Full Virtualization for GPUs Reconsidered

Hangchen Yu[1] Christopher J. Rossbach[2]

[1]The University of Texas at Austin

[2]The University of Texas at Austin and VMware Research Group

{hyu, rossbach}@cs.utexas.edu

ABSTRACT

Graphics Processing Units (GPUs) have become the tool choice in computationally demanding fields such as scientific computing and machine learning. However, supporting GPUs in virtualized settings like the cloud remains a challenge due to limited hardware support for virtualization. In practice, cloud providers elide GPU support entirely or resort to compromise techniques such as PCI pass through. GPU virtualization remains an active research area, fostering proposals for improvements at all layers of the technology stack, as well as software solutions based on API remoting, mediated pass-through, para- and full- virtualization, among others. The wealth of research leaves practitioners with much to think about but precious little in terms of usable techniques and actionable ideas.

This paper revisits GPUvm [59], a Xen-hypervisor-based full virtualization design for supporting VM access to discrete NVIDIA GPUs. Based on core primitives such as virtual memory-mapped I/O (MMIO), resource shadowing for GPU channels and page tables, GPUvm is arguably a high-water mark for transparency and feature completeness in GPGPU virtualization. We describe our experience setting up the most recent open source version, adapting new benchmarks, and re-creating experiments described in the original paper. While we are able to reproduce some reported results, we also reach a number of contrary findings. We observe that core functionalities such as para-virtual optimizations and the FIFO scheduler do not work and that optimizations such as BAR remapping are mostly ineffective. Full virtualization introduces catastrophic overhead in initialization and GPUvm's optimizations, which primarily target those overheads do so with limited success. The BAND scheduler algorithm purported to improve fairness over Xen's default CREDIT algorithm actually *increases* maximum unfairness by up to 6%, doing so at the cost of *decreasing* aggregate throughput by as much as 8%.

1. INTRODUCTION

GPUs have become ubiquitous in a number of settings thanks to steadily increasing compute density, proliferation of high-level programming frameworks [6, 58, 31], and compelling performance gains in a wide array of application domains such as machine learning [37, 8, 67, 21]. At the same time, as applications move inexorably toward the cloud, GPGPU workloads are largely left behind. Cloud providers such as Amazon Web Services and Microsoft Azure support VMs with GPUs, but GPGPU virtualization remains an open research challenge [50, 69, 65, 34, 9, 63]. GPU virtualization systems based on Xen [59, 61], KVM [35, 5], VMware [64, 24], as well as API forwarding systems [25, 64] are available, but cloud providers generally do not use them, and expose GPUs as exclusive resources through PCI passthrough [1].

A range of virtualization techniques are applicable to GPGPU compute. Device emulation [17] provides the full functionality of a GPU in software alone. While straight-forward, for GPUs, which achieve domain-specific performance gains with specialized hardware, the performance costs of emulation are untenable. API remoting (or API forwarding) [32, 24, 28, 56] forwards high-level API calls issued in VMs to the host or an appliance VM. The performance of API remoting comes at a hefty price in lost compatibility, coarse interposition, and limited isolation. Mediated passthrough [61] interposes security critical interactions with the device, avoiding costly interposition for performance-sensitive common case interactions; Xen- [61] and KVM-based [5] solutions have been proposed, but to date, support only Intel integrated GPUs. The performance, compatibility, and security tradeoffs are dramatic enough that I/O passthrough is still the most common technique used in production [66]. While it provides native performance, pass-through does not enable sharing and VMs monopolize individual physical GPUs. In *full virtualization*, the execution environment features a complete, potentially simulated virtual view of the underlying hardware. Because this enables guests to run unmodified OS and application binaries, it represents the ideal point in the design space for compatibility, interposition, and isolation. Efficient techniques for achieving full virtualization for GPUs remain elusive.

This paper revisits a recent full virtualization proposal called GPUvm [59], an open-source implementation for discrete NVIDIA GPUs in the Xen hypervisor. Virtualization is achieved by classical command forwarding, rate-limited command scheduling, and resource shadowing for page tables,

channels and device contexts. Because GPUvm interposes guest access to memory-mapped resources and shadows expensive resources such as GPU page tables, the net performance impact of full virtualization is significant. The GPUvm paper proposes a range of optimizations and para-virtual techniques that ostensibly trade transparency for improved performance, concluding that while overheads for full-virtualization can be massive, they are potentially amortizable for compute-intensive applications; that para-virtualization can improve the situation dramatically, but is still two or three times slower than pass-through or native; that fairness can be improved with a new BAND scheduling algorithm.

Our goals in undertaking this research were to assess the applicability of GPUvm and/or its core techniques in a production setting, and to gain deeper understanding of the trade-offs in hypervisor-level GPGPU virtualization. To this end, we recreate a GPUvm testbed, set up GPUvm, and port a number of additional benchmarks to the environment. We find that GPUvm is quite sensitive to configuration and hardware parameters. Not all features described in the paper are actually functional, usable, or stable, either due to platform differences we could not eliminate or due to software bugs. We recreate most experiments in the paper, validating a handful of results, but we arrive at a number of contrary findings as well. We observe significantly different-from-reported overheads for full virtualization (up to 73,700% or 23,200% on average). Para-virtual optimizations do not work, and full-virtualization optimizations are often ineffective. Scheduling algorithms purported to improve fairness can decrease fairness by as much as 6% while *reducing* aggregate performance by up to 8%.

2. BACKGROUND

2.1 Gdev

Gdev [39] is a GPU resource management ecosystem which can operate in either user- or kernel-space. It provides first-class GPU resource management schemes for multi-taksing systems and allows resource sharing between GPU contexts. To isolate the protection domains sharing GPU contexts, Gdev can virtualize the physical GPU into logical GPUs. Gdev provides a set of low-level APIs for GPGPU programming, and supports the CUDA 4.0 Driver API [6] on top of them. Our experiments, we use the Nouveau [2] device driver and Gdev as the CUDA runtime in DomainU guests.

2.2 Xen-based GPUvm

GPUvm multiplexes GPU hardware resources (such as memory, PCIe BARs, and channels, etc.) among Xen Hardware Virtual Machines (HVMs). Figure 1 shows the design overview presented by the original paper. GPUvm exposes a native GPU device model (NVIDIA Quadro 6000 NVC0 GPU) to every VM, so that guests are able to run unmodified GPU applications and device drivers. All operations upon the GPU device models are intercepted by making MMIO regions in memory inaccessible, which enables interposition based on the resulting page faults. Interposed operations are redirected to a *GPU access aggregator* using a simple client-server communication protocol.

GPUvm creates a GPU context for each VM, along with

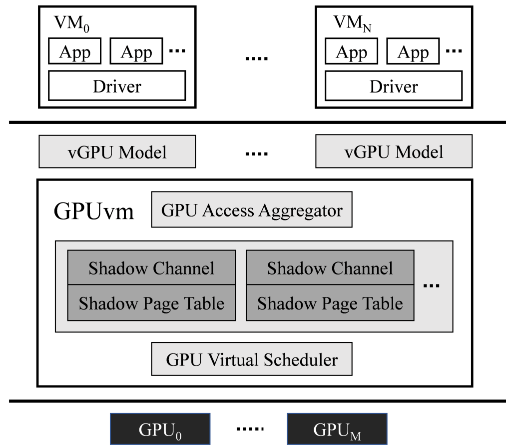

Figure 1: The design of GPUvm and system stack.

shadow contexts and *shadow GPU channels*. A context contains several virtual (guest) channels and every virtual channel maps to a shadow channel. Further, each shadow GPU channel is associated with a *shadow page table*. The GPU scheduler assigns appropriate GPU resources such as physical GPU channels to VMs. Shadowed resources enable the abstraction of privileged access to deprivileged guests, and provide an isolation mechanism whereby GPUvm can validate the correctness of changes to these structures made by guests. In a strict analogy to shadow page tables that were commonly used to virtualize memory before the emergence of hardware support for multi-level page tables, GPUvm interposes guest interactions, reflecting changes to guest contexts, channels, and page tables to their shadow counterparts. The overheads of interposition and reflecting state are the primary determinants of performance for GPUvm.

In principle, the design of GPUvm can be extended to GPUs other than NVIDIA Quadro 6000. Newer GPU device model should be exposed to VMs to support more advanced features. However, the design cannot preclude the non-trivial overheads of shadowing mechanisms. Although NVIDIA Quadro 6000 is the only one successfully evaluated in our experiments, we expect similar characterization results for other GPU hardware.

2.3 Memory Management

GPUvm supports discrete, PCIe-attached GPUs. GPU control registers and memory apertures are mapped to PCIe BARs (base address registers) making GPU configuration and memory accessible to the driver through memory operations on those BARs. The NVIDIA Quadro 6000 GPU supported by GPUvm exposes 4 memory regions (BAR 0,1,3,5) through PCIe. Memory region 0 (BAR0) is the control space of the card whose size is 32 MB. MMIO registers of different engines such as PFIFO and PGRAPH are mapped in this region using 32-bit addressed non-prefetchable addressing. Region 1 (BAR1) and Region 3 (BAR3) are prefetchable instance memory (RAMIN). BAR1 is used for submitting command requests; a 512 KB poll area resides at the head of BAR1, populated by 128 (by default) shadow channels. BAR3 is used for memory aperture. The shadow page tables

map BAR address to VRAM address. GPUvm intercepts the data access to BAR3 and then reflects the access to VRAM. GPUvm partitions BAR3 statically across a preconfigured maximum number of supported VMs, so that VMs can access the corresponding VRAM partitions. Region 5 is a 128-byte area of I/O ports used to access BAR0 or BAR1 indirectly. Para-virtualization fails to run on our machine possibly due to the hardware: it utilizes BAR4 which is not exposed through PCIe on our machine.

GPUvm utilizes 0 GB–4 GB VRAM area by default. 4 GB–6 GB region is reserved for hypervisor. GPUvm partitions poll area of BAR1, BAR3, and VRAM statically and equally, so that the physical memory space and MMIO space are partitioned into multiple sections of continuous address space. For example, each of N running VMs can use $4/N$ GB memory. The number of VMs is hard-coded and GPUvm needs to be reconfigured and recompiled each time we want to run different number of VMs simultaneously.

2.4 Optimizations

Not surprisingly, naïve full-virtualization based on shadowed resources can have non-trivial overheads, often multiple orders of magnitude slowdowns. GPUvm synchronizes guest GPU channel descriptors and shadow GPU descriptors by intercepting all data accesses through PCIe BARs, which invokes the hypervisor every time the BARs are accessed. GPUvm synchronizes the contents of guest page tables and shadow page tables on every TLB flush. Worse, GPUvm cannot intercept GPU-side page faults, so it must scan the entire guest page table to find the modified entries. These two sources of overhead are ameliorated by BAR3 remapping and lazy shadowing techniques. In the BAR3 remapping optimization, BAR3 accesses are passed-through; in the lazy shadowing optimization, GPUvm updates the shadow page tables only when they are referenced: only when memory apertures are read/written or when GPU kernels are started.

2.5 Virtual GPU Scheduler

The job of the virtual GPU scheduler is to share the GPU time across VMs fairly. The scheduler is the lowest logical layer of GPUvm, and functions by orchestrating GPU contexts submission of commands to physical GPU. Because GPUs are in general non-preemptive[1], and workloads bursty, [39, 52, 33], true fairness and performance isolation are unattainable, but best-effort heuristics which approximate them on average over time are possible. GPUvm supports three virtual GPU schedulers, including a simple FIFO implementation, the Xen default CREDIT [14], and the BAND (bandwidth-aware non-preemptive device) scheduler from Gdev [39]. In the CREDIT scheduler, each VM maintains a periodically-refreshed budget of credit and a threshold to prevent over-utilization of the GPU. The BAND scheduler extends CREDIT by lowering the virtual GPU's priority only when the budget is exhausted and utilization exceeds the bandwidth. Additionally, it inserts wait time after GPU kernel completion if the kernel over-utilizes its assigned bandwidth, so as to enable fairer GPU utilization for non-preemptive burst workloads. Because the scheduler runs in CPU code, the GPU can be idle while the scheduler executes, GPU utilization can drop below 100% [59]. To avoid this, both CREDIT and BAND schedulers account CPU time consumed by the scheduler as GPU time.

[1] While this trend is changing with the emerging NVIDIA Pascal architecture, it is true of all GPU hardware available at the time GPUvm was written, and remains true of most commonly available discrete GPUs.

Benchmark	Description
backprop	Back propagation (pattern recognition)
hotspot	Hotspot (physics simulation)
lud†	256x256 matrix LU decomposition
srad†	Speckle reducing anisotropic diffusion (imaging)
srad2	SRAD with random pseudo-inputs
loop	Long-loop compute without data
madd†	1024x1024 matrix addition ($10^4\times$)
mmul	1024x1024 matrix multiplication
fmadd	1024x1024 matrix floating addition
fmmul	1024x1024 matrix floating multiplication
memcpy†	64 MB of HtoD and DtoH
gaussian*	256x256 matrix Gaussian elimination
nn*	*k*-nearest neighbors classification
pathfinder*†	Search shortest paths through 2-D maps
needle*†	Needleman-Wunsch algorithm (bioinformatics)
idle*†	Idle for-loop
heartwall	Track movement of a mouse heart
bfs	Breadth-First Search
shm	Allocate and Access GPU shared memory
memcpy_pinned	`memcpy` using pinned host I/O memory
memcpy_async	`memcpy` asynchronously

Table 1: GPU benchmarks. Entries in green are new additions; gray entries are workloads that fail outright in our testbed. Benchmarks with † function correctly with BAR3 remapping; others do not.

3. METHODOLOGY

Experiments in this paper use a Dell Precision 3620 workstation with NVIDIA Quadro 6000 NVC0 GPU. Xen IOMMU support is disabled as GPUvm does not use Intel Virtualization Technology for Directed I/O (VT-d). Each guest is a Xen Hardware Virtual Machine (HVM) with 1 VCPU, 4 GB memory and 20 GB disk.

The documentation and comments in the code provide clues to find a combination of hardware and software that actually functioned with GPUvm [59]. However a significant effort was still required. We found that Fedora-16 as both guest and host OS minimized dependences and code modifications required to bring the system up. Linux Kernel 3.6.5 was patched with the Gdev [39] module side patches (for Nouveau). We encountered a number of bugs which cause BAR3 remapping optimization and para-virtualization to fail: related issues were posted on GitHub [3] but they remain unfixed. The errors may result from platform differences or software bugs: for example, the BAR addresses shown in PCI device list are different from those commented in the source code; the detected BAR3 address is also inconsistent with the real hardware address.

We install, run, and test the latest available open-source version of GPUvm [4] which is based on Xen 4.2.0. We use Gdev at commit `605e69e7`, which was the latest version

when GPUvm was under active development. While that version of Gdev supports Linux kernel 3.6, its kernel mode requires the much older NVCC 4.2 compiler to build CUDA binaries (cubins). GPUvm has a dependency on the C++ Boost library, requiring us to build Boost 1.4.7 manually from source (Boost $\geq$ 1.4.8 makes incompatible changes to the thread library). All components are compiled by GCC 4.6.3 and NVCC 4.2. We found that building with newer GCC forces non-trivial modifications such as fixing memset parameters and including realtime extensions library (librt), and as observed previously, the kernel-mode Gdev works only with NVCC 4.2.

We attempted to use other Linux distribution like Ubuntu 12.04 LTS and 14.04 LTS, however, we found that components including Xen 4.2, Boost 1.4.7, GPUvm A3 tool could not be built with the native system packages. In principle, GPUvm may be ported to Ubuntu 12.04 with some non-trivial fixes. Some native packages such GCC and binutils need to be downgraded, and GPUvm (Xen and A3 tool) can be compiled successfully, but Xen's PM-Timer fails consistency check on our machine during booting, so we abandoned the effort. Even Fedora-16 required significant effort to achieve a compatible configuration of the Network Security Services Utilities Library (NSS) to enable networking and fix Yum, but gave the fewest warnings and errors. NVIDIA Quadro 6000 NVC0 GPU is the only one that is usable during our test. NVIDIA Quadro 2000M NVC3 , GeForce GTX 760 NVE4, Quadro P6000 GP102 GPUs cannot be initialized by either Nouveau 3.6.5 or GPUvm itself. But reasonably, GPUvm can be extended to support these other GPU models.

3.1 Benchmarks

The GPUvm paper evaluated 16 representative benchmarks. To recreate this evaluation we ported and modified 9 Rodinia benchmarks [20] and 7 Gdev benchmarks. 11 of them are in the GPUvm paper, but because we do not have the source of original benchmarks, the implementations, configurations and test data may be slightly different.

We ported several additional benchmarks from Rodinia which were not evaluated (or reported) in the original paper, and provided data for the handful benchmarks which execute correctly in our testbed. All benchmarks are listed in Table 1. Newly ported ones are marked with an asterisk (*) and marked in green. Benchmarks marked with gray background fail to run or fail occasionally. Rodinia's `heartwall` and Gdev's `shm` fail on the Gdev version we use. `bfs`, `memcpy_pinned`, and `memcpy_async` sometimes fault in our environment. All of them hang Gdev's `__gmemcpy_to_device_p` or `__gmemcpy_to_device_np` function (copy host buffer to device memory w/o pipelining) when polling fences. We modify the kernel of `madd` to run 10,000 times so that the kernel execution time is long enough to dominate the runtime. We extend the scheduler evaluation, which in the original paper used `madd` and `long-madd` (5 times `madd`).

3.2 Platforms

We find that the para-virtualization optimizations simply do not work in our environment, and our attempts to fix them were unsuccessful. Consequently, this paper only evaluates full-virtualization and optimized full-virtualization methods. We evaluate the GPUvm platforms shown in Table 2.

Platform	Description
FV-Naïve	Full-virtualization (FV), no optimizations
FV-BAR-Remap	FV + BAR3 remapping
FV-Shadow	FV + lazy shadowing optimization.
FV-Optimized	FV + BAR3 remapping + lazy shadowing
Passthrough	Xen VGA passthrough.
Native	The host system (no Xen).

Table 2: GPUvm platforms used in this evaluation.

4. EVALUATION

We measure boot time, application performance, BAR3 accesses, shadow page table updates, and compare performance isolation (fairness). Guest boot and application performance are measured by wall-clock time and Linux timing utility. Other metrics are collected through the GPU access aggregator. As Figure 1 shows, all GPU operations are interposed by the aggregator, which provides a convenient way to probe MMIO or page table accesses. BAR3 access can be measured by counting read/write commands submitted to the context. Shadow page tables are updated when shadow channels are flushed. To measure the scheduler we use a separate sampler thread. When a command is submitted and executed by the scheduler thread, the used GPU time is reported to the sampler. The sampler wakes up every 100 ms in our tests and outputs the GPU utilization of each context (VM). We examine CREDIT and BAND schedulers on two, four, and eight VMs.

4.1 Boot

The wall-clock time for booting the different GPUvm platforms is shown in Table 3. With GPU virtualization enabled, the guest takes much longer to boot. During boot, BAR3 is written 308,932 times and read 78,736 times, and the shadow page table is updated 6 times. BAR3 remapping is more effective than lazy shadow updates because BAR3 access is quite frequent. Nonetheless, the boot time amplifications are all in the 7-8$\times$ range, which is a significant overhead to impose on VM startup.

Platform	Time (seconds)	Relative
FV-Naive	315	8.3x
FV-BAR-Remap	283	7.4x
FV-Shadow	312	8.2x
FV-Optimized	282	7.4x
FV-Passthrough	38	1.0x

Table 3: Guest boot time.

4.2 Performance

Figure 2 shows the execution time of the 16 benchmarks on the 6 platforms. The execution time, shown in logarithmic scale is shown relative to native performance. For the benchmarks evaluated in the GPUvm paper, we also plot the data from that paper for reference with textured bars. However, as the source codes of benchmarks are unavailable, the implementation, configuration, and test data may be different.

The difference can be seen in Figure 4 as well. But these reference data presents the expected performance pattern and helps determine which platforms actually work.

The data show that the non-trivial overheads of full virtualization reported by the original paper are even more non-trivial in our environment. For example, 9 of the benchmarks are slowed-down by over 100× on FV-Naïve. `idle` and `madd` have the least slowdown because the kernel execution is the dominant phase of runtime. Once a kernel is launched, its performance is very close to that on the native platform. Benchmarks with short kernel execution time (such as `backprop`) are slowed down many hundreds of times.

Contrary to the findings of the GPUvm paper, BAR3 remapping optimization does not work in some cases, and is often minimally effective at best. For example, FV-Optimized runs `lud` three times slower than FV-BAR-Remap. Figure 4 in Section 4.4 shows the breakdown of runtime which provides insight into the inefficacy of BAR3 remapping, and shows that most overhead is in initialization phases of workloads.

4.3 Performance Isolation

We find that Gdev hangs when using the FIFO scheduler for GPUvm. However, the CREDIT [14] and BAND (bandwidth-aware non-preemptive device) [39] schedulers work well in our testbed. We recreate test cases by running `madd` benchmark (1024x1024 matrix addition for 10,000 times) on two, four, and eight VMs simultaneously, and measure the GPU utilizations of each VM. Both BAR3 remapping and lazy shadowing optimizations are enabled.

GPU utilization per VM when using CREDIT and BAND schedulers are shown in Figure 5, sampled over 500 millisecond intervals. While we recreate similar experiments from the paper using 100 millisecond intervals, with similar-looking results, but we do not present the data. Utilization over 100 millisecond intervals fluctuates wildly and provides little additional insight over the 500 ms case. Both schedulers are fundamentally challenged to provide fairness over short time intervals because there is no cross-kernel concurrency, and GPU kernels cannot be preempted. The granularity can be increased with shorter scheduling periods, due to high variance in the length of GPU commands (e.g. between 1 ms and 3,000 ms in the `madd` benchmark) the shorter period only increases aggregate overhead.

The BAND scheduler is purported to provide better fairness than the CREDIT owing to two improvements [39]: better adjustment of priority and insertion of idle intervals. If a guest happens to be inactive when CREDIT scheduler replenish the credits, the guest has to wait to have GPU resources in the next scheduling. While the original paper concludes that lower amplitude fluctations indicate better fairness, we interpret the data (both theirs and ours) differently. Figure 5 primarily shows the fluctuating throughput that derives from non-preemptive time-slicing over too-short intervals. Indeed, the BAND scheduler is not necessarily always fairer than CREDIT. Table 4 and 5 shows typical breakdowns of runtime on each VM (for space, 8VM result is omitted). We compute both the Jain's fairness index [36] and the maximum unfairness for 2VM, 4VM, and 8VM cases, and under both metrics, CREDIT is actually *fairer* than BAND for all but the 8VM case, by margins as high as 6%.

Moreover, aggregate throughput of BAND is *lower* than CREDIT because of deliberately inserted idle phases, which worsens as the number of concurrently running VMs rises, as shown in Figure 3. BAND always sacrifices throughput relative to CREDIT. In 8VM case, BAND encounters as much as 8% throughput loss relative to CREDIT.

The design of GPUvm's schedulers enables CPU schedulers to be adopted as GPU schedulers. Other optimizations for CREDIT scheduler such as I/O-intensive and compute-intensive domain distinction, shared scheduling [19], and dynamic time slice [68, 22] can also be applied. Consequently, we conclude contrary to the GPUvm paper, that the CREDIT scheduler is clearly preferable.

4.4 Breakdown

A typical running application spends its time on device initialization, memory allocation, data generation, memory copy (from/to device), kernel execution, result verification, memory free, and device close. We pay close attention to the following six phases:

- `Init`: initialize Gdev, device, context, load modules.
- `MemAlloc`: device memory allocation.
- `HtoD`: data transmission from host to device.
- `DtoH`: data transmission from device to host.
- `Launch`: kernel execution time
- `Close`: unload modules, free device memory, destroy context.

Figure 4 shows the breakdown of the GPU applications' execution time. On a native machine, different types of applications may cost most time on `Init`, `Launch`, or memory copy phases `HtoD` and `DtoH`. However, on VM with full-virtualized GPU, most tested benchmarks spend major time on `Init` phase. The overheads of `Init` and `Close` are reduced by BAR3 remapping and lazy shadowing optimizations significantly. For example, Table 6 shows the slowdown of `Init` and `Close` of `needle` benchmark, for which the BAR remapping optimization works properly. The time is normalized by that on the native system.

Furthermore, the experiments show that the optimizations have tiny influence to the other phases. It also shows that some GPU optimization features like pipelining `memcpy` will not help much as the most significant overhead is caused by `Init` phase.

4.5 BAR3 Remapping

BAR3 remapping optimization is useful to reduce the execution time. The number of BAR3 accesses are measured as Table 7. With BAR3 remapping enabled, the number of BAR3 accesses received by GPU channel descriptors is reduced greatly. However, the benchmark does not speed up in most cases. As shown in Table 2, BAR3 remapping works only for `needle, srad, idle` on all platforms, and for `pathfinder, madd, lud, memcpy` for some FV optimized cases.

GPUvm paper measured the total size of data written to BAR3. The result is cited in Table 7. Typically each write is 4 bytes, so in our testbed, the system writes 100× data to BAR3. This is one of the reasons for the slowdown of GPUvm.

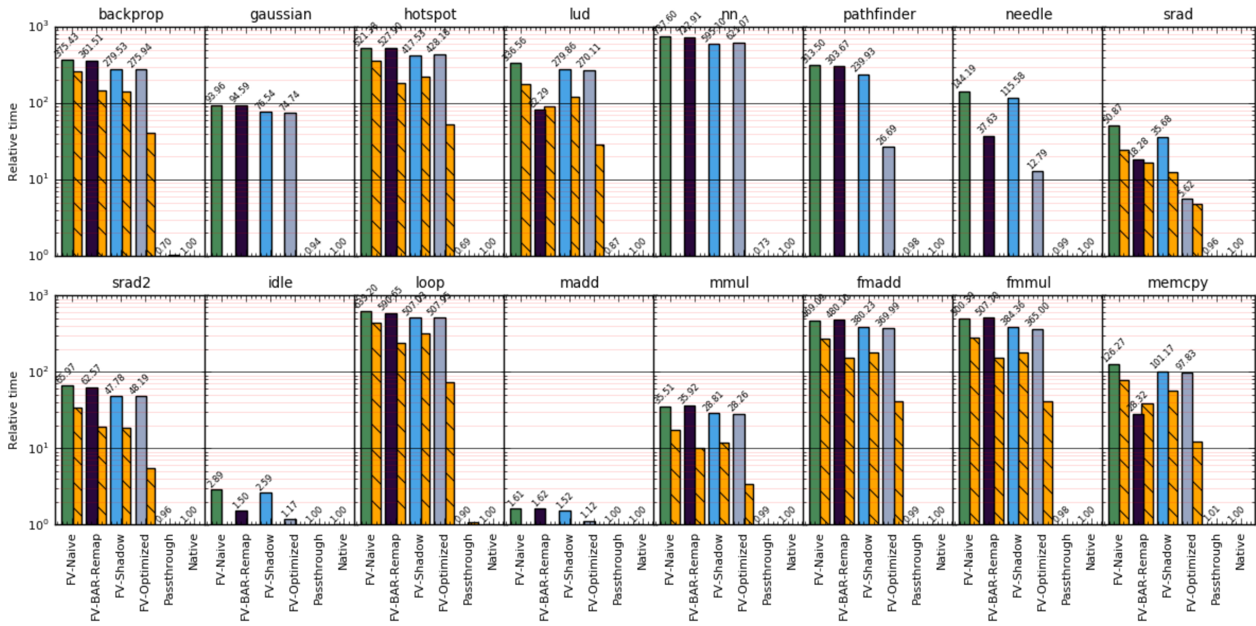

Figure 2: Relative execution time of the GPU benchmarks. Note use of logarithmic scale. Lower is better. Textured bars represent the data as reported in the original paper.

		Init	MemAlloc	HtoD	Launch	DtoH	Close	Total
2VM	VM0	1,466.12	2.71	582.94	67,781.37	17.03	142.56	69,992.74
	VM1	2,615.47	2.11	447.74	69,269.52	17.52	283.45	72,635.81
4VM	VM0	16,432.53	1.98	859.12	143,761.77	30.32	151.47	161,237.18
	VM1	14,012.28	28.43	2,355.17	141,273.89	71.07	142.27	157,883.11
	VM2	14,073.08	2.65	2,412.51	143,523.52	29.61	145.54	160,186.90
	VM3	16,447.93	2.13	887.87	134,802.56	141.45	572.07	152,854.01

Table 4: Performance isolation with CREDIT scheduler (milliseconds).

		Init	MemAlloc	HtoD	Launch	DtoH	Close	Total
2VM	VM0	3,424.71	1.99	478.37	67,544.55	19.78	339.45	71,808.84
	VM1	2,871.53	11.78	552.54	71,338.09	17.21	100.12	74,891.26
4VM	VM0	11,490.90	33.97	1,537.76	151,125.25	114.93	103.96	164,406.77
	VM1	1,456.02	782.84	2,124.68	147,844.38	71.56	97.49	152,376.97
	VM2	13,057.86	3.99	3,685.73	149,233.05	45.69	118.11	166,144.42
	VM3	16,784.81	2.31	621.93	148,983.27	13.96	275.86	166,682.13

Table 5: Performance isolation with BAND scheduler (milliseconds).

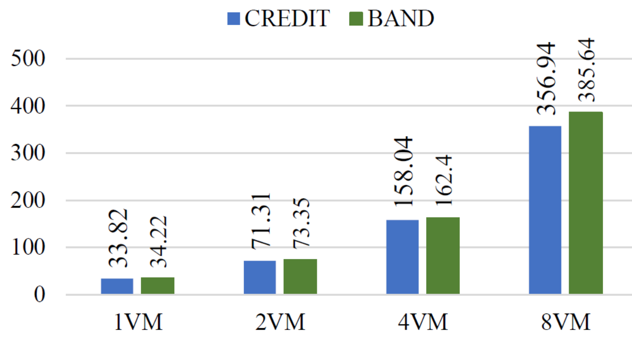

Figure 3: Average runtime of `madd` when executed simultaneously on different number of VMs (seconds).

	Naïve	BAR-remap	Shadow	Optimized
Init	850x	150x	750x	60x
Close	1,260x	1,075x	200x	165x

Table 6: Slowdown of `Init` and `Close` of `needle`.

As shown in Figure 2, `srad` has relatively small slowdown on FV-BAR-Remap and FV-Optimized, because the memory copy phase `DtoH` is the dominant and it significantly benefits from the BAR3 access optimization. `srad2` with the similar working pattern should have the analogous feature, but BAR3 remapping does not work correctly for that benchmark.

4.6 Lazy Shadowing

Since GPUvm cannot interpose page faults caused by GPU [30], it must scan the entire page table to detect mod-

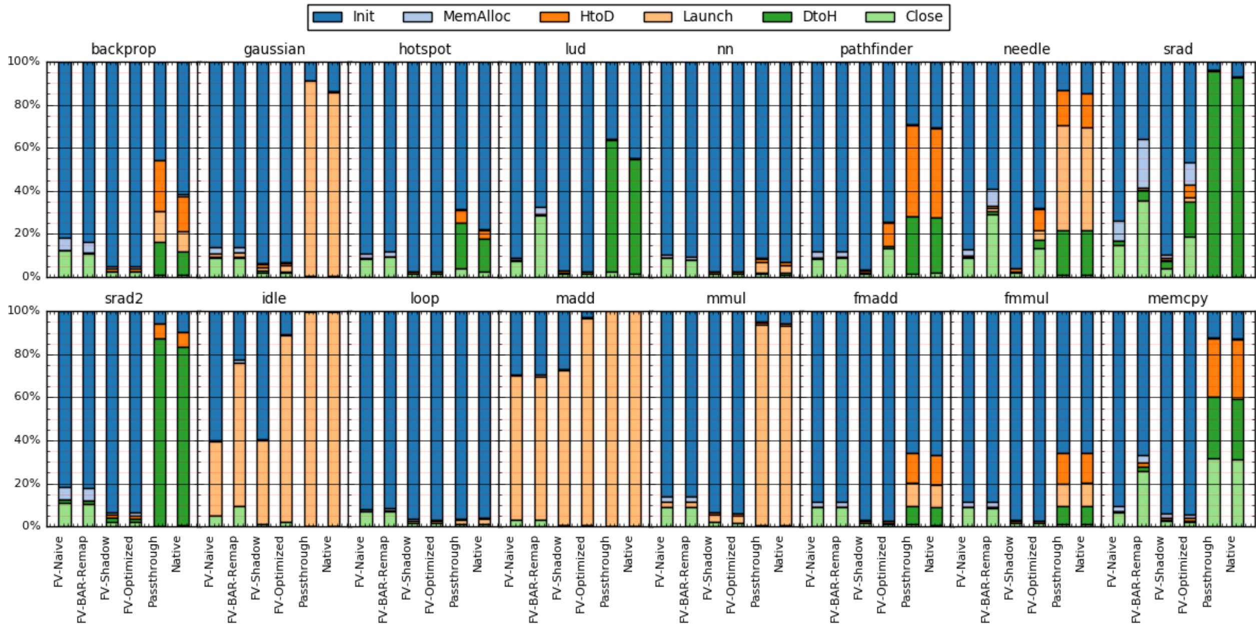

Figure 4: Breakdown of execution time for all benchmarks.

	BAR3 access				SPT update	
	R	Write Naïve	Write Remap	Write bytes†	FV-N	FV-O
backprop	0	165,600	10	7,280	40	7
gaussian	0	165,558	10	/	34	7
hotspot	0	164,916	10	6,736	34	7
lud	0	165,636	10	7,240	30	7
nn	0	165,006	10	/	32	7
pathfinder	0	166,072	10	/	34	7
needle	0	166,594	10	/	34	7
srad	0	166,696	10	6,352	52	7
srad2	0	168,122	10	6,248	40	7
idle	0	164,838	0	/	30	7
loop	0	164,836	0	6,696	30	7
madd	0	165,210	0	6,648	34	7
mmul	0	165,208	0	6,672	34	7
fmadd	0	165,216	0	6,680	34	7
fmmul	0	165,218	0	6,688	34	7
memcpy	0	166,416	0	6,168	26	6

Table 7: Count for BAR3 accesses and GPU shadow page table updates. FV-N and FV-O represent full virtualization naïve and optimized respectively. Write bytes† uses data from the original paper.

ifications in the guest page table. The TLB is flushed by device driver when a page table entry is updated, and GPUvm intercepts the TLB flush to detect page table updates, subsequently scanning and reflecting the difference to the shadow page table. Because TLB flushes can be frequent, the page table scan can introduce significant overhead. GPUvm uses a "Update-on-Reference" mechanism to reduce the overhead. only update shadow page tables

The shadow page table update count shown in Table 7 is identical to [59]. However, the performance impact is significantly different: the lazy shadowing optimization speeds up the applications by around 20%.

	name	UL	UOS	I	F	M
FV	**GPUvm [60]**	✓	✓	✓	✓	✓
	HSA-KVM [34]	✓	✓	✓	✓	
	gVirt [62]	✓	✓	✓	✓	
PT	**Amazon GPU [10]**	✓	✓			
	GPUvm [60]	✓	✓			
PV	**LoGV [29]**	✓		✓		✓
	SVGA2 [24]	✓		✓		
	GPUvm [60]	✓		✓		
API-R	**GViM [32]**				✓	
	gVirtuS [28]		✓		✓	
	vCUDA [56]					
	vmCUDA [64]					
	rCUDA [25, 51]					
	GridCuda [43]		✓			
	SnuCL [40]					
	VCL [13]					

Table 8: Comparison of existing GPU virtualization proposals, grouped by approach. The UL and UOS columns indicate ability to support unmodified guest libraries and OS. I and F indicate cross-domain isolation and some attempt to support fairness or performance isolation. M shows support for VM migration.

5. RELATED WORK

GPU Virtualization. Support for GPU virtualization at the OS [38, 39, 52, 57, 41, 53, 46, 49] and hypervisor [65, 34] are active research areas. A number of deployable VDI and graphics virtualization solutions [24, 42] support graphics frameworks [18, 55, 45]; VDI-targeted graphics cards [48, 11] simplify the design by mapping static partitions of the hard-

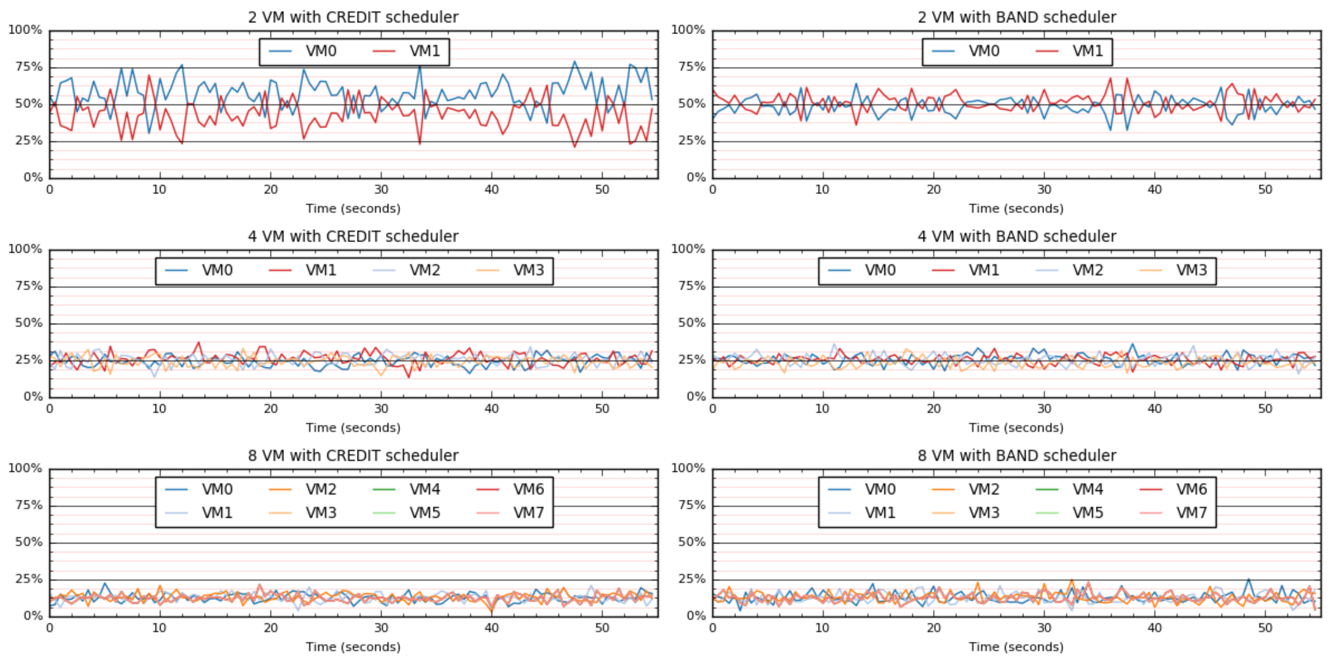

Figure 5: VMs' GPU utilizaton with the CREDIT and BAND schedulers.

ware to SR-IOV virtual functions. GPGPU support solutions have been proposed for Xen [61, 32, 59, 56, 15], KVM [35], and VMware[24]. API remoting forwards guest API calls, to a host, appliance VM, control domain OS [32, 56, 28, 64] or remote server [25, 51, 43, 13, 28, 47]. NVIDIA Docker [7] exposes GPUs in a container, enabling docker images that can be deployed on any GPU-enabled infrastructure, but GPUs are assigned to containers and not shared. Table 8 compares a number of virtualization systems from the research literature, grouping them according to fundamental technique.

GPU Scheduling. GPUvm extends CPU scheduling and virtual GPU scheduling techniques from Xen [15]. Diverse GPU resource scheduling methods have been proposed at different levels in the software stack [38, 16, 26, 52, 54] with different tradeoffs between performance and isolation.

IOMMU Emulation. GPUvm depends heavily on MMIO accesses, so an efficient IOMMU implementation would improve its performance greatly. rIOMMU [44] replaces the virtual memory page table hierarchy with a circular flat table to improve the throughput. vIOMMU [12] proposes an efficient IOMMU emulation optimized by optimistic teardown and sidecore emulation.

Performance Comparison. Previous work [27] presents a comprehensive performance comparison of virtual machines and Linux containers. The closest work in GPU virtualization compares GPU passthrough performance on virtual machines and containers [66]. These works reveal features of these virtualization techniques, and post a performance guideline for future works.

6. CONCLUSION

In this paper, we re-evaluate GPUvm with 16 benchmarks, comparing various metrics against those from the original paper. We find that para-virtualization and FIFO scheduler do not work; BAR remapping is ineffective for several workloads. We verify that GPUvm provides a full-virtualization architecture at the hypervisor, and that two optimizations can be useful to reduce initialization overheads. We additionally show intolerable slowdown for full-virtualization, amplified relative to the original results. Promising future avenues include vIOMMU [12] and SR-IOV [23]. The coarse granularity of virtual GPU schedulers results from the non-preemption of GPU commands. The static partition of PCIe BARs limits the number of VMs, but it can be solved by dynamic allocation.

7. REFERENCES

[1] Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/. Accessed: 2017-04.

[2] Freedesktop nouveau open-source driver. http://nouveau.freedesktop.org. Accessed: 2017-04.

[3] Gdev issues. https://github.com/shinpei0208/gdev/issues. Accessed: 2017-04.

[4] GPUvm source code. https://github.com/CPFL/gxen/. Accessed: 2017-04.

[5] KVMGT - the implementation of intel gvt-g(full gpu virtualization) for KVM. https://lwn.net/Articles/624516/. 2014.

[6] NVIDIA CUDA 4.0. http://developer.nvidia.com/cuda-toolkit-40. 2011.

[7] NVIDIA docker. https://github.com/NVIDIA/nvidia-docker. Accessed: 2017-04.

[8] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on

heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[9] Neha Agarwal, David Nellans, Mike O'Connor, Stephen W Keckler, and Thomas F Wenisch. Unlocking bandwidth for GPUs in CC-NUMA systems. In *HPCA*, 2015.

[10] Amazon. *Amazon Elastic Compute Cloud*, 2015.

[11] AMD. *AMD FirePro*, 2015.

[12] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. viommu: efficient iommu emulation. In *USENIX Annual Technical Conference (ATC)*, pages 73–86, 2011.

[13] A. Barak, T. Ben-Nun, E. Levy, and A. Shiloh. A package for opencl based heterogeneous computing on clusters with many gpu devices. In *Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–7, Sept 2010.

[14] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, October 2003.

[16] Mikhail Bautin, Ashok Dwarakinath, and Tzi-cker Chiueh. Graphic engine resource management. In *Electronic Imaging 2008*, pages 68180O–68180O. International Society for Optics and Photonics, 2008.

[17] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[18] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25(3):724–734, 2006.

[19] Zhibo Chang, Jian Li, Ruhui Ma, Zhiqiang Huang, and Haibing Guan. Adjustable credit scheduling for high performance network virtualization. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 337–345. IEEE, 2012.

[20] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[21] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[22] XiaoBo Ding, Zhong Ma, and XingFa Da. Dynamic time slice of credit scheduler. In *Information and Automation (ICIA), 2014 IEEE International Conference on*, pages 654–659. IEEE, 2014.

[23] Yaozu Dong, Zhao Yu, and Greg Rose. Sr-iov networking in xen: Architecture, design and implementation. In *Workshop on I/O Virtualization*, 2008.

[24] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review*, 43(3):73–82, 2009.

[25] Jose Duato, Antonio J. Pena, Federico Silla, Juan C. Fernandez, Rafael Mayo, and Enrique S. Quintana-Orti. Enabling cuda acceleration within virtual machines using rcuda. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

[26] Glenn A Elliott, Bryan C Ward, and James H Anderson. Gpusync: A framework for real-time GPU management. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, pages 33–44. IEEE, 2013.

[27] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pages 171–172. IEEE, 2015.

[28] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. A gpgpu transparent virtualization component for high performance computing clouds. In *European Conference on Parallel Processing*, pages 379–391. Springer, 2010.

[29] M. Gottschlag, M. Hillenbrand, J. Kehne, J. Stoess, and F. Bellosa. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCCEUC), 2013 IEEE 10th International Conference on*, pages 1721–1726, Nov 2013.

[30] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. Logv: Low-overhead gpgpu virtualization. In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*, pages 1721–1726. IEEE, 2013.

[31] Kate Gregory and Ade Miller. C++ amp: accelerated massive parallelism with microsoft visual c++. 2014.

[32] Vishakha Gupta, Ada Gavrilovska, Karsten Schwan, Harshvardhan Kharche, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Gvim: Gpu-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing*, pages 17–24. ACM, 2009.

[33] Vishakha Gupta, Karsten Schwan, Niraj Tolia, Vanish Talwar, and Parthasarathy Ranganathan. Pegasus: Coordinated scheduling for virtualized accelerator-based systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 3–3, Berkeley, CA, USA, 2011. USENIX Association.

[34] Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. Building a kvm-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 3–15, New York, NY, USA, 2016. ACM.

[35] Yu-Ju Huang, Hsuan-Heng Wu, Yeh-Ching Chung, and Wei-Chung Hsu. Building a kvm-based hypervisor for a heterogeneous system architecture compliant system. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 3–15. ACM, 2016.

[36] Raj Jain, Arjan Durresi, and Gojko Babic. Throughput fairness index: An explanation. Technical report, Tech. rep., Department of CIS, The Ohio State University, 1999.

[37] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[38] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011.

[39] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, pages 401–412, 2012.

[40] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. Snucl: an opencl framework for heterogeneous cpu/gpu clusters. In *Proceedings of the 26th ACM international conference on Supercomputing*, page 341âĂŞ352. ACM, 2012.

[41] Sangman Kim, Seonggu Huh, Yige Hu, Xinya Zhang, Emmett Witchel, Amir Wated, and Mark Silberstein. Gpunet: Networking abstractions for gpu programs. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 201–216, Berkeley, CA, USA, 2014. USENIX Association.

[42] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. Vmm-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 33–43, New York, NY, USA, 2007. ACM.

[43] Tyng-Yeu Liang and Yu-Wei Chang. Gridcuda: A grid-enabled cuda programming toolkit. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*, pages 141–146, March 2011.

[44] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. riommu: Efficient iommu for i/o devices that employ ring buffers. In *ACM SIGPLAN Notices*, volume 50, pages 355–368. ACM, 2015.

[45] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J.

Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM.

[46] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM.

[47] Raffaele Montella, Giuseppe Coviello, Giulio Giunta, Giuliano Laccetti, Florin Isaila, and Javier Blas. A general-purpose virtualization service for hpc on cloud computing: an application to gpus. *Parallel Processing and Applied Mathematics*, pages 740–749, 2012.

[48] NVIDIA. *NVIDIA GRID*, 2015.

[49] Sankaralingam Panneerselvam and Michael Swift. Rinnegan: Efficient resource use in heterogeneous architectures. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 373–386, New York, NY, USA, 2016. ACM.

[50] Jonathan Power, Mark D Hill, and David A Wood. Supporting x86-64 address translation for 100s of GPU lanes. In *HPCA*, 2014.

[51] C. Reano, A. J. Pena, F. Silla, J. Duato, R. Mayo, and E. S. Quintana-Orti. Cu2rcu: Towards the complete rcuda remote gpu virtualization and sharing solution. *20th Annual International Conference on High Performance Computing*, 0:1–10, 2012.

[52] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. PTask: operating system abstractions to manage GPUs as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM, 2011.

[53] Christopher J. Rossbach, Jon Currey, and Emmett Witchel. Operating systems must support GPU abstractions. In *13th Workshop on Hot Topics in Operating Systems, HotOS XIII, Napa, California, USA, May 9-11, 2011*, 2011.

[54] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. SOSP'13: The 24th ACM Symposium on Operating Systems Principles, November 2013.

[55] Mark Segal and Kurt Akeley. The opengl graphics system: A specification. Technical report, Silicon Graphics Inc., December 2006.

[56] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers*, 61(6):804–816, 2012.

[57] M. Silberstein, B. Ford, I. Keidar, and E. Witchel. Gpufs: Integrating a file system with gpus. 2013.

[58] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[59] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *USENIX Annual Technical Conference*, pages 109–120, 2014.

[60] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. Gpuvm: Why not virtualizing gpus at the hypervisor? In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 109–120, Berkeley, CA, USA, 2014. USENIX Association.

[61] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *USENIX Annual Technical Conference*, pages 121–132, 2014.

[62] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 121–132, Berkeley, CA, USA, 2014. USENIX Association.

[63] Jan Vesely, Arkaprava Basu, Mark Oskin, Gabriel H. Loh, and Abhishek Bhattacharjee. Observations and Opportunities in Architecting Shared Virtual Memory for Heterogeneous Systems. In *ISPASS*, 2016.

[64] Lan Vu, Hari Sivaraman, and Rishi Bidarkar. Gpu virtualization for high performance general purpose computing on the esx hypervisor. In *Proceedings of the High Performance Computing Symposium*, page 2. Society for Computer Simulation International, 2014.

[65] Carl Waldspurger, Emery Berger, Abhishek Bhattacharjee, Kevin Pedretti, Simon Peter, and Chris Rossbach. Sweet spots and limits for virtualization. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, pages 177–177, New York, NY, USA, 2016. ACM.

[66] John Paul Walters, Andrew J Younge, Dong In Kang, Ke Thia Yao, Mikyung Kang, Stephen P Crago, and Geoffrey C Fox. Gpu passthrough performance: A comparison of kvm, xen, vmware esxi, and lxc for cuda and opencl applications. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 636–643. IEEE, 2014.

[67] Dong Yu, Adam Eversole, Mike Seltzer, Kaisheng Yao, Zhiheng Huang, Brian Guenter, Oleksii Kuchaiev, Yu Zhang, Frank Seide, Huaming Wang, et al. An introduction to computational networks and the computational network toolkit. *Microsoft Technical Report MSR-TR-2014–112*, 2014.

[68] Lingfang Zeng, Yang Wang, Wei Shi, and Dan Feng. An improved xen credit scheduler for i/o latency-sensitive applications on multicores. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 267–274. IEEE, 2013.

[69] Tianhao Zheng, David Nellans, Arslan Zulfiqar, Mark Stephenson, and Stephen W. Keckler. Towards High Performance Paged Memory for GPUs. In *HPCA*, 2016.