active for that execution. Examining all thread paths, the analysis counts the number of threads active for each `traceRay` execution. The estimation for the maximum compaction is the ceiling of the number of the number of threads active for a given execution of `traceRay` divided by the warp size. A more realistic estimate limits thread compaction to a number of consecutive warps, the number of warps included is the *compaction window*.

Figure 8 presents the % warps compacted — rendered inactive through receiving all inactive threads — for each frame. The majority of the warps in Robot process rays that hit a skybox. These warps are very coherent and do not benefit much from compaction. Window and Sky perform both deterministic tracing and random tracing. Warps executing random path tracing benefit from thread compaction. The divergence in Cornell Box, Reflective Ball, Hairball and Sponza is significantly improved with two warps in a compaction window, increasing the window

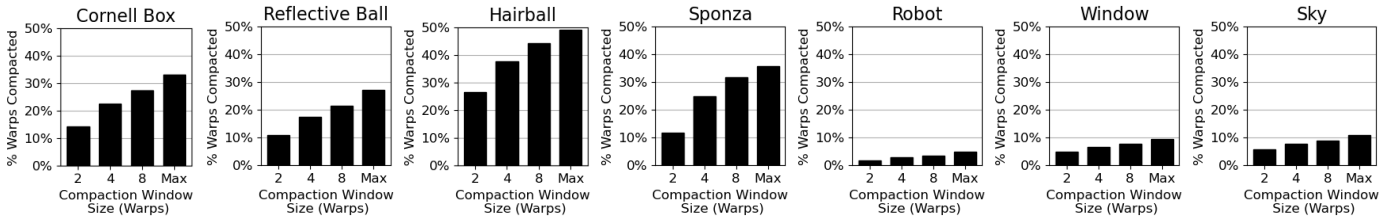

Fig. 8: Impact of Thread Compaction on number of warp executions of `traceRay`. Windows are composed of consecutive warps.

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

Fig. 9: Pseudocode for global-illumination style ray-generation shader with instrumentation for Thread Compaction analysis.

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

C. Compiler Insights

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

Fig. 10: Pseudocode that triggers thread independent scheduling on NVIDIA's Turing architecture.

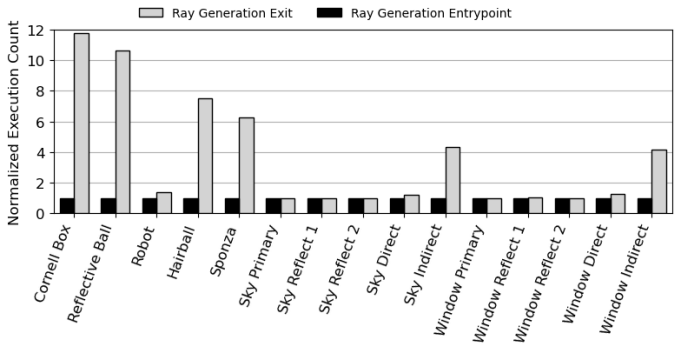

Fig. 11: Evidence of independent thread scheduling in ray tracing. Execution count of ray generation shader exit normalized to entry execution count.

In the NVIDIA Turing and Volta architectures, each thread has its own Program Counter (PC) — in earlier GPUs all threads in a warp had the same PC. Per-thread PCs allow Independent Thread Scheduling (ITS) whereby threads no longer execute a GPU kernel in lockstep. V-Vision's visualizations reveal that divergent `traceRay` calls, as shown in Figure 10, trigger ITS. Inactive threads in line 2 do not wait at the join point in line 4. ITS causes warp execution to split whereby multiple PCs are executing concurrently. The entrypoint of the graphics pipeline must be executed exactly once by each warp. Under the effects of ITS, the exit of the graphics pipeline may be executed multiple times. ITS can be quantified by comparing how many times the entrypoint and exit were executed. Figure 11 shows the exit execution count in gray, and entrypoint execution count in black. The exit count is normalized relative to the entry execution count to quantify how split the warp execution is. Control-flow divergence inherent in random path tracing triggers ITS, as evidenced by Cornell Box, Reflective Ball, Hairball, Sponza, Window (Indirect), and Sky (Indirect).

D. Application Insights

1) SIMT Efficiency: SIMT efficiency is the percentage of active threads across all basic-block executions. It measures the utilization of a SIMD GPU hardware. The low SIMT efficiency in ray tracing is due to unpredictable control flow, such as rays executing different shaders, and poor work assignment due to variation in ray bounces [28]. Existing GPU instrumentation frameworks capture SIMT efficiency of GPGPU ray tracers. However, capturing this metric in a graphics pipeline is challenging because of restrictions between shader stages. V-Vision's primitives overcome challenges and is able to track the execution across indirect function calls and shader modules.

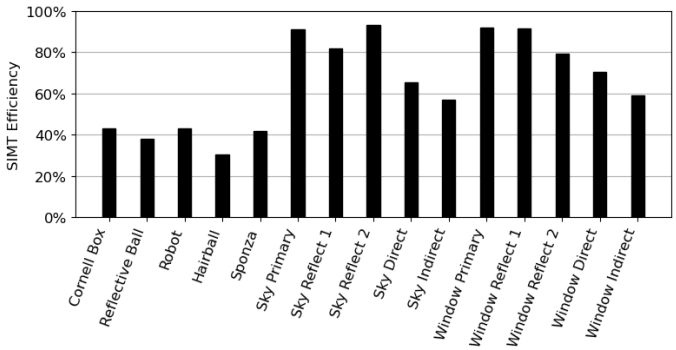

Fig. 12: SIMT efficiency of profiled frames.

Figure 12 reports the SIMT Efficiency of each pipeline invocation. The frames from RayTracingInVulkan, Cornell Box and Reflective Ball exhibit similar SIMT Efficiency despite

many differences in the frames themselves. In the Cornell Box scene, rays bounce multiple times because they are trapped in the box. Each bounce triggers control-flow divergence. Reflective Ball has fewer bounces but many divergent intersections based on material (reflection, refraction, opaque) lowering the SIMT Efficiency. The impact of geometry on SIMT efficiency is observed in Hairball and Sponza, both from ChameleonRT. With many thin hairs, Hairball triggers many incoherent ray bounces, thus lowering SIMT Efficiency when compared to Sponza. While the first three pipeline invocations of Window and Sky have high SIMT Efficiency because their effects are deterministic, the subsequent pipeline invocations, *Direct* and *Indirect*, perform random path tracing reducing SIMT Efficiency. Random path tracing in Quake II RTX has higher SIMT Efficiency than the other path tracers because it only performs 1 ray bounce per thread, compared to 3 in VK_RAYTRACE, 16 in RayTracingInVulkan and 5 in ChameleonRT, limiting the divergence.

Comparing Figure 11 to Figure 12 indicates that in general low SIMT efficiency is related to ITS. However, Robot, Sky Direct and Window Direct do not conform to this pattern. In Robot, for some warps, all the rays hit a skybox while, in other warps, the rays hit geometry. For the ones that hit the skybox, there is little work to do. In warps processing rays that hit geometry there is significant divergence that leads to ITS, resulting in the difference between entry and exit executions in Figure 11. The frame execution has low SIMT Efficiency, as shown in Figure 12, because the warps that do most of the work have high divergence. ITS occurs in both the Direct and the Indirect pipelines of Sky and Window because of a branch that tests if a surface is facing away from the Sun. The effect is more pronounced in the Indirect pipeline because of a random ray bounce before the branch.

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

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

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

Fig. 13: Simplified GLSL representation of instrumentation to characterization factors contributing to SIMT divergence.

respectively. An inactive thread that has not been captured as either an early-return divergence or as indirect-function-call divergence must be inactive due to control-flow divergence.

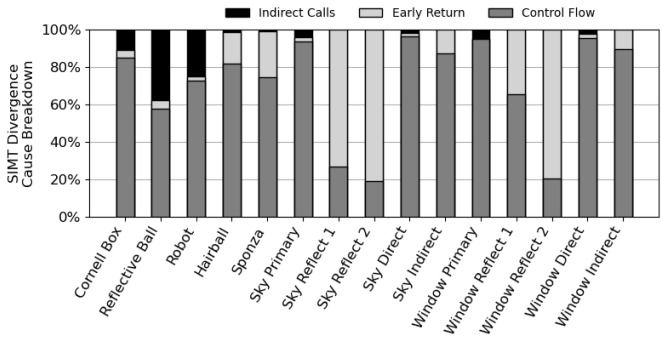

Fig. 14: Characterization of factors contributing to SIMT Divergence.

Figure 14 presents the divergence characterization. In Cornell Box, Reflective Ball, and Robot the control-flow divergence is caused by the varying ray path lengths in the tracing loops and by variations in material types that lead to different actions upon collision. Invocations of the intersection and any-hit shaders lead to indirect-call divergence. For instance, when rays collide with spheres in Reflective Ball, the intersection shader invocations lead to many divergent indirect function calls. Miss and closest-hit shaders cause much less divergence because they are invoked at most once per `traceRay` call. Thus, indirect-function-call divergence is less significant for other frames. Variable number of ray bounces account for the early-return divergence in Hairball and Sponza. Sky and Window Reflect 1 and 2 have high early-return divergence because of checks for collisions with reflective materials. The effect is less pronounced in Window Reflect 1 because part of the frame is reflective material.

3) Ray-Tracing Hotspot Detection: The behaviour of the ray-tracing pipeline is complex, involving cycles and recursion, and difficult to reason about. The same visual effects in the ray-tracing pipeline may be implemented in different shader stages. Architecture-specific complexities, such as ITS, further

complicates writing shaders. Visualizing hotspots that may be the result of geometry or other runtime factors, allows the developer to focus their refactoring efforts.

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

Figure 15 presents the dynamic instruction execution counts of each PC normalized to the maximum for Reflective Ball and Robot. Each color in the figure represents a shader in the ray-tracing pipeline. The intersection shader of Reflective Ball dominates the execution of the pipeline because all objects in the scene are spheres. The closest-hit shader is more complicated than in the closest-hits of other frames due to reflection and refraction effects. A pseudorandom number generator dominates the dynamic instruction count of Robot. Each ray requires 16 iterations of a loop to create the random direction. As most rays in Robot miss geometry, the random direction must be recomputed for nearly every ray. If the geometry becomes larger, then more rays would collide with it and perform more bounces before requiring a new pseudorandom number. This hotspot is a clear example of a bottleneck that occurs in the presence of a specific scene configuration.

4) Warp and Thread Lifetimes: Imbalanced work allocation degrades GPU performance because threads that complete their work first become inactive. The same principle applies to warps that are scheduled in groups. Imbalanced thread assignment is also a problem in path-tracing on GPGPUs [28]. The GPGPU solution of fixing work assignment with work-coarsening does not translate to graphics where frame latency is a key measure. Work assignment is also unpredictable because it is impacted by the geometry of the scene.

Execution Trace is a V-Vision's instrumentation utility that provides complete per-thread and per-warp execution traces. The lifetime of a thread or warp is defined as the number of basic blocks executed and can be derived from the execution traces. Figure 16 shows the thread and warp lifetimes for both the Hairball and Sponza frames. A large bump centered around path length of 530 is present in Sponza but not in Hairball. This bump is caused by Sponza being an enclosed space, trapping rays into a higher number of consecutive bounces. Rays that bounce off of Hairball's geometry and then miss, account for the higher incidence of path lengths in the range from 100 to 200 basic blocks. The wider range of thread path lengths causes a wider range of warp path lengths in Hairball. In comparison, Sponza has a very compressed range of warp paths due to being an enclosed space. Both scenes have a bump where rays are traced up to the maximum bounce depth. These results show that geometry has a strong effect on work assignment at the thread and warp level.

5) Ray Generation Thread Paths: The ray-generation shader generates rays and invokes `traceRay` to perform traversal and intersection, known to be expensive and to benefit from hardware acceleration [16]. Variable number of bounces and conditional calls based on material type influence the behaviour of the ray-tracing pipeline [28]. Understanding the effect of material type and geometry offers opportunities for optimizations, such as value specialization.

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

TABLE III: UNIQUE PATH COUNT AND TOP 3 FREQUENCIES OF INDIVIDUAL PATHS GENERATED BY THREAD COMPACTION UTILITY.

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

Table III shows the total number of unique thread-paths and the top 3 frequencies of individual paths. The thread paths that miss geometry are very significant when present. The frequencies 49.11% in Cornell Box, 16.1% in Reflective Ball, and 89.79% in Robot are due to rays missing geometry. Reflective Ball has many different material types and thus executes many thread paths. For instance, reflection requires a different ray than refraction while opacity does not need any rays. Excluding the rays that miss, variable ray bounce lengths cause a high path count of 126 for Cornell Box, each path having a low frequency. The Reflect 1 pipeline invocation in Window differs from Reflect 1 in Sky due to a conditional `Return` statement that quits when the object collided with is not reflective. The Window frame contains a section of floor and wall that accounts for the 26% of threads taking a different path. Sky has no reflective material so all threads take the `Return` statement. ChameleonRT has fewer unique paths due to not

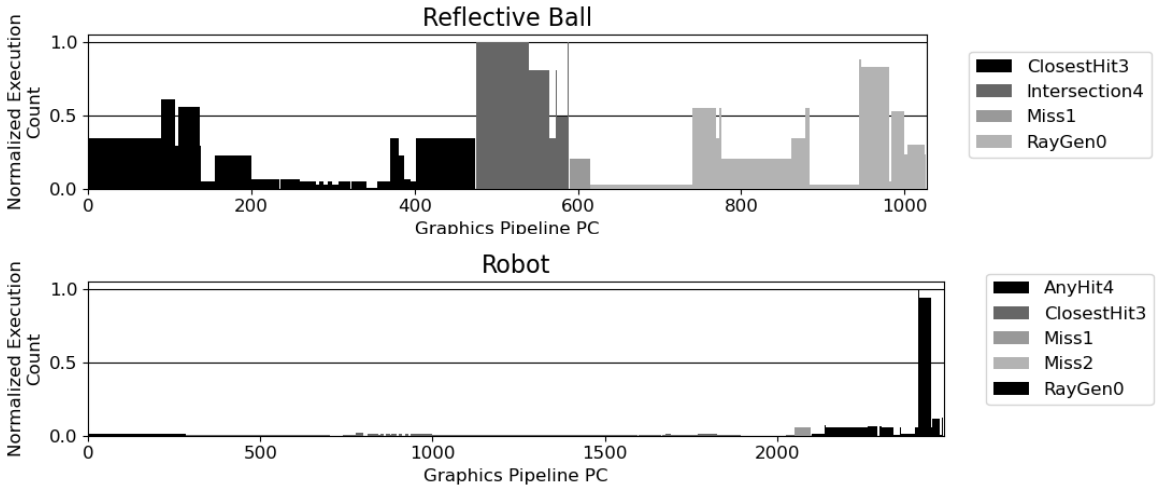

Fig. 15: Hotspots for Reflective Ball and Robot frames, normalized to maximum dynamic instruction execution count.

using different material types and tracing fewer rays per thread than RayTracingInVulkan.

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

VI. The Cost of Instrumentation

Architecture-specific data, such as register allocation, cannot be captured using SPIR-V instrumentation because SPIR-V is not bound to an architecture. Conversely, SPIR-V instrumentation has the advantage that it is supported across architectures and vendors. SPIR-V instrumentation may impact performance due to the intangible effect it has on downstream compilers. However, manual high-level instrumentation is successful in improving performance in production games, so there is value in timing and performance data [29].

Figure 17 presents the frame latency and memory overheads of the respective utilities to produce the results in Section V. Each utility may generate multiple outputs. For example, SIMT Efficiency creates the dynamic instruction count, graphics pipeline hotspots visualization, and full SIMT Efficiency analysis. The overheads range from $14\times$ to $1502\times$ increased latency over no instrumentation. In the worst case of SIMT efficiency for Reflective Ball, the complete execution time is 100 seconds. This performance degradation is reasonable because program behaviour cannot otherwise be captured. Instrumentation and analysis happen offline during design of architecture or code generation solutions, therefore this execution time is manageable. Figure 17 also illustrates that analysis contributes significantly more overhead than the instrumentation itself. Execution Trace and SIMT Efficiency collect the same amount of data but have vastly different overheads. The most data produced by the instrumentation is 566 MB which is 9.4% of the available memory of the 1660Ti.

VII. Related Work

Proprietary tools: NSight Graphics collects samples of the graphics pipeline SASS binary being executed and accumulates hardware performance counters [30]. Similar to NSight Graphics, Intel GPA is uses vendor-specific hardware performance counters to characterize application performance on Intel Graphics Hardware [11]. Microsoft Pix is a graphics profiling tool that achieves cross-vendor support by leveraging the debugging capabilities of a single graphics API [12]. In comparison to these tools, V-Vision provides fine-grained data from the code executing on the GPU with a commensurately higher overhead. Thus, V-Vision may complement the proprietary tools in situations where performance bottlenecks occur in the graphics pipeline itself. V-Vision sets itself apart as an open-source framework for developing studies, such as Thread Compaction, which require precise execution data.

NVBit [4] uses library injection to create wrapper functions for CUDA driver calls to perform on-the-fly instrumentation. NVBit instruments SASS allowing it perform analyses, such as register allocation, that are impossible for V-Vision at the SPIR-V level. V-Vision provides instruction primitives to overcome graphics instrumentation challenges similar to NVBit's abstraction of SASS instructions that overcome binary instrumentation challenges.

Zeroploit implements value-specialization for DirectX using IR instrumentation [31]. Zeroploit observes, with manual instrumentation, that many common shader operations produce a value of 0. V-Vision's instruction primitives can collect frequencies of a given value to automatically detect such opportunity in existing games using Vulkan.

Open source tools: Strengert *et al.* developed a debugging tool for OpenGL applications that instruments shaders based on user guidance [13]. Their tool uses library injection which allows for modification of a shader before it is executed

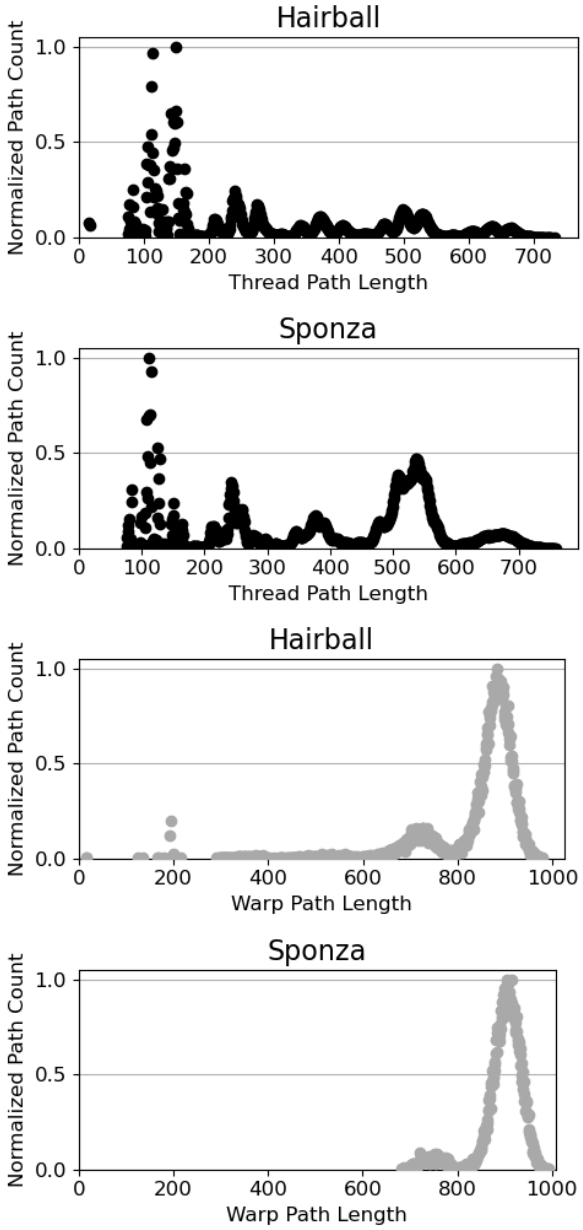

Fig. 16: Thread and Warp lifetimes for Hairball and Sponza, normalized to maximum path count. Other frames omitted for brevity.

with a custom source-to-source transpilation. Their approach is geared towards providing values from variables indicated by the user. This limits the tool, in its current form, to not support capturing general information, such as the SIMT efficiency. V-Vision's goal is not shader debugging as that is already fulfilled by the `debugPrintfEXT` GLSL builtin. V-Vision leverages existing compiler infrastructure for SPIR-V instrumentation, so implementing auto-instrumentation is more productive than in a custom transpiler.

SIMT Divergence: Existing research in improving SIMT efficiency in path-tracing on GPGPUs judge their success based on MRays per second or Frames Per Second [22], [32]. Damani *et al.* study improving SIMT efficiency by introducing speculative reconvergence and note that the relationship between

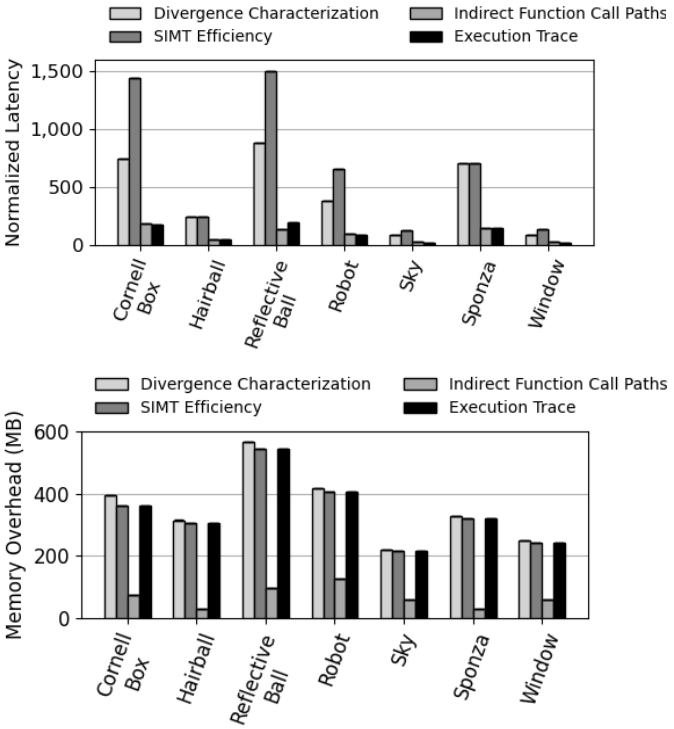

Fig. 17: **Top:** Overhead of data collection and analysis for each mode. **Bottom:** Device data overhead for each mode.

SIMT efficiency and performance is dependant on factors like work creation cost [28]. To this end, Damani *et al.* use the nvprof [6] profiler to collect hardware performance counters which sample an applications SIMT efficiency. V-Vision allows developers to relate SIMT Efficiency to performance in their case.

VIII. Conclusion

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

APPENDIX

A. Abstract

V-Vision source code and documentation is available at https://gitee.com/mindspore/mindinsight with support for the recent `vk_KHR_Ray_Tracing_Pipeline` extension.

Our artifact provides a workflow to automatically apply V-Vision to Vulkan applications. Our artifact also provides the Windows binaries for frames used in the insights sections as well as instructions and scripts to reproduce the frames. Using the artifact on an NVIDIA RTX GPU with beta drivers, all the case studies can be regenerated. The artifact also provides the ability to apply V-Vision on non-RTX GPUs to rasterization applications. Finally, the artifact includes the reference data used to generate the figures in the manuscript as well as the source-code for V-Vision.

B. Artifact Checklist

- **Program:** Quake II RTX, ChameleonRT, RayTracingInVulkan, VK_RAYTRACE. Scripts provided to fetch and build for Windows 10.
- **Compilation:** NVIDIA Driver 451.79. Driver needs to support `VK_NV_ray_tracing` and `VK_KHR_ray_tracing`
- **Transformations:** Auto-instrumentation SPIR-V passes.
- **Binary:** Included for Windows 10. Source code and scripts for building on Linux (tested on Ubuntu 18.04) and Windows.
- **Run-time environment:** Scripts for complete workflow provided and tested for Windows(Windows 10). Scripts for workflow verification available and tested for Linux (Ubuntu 18.04).
- **Hardware:** For ray-tracing case studies, RTX GPU required. For other studies, any Vulkan-capable GPU required.
- **Execution:** Overhead introduced by V-Vision varies, but longest wall-clock time observed for a single frame is 100 seconds.
- **Output:** All the data for the SIMT Efficiency, Divergence Characterization, Independent Thread Scheduling Study, Pipeline Hotspots, TraceRay Thread Paths Study, Thread Compaction Study, and Thread and Warp lifetimes is produced. Using the graphing scripts, this data can reproduce all the figures in the manuscript. In addition, heatmap visualizations are produced.
- **How much disk space required (approximately)?:** 12 GB for frames, 2 GB for code.
- **How much time is needed to prepare workflow (approximately)?:** 1 hour
- **How much time is needed to complete experiments (approximately)?:** 8 hours
- **Publicly available?:** Yes.
- **Archived (provide DOI)?:** 10.5281/zenodo.4281460

C. Description

1) How Delivered: V-Vision with support for finalized Vulkan ray-tracing specification is available from MindInsight: https://gitee.com/mindspore/mindinsight. The artifact, requiring outdated drivers supporting `VK_KHR_ray_tracing`, available from Zenodo: https://doi.org/10.5281/zenodo.4281460.

2) Hardware Dependencies: GPU must be compatible with Vulkan. GPU must support `VK_NV_ray_tracing` or `VK_KHR_ray_tracing` extension for ray-tracing studies.

3) Software Dependencies: To reproduce case studies, applications required: Quake 2 Rtx, ChameleonRt, RayTracingInVulkan, VK_RAYTRACE.

NVIDIA Driver version that supports `VK_NV_ray_tracing` or `VK_KHR_ray_tracing`

D. Installation

1) Download from https://gitee.com/mindspore/mindinsight
2) Follow README instructions to clone and build.

E. Experiment Workflow

To execute experiment execute `python3 run.py config-file`. Available config files:

1) **data_run.ini** configuration for regenerating case studies.
2) **overhead_run.ini** configuration for regenerating overhead data.
3) **sanity.ini** configuration for experiment on vkcube which works on any vulkan-capable GPU.
4) **sanity_linux.ini** configuration for experiment on vkcube with linux paths.
5) **visualizations_run.ini** configuration for generating ray-tracing visualizations.

The configuration file defines which V-Vision mode to execute and also how to find and invoke application. For each run of the experiment, the order of the applications and modes are scrambled to avoid potential bias in the data.

F. Evaluation and Expected Result

To evaluate the artifact it is sufficient to run the experimental workflow and then graph the results. A script `graph_all` is provided for Linux and Windows to automate producing the graphs. The case study data based on program behaviour, Thread Compaction, should match exactly if executed on the same GPU with the same Driver version. The other case studies may experience a very minuscule variance but this should not even be visible. If executed on a different GPU then studies that assess hardware utilization (SIMT Efficiency, Divergence Characterization) may report different values due to architectural differences (such as having RTCores).

G. Experiment Customization

The configuration files provides support for executing an arbitrary set of V-Vision modes on an arbitrary set of different Vulkan applications. The configuration files and workflow are cross-platform and differ only in the file paths provided by the user for the local install locations. An exhaustive list of the options available is included in the artifact README.

The graphs generated require domain-specific knowledge, such as properly naming the Quake II RTX pipeline invocations for their respective purposes. Therefore, the graphing scripts require some very minor code changes to support new applications.

H. Notes

It is highly recommended to use Windows 10 to regenerate case study results as a VVision binary is provided as well as scripts to fetch and build the applications.

I. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

References

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

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

[3] NVIDIA, "CUDA C++ Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, accessed: 2020-08-12.

[4] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *International Symposium on Microarchitecture (MICRO)*, Columbus, OH, USA, 2019, p. 372–383. [Online]. Available: https://doi.org/10.1145/3352460.3358307

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

[6] NVIDIA, "CUDA Pro Tip: nvprof is Your Handy Universal GPU Profiler," https://developer.nvidia.com/blog/cuda-pro-tip-nvprof-your-handy-universal-gpu-profiler/, accessed: 2020-08-21.

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

[8] Apple, "Metal, accelerating graphics and much more." https://developer.apple.com/metal/, accessed: 2020-08-12.

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

[10] NVIDIA, "NVIDIA NSight Graphics," https://developer.nvidia.com/nsight-graphics, accessed: 2020-08-12.

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

[12] Microsoft, "PIX, Introduction," accessed: 2020-08-25.

[13] M. Strengert, T. Klein, and T. Ertl, "A hardware-aware debugger for the opengl shading language," in *Graphics Hardware (GH)*, San Diego, CA, USA, 2007.

[25] NVIDIA, "Quake II RTX," https://github.com/NVIDIA/Q2RTX, accessed: 2020-08-12.

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

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

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

[17] Imagination, "PowerVR Ray Tracing," accessed: 2020-08-25.

[18] Khronos, "Architecture of the Vulkan Loader Interfaces," https://github.com/KhronosGroup/Vulkan-Loader/blob/master/loader/LoaderAndLayerInterface.md, accessed: 2020-08-11.

[19] Valve, "Steam Overlay," https://partner.steamgames.com/doc/features/overlay, accessed: 2020-08-11.

[20] LunarG, "GFXReconstruct," https://github.com/LunarG/gfxreconstruct/blob/dev/README.md, accessed: 2020-08-11.

[21] ——, "GPU-Assisted Validation," https://vulkan.lunarg.com/doc/view/1.1.114.0/windows/gpu_validation.html, accessed: 2020-08-11.

[22] I. Wald, "Active Thread Compaction for GPU Path Tracing," in *High Performance Graphics (HPG)*, 2011, p. 51–58. [Online]. Available: https://doi.org/10.1145/2018323.2018331

[23] W. Usher, "ChameleonRT," https://github.com/Twinklebear/ChameleonRT, 2019.

[24] M. McGuire, "Computer graphics archive," July 2017. [Online]. Available: https://casual-effects.com/data

[26] GPSnoopy, "Ray Tracing In Vulkan," https://github.com/GPSnoopy/RayTracingInVulkan, accessed: 2020-08-12.

[27] NVIDIA, "VK_RAYTRACE," https://github.com/nvpro-samples/vk_raytrace, accessed: 2020-08-12.

[28] S. Damani, D. R. Johnson, M. Stephenson, S. W. Keckler, E. Yan, M. McKeown, and O. Giroux, "Speculative Reconvergence for Improved SIMT Efficiency," in *Intern. Symp. on Code Generation and Optimization (CGO)*, 2020, p. 121–132. [Online]. Available: https://doi.org/10.1145/3368826.3377911

[29] J. MacArthur and M. Stich, "Profiling DXR Shaders with Timer Instrumentation," accessed: 2020-08-29.

[30] R. Kerschner and J. Klei, "Speed of Light DXR Ray Tracing with NVIDIA Nsight Graphics (Presented by NVIDIA)," https://www.gdcvault.com/play/1026187/Speed-of-Light-DXR-Ray, accessed: 2020-08-12.

[31] R. Rangan, M. W. Stephenson, A. Ukarande, S. Murthy, V. Agarwal, and M. Blackstein, "Zeroploit: Exploiting zero valued operands in interactive gaming applications," *ACM Trans. Archit. Code Optim.*, vol. 17, no. 3, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3394284

[32] K. Garanzha and C. Loop, "Fast ray sorting and breadth-first packet traversal for GPU ray tracing," in *Computer Graphics Forum*, vol. 29, no. 2. Wiley Online Library, 2010, pp. 289–298. [Online]. Available: https://doi.org/10.1111/j.1467-8659.2009.01598.x