2021 IEEE International Performance, Computing, and Communications Conference (IPCCC) | 978-1-6654-4331-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/IPCCC51483.2021.9679439

Diagnosing the Interference on CPU-GPU Synchronization Caused by CPU Sharing in Multi-Tenant GPU Clouds

Youssef Elmougy
Hofstra University
New York, USA
yelmougy1@pride.hofstra.edu

Weiwei Jia
Xiaoning Ding
New Jersey Institute of Technology
New Jersey, USA
wj47@njit.edu
xiaoning.ding@njit.edu

Jianchen Shan
Hofstra University
New York, USA
jianchen.shan@hofstra.edu

***Abstract*—The GPU-accelerated cloud, enabled by maturing GPU virtualization techniques, has become the most attractive platform for high-performance computing and machine learning workloads. However, it is notoriously challenging to build the multi-tenant GPU cloud where resources, like CPUs and GPUs, can be shared. One well-known and heavily studied reason is that workloads suffer from poor performance isolation and low GPU utilization when GPUs are shared. But little attention has been paid to another fundamental yet under studied problem: how sharing CPUs among GPU instances could affect the workload performance?**

Targeting this problem, the paper conducts experiments to measure the performance slowdown and vGPU utilization decrease under interference from CPU sharing. The results show that GPU workloads suffer from poor and unpredictable performance and heavy vGPU under-utilization because of CPU sharing. We find that such interference is the result of the complex interplay between the characteristics of CPU-GPU interactions and the special behavior of shared vCPUs: vCPU discontinuity. To diagnose how vCPU discontinuity causes the interference, the paper leverages NVIDIA Nsight Systems for fine-grained profiling and has the following findings: 1) vCPU discontinuity causes inefficient CPU-GPU synchronizations; 2) vCPU discontinuity delays task offloading to the vGPU; 3) Polling-based CPU-GPU synchronization suffers from interference more than blocking-based CPU-GPU synchronization; 4) GPU workloads with frequent task offloads and synchronizations are more vulnerable. Based on the findings, the paper proposes a novel *polling-then-blocking* CPU-GPU synchronization primitive. Evaluation shows that it can improve the performance by 4.2x.

I. Introduction

With the maturity of GPU virtualization techniques, GPU-based clouds are starting to gain momentum. Studies show that the GPU Passthrough [1], [2] technique allows virtual machines (VMs) to access GPUs with almost native performance. Hence, major cloud providers [3]–[7] now offer instances accelerated by virtual GPUs (vGPUs). For example, Amazon AWS provisions P3 instances on the EC2 platform containing up to 8 vGPUs and 96 virtual CPUs (vCPUs). In addition, some GPU instances are specifically optimized for high-performance computing and machine learning [3], [4].

However, it is still challenging to build multi-tenant GPU clouds where resources, like GPUs and CPUs, are shared among instances. One reason being that workloads suffer from poor performance isolation and low utilization when instances share GPUs. Many efforts have been made to address these issues [8]–[12]. But little attention has been paid to another fundamental yet under studied problem: *how sharing CPUs among GPU instances could affect the workload performance?*

It is crucial to answer this question for two reasons. First, many cloud providers have already started provisioning GPU instances with shared vCPUs (e.g. Oracle VM.GPU2/3 and IBM AC1/2). Analyzing the performance interference can provide a guideline for users to make more informed decisions regarding workload deployment and motivate cloud providers to make related optimizations. Second, sharing vCPUs can increase the system throughput and lower cost, since vCPUs only handle light tasks, such as device communication and synchronization, while the heavy tasks are offloaded to vGPUs. Though these vCPU tasks are often multi-threaded and remain in the critical path of the GPU workflow, thus, are vulnerable to any interference.

Through extensive experiments, we observed that workloads may suffer from poor and unpredictable performance under interference caused by CPU sharing in multi-tenant GPU clouds. Figure 1 shows performance degradation of various GPU workloads, including NAMD [13], GROMACS [14], and two programs from the Rodinia Benchmark Suite [15], under different interference scenarios caused by a while(1) loop workload (i.e. synthetic) and two real workloads from the PARSEC Benchmark Suite [16]. To mimic the multi-tenant GPU clouds, two VMs sharing CPUs are created with one running a GPU workload and the other running a CPU workload as the background interference. Using GPU Passthrough, one dedicated GPU is attached to the VM running GPU workloads to eliminate the effects from vGPU sharing (see Section III for detailed settings).

As shown in Figure 1, the performance slowdowns vary substantially across different GPU workloads. For example, the CPU sharing slowed down NAMD and GROMACS by up to 6.7x while the performance of r.particle_filter and r.srad are barely degraded under different interference scenarios. Also, performance degradation becomes unpredictable when the co-running interference changes. For example, while NAMD suffers a 1.8x slowdown with streamcluster and fluidanimate, its degradation with the synthetic workload is as high as 6.7x.

978-1-6654-4331-9/21/$31.00 ©2021 IEEE

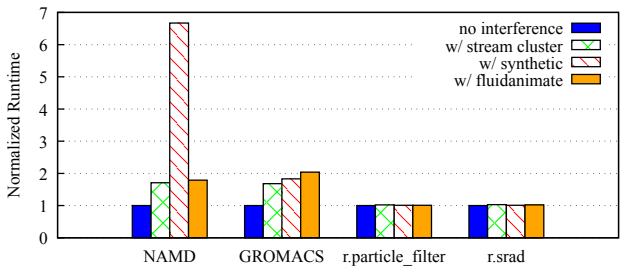

Fig. 1: The runtime slowdowns of GPU workloads under different CPU-sharing interference in multi-tenant clouds.

Similarly, the slowdowns of GROMACS ranges from 1.7x to 2.1x under different background interference. More interestingly, the performance slowdowns of different workloads vary even under the same type of interference. For example, both NAMD and GROMACS are slowed down by around 1.7x with streamcluster. In contrast, while the synthetic workload causes a 6.7x slowdown to NAMD, it only degrades the performance of GROMACS by 1.8x.

The slowdown and its unpredictability under different interferences are the result of complex interplay between the characteristics of the GPU workload, especially tasks handled by vCPUs, and the behaviours of the shared vCPUs. To enable fair CPU sharing with low latency, the hypervisor would constantly deschedule one vCPU from a CPU to reschedule another vCPU. This unique behaviour is called vCPU discontinuity. Many studies [17]–[20] have been conducted to understand, in multi-tenant clouds, how vCPU discontinuity influences performance of different types of CPU workloads, such as synchronization-intensive workloads [21] and I/O-intensive workloads [22]. However, how vCPU discontinuity interacts with GPU workloads remains unknown, and understanding it would be crucial to analyzing performance interference caused by CPU sharing in multi-tenant GPU clouds.

The paper aims to diagnose the performance interference by revealing how vCPU discontinuity affects GPU workloads. The investigation is done by leveraging NVIDIA Nsight Systems [23], which can visualize the timeline of a workload's CPU and GPU activities, to compare the workload execution timelines with and without vCPU discontinuity side-by-side. With the help of the fine-grained profiling, the paper has the following findings: 1) vCPU discontinuity causes inefficient CPU-GPU synchronizations; 2) vCPU discontinuity delays task offloading to the GPU and detection of kernel completion, with the delays observed to be in the order of milliseconds; 3) Polling-based CPU-GPU synchronization suffers from the previous two problems more than blocking-based CPU-GPU synchronization due to its quick depletion of the vCPU time slice leading to more frequent vCPU descheduling; 4) The delays in CPU-GPU synchronizations and GPU offloading increase the workload runtime and result in heavy vGPU under-utilization. In other words, GPU workloads with frequent task offloads and synchronizations are vulnerable to the performance interference from vCPU discontinuity.

To demonstrate the above findings, the paper thoroughly analyzes and profiles the NAMD benchmark as a case study. NAMD is selected for the following reasons. First, NAMD is a representative scientific workload. The observations from studying it can be applicable to many other similar workloads. Second, NAMD is implemented using CUDA [24] which is a widely used paradigm in GPU-based clouds. The application offers two versions which use polling-based and blocking-based CPU-GPU synchronizations respectively, allowing the paper to comprehensively analyze both scenarios. Last but not least, NAMD frequently launches fine-grained kernels and suffers from poor and unpredictable performance, which makes it the perfect example to demonstrate the performance influence and vGPU under-utilization in detail.

Based on the findings, the paper proposes a novel *polling-then-blocking* CPU-GPU synchronization primitive to mitigate the performance interference and vGPU under-utilization. This hybrid optimization is driven by the predictability of kernel execution times, which deviate only by 2%-5% from their average value. This inspired a design that runs a continuously updated online profiler gathering optimal polling intervals then allowing vCPU to poll for this defined period before blocking for completion in order to save the vCPU time slice for critical path tasks. The evaluations show that this hybrid CPU-GPU synchronization method improves the performance of NAMD by 4.2x and 1.9x compared to NAMD using polling-based and blocking-based CPU-GPU synchronizations respectively.

The contributions of the paper are summarized as follows:

1) The paper is the first to systematically diagnose how CPU sharing among GPU instances in clouds can affect the GPU workloads.
2) The paper leverages fine-grained profiling to reveal how vCPU discontinuity causes GPU workload performance degradation and vGPU under-utilization.
3) The paper thoroughly investigates the NAMD implemented with CUDA as a case study to demonstrate the delays of CPU-GPU synchronizations and GPU kernel offload.
4) The paper proposes a novel *polling-then-blocking* CPU-GPU synchronization method to mitigate the performance interference.

The remainder of the paper is organized as follows. Section II provides the background of GPU virtualization, vCPU discontinuity, and the workflow of GPU workloads. Section III describes the motivating experiments that evaluate GPU workload performance degradation and vGPU under-utilization. Section IV thoroughly profiles and analyzes NAMD using NVIDIA Nsight Systems. Section V proposes a novel *polling-then-blocking* CPU-GPU synchronization method to optimize performance in the GPU cloud. Section VI introduces the related works, and Section VII concludes the paper.

II. Background

In this section, we aim to introduce the most prevailing details regarding multi-tenant GPU clouds. Firstly, the current utilized techniques for attaching shared or dedicated GPU resources to VMs in the GPU cloud are presented

(Section II-A). Secondly, the behavior of vCPUs utilizing shared CPU resources in the GPU cloud is detailed, with an introduction to the issue of vCPU discontinuity (Section II-B). Lastly, we analyze the precise workflow of GPU applications as well as the CPU-GPU synchronization techniques and their respective use of CUDA events (Section II-C).

A. GPU Cloud Powered by Virtualization

GPUs are massively parallel processing devices which are designed for compute-intensive and high throughput applications such as those in machine learning, academic research, scientific simulation, biomolecular dynamics, image processing, and much more. Their ability to exploit hundreds to thousands of cores in accelerating parallel computing tasks has made it possible to offload critical tasks to the GPUs. Our focus is geared towards NVIDIA GPUs due to their widespread use in the GPU cloud, as well as their popular programming paradigm NVIDIA's Compute Unified Device Architecture (CUDA) [24]. In virtualized environments, cloud instances can access either shared or dedicated vGPU resources through API forwarding, device emulation, or hardware virtualization. GPU cloud instances powered by shared resources involves sharing of a physical GPU device simultaneously among several cloud instances. The virtualization technologies that make sharing of vGPU devices on the cloud possible include NVIDIA GRID vGPU [25], Intel KVMGT [26], and AMD MxGPU [27].

Allocating a dedicated vGPU device to a cloud instance allows for bare-metal performance [28], [29]. GPU-cloud instances powered by dedicated resources involves gaining direct control on the GPU device, with exclusive access for the full duration of the cloud instance. The virtualization technology that is most widely used in the cloud is PCI Passthrough [1]. Since this technique allows the GPU device to perform as if it were physically attached to the cloud instance, it provides increased performance and better fidelity.

Critical compute-intensive tasks being offloaded to vGPUs means cloud instances require higher distribution of resources towards the vGPU as compared to the vCPU. Hence, GPU cloud instances desiring high performance benefits favour utilizing dedicated vGPU devices while sharing vCPU resources with other instances due to its weighted benefits.

B. vCPU Discontinuity Interference

Hypervisors in virtualized environments multiplex CPU resources to provide vCPU resources for VMs. In multi-tenant clouds, it is a very common occurrence that the number of active vCPUs surpasses the number of available CPU cores. Due to this overcommitting, active vCPUs are forced to share CPU resources, with each vCPU receiving an equal share of the time slice intermittently. In order to provide each vCPU with its fair share of resources, the hypervisor deschedules a vCPU upon the completion of its full time slice to reschedule another vCPU. This is referred to as vCPU discontinuity. To increase efficiency and reduce costs associated with context switch, the length of time slices is commonly in the tens of milliseconds.

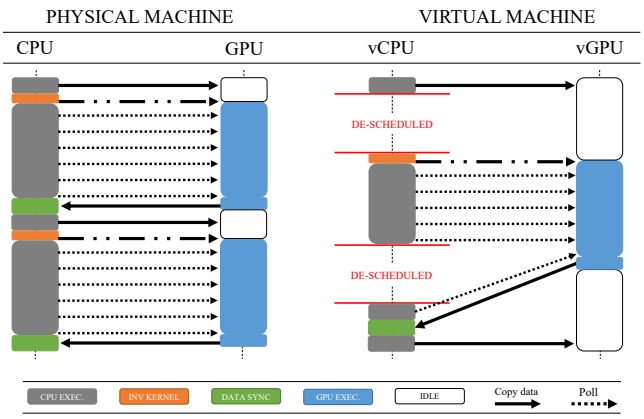

Fig. 2: Comparison of timelines of the execution cycle of a GPU application using Polling CPU-GPU synchronization technique on a PM and a VM. The CPU continuously polls for GPU kernel completion utilizing *cudaEventQuery*.

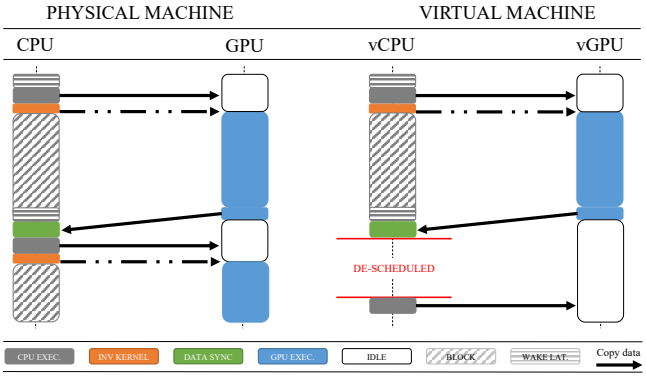

Fig. 3: Comparison of timelines of the execution cycle of a GPU application using Blocking CPU-GPU synchronization technique on a PM and a VM. The CPU blocks until notification of GPU kernel completion by *cudaEventSynchronize*.

The immediate descheduling of a vCPU at the completion of its time slice releases the vCPU's ability to offload kernels or to process its own light tasks. The vCPU is only able to process these tasks when it receives its next time slice. In comparison, physical machines are able to immediately process these tasks without risk of CPU descheduling. Figures 2 and 3 show this comparison in workflow execution. In virtual machines, the continuous descheduling caused by vCPU discontinuity induces heavy interference to workloads due to its effect of delaying tasks on the critical path such as kernel invocation or data synchronization. This delay is greatly affected by the length of the time slice, where the delay can be in the order of tens of milliseconds dependant on the number of vCPUs sharing the available CPU cores.

C. Workloads in GPU Cloud

The regular workflow of a GPU application is a cycle of: (1) initialization of variables, memory allocation and transfer, and execution of light functions by the CPU; (2) offloading of computational intensive tasks to the GPU in the form of

GPU kernels; (3) asynchronous execution of tasks and continuous CPU-GPU synchronization probing for GPU kernel completion by the CPU; (4) memory transfer and execution of completed kernel on the CPU. The timeline of this workflow on the physical machine and virtual machine is shown in Figures 2 and 3, with the virtual machine workflow affected by vCPU discontinuity interference. It is important to notice that applications and workloads with high frequency of CPU-GPU synchronization are more vulnerable to performance degradations, especially while sharing vCPUs. If continuous CPU-GPU synchronization and communication is required as part of the critical path, then vCPU discontinuity interference can delay the required and critical device synchronization. Hence, *synchronization-intensive workloads* are most vulnerable in the GPU cloud.

It is worth mentioning some details of the execution sequence of GPU applications for clarity in subsequent discussions. CPUs run multiple *threads* which execute CPU tasks in parallel, where the number of threads determines the number of tasks that can be performed simultaneously. A function in the application code with the purpose of execution on the GPU is referred to as a *kernel*. GPU kernels consist of the computationally intensive and high throughput tasks of an application to maximize on the GPU's programming power. After initialization of variables and memory allocation, a CPU is responsible for copying data to the GPU's memory and launching a kernel. As the kernel is executed, the CPU resumes with other tasks while synchronously coordinating communication between itself and the GPU. It must be noted that GPUs do not have the ability to directly communicate with CPUs, hence the only signaling mechanism available to signal kernel completion is by setting a flag value in memory [30]. The process of a CPU checking for this flag value is referred to as *CPU-GPU synchronization*. Different CPU-GPU synchronization techniques are discussed further in this section. Since the CPU's performance is dire to this cycle, multiple shared CPUs are used to establish better task parallelization and increase the performance of the application.

In order for CPUs to receive notification of completed kernels, in CUDA, two specific event management functions, *cudaEventQuery* and *cudaEventSynchronize* [31], can be used. CUDA events are synchronization markers, where the event is setup to check the status of all the work in a GPU kernel. The former function, *cudaEventQuery*, is used to query the status of the work captured by an event, while the latter function, *cudaEventSynchronize*, is used to block and wait until the completion of all work captured by an event.

CPUs synchronize and communicate with the GPU through CUDA events that check the status of completion of invoked kernels. There are two main CPU-GPU synchronization techniques: *polling* and *blocking* [32]. In the former synchronization technique, a CPU thread continuously calls *cudaEventQuery* to poll the status of the kernel until the event is returned a success indicating kernel completion. Due to the nature of constant checking for completion, polling uses the full resources and time slice of the vCPU. In the latter synchronization technique, a CPU calls *cudaEventSynchronize* which causes the CPU to sleep or block until the event is returned a success indicating kernel completion. Unlike polling, blocking does not use the full resources of the vCPU, but it introduces a crucial latency in waking up the vCPU thread, due to the costly vCPU switches [33], and other potential issues discussed in later sections.

III. Measuring Performance Interference

To understand how vCPU discontinuity interference affects GPU workloads, we conducted experiments to measure the performance slowdowns and decreases in vGPU utilization under conditions as those in multi-tenant GPU clouds. In this section, we detail our cloud environment as well as the CPU and GPU benchmarks used. We then present the experiments that leverage this cloud environment and discuss the results.

A. Environmental Setup

Our cloud environment consisted of two VMs: one VM (named `corunner`) serving as a CPU cloud instance is equipped with a specified number of vCPUs, and the other VM (named `GPUrunner`) serving as a GPU cloud instance is equipped with the same specified number of vCPUs and one vGPU. The vCPUs of both the `corunner` and the `GPUrunner` VMs share the same set of CPUs on creation using the technique of `vcpupin`. CPU bandwidth control is used to mitigate the performance isolation issue by ensuring both VMs use a fair share of their allotted vCPU time. Furthermore, a dedicated GPU device was attached to the `GPUrunner` VM to eliminate the interference from GPU sharing between instances.

The host system used is a HPE ProLiant DL385Gen10 server with 256GB of memory, 4 Intel Xeon Gold 6138 20-core processors, and 2 NVIDIA Tesla P100 GPUs. We created VMs on the server with 16, 32, and 64 vCPUs to measure the workload performance and vGPU utilization in relation to the number of vCPUs. The VMM is KVM [34]. Ubuntu Server 20.04.2 (kernel 5.8.0) is the OS for both the host and the guests. The GPU, with CUDA 10.1 as the driver, is attached as vGPU to the `GPUrunner` VM using PCI Passthrough.

B. Benchmarks

The CPU-intensive benchmarks used to provide background interference were a synthetic benchmark, consisting of a while(1) loop, matrix multiplication program (i.e. matmul), and p.stream_cluster, p.dedup, p.x264, and p.fluidanimate from the PARSEC Benchmark Suite [16]. The CPU-GPU synchronization-intensive benchmarks executing in the GPU instance `GPUrunner` were NAMD [13] and GROMACS [14].

C. Scenarios

To measure the performance of the vGPU while sharing vCPU resources and accurately compare the findings with baseline performance results, we conducted three sets of experiments for each CPU-GPU synchronization-intensive benchmark: (1) **Shared1**: `GPUrunner` VM executing a benchmark

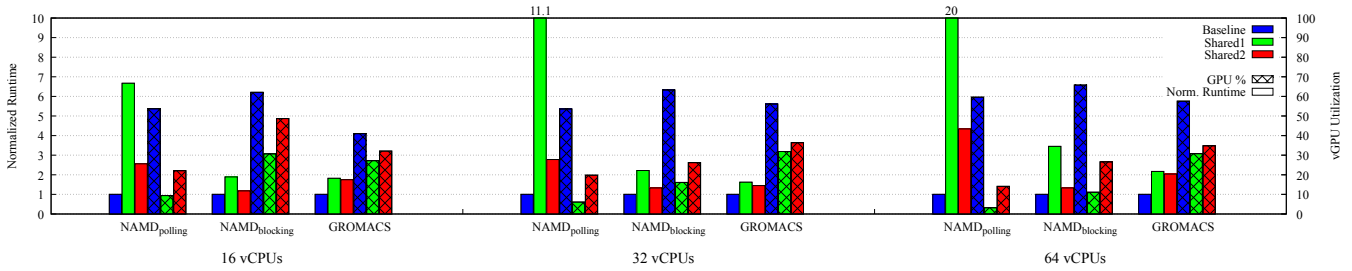

Fig. 4: The runtime slowdowns of GPU workloads under interference from synthetic benchmark sharing 16, 32, 64, vCPUs.

simultaneously along with the `corunner` VM executing a CPU-intensive benchmark *without* performance isolation enforced for CPU sharing, (2) **Shared2**: `GPUrunner` VM executing a benchmark simultaneously along with the `corunner` VM executing a CPU-intensive benchmark *with* performance isolation enforced for CPU sharing using CPU bandwidth control, and (3) **Baseline**: `GPUrunner` VM executing a benchmark with half the number of specified vCPUs along with no execution from the `corunner` VM. These experimental setups are summarized in Table I.

TABLE I: Experimental Setups

Experiment	`corunner`	`GPUrunner`	Perf. Iso.	#vCPUs
Shared1	✓	✓	✗	16, 32, 64
Shared2	✓	✓	✓	16, 32, 64
Baseline	✗	✓	N.A	8, 16, 32

The percentage utilization of the vCPU, vGPU, and workload execution time while running each benchmark was recorded. Each experiment was performed on VMs running a set of 16, 32, or 64 vCPUs. The Baseline experiment type provides the ideal comparable performance of a GPU cloud instance without interference from CPU sharing. Assuming there is no degradation issue from sharing vCPUs, then a scenario where a VM is assigned to 16 shared vCPUs and a single vGPU should perform equivalently to a scenario where a VM is assigned to 8 dedicated vCPUs and a single vGPU. However, we know that sharing vCPUs among cloud instances may cause interference, hence we quantify this scenario using the Shared1 and Shared2 experiment types. Shared1 provides a scenario where 2 cloud instances execute intensive workloads with no mechanism controlling the fair utilization of the shared vCPU resources. On the other hand, Shared2 provides a similar scenario where 2 cloud instances execute intensive workloads employing a performance isolation mechanism to fairly divide shared resources among the instances.

D. Results

The benchmark runtime slowdown results for 16, 32, and 64 vCPUs while running against interference from the synthetic benchmark are shown in Figure 4, while Figure 5 shows the runtime slowdown of the benchmarks co-run with different interference benchmarks on 16 vCPUs. It can be seen that there is a significant slowdown in workload execution time and heavy vGPU under-utilization across all benchmarks, which can be credited to the interference issues caused by vCPU discontinuity. In this section, we leverage the cloud environment to analyze the performance of GPU workloads under interference from vCPU discontinuity and resource overcommitment. The performance of the benchmarks co-run with interference from the synthetic benchmark is used to analyze these findings due to its consistent maximum interference across all benchmarks. The findings are analyzed below.

1) Degradation in Runtime Performance: Across all benchmarks, the program execution time was slowed down by 1.2x to as much as 20x as shown in Figure 4. In the Shared1 scenario, as compared to the Baseline, the performance of the NAMD benchmarks' execution times were decreased by 47% to 85% and that of GROMACS was decreased by 45% while sharing 16 vCPUs. Likewise, in the Shared2 scenario, as compared to the Baseline, the performance of the NAMD with Polling, NAMD with Blocking, and GROMACS benchmark's execution times were decreased by 15%, 61%, and 43% respectively while sharing 16 vCPUs. The reason for this evident degradation can be credited to vCPU discontinuity interference on critical path tasks. The constant descheduling and rescheduling of vCPUs during program execution causes vulnerability to vCPU inability to offload kernels or execute light tasks. With interference to these critical tasks, the benchmarks experience a constantly increasing delay at each time slice, extending their program execution time.

2) vGPU Under-utilization: The vGPU workload performance is highly coupled with the ability to timely receive offloaded kernels and immediate execution of completed kernels. Due to the critical path task interference caused by vCPU discontinuity, there is a delay in vGPU tasks in turn decreasing the highly coupled vGPU workload performance. As shown in Figure 4, the vGPU performance is slowed down 1.3x to 20x. The vGPU performance of the NAMD benchmarks decreases by 51% to 82% and that of GROMACS decreases by 34% while sharing 16 vCPUs in the Shared1 scenario, and by 22% to 59% and by 22% respectively while sharing 16 vCPUs in the Shared2 scenario. More specifically, benchmarks experiencing a comparably larger degradation in runtime performance as well experience heavy vGPU under-utilization, due to the high coupling.

3) Unpredictability in Performance: Although runtime performance degradation is observed across all benchmarks, the degradation varies over different benchmarks (Figure 4) and under different co-run interferences (Figure 5). For instance,

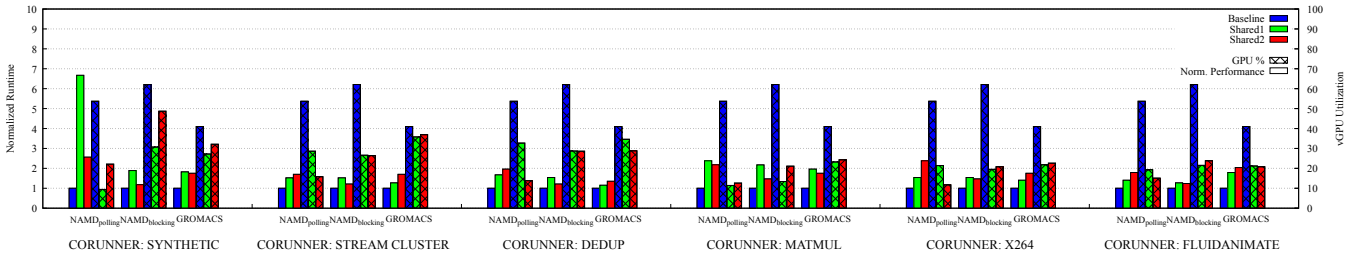

Fig. 5: The runtime slowdowns of GPU workloads under different CPU interference sharing 16 vCPUs.

the runtime performance of NAMD with Polling is degraded by 6.7x while sharing 16 vCPUs, compared to a degradation of 1.8x for GROMACS. Similarly, as seen in Figure 5, the runtime performance of NAMD with Polling is degraded by 6.7x with interference from the synthetic benchmark, but experiences a degradation of 1.4x with interference from p.fluidanimate. It is established from earlier arguments that workloads experience interference from vCPU discontinuity, although this interference is evidently inconsistent. The behavior of this interference greatly depends on the patterns and CPU intensity of the co-run programs, and their effect on vCPU discontinuity. Hence, the inconsistency across different co-run interferences induces an unpredictable nature in GPU workload performance.

4) Increased Degradation Under Varied #vCPUs: Intuitively, attaching more vCPUs to a VM should increase workload performance due to the increased task parallelism. Although, from our experiments we identify that the performance degradation is amplified as the number of vCPUs increases. For example, the workload runtime performance of the NAMD with Polling benchmark is degraded by a factor of 6.7x, 11.1x, and 20x when sharing 16, 32, and 64 vCPUs respectively. More interestingly, the absolute runtime of all workloads under interference consistently increased as the number of vCPUs increased. The key to this observation is as the number of vCPUs attached to a VM increases, the number of tasks executed by the vCPUs and the number of tasks offloaded to the vGPU increase accordingly. Consequently, the increase in kernel invocation frequency increases the synchronization intensity and rate at which CPU-GPU synchronization and communication occurs. This further amplifies the interference leading to runtime slowdown and vGPU utilization degradation. Although, increasing the number of vCPUs available to the VM increases the amount of tasks that can be offloaded to the vGPU, it also increases the vCPU's vulnerability to interference issues caused by vCPU discontinuity.

5) Increased Degradation Under Varied #VMs: The performance of GPU workloads suffer drastically in relation to the number of other VMs in the multi-tenant cloud sharing the same CPU resources. The essence of multi-user resource sharing in the cloud is referred to as an overcommitted scenario, where there is an over-commitment to sharing available physical resources across multiple VMs. Figure 6 shows the performance of GPU workloads in varying overcommitted scenarios. The no interference experiments are where the VMs

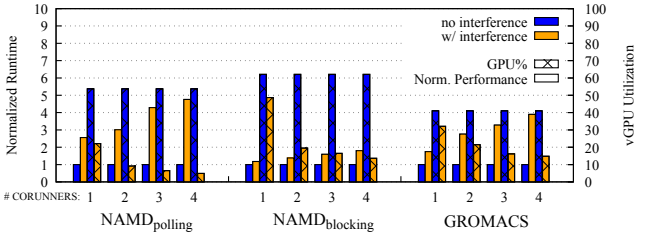

Fig. 6: The performance of GPU workloads in varying overcommitted scenarios with 1-4 co-run VMs causing interference using the synthetic benchmark.

receive dedicated CPU and GPU resources and hence are not affected by outside interference, while with interference experiments are where the VMs share CPU resources among 1-4 other VMs. In the presence of only one other VM sharing the same CPU resources, it can be seen that the runtime performance is slowed down by 1.2x to 2.6x across the benchmarks. Although, in the presence of four other VMs sharing the same CPU resources, it can be seen that the runtime performance is degraded further by up to 4.8x. It is important to note that resource over-commitment is very common in multi-tenant clouds due to cost efficiency, though it has a dire effect on workload performance.

IV. Diagnose: Case Study of NAMD

The efficiency of vCPU critical tasks directly determines the workload performance and vGPU utilization. In time-sharing multi-tenant clouds, vCPUs get descheduled, leaving them with the inability to process necessary tasks, in turn delaying the critical path execution. Thus, to fully understand the performance degradation and vGPU under-utilization, it is crucial to understand how vCPU discontinuity affects these vCPU tasks. We found that the major sources of latencies due to vCPU discontinuity are during kernel invocation (Section IV-A) and in receiving completed kernels (Section IV-B). These observations show that workloads with more frequent number of kernels are more vulnerable to performance degradation. GPU workloads such as r.particle_filter and r.srad only invoke an average of 1 kernel per second during execution, while *synchronization-intensive workloads*, such as NAMD, can invoke as much as 370 kernels per second, and are therefore more vulnerable to critical path latencies, inducing higher performance degradations, as was shown in Figure 1.

To study the effect of vCPU discontinuity on critical vCPU tasks, it is essential to analyze synchronization-intensive work-

loads due to their high vulnerability to performance degradations. In this section we analyze the performance results of the synchronization-intensive benchmark NAMD, which utilizes both polling and blocking CPU-GPU synchronization techniques, as a case study of the vCPU discontinuity interference in the cloud by providing and investigating NVIDIA Nsight Systems [23] visual profiling examples. In the diagnose, we conduct profiling using the Shared2 experiment type to provide fair resource sharing among tenants, and use the synthetic benchmark for maximum background interference.

A. Latency in Kernel Invocation

Offloading tasks to the vGPU is an important vCPU task that, if delayed, induces a delay on future critical path tasks. The effect of descheduling of the vCPU at the completion of its time slice in time-sharing multi-tenant clouds, vCPU discontinuity interference, is different among both CPU-GPU synchronization techniques. In the polling scenario, the vCPU is always busy querying the status of kernel execution, thus, utilizing the full CPU time slice, making the vCPU vulnerable to getting descheduled immediately before it is able to invoke a GPU kernel, as explained by Figure 2. This was observed through the visual profiling as shown in Figure 7. The vCPU starts the kernel launcher but is immediately descheduled, which causes the vCPU to wait until its next time slice to successfully launch the GPU kernel. The latency induced due to this behavior was observed to be 29.79*ms* on average. This induced latency due to vCPU discontinuity interference can as well be observed through the number of kernel invocations made per second. Without sharing vCPUs, hence unaffected by vCPU discontinuity, NAMD with Polling invokes an average of 171 kernels per second. Although, this drops to an average of 95 kernel invocations per second when affected by vCPU discontinuity. The top workflow in Figure 7 displays the evident delay between the time of the start of the kernel launcher and the time that the vCPU is rescheduled to successfully launch the kernel. This is in comparison to the bottom workflow which displays a normal execution cycle where the vCPU has the ability to immediately launch the GPU kernel.

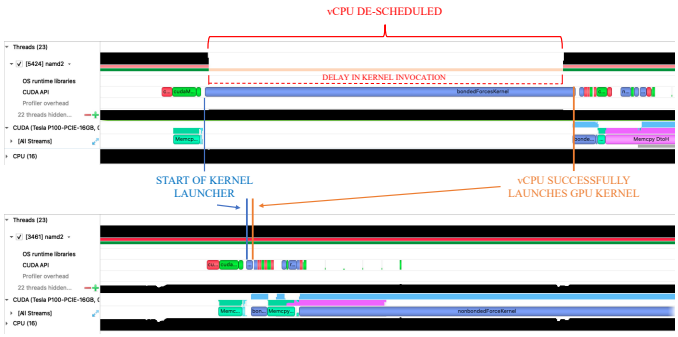

Fig. 7: The profiling timeline for NAMD with Polling with (top) and without (bottom) vCPU discontinuity interference during kernel invocation.

Conversely, in the blocking scenario, the vCPU is woken up to invoke the GPU kernel and returns to the blocking mode, only to be woken up at the notification of kernel completion. Thus, latency is only experienced in waking up the vCPU, as explained by Figure 3. Once the vCPU is ready to execute the next set of kernels, it is woken up to begin the cycle, hence it is not vulnerable to getting descheduled immediately before kernel invocation. Although the latency in waking up the vCPU varies in an overcommitted scenario, it has no direct effect on subsequent kernel invocations.

B. Latency in Receiving Completed Kernels

In the time-sharing multi-tenant cloud, if the vCPU is descheduled due to the completion of its time slice, then there will be an observed delay between the GPU kernel completion and the data synchronization call on the vCPU. In the polling scenario, the vCPU may utilize its full time slice and become vulnerable to getting descheduled during kernel execution or immediately before kernel completion, as explained by Figure 2. This was observed through the visual profiling as shown in Figure 8. Once the GPU kernel is complete, the vCPU is unable to receive and process the completed data because it is in-active. The latency induced was observed to be 28.65*ms* on average. The top workflow in Figure 8 displays the evident delay due to the inability to immediately execute the data from the completed kernel. This is in comparison to the bottom workflow which displays a normal execution cycle where the vCPU is active to immediately receive and process the data, avoiding delays to future critical path tasks.

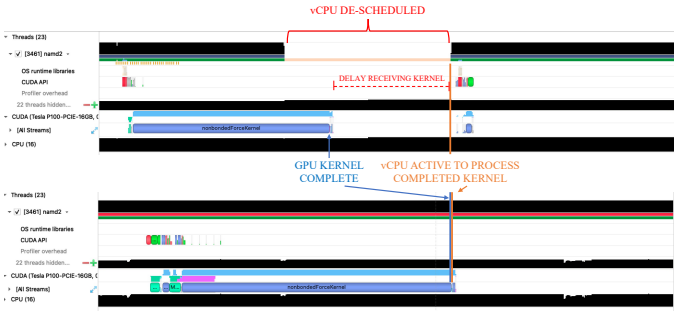

Fig. 8: The profiling timeline for NAMD with Polling with (top) and without (bottom) vCPU discontinuity interference while receiving a completed kernel.

In the blocking scenario, the delay was similar to that of the polling scenario. This as well is credited to the depletion of shared CPU resources between instances in the multi-tenant cloud, as explained by Figure 3. This was observed through the visual profiling as shown in Figure 9. The kernel is completed on the vGPU, but is not processed immediately due to descheduling of the vCPU. The latency induced was observed to be 13.58*ms* on average. The top workflow in Figure 9 displays that upon kernel completion, the vCPU is unable to immediately receive and process the data due to its descheduling. This in-completion of tasks causes a significant delay, inducing further delays on critical path tasks of future time steps. The delay on future time steps can be observed through the kernel invocations made per second, where NAMD with Blocking invokes 191 and 248 kernels per second, with

and without interference from vCPU discontinuity respectively. The bottom workflow in Figure 9 displays a normal execution cycle where the vCPU is able to synchronize and immediately receive and process completed kernels.

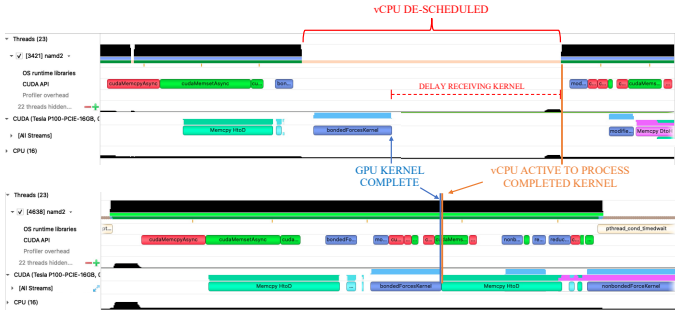

Fig. 9: The profiling timeline for NAMD with Blocking with (top) and without (bottom) vCPU discontinuity interference while receiving a completed kernel.

The latency associated with each CPU-GPU synchronization technique is summarized in Table II. Both techniques are vulnerable to delays in critical path tasks due to the continuous vCPU descheduling. In section V, we propose a novel CPU-GPU synchronization primitive to mitigate these delays.

V. Optimizing Performance in GPU Clouds

Our analysis found that the nature of synchronization-intensive applications contributes to a substantial waste of the vCPU time slice due to the continuous device communication. Through profiling of NAMD, we observed that the execution time of each kernel was consistent and predictable, deviating only 2%-5% from their average value regardless of background interference. Inspired by the characteristics of synchronization-intensive applications and predictability of kernel execution times, we propose a novel *polling-then-blocking* CPU-GPU synchronization primitive to improve workload performance in the multi-tenant GPU cloud. This hybrid synchronization model reduces vCPU time slice waste by polling for a defined period before blocking for completion, allowing saved time to be focused on execution of critical path tasks. This optimization serves as an adaptive synchronization model between the polling and blocking CPU-GPU synchronization techniques.

The *polling-then-blocking* primitive uses Algorithm 1. This hybrid optimization approach utilizes a dictionary of average kernel execution times, accessed by Kernel ID, to determine the most efficient synchronization technique to use after launching a kernel. The time during execution of each invoked kernel is recorded, and the respective average time is continually adjusted in the dictionary to ensure data accuracy and maintain optimal polling intervals. Following kernel invocation, a call to the hybrid synchronization primitive is done. The primitive initially ensures that the vCPU is blocked by calling *cudaEventSynchronize* if the kernel's execution time is longer than the maximum threshold. This threshold should be equal to the remaining vCPU time slice, ensuring that the vCPU immediately blocks if it cannot perform polling without interruption. The remaining vCPU time slice is calculated by

Algorithm 1 Kernel Latency Optimization

```
1:  D_k: kernel exec. dictionary; T_k: kernel exec. time value; T: timestep ending polling; Event: cudaEvent; max_threshold: max. allowed spin time
2:  function HYBRIDSYNC(Event, kernel.id)
3:      T_k ← D_k[kernel.id]
4:      if T_k > max_threshold then
5:          cudaEventSynchronize(Event)        ▷ block instantly
6:          D_k[kernel.id] ← updated T_k
7:      else  ▷ poll first, then block for completion if needed
8:          T = time() + T_k
9:          while T > time() do   ▷ poll for at most T_k time
10:             if cudaEventQuery(Event) then
11:                 D_k[kernel.id] ← updated T_k
12:                 return                      ▷ kernel complete
13:             end if
14:         end while
15:         cudaEventSynchronize(Event)
16:         D_k[kernel.id] ← updated T_k
17:     end if
18: end function
19: kernel<<<···>>>()
20: cudaEventRecord(Event, Stream=0)
21: hybridSync(Event, kernel.id)
```

methods in [22]. For regular kernels, the primitive initially allows the vCPU to use the polling technique to check for kernel completion for the full duration of the stored polling interval using *cudaEventQuery*, returning from synchronization if completed within this period. If the kernel is not completed within this period, then a call to *cudaEventSynchronize* is made to block the vCPU and wait upon the completion of the kernel. Importantly, before returning to the main function, the optimal polling period is always updated and stored into the dictionary. Thus, since the kernel synchronization is bound to less delays due to avoiding high vCPU time slice consumption, this will cause an overall less delay to the critical path and as well mitigate the latency in invoking kernels. This implementation is portable and scalable to other applications due to its online gathering of moving averages during execution, providing optimal polling intervals for each kernel at runtime while taking into consideration workload and environment changes.

Figure 10 shows the comparable optimized performance using the proposed hybrid CPU-GPU synchronization technique. The application was run under interference from the synthetic benchmark for maximum interference. The results show that the NAMD application achieved the highest increase in performance of 4.2x and 5.4x for execution time and vGPU utilization respectively while implemented with the *polling-then-blocking* CPU-GPU synchronization technique. Moreover, the latency in kernel invocation and in receiving completed kernels were drastically reduced under the hybrid synchronization model. Table II shows a comparison of the latency under

TABLE II: Latency Using Different CPU-GPU Synchronization Techniques Compared to Optimized Hybrid Model.

	CPU-GPU Synchronization Techniques		
	Polling	Blocking	Hybrid
Invoking Kernel	High Latency (29.79*ms*)	Negligible Latency	Low Latency (5.42*ms*)
Receiving Kernel	High Latency (28.65*ms*)	High Latency (13.58*ms*)	Low Latency (1.49*ms*)

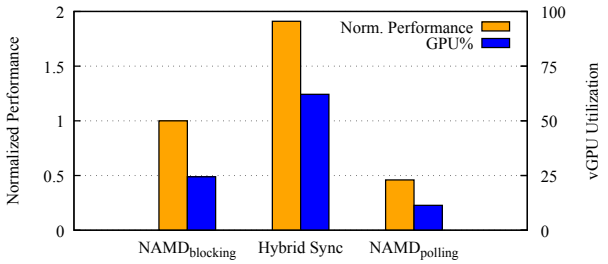

Fig. 10: Normalized performance and vGPU utilization of NAMD under polling, blocking, and the proposed hybrid CPU-GPU synchronization techniques using NAMD with Blocking as a baseline. A larger bar indicates higher performance.

different CPU-GPU synchronization techniques. In NAMD, a single execution cycle has an average length of 4.5*ms*, from variable initialization to execution of the completed kernel on the CPU. As seen in Table II, the latency incurred while utilizing the polling and blocking synchronization techniques is 13.58*ms* to 29.79*ms* which induces severe performance degradations considering that the latency can be as much as 6.6x the length of a single execution cycle. The hybrid model outperforms both polling and blocking as is accurate with our analysis of allowing the vCPU to extend its active time just in time to receive and process the completed kernel, otherwise blocking in order save the vCPU time slice and avoid delays in the critical path. Under this hybrid synchronization model, the performance of the application replicates that of both the polling and blocking synchronization techniques, while producing an increased performance in comparison.

It's important to mention that the *polling-then-blocking* primitive increases system throughput due to less waste of the vCPU time slice for excessive polling, hence improvement of other co-runner performance is possible.

VI. Related Work

QoS of GPU Cloud Sharing of vCPUs and vGPUs is vulnerable to poor performance in the GPU cloud due to the underlying default mechanisms of resource sharing and scheduling. This causes substandard utilization and performance fluctuations. Many studies have been done to improve the resource sharing and Quality of Service (QoS) of the GPU cloud, with solutions including VGRIS [8], vGASA [9], gScale [10], FairGV [11], and gQoS [12]. These solutions focus on improving the QoS of sharing vGPU devices in the GPU cloud. The work in this paper instead focuses on improving the QoS of the GPU cloud by improving the resource sharing of vCPU devices between GPU cloud instances.

Improving NAMD Performance NAMD is a highly popular molecular dynamics application that has been applied as a case study benchmark in many research studies. Due to its widespread use in research, many solutions have been implemented to improve the performance of the NAMD application [35]–[38]. However, all solutions have solely focused on improving the performance of NAMD on physical machines. The work in this paper focuses on improving the performance of NAMD in the virtualized cloud environment.

CPU-GPU Synchronization CPUs and GPUs have separate memory resources, hence there is an inherent latency in data transfer and communication between the devices. Several studies have been done to improve CPU-GPU synchronization, with solutions such as CGCM [39], BigKernel [30], and taking advantage of F/E bits in GPU DRAM [40]. Although, the primary focus of these studies is on implementing synchronization improvements at the hardware and compiler level. The work in this paper focuses on improving CPU-GPU synchronization on a system and application level. The proposed *polling-then-blocking* hybrid primitive is a synchronization method that is embedded within an application's source code.

VII. Conclusion

In this paper, we diagnose the effect of vCPU discontinuity on GPU workloads in multi-tenant GPU clouds by thoroughly experimenting, profiling, and analyzing the behavior of NAMD as a case study. We found that vCPU discontinuity leads to inefficient CPU-GPU synchronization, which in turn delays critical path tasks such as kernel offloading and receiving. Consequent delays in these tasks postpones subsequent critical path tasks, inducing latencies in the order of milliseconds. Furthermore, the incurred delays in CPU-GPU synchronizations and critical path tasks due to vCPU interference produces an increase in workload runtime and results in heavy vGPU under-utilization. Through extensive experiments, the performance interference caused by vCPU discontinuity attributed to the following issues: (1) vCPU discontinuity increases the vulnerability of vCPUs to be descheduled when the vCPU is needed to execute critical path tasks, thus, causing a degradation in runtime performance and heavy vGPU under-utilization; (2) the inconsistency in vCPU discontinuity interference correlates with the patterns of co-run interferences, thus, produces unpredictable GPU workload performance; (3) increasing the number of shared vCPUs in turn increases the performance degradation of the GPU workload due to the inherent increase in synchronization frequency inducing higher vulnerability to interference by vCPU discontinuity; (4) a higher number of VMs in the multi-tenant cloud induces increased performance degradation of the GPU workload due to the inefficiency of the increasing resource over-commitment.

Inspired by these findings, we propose a *polling-then-blocking* technique that provides a hybrid optimization to allow for vCPUs to remain active just in time to receive and process completed kernels, avoiding delays to the critical path. The thorough diagnose of the performance interference caused by CPU sharing in multi-tenant GPU clouds done in this paper should provide a guideline for users, to make informed decisions in relation to workload deployment, and cloud providers, to make related optimizations and start provisioning efficient multi-tenant GPU cloud instances sharing CPU resources.

REFERENCES

[1] J. P. Walters, A. J. Younge, D. I. Kang, K. T. Yao, M. Kang, S. P. Crago, and G. C. Fox, "Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications," in *2014 IEEE 7th international conference on cloud computing*. IEEE, 2014, pp. 636–643.

[2] K. Tian, Y. Dong, and D. Cowperthwaite, "A full gpu virtualization solution with mediated pass-through," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 121–132.

[3] (2020) Microsoft Azure: GPU optimized virtual machines. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/sizes-gpu

[4] (2021) Amazon AWS EC2 Instance Types. [Online]. Available: https://aws.amazon.com/ec2/instance-types/

[5] (2021) GPUs on Google Compute Engine. [Online]. Available: https://cloud.google.com/compute/docs/gpus/#gpu_comparison

[6] (2021) Oracle Cloud: GPU-Virtual Machines and Bare Metal. [Online]. Available: https://www.oracle.com/cloud/compute/gpu.html

[7] (2021) IBM Cloud: GPUs for cloud servers. [Online]. Available: https://www.ibm.com/cloud/gpu

[8] Z. Qi, J. Yao, C. Zhang, M. Yu, Z. Yang, and H. Guan, "Vgris: Virtualized gpu resource isolation and scheduling in cloud gaming," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 2, pp. 1–25, 2014.

[9] C. Zhang, J. Yao, Z. Qi, M. Yu, and H. Guan, "vgasa: Adaptive scheduling algorithm of virtualized gpu resource in cloud gaming," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 11, pp. 3036–3045, 2013.

[10] M. Xue, K. Tian, Y. Dong, J. Ma, J. Wang, Z. Qi, B. He, and H. Guan, "gscale: Scaling up gpu virtualization with dynamic sharing of graphics memory space," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 579–590.

[11] C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "Fairgv: fair and fast gpu virtualization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 12, pp. 3472–3485, 2017.

[12] Q. Lu, J. Yao, H. Guan, and P. Gao, "gqos: A qos-oriented gpu virtualization with adaptive capacity sharing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 4, pp. 843–855, 2019.

[13] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kale, and K. Schulten, "Scalable molecular dynamics with namd," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[14] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. Berendsen, "Gromacs: fast, flexible, and free," *Journal of computational chemistry*, vol. 26, no. 16, pp. 1701–1718, 2005.

[15] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[17] J. Ahn, C. H. Park, and J. Huh, "Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014, pp. 394–405.

[18] L. Cheng, J. Rao, and F. C. Lau, "vscale: Automatic and efficient processor scaling for smp virtual machines," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–14.

[19] S. Kashyap, C. Min, and T. Kim, "Scaling guest os critical sections with ecs," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 159–172.

[20] Y. Zhao, J. Rao, and Q. Yi, "Characterizing and optimizing the performance of multithreaded programs under interference," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, 2016, pp. 287–297.

[21] S. Schildermans, J. Shan, K. Aerts, J. Jackrel, and X. Ding, "Virtualization overhead of multithreading in x86 state of the art & remaining challenges," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[22] W. Jia, C. Wang, X. Chen, J. Shan, X. Shang, H. Cui, X. Ding, L. Cheng, F. C. Lau, Y. Wang *et al.*, "Effectively mitigating i/o inactivity in vcpu scheduling," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 267–280.

[23] (2021) NVIDIA Nsight Systems. [Online]. Available: https://developer.nvidia.com/nsight-systems

[24] (2021) NVIDIA CUDA Compute Unified Device Architecture Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[25] A. Herrera, "Nvidia grid: Graphics accelerated vdi with the visual performance of a workstation," *Nvidia Corp*, pp. 1–18, 2014.

[26] J. Song, Z. Lv, and K. Tian, "Kvmgt: A full gpu virtualization solution," in *KVM Forum*, 2014.

[27] (2016) AMD MxGPU: Hardware-based virtualization. [Online]. Available: https://www.amd.com/en/graphics/workstation-virtual-graphics

[28] C.-T. Yang, H.-Y. Wang, W.-S. Ou, Y.-T. Liu, and C.-H. Hsu, "On implementation of gpu virtualization using pci pass-through," in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*. IEEE, 2012, pp. 711–716.

[29] A. J. Younge, J. P. Walters, S. Crago, and G. C. Fox, "Evaluating gpu passthrough in xen for high performance cloud computing," in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. IEEE, 2014, pp. 852–859.

[30] R. Mokhtari and M. Stumm, "Bigkernel–high performance cpu-gpu communication pipelining for big data-style applications," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 819–828.

[31] (2021) CUDA Toolkit Documentation: cudaEventQuery and cudaEventSynchronize. [Online]. Available: https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__EVENT.html

[32] J. A. Stuart and J. D. Owens, "Efficient synchronization primitives for gpus," *arXiv preprint arXiv:1110.4623*, 2011.

[33] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan, "Gleaner: Mitigating the blocked-waiter wakeup problem for virtualized multicore applications," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 73–84.

[34] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, no. 8. Dttawa, Dntorio, Canada, 2007, pp. 225–230.

[35] S. Kumar, C. Huang, G. Almasi, and L. V. Kalé, "Achieving strong scaling with namd on blue gene/l," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[36] J. C. Phillips, Gengbin Zheng, S. Kumar, and L. V. Kale, "Namd: Biomolecular simulation on thousands of processors," in *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, 2002, pp. 36–36.

[37] R. K. Brunner, J. C. Phillips *et al.*, "Scalable molecular dynamics for large biomolecular systems," *Scientific Programming*, vol. 8, no. 3, pp. 195–207, 2000.

[38] B. Acun, D. J. Hardy, L. V. Kale, K. Li, J. C. Phillips, and J. E. Stone, "Scalable molecular dynamics with namd on the summit system," *IBM journal of research and development*, vol. 62, no. 6, pp. 4–1, 2018.

[39] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 142–151.

[40] D. Lustig and M. Martonosi, "Reducing gpu offload latency via fine-grained cpu-gpu synchronization," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2013, pp. 354–365.