Supporting Dynamic GPU Computing Result Reuse in the Cloud

Husheng Zhou, Yangchun Fu, and Cong Liu
Department of Computer Science, The University of Texas at Dallas

Abstract

Graphics processing units (GPUs) have been adopted by major cloud vendors, as GPUs provide orders-of-magnitude speedup for computation-intensive data-parallel applications. In the cloud, efficiently sharing GPU resources among multiple virtual machines (VMs) is not so straightforward. Recent research has been conducted to develop GPU virtualization technologies, making it feasible for VMs to share GPU resources in a reliable manner. This paper seeks to improve the efficiency of sharing GPU resources in the cloud for accelerating general-purpose workloads. Our key observation is that redundant GPU computation requests are being seen in many GPU-accelerated workloads in the cloud, such as cloud gaming where multiple clients playing the same game call GPUs to perform physics simulation. We have measured this redundancy using a gaming case study, and found that more than 24% (47%) of the GPU computation requests called by the same VM (multiple VMs) are identical. To exploit this redundancy, we present GRU (GPU Result re-Use), a GPU sharing, result memoization and reuse ecosystem in a cloud environment. GRU transparently enables VMs in the cloud to share a single GPU efficiently, and memoizes GPU computation results for reuse. It leverages the GPU full-virtualization technology, which enables GPU result memoization and reuse without modification of existing device drivers and operating systems. We have implemented GRU on top of the Xen hypervisor. Preliminary experiments show that GRU is able to achieve a significant speedup of up to 18 times compared to the state-of-the-art GPU virtualization framework, while adding a rather small amount of runtime overheads.

1 Introduction

Graphics processing units (GPUs) are powerful for accelerating computation-intensive data-parallel applications, due to their highly multithreaded architecture and high-bandwidth memory. Along with the support of various programming models and runtime environments such as the *compute unified device architecture* (CUDA) [21] from NVIDIA and OpenCL [15] from Apple, GPUs can be easily used for general-purpose computing (*a.k.a.*, GPGPU) in addition to dedicated graphics applications. GPUs have already been widely used to accelerate general-purpose workloads in many application domains such as supercomputing systems. Unfortunately, enterprise and cloud computing domains have inefficient access to GPU virtualization technology because the required resource isolation of multiple virtual machines (VMs) accessing the same set of GPU resources cannot be efficiently guaranteed.

Efficiently utilizing GPU resources in the cloud environment is not so straightforward. Current commercial cloud providers (e.g., Amazon Elastic Compute Cloud—EC2) dedicate an individual physical instance of GPUs to a VM, which causes resource under-utilization. A feasible solution to make GPU a truly shared and schedulable resource in the cloud is through GPU virtualization, which allows multiple VMs to access the same set of GPUs simultaneously. Recent research [8, 9, 14, 20, 26, 27] implements several GPU virtualization technologies for graphics rendering applications as well as GPGPU computing workloads.

With GPU virtualization technologies and the corresponding operating systems (OS) support, it is feasible for multiple VMs to share underlying GPUs in a relatively reliable manner. In this paper, we seek to further improve the efficiency of sharing GPU resources in the cloud. Our key observation is that redundancy has been seen in many cloud applications, such as scientific computing and cloud gaming where multiple clients playing the same game call GPUs to perform physics simulation for the same scenes. For example, the popular game *Call of Duty: Ghosts* [28] uses GPGPU to enhance smoke particles. When two VM instances are running the same game, the same game scenes and cutscenes will generate redundant GPGPU computations. Moreover, most GPGPU computation-related commercial software and applications prefer to use libraries provided or maintained by NVIDIA, since such libraries are more stable and highly-optimized. This fact increases the probability of seeing redundant GPGPU computing requests. For example, scientific applications use cuBLAS library for matrix operation, cuFFTW library for fast fourier transform (FFT). The standardization of GPGPU interface unifies the kernel code of GPU computation (a piece of GPU-accelerated code which is often relatively simple), making input data be the only difference among GPGPU computations.

Motivated by this, we explore the possibility of correctly and efficiently reusing GPU computing results among multiple VMs as well as for each individual VM. CPU instruction reuse and memoization have been proposed to make program execution on CPUs faster and more predictable [6, 7, 25]. Compared to CPU instruction reuse, reusing results processed on GPUs may be more viable, due to the nature of GPU processing. Employing GPU for general-purpose computing is usually a triple-step procedure: copying input data to GPU, launching a kernel, and acquiring output. This triple-step procedure makes GPGPU a "blackbox"—given input data and computation code, GPU will output the result—which provides a more straightforward processing environment compared to computing using CPUs. On GPUs, computation requests with the same input data and kernel will generate the same results. Similar to the dynamic CPU instruction reuse technology, it is possible to dynamically eliminate redundant GPGPU computations by buffering reusable results, so that future requests with the same input (data and kernel) can directly reuse the results without incurring redundant computations. We call this GPU computing result reuse, which has great potential of improving the throughput, response, and energy efficiency of using cloud resources.

This paper presents GRU — a GPU sharing, result memoization and reuse ecosystem in the cloud. GRU extends GPUvm [26] which is an open-source GPU virtualization tool built on top of Xen [4], exposing physical GPU model to guest VMs and bridging the semantic gap between domU (i.e., domain U) and dom0 (i.e., domain 0). Leveraging the GPU full-virtualization technology implemented in GPUvm [26], GRU transparently allows GPU result reuse and memoization without modification of existing device drivers and operating systems. GRU is composed of three main components: (*i*) *Qemu-Xen* which collects the MMIO (memory mapped I/O) operations and forwards GPU commands, (*ii*) the *aggregator* which interprets the parameters of GPU requests and manages the GPU memory, and (*iii*) the reuse engine which uses LUT (look up table) to identify cached results and stores computation results in a "shadow result memory". Once GPU computational requests with the same input data and kernel code are received and identified, GRU directs the results back from shadow result memory without incurring redundant computations. Preliminary experiments show that our prototype can improve the throughput of GPUvm by up to 18 times, while adding a rather small amount of runtime overheads.

2 Case Study

We conducted a measurements-based case study to motivate the necessity of GPU computing result reuse. We choose a popular game named "Mirror's Edge" for the case study, as it not only uses GPU to display but utilizes GPU for general-purpose computing as well. *Mirror's Edge* is an action-adventure 3-D video game published by Electronic Arts. It is powered by the Unreal Engine 3 [10] which features support for NVIDIA's PhysX [22]. PhysX is a physics simulation engine build on top of CUDA, which utilizes GPU to simulate debris, glass shard, spark effects, dynamic fog, *etc*.

Table 1: The statistics of executed kernels on GPU, from two execution traces of Mirror's Edge

	#Launch	Uniq	Exe	Redundant	Identical
R1	173766	24	22.31s	24.71%	47.12%
R2	123175	19	17.10s	25.63%	66.47%

We used NVIDIA's command line profiling tool *nvprof* [23] to monitor the game execution and collect the GPU computation trace. We also wrote an inspection tool employing dll injection and API hook to intercept the parameters of each GPU-related operation. We executed the game twice, each for ten minutes from the prologue chapter. For each execution, we performed random actions for the game play. The statistics of tracing these two executions are shown in Table 1. During the first execution, we collected 173,766 CUDA kernel launches that originate from 24 unique kernels. The total execution time of GPGPU computing due to these kernel launches adds up to 22.31 seconds, which implies that for the eight-minute gaming time plus the two-minute logo displaying and opening animation time, nearly 5% is spent on GPGPU computing. During the second execution, 123,175 CUDA kernel launches that originate from 19 unique kernels are collected. The total GPGPU computing time is 17.1 seconds.

We use a hash table to store the combination of kernel and its parameters, which include the input data and kernel configuration. For each individual execution, if a kernel launch can find a tuple with the same input (kernel and parameters) in the hash table, then it is considered as a redundant launch; otherwise this launch is stored in the hash table as a new tuple. As shown in Table 1, the percentage of redundant launches for the two executions reaches up to 24.71% and 25.63%, respectively. Moreover, the identical launch percentage shown in Table 1 indicates the percentage of kernel launches in both executions with the same kernel and data. The number of identical launches is 81,879, accounting for 47.12% and 66.47% of the kernel launches in the first and second execution, respectively.

Through this case study of running real-world games, we observe a large percentage of redundant kernel launches, either within a VM or among multiple VMs running the same game. This observation can be exploited in the cloud gaming scenario, where thousands of VM instances run the same game program. We thus seek

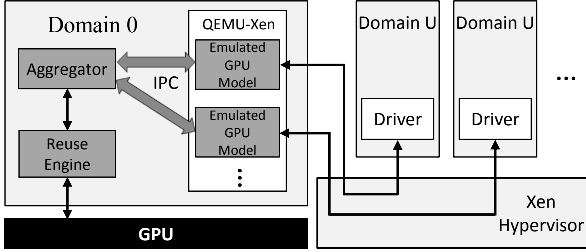

Figure 1: Ecosystem of sharing and reusing framework

to eliminate redundant GPU computations by reusing previous results, which leads to faster execution, higher throughput and energy efficiency.

3 Prototype Implementation

In this section, we present the design and implementation details of the GRU ecosystem.

3.1 GRU Ecosystem Overview

GRU is built on top of GPUvm [26]. GPUvm is an open source Xen-based GPU virtualization software providing virtualization approaches by exposing host GPU device model to guest device drivers. The ecosystem of GRU is described in Fig. 1, which is composed of an extended Qemu-Xen, an aggregator, and a reuse engine. We extend the Qemu-Xen emulator and GPUvm's aggregator to intercept GPU memory-copy and computation operations, *i.e.*, kernel code and the corresponding input data. To enable result reuse, we implement a reuse engine in dom0, which closely interacts with the aggregator.

The extended Qemu-Xen component emulates GPU device models and exposes them to domU, as shown in Fig. 1. Currently, discrete GPUs are treated as I/O devices by the operating system. Therefore, all communications between GPU and operating system are via MMIO. All write- and read-related operations on virtual MMIO are intercepted and forwarded to the aggregator through inter process communication (IPC). Similar to the original aggregator implementation presented in GPUvm, GRU's extended aggregator is still in charge of GPU scheduling and memory management. The difference is that our aggregator also needs to identify and interpret high-level behaviors (data transfer or kernel launch) and related parameters (input memory address, size, kernel configuration, *etc*.) from received GPU operations. The interpreted information is then sent to the reuse engine which lies in dom0 bridging the aggregator and GPU. The reuse engine maintains a LUT to index previous GPU computation results. It also selectively caches results in a "shadow result memory", acting as an in-memory database. After receiving aggregated requests from the aggregator and querying cached results from its LUT, the reuse engine decides whether to issue

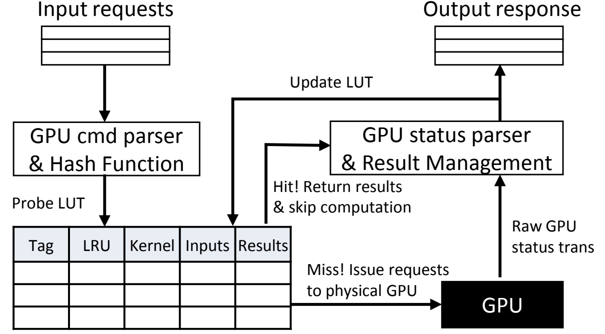

Figure 2: Workflow of memoization and reuse system.

requests to GPU; or if it is a redundant request, the reuse engine directly returns the results that are stored in the shadow result memory.

3.2 Bridging Semantic Gap

GPU is controlled by the CPU using GPU commands that are architecture-specific. A single GPU command is an atomicity operation, which is essentially a value written by CPU and fetched by GPU at a specific address. A series of such commands combine together to perform a high-level GPU operation, such as host-to-device memory-copy or kernel launch. However, interpreting high-level operations from GPU commands is very challenging, particularly given that current NVIDIA GPU drivers and hardware architectures are close-source. This is the semantic gap we need to bridge at the hypervisor level when implementing GRU.

According to the full-virtualization implementation of GPUvm, the device driver in domU considers the emulated GPU model as an ordinary physical GPU instance and write GPU commands to it. Qemu-Xen can intercept all the commands related to I/O and forward them to the aggregator. GPUvm has implemented data structures in its aggregator to parse GPU commands. Unfortunately, this is not enough for implementing our reuse framework. Different from GPUvm, our reuse framework not only parses and forwards the GPU commands, but also need to interpret what operations these commands will perform. For example, when we capture a series of memory write operations, we need to identify which combination of operations indicates a host-to-device data transfer operation, and trace the addresses of source and destination together with its size. Each of these steps is quite challenging because NVDIA makes its device details closed to public. Thanks to the reverse engineering work on GPU device and drivers [18, 11, 17], we borrowed and implemented their data structures at GRU's aggregator to trace all the needed information (e.g., input data address and size). After the aggregator gets the information, it sends the interpreted requests to the reuse engine in dom0.

3.3 Result Memoization and Reuse

The reuse engine is a user process in dom0, which receives requests from the aggregator, checks reuse possibility, and then returns the reusable computation results to the aggregator. Fig. 2 depicts a detailed block diagram of the various components implemented in the reuse engine and how they operate in general. Specifically, an input request is first parsed by the command parser to check whether it is a computation-related request. If it is a host-to-device memory transfer request, the reuse engine delays issuing it to the physical GPU device till the reusability of the corresponding kernel launch request has been checked. The kernel code together with the input data of a kernel launch request are hashed by the hashing function. The LUT uses the corresponding hash value to identify a redundant computation request. Each entry of the LUT contains the address of a reusable result in shadow result memory. We employ pseudo LRU [1] as the replacement policy.

A hit occurs in the LUT probing when we have a memoized result, in which case we skip the actual computation on GPU and generate a faked interrupt to the emulated GPU model to notify the completion of computation. Address of the cached result in shadow result memory is returned to the result manager, which locates the cached result and prepares the data transfer to domU. When the result manager receives the followed GPU request of transferring back the result, it performs the actual result transfer. The GPU status parser is in charge of all the faking work for the status change in the emulated GPU model to notify the completion of computations.

On the contrary, a miss in LUT indicates that result of the current computation request is not available. Such computation requests are then issued to the physical GPU. A new entry is reserved for this computation in LUT, replacing a previous entry according to pseudo-LRU if there is not enough space. After the actual computation on GPU completes, the result will be stored in the shadow result memory and the corresponding tuple in LUT will be updated accordingly.

4 Preliminary Evaluation

We have conducted a preliminary set of experiments to evaluate: 1) the overhead incurred by GRU; 2) the speedup in terms of throughput under different scenarios. These experiments are conducted on a desktop machine with i7-4790K 4.0 GHz processor, 16 GB memory, and NVIDIA Quadro 6000 GPU. The tool stack of Xen is extended from GPUvm for better supporting address translation and GPU command interpretation. In both dom0 and domU, Linux 3.6.5 is running as the operating system with Nouveau (an open-source device driver for NVIDIA GPUs) as the GPU device driver, and Gdev [17], which is an open-source GPGPU runtime and driver software, is running as the CUDA engine. GPU full-virtualization with optimizations provided by GPUvm is employed as the GPU sharing framework. In the experiments, we start two domU instances, assigning 1GB of memory to each domU and 8GB to dom0.

Table 2: Benchmarks used in evaluation

NAME	Description
chess	Chinese Chess game with naive AI
madd	Matrix addition
mmul	Matrix multiplication
srad	Speckle reducing anisotropic diffusion
srad2	Another version of srad with pseudo-inputs
backprop	Back propagation
hotspot	Physics simulation
lud	LU Decomposition

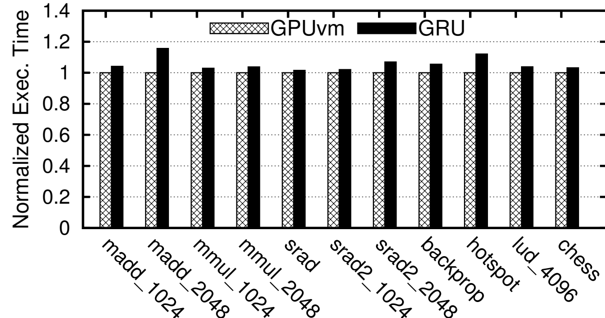

Figure 3: Normalized execution time.

The benchmarks used in this evaluation include Rodinia benchmarks [5] and one synthesized CUDA game, as listed in Table 2. First we evaluated the runtime overhead incurred by GRU, compared to the original GPUvm. The overhead mainly comes from three sources: (*i*) data movement between domU and dom0, (*ii*) hashing and probing in LUT, and (*iii*) storing results in the shadow result memory. Fig. 3 depicts the normalized execution time under GPUvm and GRU when executing each benchmark once. The x-axis of Fig. 3 represents the benchmark with a specific configuration. For example, madd_1024 means the input data size of the matrix addition kernel is 1024×1024. We observe that the overheads incurred under GRU cause a less than 16% increase in execution times for all benchmarks. For a majority of the considered benchmarks (9 out of 11), the overheads cause a less than 7% increase in execution times. As seen in Fig. 3, the madd_2048 benchmark yields the highest overhead (16%), which is because the relatively large input data of the madd benchmark cause more overheads in LUT hashing and data movement.

In the second set of experiments, we evaluated the throughput performance under GRU compared to GPUvm. We synthesize each benchmark to execute (both memory-copy and kernel launch operations) re-

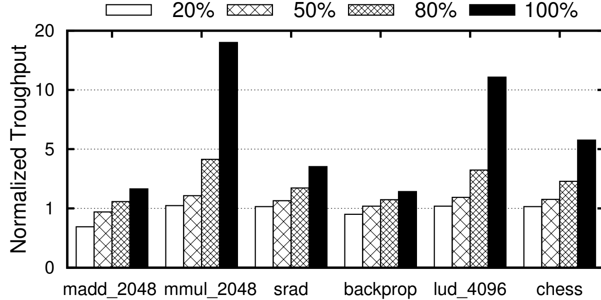

Figure 4: Normalized throughput

peatedly for 10 times after CUDA context initialization. We configure the benchmarks to execute in four scenarios with two (five, eight, and ten, respectively) times of LUT hit, which implies 20% (50%, 80%, and 100%, respectively) of the executions will benefit from reusing the cached results. For the chess game, we simulated game play against a chess AI for ten moves, where 20% (50%, 80%, 100%, respectively) of the AI computations are previously cached. We define throughput as the reciprocal of execution time.

Fig. 4 depicts the normalized throughput of benchmarks in four configurations, where GPUvm is used as the baseline. We observe that the throughput of two benchmarks (madd and backprop) are lower than GPUvm when the hit rate is 20%. This is because the overheads GRU introduces negate the benefits of result reuse. When the hit rate is greater than 20%, GRU outperforms GPUvm for all benchmarks, sometimes by a wide margin. For example, when the hit rate equals 80% (100%), GRU is able to achieve a speedup of 4.5 (18.01) and 3.6 (12.15) for mmul_2048 and lud_4096, respectively. The performance improvement for these two benchmarks is the most significant due to their computation-intensive nature. That is, they both have relatively small input data size but rather heavy computation workloads.

The above-discussed experimental evaluation, although in its preliminary form, show that GRU is promising in achieving significant performance improvement, particularly when the redundancy seen in GPU computation requests increases, while adding a rather small amount of runtime overheads to the current GPU virtualization framework.

5 Related Work

Using GPUs in the cloud. Current approaches to utilize GPUs in the cloud are classified into I/O pass-through [2], API remoting [9, 13, 16, 19, 24], para-virtualization [8, 14, 26] and full-virtualization [26, 27]. I/O pass-through, which dedicates an individual physical instance of GPUs to an VM, does not allow GPU sharing among VMs. API remoting, in which GPU calls are forwarded from the guest VMs to the host, needs to change the software stack in both guest and host. The para-virtualization approach provides multiplexing access to GPU through an ideal device model, with the drawback of modifying the device drivers in guest VMs. The full-virtualization approach is able to support GPU sharing among VMs without modification of existing device drivers and operating systems. However, all these aforementioned work does not exploit the idea of GPU computing result reuse. Our framework leverages the GPU full-virtualization approach and implements GPU computing result reuse to further improve the efficiency of sharing GPU resources in the cloud.

Computing result reuse. Result reuse for CPU computation has been proposed for decades. Sodani *et al.* [25] proposed dynamic instruction reuse to eliminate redundant execution of instruction groups with same inputs. Connors *et al.* proposed a compiler-directed computation reuse approach [6] and explored the hardware support model [7]. Fu *et al.* proposed VMST [12] that reuses large blob of binary code to achieve virtual machine introspection. The similarity of computation between CPU and GPU (instructions and inputs) makes result reuse applicable to GPU. Recently, Arnau *et al.* [3] presented a hardware memoization approach to eliminate redundant fragment shader executions on a mobile GPU. Different from these work, we focus on realizing GPU computing result reuse in a cloud computing environment.

6 Conclusion and Future Work

Redundant GPU computation requests exist in many GPU-accelerated workloads in the cloud. This observation motivates us to explore the possibility of correctly and efficiently reusing GPU computing results among multiple VMs as well as for an individual VM. We propose GRU, a GPU sharing, result memoization and reuse ecosystem for cloud computing. Preliminary experiments show that GRU is able to improve the throughput of eleven GPGPU benchmarks, often by a wide margin, compared to a state-of-art GPU virtualization framework.

The GRU system framework is still at its initial stage. In the near future, we plan to complete and optimize the implementation of GRU by performing the following tasks: (*i*) implementing the framework in a more stabilized and generalized manner, which can reliably support more complicated kernels (e.g., those with data dependencies) and applications (e.g., cloud gaming). (*ii*) introducing reuse-measuring metrics or compiler-assisted technique to identify partially reusable results. (*iii*) for result-resilient computation or approximate computation where results do not need to be exactly accurate, exploring methods to enlarge the reusability or introducing signature-based match.

Discussion Topics

We hope to receive the reviewers' comments on the idea, design, and implementation of the GRU ecosystem presented in this paper. Any suggestions in particular on improving the efficiency and reliability of our current design and implementation would be greatly appreciated. The controversial point of this paper mainly lies in the efficiency of the current GPU full-virtualization system framework on top of which our work builds upon. Since current GPU virtualization techniques are not yet mature enough, the overhead and latency caused by virtualizing GPUs could be large under certain circumstances.

This paper may generate interesting discussions about how to utilize and share GPUs in a cloud environment in a more feasible and efficient manner. Researchers from related domains, including GPU architecture, GPGPU computing, operating systems, data/computing reuse, high performance computing, cloud gaming, and cloud computing in general, may be interested in and inspired by this work.

We would like to point out to the reviewers that our system currently only supports the reuse of computations with the exact same kernel codes and input data. We are currently working on supporting partial result reuse. Also, our current GRU implementation does not support graphics-related rendering and multiplexing applications. As seen in the preliminary experiments, for scenarios where little redundancy is seen in GPU computation requests, the benefits brought by GRU could be trivial (even if the GRU-incurred overheads are reasonably small). Thus, it is important to explore potential techniques that can improve the result reusability under different scenarios. Finally, if the efficiency (w.r.t. overheads) and reliability of current GPU virtualization technologies do not improve to reach a mature stage, the whole ideal of GPU computing result reuse might fall apart.

References

[1] AL-ZOUBI, H., MILENKOVIC, A., AND MILENKOVIC, M. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *ACM-SE* (2004), ACM, pp. 267–272.

[2] AMAZON.COM. Amazon elastic compute cloud (amazon ec2).

[3] ARNAU, J.-M., PARCERISA, J.-M., AND XEKALAKIS, P. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. In *ISCA* (2014), IEEE, pp. 529–540.

[4] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *SOSP* (2003), ACM, pp. 164–177.

[5] CHE, S., BOYER, M., MENG, J., TARJAN, D., SHEAFFER, J. W., LEE, S.-H., AND SKADRON, K. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC* (2009), pp. 44–54.

[6] CONNERS, D., AND HWU, W.-M. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO* (1999), IEEE, pp. 158–169.

[7] CONNORS, D. A., HUNTER, H. C., CHENG, B.-C., AND HWU, W.-M. W. Hardware support for dynamic activation of compiler-directed computation reuse. In *ASPLOS* (2000), ACM, pp. 222–233.

[8] DOWTY, M., AND SUGERMAN, J. Gpu virtualization on vmware's hosted i/o architecture. *ACM SIGOPS Operating Systems Review 43*, 3 (2009), 73–82.

[9] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTÍ, E. S. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *HPCS* (2010), IEEE, pp. 224–231.

[10] EPIC GAMES, INC. Unreal Engine. `https://www.unrealengine.com`, 2004.

[11] FREEDESKTOP. Nouveau Open-Source Driver. `http://nouveau.freedesktop.org`.

[12] FU, Y., AND LIN, Z. Space traveling across vm: Automatically bridging the semantic gap in virtual machine introspection via online kernel data redirection. In *SP* (2012), IEEE, pp. 586–600.

[13] GIUNTA, G., MONTELLA, R., AGRILLO, G., AND COVIELLO, G. A gpgpu transparent virtualization component for high performance computing clouds. In *Euro-Par* (2010), Springer, pp. 379–391.

[14] GOTTSCHLAG, M., HILLENBRAND, M., KEHNE, J., STOESS, J., AND BELLOSA, F. Logv: Low-overhead gpgpu virtualization. In *FHC* (2013), IEEE.

[15] GROUP, K. O. W. OpenCL-The open standard for parallel programming of heterogeneous systems. `https://www.khronos.org/opencl`, 2008.

[16] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. Gvim: Gpu-accelerated virtual machines. In *HPCVirt* (2009), ACM, pp. 17–24.

[17] KATO, S., MCTHROW, M., MALTZAHN, C., AND BRANDT, S. A. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of USENIX Annual Technical Conference* (2012), pp. 401–412.

[18] KOSCIELNICKI, M. envytools. `git://0x04.net/envytools.git`, 2012.

[19] LAGAR-CAVILLA, H. A., TOLIA, N., SATYANARAYANAN, M., AND DE LARA, E. Vmm-independent graphics acceleration. In *VEE* (2007), ACM, pp. 33–43.

[20] MOSIX. VirtualCL Cluster Platform. `http://www.mosix.org/txt_vcl.html`.

[21] NVIDIA. Compute unified device architecture programming guide.

[22] NVIDIA. PhysX. `http://www.geforce.com/hardware/technology/physx`, 2004.

[23] NVIDIA. CUDA Toolkit Documentation. `http://docs.nvidia.com/cuda/profiler-users-guide`, 2013.

[24] SHI, L., CHEN, H., SUN, J., AND LI, K. vcuda: Gpu-accelerated high-performance computing in virtual machines. *IEEE Transactions on Computers 61*, 6 (2012), 804–816.

[25] SODANI, A., AND SOHI, G. S. Dynamic instruction reuse. In *ISCA* (1997), ACM, pp. 194–205.

[26] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. Gpuvm: Why not virtualizing gpus at the hypervisor? In *ATC* (2014), USENIX, pp. 109–120.

[27] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In *ATC* (2014), USENIX, pp. 121–132.

[28] WARD, I. Call of Duty: Ghosts. `http://en.wikipedia.org/wiki/Call_of_Duty:_Ghosts`, 2013.