



Titre: Analyzing GPU Performance in Virtualized Environments: A Case Study
Title:

Auteurs: Adel Belkhiri, & Michel Dagenais
Authors:

Date: 2024

Type: Article de revue / Article

Référence: Belkhiri, A., & Dagenais, M. (2024). Analyzing GPU Performance in Virtualized Environments: A Case Study. Future Internet, 16(3), 72 (18 pages).
Citation: <https://doi.org/10.3390/fi16030072>

 **Document en libre accès dans PolyPublie**
Open Access document in PolyPublie

URL de PolyPublie: <https://publications.polymtl.ca/57587/>
PolyPublie URL:

Version: Version officielle de l'éditeur / Published version
Révisé par les pairs / Refereed

Conditions d'utilisation: CC BY
Terms of Use:

 **Document publié chez l'éditeur officiel**
Document issued by the official publisher

Titre de la revue: Future Internet (vol. 16, no. 3)
Journal Title:



Maison d'édition: Multidisciplinary Digital Publishing Institute
Publisher:

URL officiel: <https://doi.org/10.3390/fi16030072>
Official URL:

Mention légale: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).
Legal notice:

Article

Analyzing GPU Performance in Virtualized Environments: A Case Study

Adel Belkhiri *  and Michel Dagenais 

Department of Computer and Software Engineering, École Polytechnique de Montréal, Montréal, QC H3T 1J4, Canada; michel.dagenais@polymtl.ca

* Correspondence: adel.belkhiri@polymtl.ca

Abstract: The graphics processing unit (GPU) plays a crucial role in boosting application performance and enhancing computational tasks. Thanks to its parallel architecture and energy efficiency, the GPU has become essential in many computing scenarios. On the other hand, the advent of GPU virtualization has been a significant breakthrough, as it provides scalable and adaptable GPU resources for virtual machines. However, this technology faces challenges in debugging and analyzing the performance of GPU-accelerated applications. Most current performance tools do not support virtual GPUs (vGPUs), highlighting the need for more advanced tools. Thus, this article introduces a novel performance analysis tool that is designed for systems using vGPUs. Our tool is compatible with the Intel GVT-g virtualization solution, although its underlying principles can apply to many vGPU-based systems. Our tool uses software tracing techniques to gather detailed runtime data and generate relevant performance metrics. It also offers many synchronized graphical views, which gives practitioners deep insights into GVT-g operations and helps them identify potential performance bottlenecks in vGPU-enabled virtual machines.

Keywords: GPU virtualization; GVT-g; performance analysis; software tracing



Citation: Belkhiri, A.; Dagenais, M. Analyzing GPU Performance in Virtualized Environments: A Case Study. *Future Internet* **2024**, *16*, 72. <https://doi.org/10.3390/fi16030072>

Academic Editors: Jerry Chou and Wu-Chun Chung

Received: 29 January 2024

Revised: 19 February 2024

Accepted: 21 February 2024

Published: 23 February 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Accelerators are specialized processors that have been developed and integrated into computer systems in response to the growing demand for high-performance computing. These accelerators aim to assist the central processing unit (CPU) in executing certain types of computations. Several studies have demonstrated that offloading specific tasks to accelerators can considerably boost the overall system performance [1]. Hence, the use of accelerators has become a typical approach to handling the high computational demands in various industries and research fields. This has motivated hardware manufacturers to develop a variety of specialized accelerators, including accelerated processing units (APUs), floating-point units (FPUs), digital signal processing units (DSPs), network processing units (NPUs), and graphics processing units (GPUs).

The GPU, which is considered one of the most pervasive accelerators, was initially devised for graphics rendering and image processing. However, in the last few years, its computational power has been increasingly harnessed for parallel number crunching. Nowadays, GPU-accelerated applications are present in many domains that are often unrelated to graphics, such as deep learning, financial analytics, and general-purpose scientific calculations. On the other hand, virtualization plays a fundamental role, as it enables many modern computing concepts. Its primary function is facilitating the sharing and multiplexing of physical resources among different applications. Particularly, virtualization significantly enhances GPU utilization by enabling more efficient allocation of its computational power. In general, the main advantage of virtualizing computing resources lies in reducing energy consumption in data centers, which, in turn, contributes to reducing operational costs. Hence, a variety of new applications that leverage the

capabilities of virtual GPUs (vGPUs) have emerged. Examples of such applications that benefit from vGPU acceleration are deep learning, virtual desktop infrastructure (VDI), and artificial intelligence applications.

It is worth noting that virtualizing GPUs presents more complex challenges than the virtualization of CPUs and most I/O devices, as the latter rely on well-established technologies. These challenges stem from several key obstacles. First, there is significant architectural diversity in hardware among GPU brands, complicating the development of a universal virtualization solution. Second, using closed-source drivers from popular GPU brands, such as NVIDIA, poses a significant challenge for third-party developers in developing virtualization technology for these devices. Third, most GPU designs lack inherent sharing mechanisms, leading to a GPU process gaining exclusive access to its resources, thereby blocking other processes from preemption. In addition, several studies have shown that the overhead involved in process preemption is substantially higher in GPUs than in CPUs [2,3]. This increased overhead is essentially due to the higher number of cores and context states in GPUs. It is important to note that some recent GPUs, such as the NVIDIA Pascal GPU [4], include support for preemption at the kernel, thread, and instruction levels to mitigate these challenges.

Before developing advanced GPU virtualization technologies, practitioners used the passthrough technique to enable VMs to access physical GPU (pGPU) resources directly. This approach, however, has limitations, such as the inability to share a GPU among multiple VMs and the lack of support for live migration. Major GPU manufacturers such as NVIDIA, AMD, and Intel have introduced their brand-specific virtualization solutions to address these issues. These include NVIDIA GRID [5], AMD MxGPU [6], and Intel's Graphics Virtualization Technology—Grid generation (GVT-g) [7]. The first two virtualization solutions are based on hardware virtualization capabilities, whereas the third one, Intel's GVT-g, provides a software-based solution for full GPU virtualization. GVT-g is open source and has been integrated into the Linux mainline kernel, which makes it a desirable option due to its accessibility and potential for broader integration.

On the other hand, performance analysis tools for GPUs are important for debugging performance issues in GPU-accelerated applications [8]. These tools help understand how the GPU resources are allocated and consumed, and they facilitate the diagnosis of potential performance bottlenecks. They are particularly crucial in virtualized environments, where resource sharing in vGPUs and its impact on performance need to be better understood. However, developing practical tools for monitoring and debugging vGPUs remains challenging. This is because virtualized environments often present many layers, encompassing hardware, middleware, and host and guest operating systems, which increase the isolation and abstraction of GPU resources, making it difficult to pinpoint the causes of performance issues.

The landscape of GPU performance analysis features diverse tools, with academic contributions tailored to specific GPU programming models or diagnosing particular GPU-related performance issues. For instance, the works in [9,10] leveraged library interposition to capture runtime events from OpenCL- and HSA-based programs, respectively. While they enabled the analysis of GPU kernel execution and CPU–GPU interaction, they presented high overhead and faced challenges for broader analysis and portability. On the other hand, vendor-provided options, such as vTune Profiler [11] and Nsight Systems [12], leverage dynamic instrumentation and hardware counters to gather performance data and unveil application behavior in various aspects. These tools offer interesting analyses covering kernel execution, memory access patterns, and GPU API call paths. Nevertheless, they are often limited in terms of openness and cross-architecture applicability. In summary, while each of these tools has its strengths and weaknesses, a common shortfall is their lack of support for vGPUs—with the exception of Nsight Systems, which, however, is proprietary and closed-source software.

This paper presents a novel performance analysis framework dedicated to GPUs virtualized with the GVT-g technology. Our framework uses host-based tracing to gather

performance data efficiently and with minimal overhead. Tracing is a proven technique for collecting detailed performance data from complex systems. A key advantage of our approach is that it does not necessitate the installation of any agents or tools within the VMs, as tracing is confined to the host operating system's kernel space. The benefits of this methodology include ease of deployment, preservation of VM owner privacy, and cost-effectiveness. This study's contributions are as follows:

1. Efficient instrumentation of KVMGT, the open-source implementation of GVT-g for KVM [13], and integrated modules within the Linux Trace Toolkit Next Generation (LTTng) [14] to collect performance data from both the virtualization solution and the Intel GPU driver.
2. Development of a unified, stateful model capable of filtering and organizing the collected trace data. This model simplifies data processing and supports our analysis in computing many performance metrics.
3. Implementation of various analyses within Trace Compass [15], an open-source performance analyzer, and a series of synchronized graphical views. These views aim to assist practitioners in understanding GVT-g's internal mechanisms and in diagnosing many performance issues related to vGPU usage.

The remainder of this paper is organized as follows: In Section 2, we conduct a comprehensive literature review of the methodologies and techniques applied in GPU virtualization. Section 3 introduces prominent GPU profiling and performance analysis tools. Section 4 details the design of our proposed framework and outlines several pertinent performance metrics. Next, Section 5 presents three practical scenarios that demonstrate the effectiveness of our framework. Following this, Section 6 assesses and discusses the overhead associated with using our framework. Finally, we summarize our findings and contributions in Section 7.

2. Background

In this section, we delve into the methodologies and technologies that form the foundation of GPU virtualization. Understanding these virtualization technologies is crucial for developing performance analysis solutions, as they directly impact the allocation and management of host resources. Despite challenges, such as proprietary GPU drivers and non-standardized GPU designs, a number of virtualization techniques have been developed [1]. We categorize these techniques into four primary categories: PCI passthrough, single-root input/output virtualization (SR-IOV), para- and full virtualization, and API remoting.

2.1. PCI Passthrough

PCI passthrough utilizes hardware virtualization technologies—specifically, Intel VT-d [16] and AMD-Vi [17]—to allow VMs to access the pGPU of the host directly. While this method offers near-native GPU performance, it limits one GPU to a single VM, leading to potential resource underutilization. Google Cloud is an example of a cloud service employing this technology [18].

2.2. SR-IOV

SR-IOV addresses the resource-sharing limitation of PCI passthrough. Partitioning a single physical device's hardware functions into physical functions (PFs) and virtual functions (VFs) allows multiple VMs to share a single device. Supported by major hypervisors such as KVM and Xen [19,20], its implementations include NVIDIA GRID [5] and AMD MxGPU [6]. SR-IOV offers balanced performance, equitable pGPU distribution, and enhanced security via vGPU isolation.

2.3. API Remoting

API remoting virtualizes GPUs by intercepting and processing API calls in the guest OS. While effective, it requires frequent updates to wrapper libraries to stay aligned with evolving GPU frameworks [21–25]. This approach is notable for its high-level virtualization capabilities.

2.4. Para- and Full Virtualization

GPUs in this category are virtualized at the driver and hypervisor levels. Paravirtualization employs customized GPU drivers in the guest OS, whereas full virtualization uses standard drivers. A notable example is Intel GVT-g [7], a technology that utilizes mediated passthrough (MPT) for efficient and isolated vGPU access. This method supports advanced features such as live migration [26] and enhances vGPU density [27,28], contributing to its comprehensive utility in virtualized environments.

Although not widely adopted in cloud environments due to competition from proprietary solutions, GVT-g holds significant value for its distinctive technical advantages. It seamlessly integrates with the Linux kernel as an open-source software-based solution, enabling community customization and broader applicability compared to closed-source alternatives. Furthermore, its lightweight design minimizes performance overhead and hardware dependence, making it a cost-effective option for diverse computing scenarios.

3. Related Work

As the complexity of GPU-accelerated applications continues to grow, the need for effective performance analysis methods becomes increasingly critical, particularly in vGPU-based systems. Our study of existing GPU performance analysis tools shows that they offer different levels of analysis, and they are mostly dedicated to specific GPU architectures. High-end production-quality tools such as vTune Profiler [11] and Nsight systems [12] are notable for their comprehensive approach to analyzing GPU-accelerated applications. They provide a holistic understanding of the application runtime behavior, particularly unveiling the interaction between CPU and GPU, which is crucial for effective optimization. In contrast, tools proposed in academic research are often tailored to specific GPU programming models or addressing particular GPU-related performance issues.

Many vendors offer dedicated software for profiling GPU applications, such as NVIDIA Nsight Systems, Intel vTune Profiler, and AMD Radeon GPU Profiler [29]. These tools leverage various techniques such as binary instrumentation, hardware counters, and API hooking to gather detailed performance events. Despite providing rich insights into kernel execution, CPU–GPU interaction, memory access patterns, and GPU API call paths, these tools are often limited in terms of openness, flexibility, and cross-architecture applicability. In addition to proprietary offerings, the GPU performance analysis ecosystem encompasses feature-rich open-source tools. For example, HPCToolkit [30,31] and TAU [32] are two versatile tools tailored for analyzing heterogeneous systems' performance. These tools offer valuable diagnostic capabilities for pinpointing GPU bottlenecks and determining their root causes. For instance, through call path profiling, they provide insights for kernel execution and enable the identification of hotspots in the program's code.

Aside from the established profilers, academic research also presents many innovative tools for the diagnosis of performance issues in GPU-accelerated applications. Zhou et al. proposed GVProf [33], a value-aware profiler for identifying redundant memory accesses in GPU-accelerated applications. Their follow-up work [34] focused on improving the detection of value-related patterns (e.g., redundant values, duplicate writes, and single-valued data). The main objective of their work was to identify diverse performance bottlenecks and provide suggestions for code optimization. GPA (GPU Performance Advisor) [35] is a diagnostic tool that leverages instruction sampling and data flow analysis to pinpoint inefficiencies in the application code. DrGPU [36] uses a top-down profiling approach to quantify and decompose stall cycles using hardware performance counters. Based on the stall analysis, it identifies inefficient software–hardware interactions and their root causes, thus helping make informed optimization decisions. CUDAAdvisor [37], built on top of LLVM, instrumentalizes application code on both the host and device sides. It conducts code- and data-centric profiling to identify performance bottlenecks arising from competition for cache resources and memory and control flow divergence. The main disadvantages of these tools lie in their considerable overhead and exclusive applicability to NVIDIA GPUs. On the other hand, several profiling tools leverage library interposition

and userspace tracing to capture runtime events, enabling the correlation of CPU and GPU activities. For example, CLUST [9] and LTTng-HSA [10] employ these techniques to profile OpenCL- and HSA-based applications, respectively. However, a substantial drawback of these tools is their tight coupling with specific GPU programming frameworks, which limits their capability to provide a system-wide analysis.

To sum up, the current landscape of GPU performance analysis tools is diverse, with each tool differing in openness, targeted GPU architectures, and depth of analysis. A significant limitation among these tools, with the notable exception of those offered by NVIDIA, is their lack of support for vGPUs (Table 1). This gap is particularly significant considering the growing importance of GPUs in virtualized environments. Our research aims to address this oversight by proposing a method for analyzing the performance of GPUs virtualized via GVT-g. We detail this method and the framework that it implements in the following section.

Table 1. A comparative analysis of a few state-of-the-art tools alongside our framework.

Tool	GPU Arch.	GPU APIs	Analysis Levels	vGPU Support	Overhead
ValueExpert [34]	NVIDIA	CUDA	Thread	No	Moderate
GPA [35]	NVIDIA	CUDA	Thread	No	High
CLUST [9]	-	OpenCL	Thread	No	Moderate
LTTng-HSA [10]	-	HSA	Thread	No	Moderate
DrGPU [36]	NVIDIA	CUDA	Thread, instruction	No	High
GVProf [33]	NVIDIA	CUDA	Thread, instruction	No	Moderate
CUDAAdvisor [37]	NVIDIA	CUDA	Thread, instruction	No	High
Our framework	Intel	-	System-wide, thread	Yes	Low

4. Proposed Solution

This paper presents a new performance analysis framework for vGPUs managed with GVT-g. The architecture of this framework is depicted in Figure 1. It comprises three main subsystems: a data collection subsystem, an analysis subsystem, and a visualization subsystem. The first subsystem is in charge of collecting meaningful performance data from GVT-g and the Intel GPU driver. The second subsystem processes the collected data and derives relevant performance metrics. The visualization subsystem consists of several graphical views that display the results of the analysis conducted by the second subsystem. We implemented the visualization and analysis subsystems as extensions of Trace Compass [15], an open-source trace analyzer. This tool can analyze a massive amount of tracing data to enable the user to diagnose a variety of performance bottlenecks. Moreover, Trace Compass supports interactive graphical interfaces, pre-built analyses, event filtering, and trace synchronization. We present the design details of those subsystems in the following subsections.

In addition, we used KVM [38] to build our virtualization environment since it is the default hypervisor of Linux. The role of KVM is to enable VMs to leverage the hardware virtualization capabilities of the host machine (e.g., VT-x and AMD-V on Intel and AMD machines, respectively). In Linux, KVM is implemented as three kernel modules: `kvm.ko`, `kvm-intel.ko`, and `kvm-amd.ko`. At the user-space level, we used QEMU [39] to run the guest OSs of our VMs.

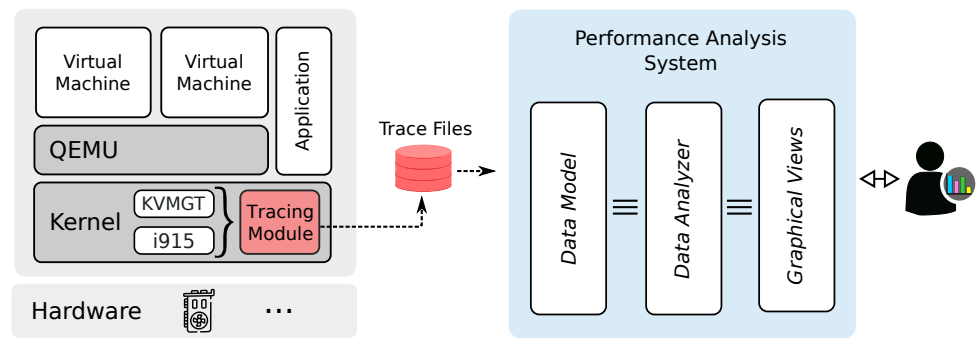


Figure 1. Architecture of the proposed framework.

4.1. Data Collection

Logging and tracing are fundamental techniques for collecting runtime data from software. Logging involves recording data in one or more log files and detailing application failures, misbehavior, and the status of its ongoing operations. These high-level human-readable data help in swift bug troubleshooting. In contrast, tracing captures low-level details of software execution, focusing on diagnosing complex functional issues and identifying performance bottlenecks. The resultant data tend to be extensive and detailed. Unlike logging, tracing requires a specialized tool known as a tracer. A multitude of tracers are available for nearly all modern operating systems. For example, in Linux systems, notable tracers include Ftrace [40], Perf [41], and LTTng.

Our framework relies on the tracing technique for gathering low-level data for our analysis. It uses LTTng as a tracer because of its versatility and minimal overhead. It is worth noting that we have confined our tracing scope to the host machine to minimize intrusiveness. In addition, as GVT-g is a kernel module (named *kvmgt.ko* in Linux systems), our focus is restricted to kernel space tracing. This involves instrumenting the GVT-g module’s source code to introduce new tracepoints and using a selection of existing tracepoints in *i915*, the host GPU driver. Table 2 details a subset of these tracepoints, which are essential for our analysis.

Table 2. A subset of the kernel events required for our analysis.

Tracepoint	Description
<i>gvt_workload_queue</i>	Indicates the addition of a GPU request to the vGPU queue after submission by a VM process.
<i>gvt_workload_submit</i>	Marks the forwarding of a request from GVT-g threads to the host graphics driver (<i>i915</i>).
<i>gvt_workload_complete</i>	Fires upon completion and removal of a GPU request from GVT-g data structures.
<i>gvt_sched_switch</i>	Reports the remaining time slice for each vGPU upon scheduling out.
<i>i915_gem_request_add</i>	Indicates the queuing of a GPU request by the host GPU driver.
<i>i915_gem_request_submit</i>	Fires when a request’s dependencies are resolved and it is ready for execution.
<i>i915_gem_request_in</i>	Indicates the sending of a GPU request to the hardware for execution.
<i>i915_gem_request_out</i>	Marks the removal of a request from the host GPU driver’s data structures.
<i>i915_intel_engine_notify</i>	Indicates the completion of a GPU request by one of the GPU engines.
<i>i915_gem_object_create</i>	Triggered when a new GEM object is created, indicating the allocation of a new chunk of memory within the GPU’s memory space.
<i>i915_gem_object_destroy</i>	Occurs when a GEM object is destroyed, signaling the release of the memory associated with that object.

4.2. Data Analysis

The data generated by tracers are inherently low-level and complex, making any manual analysis very complex. The trace events are semantically interconnected and can be

fully understood only within their specific context. Consequently, our principal objective is to develop an automated analysis system that improves the interpretability of the collected data. This system aims to assist practitioners in rapidly identifying performance issues arising from utilizing GPU resources in virtualized environments.

4.2.1. Data Model

As it is designed for offline analysis, our framework requires a robust data model to organize the data collected from the traced kernel subsystems efficiently. An efficient model is indeed essential for generating metrics and graphical representations effectively. Thus, it is necessary to have a thorough understanding of the GPU request’s lifecycle—from when a VM process initiates a request to the point where the hardware executes it and returns a response. Such a comprehensive insight is paramount for diagnosing intricate performance anomalies effectively.

Figure 2 illustrates the various states that a GPU request undergoes throughout its lifecycle in the system. Initially, a process within a GPU-accelerated VM generates a request for GPGPU or graphics processing operations. This request is first received by the VM’s GPU driver, which places it in a waiting queue (1). As GPU requests often depend on others, these dependencies must be resolved before the request attains the “Submitted” state (2). Once all preceding requests in the queue have been executed, the guest OS’s GPU driver removes the request from its waiting queue and forwards it to the vGPU for execution (3). Subsequently, GVT-g intercepts the request and queues it in the vGPU’s waiting queue associated with the VM (4). When the vGPU’s scheduler allocates time for this vGPU, the request is passed to the host GPU driver (5), which then places it in another waiting queue (6). The request awaits its turn for execution, pending the resolution of its dependencies (7) and completing all prior queued requests. Once these conditions are met, the host GPU driver dispatches the request to the hardware for execution (8). Upon completing the request, GVT-g receives a notification (9) and informs the guest OS via a virtual interrupt.

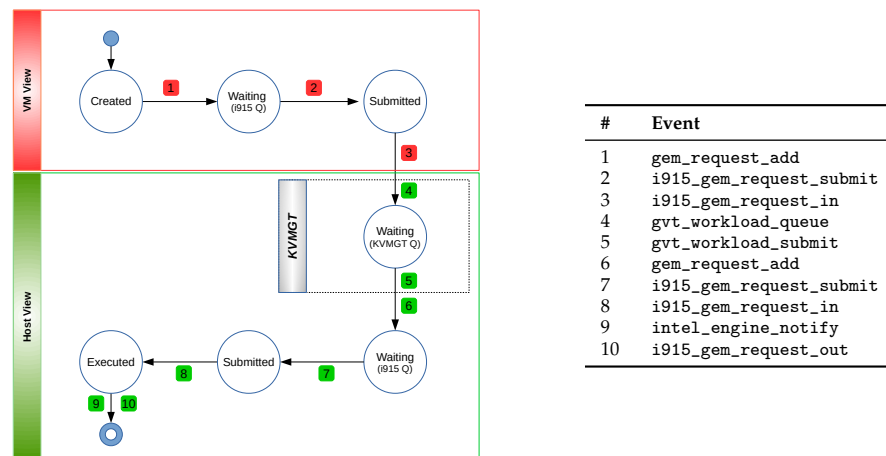


Figure 2. States of a GPU request along with the corresponding tracing events.

Modeling the collected performance data is mandatory for computing relevant performance metrics, such as the average wait and execution times of requests per vGPU. To this end, we used Trace Compass [15], a powerful trace analyzer that integrates several interesting pre-built performance analyses. These analyses save the data representing the states of the system to be monitored in a disk-based data structure called a state history tree (SHT) [42]. We store the states of our system in an attribute–tree data structure (Figure 3). This data structure exhibits sets of hierarchical attributes, which are updated each time one of the events described in Section 4.1 is handled. Thus, according to our model, one or many vGPUs can be attached to one pGPU. Both a pGPU and a vGPU each have their

own driver waiting queue. A GPU request is identified by its sequence number (*seqno*), its context (*ctx*), and the engine to which it should be sent for execution (*ring*).

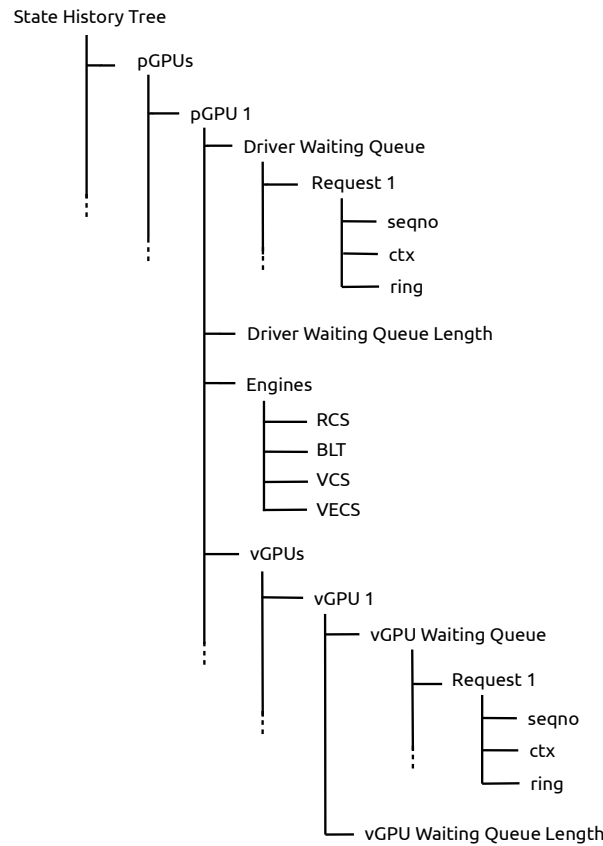


Figure 3. Excerpt of the tree-based data structure upon which our analyses are based.

4.2.2. Performance Metrics

A performance metric is a quantitative measure of a system’s performance in a specific aspect. Mainly, performance metrics help assess the status of a system and pinpoint potential performance problems. This section introduces several performance metrics that our framework calculates.

GPU Utilization

This metric quantifies the occupancy levels of both the pGPU and the vGPU. It is expressed as the percentage of the time during which the GPU is actively processing requests. Considering that the GPU operates multiple engines to process requests in different queues concurrently, we calculate this metric on a per-engine basis. The formula below uses t_s and t_e as markers, representing the start and end timestamps, respectively, for the execution of an individual request denoted by i . Thus, the GPU utilization metric U is calculated as the sum of the execution durations of all requests tracked during T , the observation period. This sum is then divided by T to obtain the utilization percentage.

$$U = \sum_i \frac{t_{e_i} - t_{s_i}}{T} \times 100$$

Waiting Queue Length

This metric indicates the number of requests awaiting execution or processing. It is used by practitioners to identify bottlenecks or performance degradation, especially when examined alongside other metrics. The waiting queue length metric varies based on factors such as the frequency of request issuance, the size of the requests, and the hardware’s

execution rate. It functions as a counter that increments with each request added to the queue and decrements when a request is removed.

Average Wait Time

As previously noted, the hardware does not execute GPU requests immediately; they spend some time in various queues, awaiting processing by the pGPU. This delay, or waiting time, starts when a request is queued in the waiting queue of the guest GPU driver and ends when it is dispatched for execution. It is indeed a key factor in evaluating the performance efficiency of the vGPU, as it directly impacts the total time required to process GPU requests. In addition, analyzing the waiting time can provide some insights into potential performance bottlenecks or inefficiencies in the GPU request handling process. Our framework estimates the average wait time (WT) of GPU requests using the formula below. It calculates it by summing the waiting durations (W) of individual requests observed within a specific period and then dividing this sum by n, the total number of requests.

$$WT = \sum_i \frac{W_i}{n}$$

Average Latency

Furthermore, our framework calculates the average time taken to process GPU requests from issuance to completion. As depicted in Figure 4, this metric encompasses both the waiting time and the execution time. The latter refers to the duration that a GPU engine requires to execute the request. To calculate the average latency, denoted by L in the formula below, our framework sums the waiting (W) and execution (E) times for all requests processed within a given observation period. This sum is then divided by the number of requests, n, to yield the average latency value.

$$L = \sum_i \frac{W_i + E_i}{n}$$

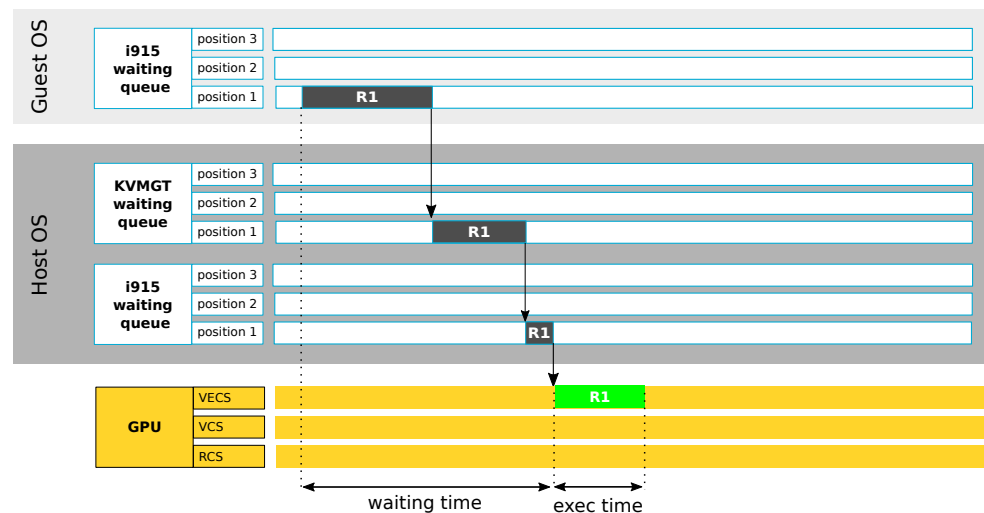


Figure 4. The queuing and processing phases of GPU requests.

GPU Memory Usage

The i915 GPU driver provides deep insights into memory usage through specific events (e.g., `i915_gem_object_create` and `i915_gem_object_destroy`). By tracking these events, we can measure the total memory allocated and subsequently released by Graphics Execution Manager (GEM) objects, representing the dynamic footprint of GPU memory usage. This metric is crucial for understanding how memory allocation and usage impact performance, particularly in virtualized environments where optimizing resource utiliza-

tion is paramount. Using this metric helps pinpoint potential memory-related bottlenecks and inefficiencies both in the host machine and within VMs.

4.3. Visualization

The presentation of information in a trace analysis framework should be made in a manner that enhances the user’s comprehension of the analyzed system. Hence, our framework provides various views, each tailored to illustrate a specific aspect of GVT-g operations. These views have been integrated as plug-ins within Trace Compass, a versatile open-source trace visualization tool. Examples of the visualizations our framework provides include the *GPU usage view*, an XY chart that graphically represents the percentage of utilization of vGPUs and pGPUs. Another visualization is the *latency view*, which displays the durations spent by GPU requests in queues and during execution. This view employs a time graph format, where colored rectangles represent the latencies of individual GPU requests, each marked with its corresponding request ID. Our framework can display the transitions of a request between queues using arrows from the source to the destination.

5. Use Cases

This section aims to demonstrate the practicality and effectiveness of our framework in diagnosing and examining performance issues in GPU-accelerated VMs. We detail the configurations used in our investigations in Table 3.

Table 3. Experimental configurations of the hardware and software.

Host Machine		Virtual Machine	
CPU	Intel Core i7-7500U @2.70GHz	vCPU	2
GPU	HD Graphics 620	vGPU Mem.	Low: 64 MB/High: 384 MB
RAM	16 GB	RAM	4 GB
OS	Ubuntu 16.04 (Kernel 4.14.15)	OS	Ubuntu 16.04 (Kernel 4.15)
Qemu	v2.11.1		
LTtng	v2.10.8		

5.1. GVT-g Scheduling

In this case study, we use our framework to examine the internal mechanics of the GVT-g scheduler, as its operations may impact the performance of GPU virtualization solutions. Specifically, GVT-g uses a kernel thread named *got_service_thr* and multiple additional threads, which are referred to as *workloads thread X*, where “X” represents a specific engine supported by the pGPU, such as render or video command streamers. The *got_service_thr* is responsible for managing vGPU scheduling and context switching, and the *workloads thread X* threads are initiated when a vGPU is activated. These threads concurrently monitor the waiting queue, facilitating the transfer of GPU requests to the host GPU driver, as depicted in Figure 5.

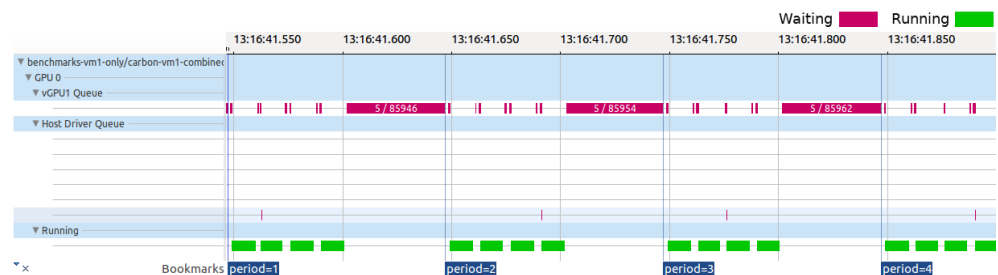


Figure 5. Time-graph view illustrating the wait times of GPU requests in GVT-g and host driver queues, as well as their respective execution times. It also shows how the scheduler defers the execution of vGPU1 requests to the next period when its allocated time slice is exhausted.

Our initial experiment involved two GPU-accelerated virtual machines (VM1 and VM2) sharing a single pGPU via GVT-g, with each VM hosting a similarly configured vGPU and equal resource allocation weights. These weights, assigned by GVT-g at instantiation, determine each vGPU’s share of pGPU resources. We executed a series of benchmarks from the Rodinia GPU benchmark suite [43] on VM1. We repeated these benchmarks after completely shutting down VM2. Figure 6 presents histograms comparing the benchmarks’ execution times in both scenarios. Notably, the benchmarks executed slower when VM2 was idle than when it was shut down, with significant time discrepancies in the Gaussian, Lud, and Heartwall benchmarks. This study considered a VM idle if it did not engage in significant GPU workloads.

To investigate the performance difference, we analyzed the execution of the Gaussian benchmark on VM1 using our framework. Figure 7 illustrates the benchmark’s execution duration and GPU utilization. Initially, the benchmark lasted 59 s with an average GPU utilization of about 45%. In contrast, after shutting down VM2, the benchmark was completed in 27 s with a GPU utilization near 80%. This comparison highlights the performance impact of VM2’s concurrent operation, despite it being inactive.

Further examination of the GVT-g scheduler’s operations revealed its reliance on time multiplexing for fair pGPU resource distribution among vGPUs. The scheduler, which was operated via the *got_service_thr* thread, simultaneously managed all of the GPU engines to minimize scheduling complexity and avoid synchronization bottlenecks. It assigned time slices to vGPUs based on their weights, deducting the aggregate execution time of a vGPU’s requests from its time slice (see the formula below). If a vGPU depletes its time slice within a certain interval (100 ms), the scheduler suspends its operations, recalibrating time slices every ten intervals (one second) and ignoring past debts or credits.

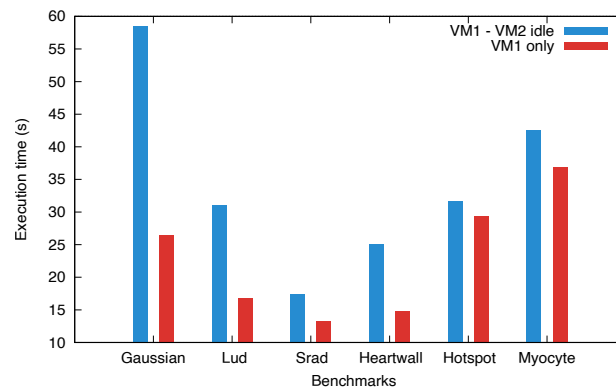


Figure 6. Latencies of benchmarks when executed in multi-VM and single-VM contexts.

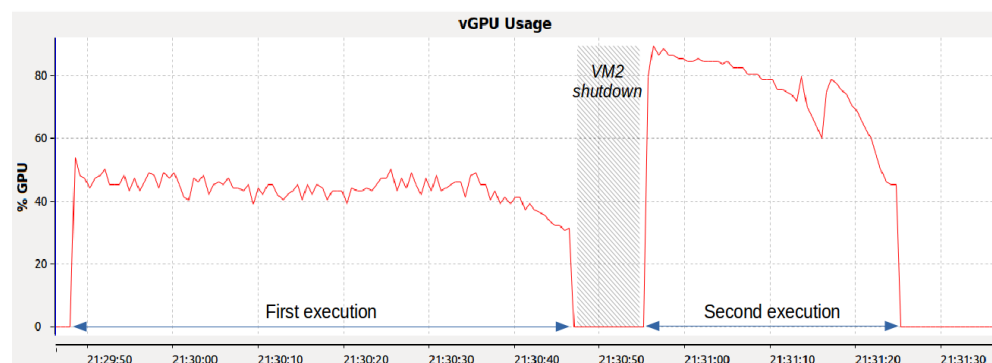


Figure 7. XY view depicting the GPU usage percentages for two consecutive runs of the Gaussian benchmark in VM1. It also highlights the significant increase in GPU usage following the shutdown of VM2, which led to faster execution of the benchmark.

$$Time\ Slice = 100\ ms \times \frac{vGPU_{weight}}{\sum_i GPU_{weight}^i}$$

Example 1. Let us consider two vGPUs sharing the host GPU’s resources, and each is assigned an identical weight. In this scenario, the GVT-g scheduler assigns each vGPU a 50 ms time slice per period. If one vGPU is removed, the remaining one receives the entire 100 ms time slice per period, fully utilizing the available resources.

The GVT-g scheduler follows a round-robin policy, cycling every 1 millisecond to inspect vGPU queues for new requests. Upon finding requests, the scheduler activates the underlying vGPU for processing if its time slice remains. Concurrently, the ‘workloads threads X’ handle the active vGPU’s requests and update statistics for the previously active vGPU. If a vGPU exhausts its time slice, the scheduler then schedules another vGPU, as shown in Figure 5.

In short, our analysis shows that a VM cannot use the idle time slice of another VM due to the GVT-g scheduler policy. While this aligns with typical cloud computing models, it is suboptimal for private cloud data centers seeking to maximize vGPU acceleration. We propose enhancing the GVT-g scheduler with an algorithm that allows a VM to process its GPU workloads when the host GPU resources are unoccupied. A configuration option can be set during startup to allow GVT-g to apply the appropriate scheduling algorithm based on the deployment environment, enabling more efficient GPU resource management.

5.2. Analysis of Contention for Resources

This second use case looks into the extent to which the performance of a vGPU-enabled VM can be affected by the presence of other VMs. In our experiment, two collocated VMs shared host GPU resources via GVT-g, and their vGPUs were configured identically, as detailed in Table 3. On VM1, we ran the Gaussian benchmark, while VM2 executed a variety of benchmarks characterized by various intensities of host GPU resource usage (e.g., varying workload sizes, diverse request frequencies, etc.). For example, Figure 8 demonstrates the effects of workload sizes from these benchmarks on the Gaussian benchmark’s execution time. Notably, the execution time of the Gaussian benchmark increased significantly when run concurrently with the Hotspot benchmark. Our framework view, which is depicted in Figure 9, further illustrates that during the Hotspot benchmark’s execution, GPU usage for the Gaussian benchmark did not exceed 20%, but this rose to 40% after the Hotspot benchmark ended.

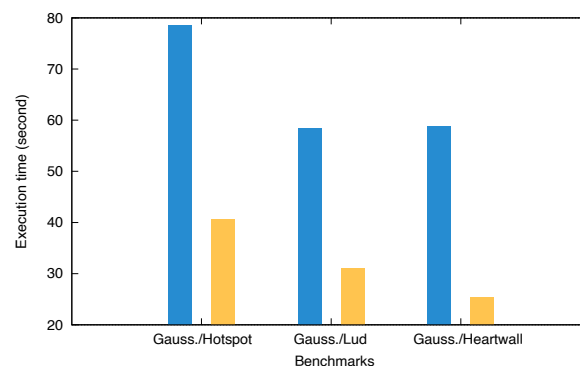


Figure 8. The execution time of the Gaussian benchmark (VM1) increased when it was concurrently executed with the Hotspot benchmark (VM2). Contention for GPU resources between the two VMs is a plausible cause.

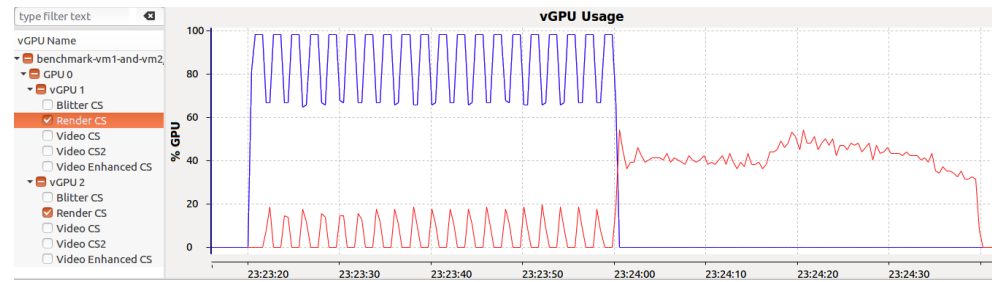


Figure 9. View depicting the GPU usage during concurrent runs of the Gaussian/VM1 (red) and Hotspot/VM2 (blue) benchmarks. It also demonstrates the negative impact of the size of workloads executed by vGPU2 on the performance of vGPU1.

This experiment shows that the performance of one vGPU can be considerably affected by another vGPU that is co-located. Our analysis indicates that this is primarily caused by the disparity in workload sizes: approximately 18 ms for vGPU1 (Gaussian benchmark) versus 800 ms for vGPU2 (Hotspot benchmark). The i915 GPU driver's scheduler lacks preemptive multitasking capabilities, meaning that it cannot interrupt an ongoing workload. Thus, the GVT-g scheduler uses a coarse-grained sharing approach for the host GPU time to compensate. Time slices are allocated to vGPUs in proportion to their weights, and the execution times of GPU requests are subtracted from these slices. Outstanding balances (debts and lefts) from previous periods are carried over for up to ten periods, explaining why the large workloads from vGPU2 adversely affected vGPU1's performance.

Despite enhancing scheduling fairness among vGPUs and, thus, improving VM application responsiveness, this method's drawback lies in its coarse-grained approach to host GPU time-sharing. This becomes particularly problematic when a vGPU's workload size exceeds the period length.

In summary, this use case demonstrates that the GVT-g scheduler struggles to maintain fairness between vGPUs under certain conditions, which is largely due to the i915 driver's lack of preemption support and the implementation of its coarse-grained sharing mechanism. Because the scheduling algorithm's focus is to maintain interactive program responsiveness, we suggest disabling the debt and leftover management feature when VMs are tasked with running non-interactive tasks.

5.3. Virtual Machine Placement

Live migration, a key feature supported by most modern hypervisors, involves relocating VMs to balance the load on physical host machines. This process not only optimizes physical resource utilization within data centers but also ensures adherence to service-level agreements. VM placement, which is a critical component of VM migration, entails selecting the most suitable physical machine to host a VM. Many placement algorithms have been developed over time to maximize resource utilization in cloud data centers and minimize energy consumption. The efficiency of a placement algorithm is paramount, as studies show a strong correlation between resource optimization and the reduction of operational costs in data centers [44].

Effective VM placement algorithms often depend on sophisticated monitoring tools. These tools assess VM resource requirements and identify performance issues stemming from resource overcommitment. Overcommitment leads to VMs competing for shared resources, causing frequent interruptions and prolonged execution times for running programs. To mitigate this, several algorithms co-locate VMs with complementary resource demands on the same host. For example, pairing a CPU-intensive VM with a disk or network-intensive VM can optimize resource utilization and enhance overall performance. Similarly, aligning VMs that alternate between CPU and GPU resource demands within the same host proves beneficial. Hence, co-locating CPU- and GPU-bound VMs optimizes the use of both processing units, which lowers the frequency of VM preemption and boosts the overall system performance.

Seeking to optimize VM placement within our private cloud, which was managed via OpenStack [45], we leveraged our performance analysis framework to incorporate GPU usage data from running VMs as a key input parameter. This optimization involved migrating VMs that exceeded a predefined CPU usage threshold over a certain period to host machines that were mostly engaged with GPU-intensive tasks. This reallocation significantly enhanced the resource utilization across our cloud infrastructure and reduced the execution times of applications within the VMs. The view from our tool in Figure 10 illustrates this strategy, showcasing a VM that alternates between CPU and GPU computing phases.

To validate our approach, we used the *sched_switch*, *kvm_entry*, and *kvm_exit* events to compute the number of VM preemptions. Our method involved tracing the operating system kernels of the cloud host machines and categorizing VMs as either GPU-intensive or CPU-intensive using our analysis framework. For example, we observed several CPU-intensive VMs, such as VM-532 and VM-533, co-located on the same host and frequently preempting each other for CPU resources, as depicted in Figure 11. Conversely, other hosts housing VMs primarily running GPU-intensive programs with minimal CPU dependency were identified. We then tagged VMs accordingly as ‘CPU-intensive’ and ‘GPU-intensive’. Our refined placement algorithm reassigned CPU-intensive VMs to hosts with GPU-intensive VMs when CPU-overcommitment-induced preemptions surpassed a specific threshold. After the implementation of this optimized algorithm, a marked decrease in VM preemptions was observed. Figure 12 illustrates this improvement, showing a significant reduction in the preemptions of VM-532 after VM-535’s relocation, with preemption counts dropping from 23,125 to 15,087.

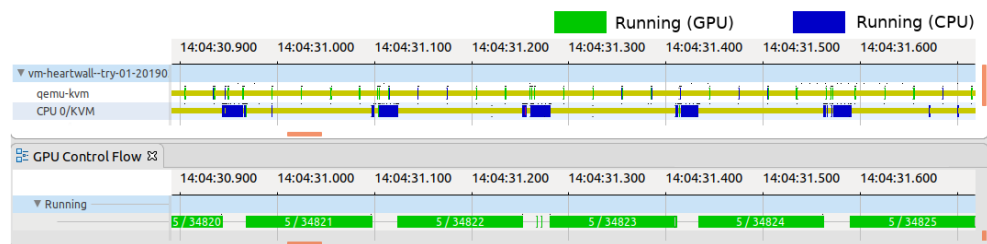


Figure 10. View showing a VM deployed in our private cloud that alternated between CPU-intensive and GPU-accelerated computations.

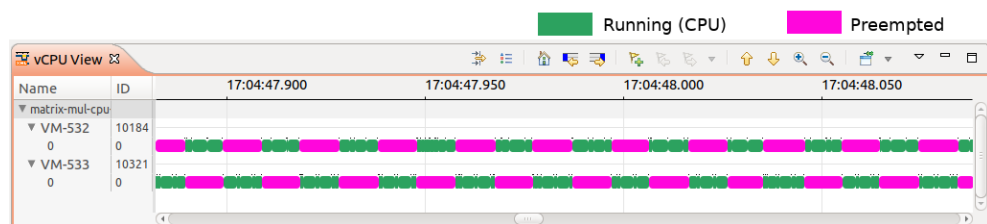


Figure 11. View showing how VM-532 and VM-533 preempted each other as they competed for the same CPU resources.

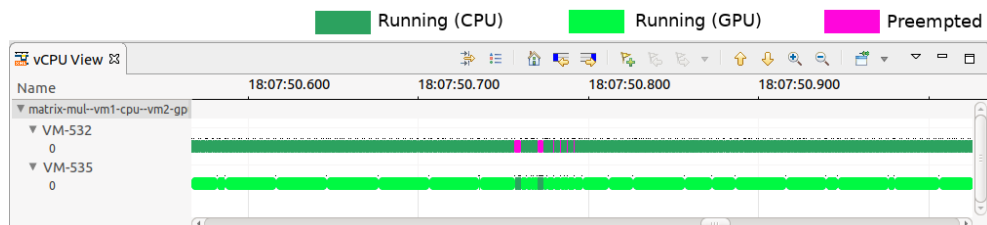


Figure 12. Our analysis shows that after applying the optimized VM placement algorithm, VM-532 was rarely preempted by the GPU-intensive VM-535.

6. Tracing Cost Analysis

This section examines the cost implications associated with the use of our framework—particularly the trace data collection phase. As noted before, our approach relies mainly on efficient

host kernel tracing. However, the overhead introduced by this tracing must be minimal to prevent any significant impact on the performance or behavior of the system under observation.

To estimate the overhead incurred by our tracing method, we conducted a series of experiments using a VM, the specifications of which are detailed in Table 3. This VM was utilized to execute selected benchmarks from the Rodinia benchmark suite [43], which were chosen due to their varied workload sizes and submission frequencies. Our results, as summarized in Table 4, indicate that the operational overhead from active tracepoints was modest, averaging around 1.01%. This low overhead confirmed the efficiency of our tracing approach in gathering runtime data without significantly burdening the system.

Storing trace files can be problematic for most tracing tools, as they generate data in large amounts. This storage cost is mostly influenced by two factors: the event frequency and the chosen storage format. Our findings, which are illustrated in Table 4, show that the size of the trace files did not directly correlate with the benchmark execution times. Rather, it was more closely related to the number of events generated, which, in our context, corresponded to the volume of GPU request submissions. Owing to our strategy of selectively activating a limited number of tracepoints, we managed to keep the trace file sizes within reasonable limits, thus mitigating potential storage issues.

Table 4. Tracing overhead.

Benchmark	Exec Time (ms)	Exec Time w/ Tracing (ms)	# Events	Trace Size (KB)	Overhead * (%)
Gaussian	26,393	26,906	49,255	1228	1.906
Hotspot	29,455	30,161	9122	248	2.341
Lud	16,736	16,862	19,866	512	0.743
Myocyte	36,935	37,061	158,587	3788	0.341
Srad	13,206	13,238	73,973	1843	0.237
Heartwall	14,795	14,873	4110	148	0.525

* The overhead is computed as the percentage increase in execution time with tracing enabled.

7. Conclusions

A GPU is a powerful computing unit with remarkable characteristics, including a highly parallel architecture and energy efficiency. Hence, this device has found extensive application in diverse domains, such as heterogeneous and embedded systems, where it greatly improves application performance. These applications range from autonomous driving to media transcoding and graphics-intensive applications. The interesting attributes of this accelerator have prompted the introduction of vGPU-accelerated VMs, opening up opportunities to address CPU bottlenecks in data centers and enabling end-users to harness the power of high-performance computing devices for a variety of applications.

While our review of the existing literature shows various performance analysis tools for GPUs, support for vGPUs remains a significant gap. Unfortunately, the only tool offering vGPU support, NVIDIA Nsight Systems, is closed-source software and lacks publicly available documentation. To address this gap in research, our study introduces a novel diagnostic tool that is specifically designed to analyze vGPU performance. Utilizing GVT-g, Intel's virtualization technology for on-die GPUs, as a foundational platform, our framework provides detailed insights into GVT-g operations. It facilitates an exhaustive analysis of VMs equipped with vGPUs, thereby enhancing the understanding and optimization of their performance.

Our approach leverages LTTng, a low-overhead tracer, to collect the low-level data required for our analyses. Through this tool, we traced the host machine kernel to gain a better understanding of vGPU resource utilization. Furthermore, we developed a suite of auto-synchronized views in Trace Compass to unveil the intricate dynamics of vGPU-accelerated VMs as they interact with the host GPU resources. Finally, our analysis reveals that the tracing overhead is minimal at approximately 1.01%. Looking ahead, future iterations of our work will expand this approach to encompass other GPU virtualization technologies,

such as MxGPU and NVIDIA Grid. These technologies have gained prominence in the data centers of major cloud providers, making our research even more pertinent and valuable in the evolving landscape of GPU virtualization.

Author Contributions: Conceptualization, A.B.; Methodology, A.B.; Software, A.B.; Validation, A.B.; Investigation, A.B.; Writing – original draft, A.B.; Supervision, M.D.; Funding acquisition, M.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was financed by the Natural Sciences and Engineering Research Council of Canada (NSERC), through the NSERC Alliance project 554158-20, in partnership with Prompt, Ericsson, Ciena, AMD, and EfficiOS. The APC was funded by Adel Belkhiri.

Data Availability Statement: The source codes of the proposed framework are available on the author’s GitHub: <https://github.com/adel-belkhiri> (accessed on 20 February 2024).

Acknowledgments: We would like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena, AMD, and EfficiOS for supporting this research.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hong, C.H.; Spence, I.; Nikolopoulos, D.S. FairGV: Fair and Fast GPU Virtualization. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 3472–3485. [CrossRef]
2. Ji, Z.; Wang, C.L. Compiler-Directed Incremental Checkpointing for Low Latency GPU Preemption. In Proceedings of the 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Lyon, France, 30 May–3 June 2022; pp. 751–761. [CrossRef]
3. Hong, C.H.; Spence, I.; Nikolopoulos, D.S. GPU Virtualization and Scheduling Methods: A Comprehensive Survey. *ACM Comput. Surv.* **2017**, *50*, 1–37. [CrossRef]
4. NVIDIA. GP100 Pascal Whitepaper. 2016. Available online: <https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf> (accessed on 20 February 2024).
5. Nvidia Grid: Graphics Accelerated VDI with the Visual Performance of a Workstation. 2013. Available online: <http://www.nvidia.com/content/grid/vdi-whitepaper.pdf> (accessed on 20 February 2024).
6. AMD MxGPU. 2024. Available online: <https://www.amd.com/en/graphics/workstation-virtualization-solutions> (accessed on 20 February 2024).
7. Tian, K.; Dong, Y.; Cowperthwaite, D. A Full GPU Virtualization Solution with Mediated Pass-through. In Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC’14, Philadelphia, PA, USA, 19–20 June 2014; USENIX Association: Berkeley, CA, USA, 2014; pp. 121–132.
8. Aceto, G.; Botta, A.; Donato, W.; Pescapè, A. Cloud monitoring: A survey. *Comput. Netw.* **2013**, *57*, 2093–2115. [CrossRef]
9. Couturier, D.; Dagenais, M.R. LTTng CLUST: A System-wide Unified CPU and GPU Tracing Tool for OpenCL Applications. *Adv. Softw. Eng.* **2015**, *2015*, 940628. [CrossRef]
10. Margheritta, P.; Dagenais, M.R. LTTng-HSA: Bringing LTTng tracing to HSA-based GPU runtimes. *Concurr. Comput. Pract. Exp.* **2019**, *31*, e5231. [CrossRef]
11. Intel VTune Amplifier. 2024. Available online: <https://software.intel.com/en-us/intel-vtune-amplifier-xe> (accessed on 20 February 2024).
12. Nvidia Nsight Graphics. 2024. Available online: <https://developer.nvidia.com/nsight-graphics> (accessed on 20 February 2024).
13. Song, J. KVMGT: A Full GPU Virtualization Solution. In Proceedings of the KVM Forum, Düsseldorf, Germany, 14–16 October 2014; The Linux Foundation: San Francisco, CA, USA, 2014.
14. LTTng Project. The LTTng Documentation. 2024. Available online: <https://ltnng.org/docs/v2.10/> (accessed on 20 February 2024).
15. Trace Compass. 2024. Available online: <http://tracecompass.org/> (accessed on 20 February 2024).
16. Abramson, D.; Jackson, J.; Muthrasanallur, S.; Neiger, G.; Regnier, G.; Sankaran, R.; Schoinas, I.; Uhlig, R.; Vembu, B.; Wiegert, J. Intel Virtualization Technology for Directed I/O. *Intel Technol. J.* **2006**, *10*, 179–192. [CrossRef]
17. van Doorn, L. Hardware Virtualization Trends. In Proceedings of the 2nd International Conference on Virtual Execution Environments, VEE ’06, Ottawa, ON, Canada, 14–16 June 2006; ACM: New York, NY, USA, 2006; p. 45.
18. GPUs on Compute Engine. 2024. Available online: <https://cloud.google.com/compute/docs/gpus/> (accessed on 20 February 2024).
19. Zhang, J.; Lu, X.; Panda, D.K. Performance Characterization of Hypervisor-and Container-Based Virtualization for HPC on SR-IOV Enabled InfiniBand Clusters. In Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), Chicago, IL, USA, 23–27 May 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1777–1784.
20. Zhang, J.; Lu, X.; Panda, D.K. High-Performance Virtual Machine Migration Framework for MPI Applications on SR-IOV Enabled InfiniBand Clusters. In Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Orlando, FL, USA, 29 May–2 June 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 143–152.

21. Gupta, V.; Gavrilovska, A.; Schwan, K.; Khariche, H.; Tolia, N.; Talwar, V.; Ranganathan, P. GViM: GPU-accelerated Virtual Machines. In Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing, HPCVirt '09, Nuremberg, Germany, 31 March 2009; ACM: New York, NY, USA, 2009; pp. 17–24.
22. Gupta, V.; Schwan, K.; Tolia, N.; Talwar, V.; Ranganathan, P. Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems. In Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference, USENIXATC'11, Portland, OR, USA, 15–17 June 2011; USENIX Association: Berkeley, CA, USA, 2011; p. 3.
23. Duato, J.; Peña, A.J.; Silla, F.; Mayo, R.; Quintana-Orti, E.S. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In Proceedings of the International Conference on High Performance Computing Simulation, Caen, France, 28 June–2 July 2010; IEEE: Piscataway, NJ, USA, 2010; pp. 224–231.
24. Lee, C.; Kim, S.W.; Yoo, C. VADI: GPU Virtualization for an Automotive Platform. *IEEE Trans. Ind. Inform.* **2016**, *12*, 277–290. [[CrossRef](#)]
25. Shi, L.; Chen, H.; Sun, J.; Li, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. *IEEE Trans. Comput.* **2012**, *61*, 804–816. [[CrossRef](#)]
26. Ma, J.; Zheng, X.; Dong, Y.; Li, W.; Qi, Z.; He, B.; Guan, H. gMig: Efficient GPU Live Migration Optimized by Software Dirty Page for Full Virtualization. In Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '18, Williamsburg, VA, USA, 24–25 March 2018; ACM: New York, NY, USA, 2018; pp. 31–44.
27. Dong, Y.; Xue, M.; Zheng, X.; Wang, J.; Qi, Z.; Guan, H. Boosting GPU Virtualization Performance with Hybrid Shadow Page Tables. In Proceedings of the USENIX Annual Technical Conference, Santa Clara, CA, USA, 8–10 July 2015; USENIX Association: Berkeley, CA, USA, 2015; pp. 517–528.
28. Xue, M.; Tian, K.; Dong, Y.; Ma, J.; Wang, J.; Qi, Z.; He, B.; Guan, H. gScale: Scaling up GPU Virtualization with Dynamic Sharing of Graphics Memory Space. In Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC 16), Denver, CO, USA, 22–24 June 2016; USENIX Association: Berkeley, CA, USA, 2016; pp. 579–590.
29. Devices, A.M. AMD GPU Open-Radeon GPU Profiler. 2024. Available online: <https://gpuopen.com/rgp/> (accessed on 20 February 2024).
30. Gupta, R.; Shen, X.; Zhou, K.; Krentel, M.; Mellor-Crummey, J. A tool for top-down performance analysis of GPU-accelerated applications. In Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA, 22–26 February 2020; pp. 415–416. [[CrossRef](#)]
31. Cherian, A.T.; Zhou, K.; Grubisic, D.; Meng, X.; Mellor-Crummey, J. Measurement and Analysis of GPU-Accelerated OpenCL Computations on Intel GPUs. In Proceedings of the 2021 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools), St. Louis, MO, USA, 14 November 2021. [[CrossRef](#)]
32. TAU Performance System. 2024. Available online: <http://www.paratools.com/tau> (accessed on 20 February 2024).
33. Zhou, K.; Hao, Y.; Mellor-Crummey, J.; Meng, X.; Liu, X. GVPROF: A Value Profiler for GPU-Based Clusters. In Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis, Atlanta, GA, USA, 9–19 November 2020; pp. 1–16. [[CrossRef](#)]
34. Falsafi, B.; Ferdman, M.; Lu, S.; Wenisch, T.; Zhou, K.; Hao, Y.; Mellor-Crummey, J.; Meng, X.; Liu, X. ValueExpert: Exploring value patterns in GPU-accelerated applications. In Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022; pp. 171–185. [[CrossRef](#)]
35. Zhou, K.; Meng, X.; Sai, R.; Grubisic, D.; Mellor-Crummey, J. An Automated Tool for Analysis and Tuning of GPU-Accelerated Code in HPC Applications. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *33*, 854–865. [[CrossRef](#)]
36. Hao, Y.; Jain, N.; Van der Wijngaart, R.; Saxena, N.; Fan, Y.; Liu, X. DrGPU: A Top-Down Profiler for GPU Applications. In Proceedings of the 2023 ACM/SPEC International Conference on Performance Engineering, London, UK, 7–11 May 2023; pp. 43–53. [[CrossRef](#)]
37. Knoop, J.; Schordan, M.; Johnson, T.; O'Boyle, M.; Shen, D.; Song, S.L.; Li, A.; Liu, X. CUDAAdvisor: LLVM-based runtime profiling for modern GPUs. In Proceedings of the 2018 International Symposium on Code Generation and Optimization, Vienna, Austria, 24–28 February 2018; pp. 214–227. [[CrossRef](#)]
38. Kernel Virtual Machine. 2019. Available online: <https://www.linux-kvm.org> (accessed on 20 February 2024).
39. QEMU. 2024. Available online: <https://www.qemu.org/> (accessed on 20 February 2024).
40. Ftrace-Function Tracer. 2018. Available online: <https://www.kernel.org/doc/Documentation/trace/ftrace.txt> (accessed on 20 February 2024).
41. Perf: Linux Profiling with Performance Counters. 2024. Available online: <https://perf.wiki.kernel.org> (accessed on 20 February 2024).
42. Montplaisir-Gonçalves, A.; Ezzati-Jivan, N.; Wininger, F.; Dagenais, M.R. State History Tree: An Incremental Disk-Based Data Structure for Very Large Interval Data. In Proceedings of the 2013 International Conference on Social Computing, Alexandria, VA, USA, 8–14 September 2013; IEEE: Piscataway, NJ, USA, 2013; pp. 716–724.
43. Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J.W.; Lee, S.; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 4–6 October 2009; IEEE: Piscataway, NJ, USA, 2009; pp. 44–54.
44. Masdari, M.; Nabavi, S.S.; Ahmadi, V. An overview of virtual machine placement schemes in cloud computing. *J. Netw. Comput. Appl.* **2016**, *66*, 106–127. [[CrossRef](#)]

-
45. Open Infrastructure Foundation. OpenStack: Open Source Cloud Computing Infrastructure. 2024. Available online: <https://www.openstack.org/> (accessed on 20 February 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.