2016 IEEE International Conference on Computer and Information Technology

G-KVM: A Full GPU Virtualization on KVM

Hong-Cyuan Hsu
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan
hsuquan@hotmail.com

Che-Rung Lee
Department of Computer Science
National Tsing Hua University
HsinChu, Taiwan
cherung@cs.nthu.edu.tw

***Abstract*—Graphics processing Units (GPUs), which originally designed for computer graphics applications, have been widely adopted to general purpose computing in many domains owing to their massive computational power. In the era of cloud computing, GPU virtualization becomes an important technique for the better management of GPUs in data centers. However, most of current solutions are not full virtualization. They either need to modify the guest drivers or libraries, or restrict the hardware sharing capability. The only full GPU virtualization solution is GPUvm, which however can only be executed on Xen hypervisors. In this paper, we present a full GPU virtualization solution on KVM (Kernel-based Virtual Machine), called G-KVM. Our work is not merely a direct porting of GPUvm to KVM, since Xen and KVM have fundamental differences in their system architectures. Two major changes of G-KVM are aggregator and QEMU device model. The experiments show that G-KVM has better performance for MMIO operations than GPUvm on Xen hypervisor. For the compute-extensive experiments, execution time of G-KVM can achieve nearly 82% of native performance, which is similar to GPUvm. The performance scaling experiment shows that the performance of single machine with G-KVM can be scaled up to multiple virtual machines.**

I. Introduction

Graphics processing units (GPUs), originally designed for computer graphics applications, have been widely adopted to general purpose computing owing to its massive computational power and the maturity of GPU programming languages, such as CUDA [1] or OpenACC [2]. Applications of general purpose GPU (GPGPU) include scientific computing, computational finance, data mining, machine learning, and many more areas that require high performance computing. The recent success of deep learning, in which GPGPU plays a critical role to provide necessary computational power, also boosts the demands of GPGPU.

The increasing demand of GPGPU drives the provision of GPGPU instances in cloud infrastructure services. For example, Amazon EC2 provides several types of GPU instances [3]. However, provisioning GPGPU in cloud platforms faces a critical challenge: virtualization. Normally, cloud service providers utilize virtualized hardware to provision infrastructure for better system management and machine utilization. Nevertheless, the GPGPU virtualization techniques are still immature, comparing to the virtualization of other computing resources.

In the spectrum of GPGPU virtualization techniques, the simplest one is the device pass-through, which is also used in Amazon EC2. Using the IOMMU support in CPU extension, such as VT-d [4] for Intel and AMD-vi [5] for AMD, the pass-through technique can assign the I/O device directly to a specific virtual machine. IOMMU helps the mapping of the device DMA address to the guest address space. In addition, the guest can use IOMMU to directly access the MMIO areas. The interrupt requests are also directly forwarded to guest. Although pass-through technique has good security and near native performance, it can only assign one GPU device to a single virtual machine, which makes VM consolidation impossible.

Another common GPGPU virtualization technique is the API-remoting technique [6][7][8], in which the APIs of guest GPU programs are intercepted and forwarded to the host, and the host GPU executes the APIs and sends back the results to the guest. In API-remoting, the GPU related libraries and drivers in the guest need to be modified to intercept the API calls from user programs and to take care the responses from the host. The major difference among various API-remoting techniques is the communication channels between host and guest. Some of them use RPC (Remote Procedure Call) that allows remote communication; others use IPC (Inter-Process Communication) for faster local communication. The API-remoting technique is relatively easy to implement and for some GPU programs, and the performance degradation is small. However, it lacks the flexibility and compatibility, because the library version and drivers are limited to the modified ones.

Some of the GPU vendors provide virtualization support for their devices. For example, Nvidia GRID technology uses a vGPU manager to share GPU based on time slices [9]. The graphics commands of each VM are passed directly to the GPU, and a single GPU can be assigned to multiple virtual machines. AMD also provides Multiuser GPU [10] with SR-IOV (Single Root I/O virtualization) based GPU virtualization. SR-IOV [11] is an extension of PCIe specification, in which the real devices can provide multiple virtual devices. SR-IOV has two PCI functions: physical functions and virtual functions. Physical Functions(PFs) are real PCIe devices that has SR-IOV capabilities; Virtual Functions(VFs) are hardware virtualized functions derived from a physical function. The virtual functions can be assigned to virtual machines by hypervisors. The number of virtual function is limited by the hardware. Hardware assisted GPGPU virtualization provides

978-1-5090-4314-9/16 $25.00 © 2016 IEEE
DOI 10.1109/CIT.2016.30

full features, good performance as well as isolation. However, GPUs with hardware assisted virtualization are much more complicated and therefore more expensive.

The most difficult and seldom GPGPU virtualization is full virtualization, which means the native driver can be run on the guest without any modification of libraries and drivers. The virtualization model of full GPU virtualization can be applied to any kind of GPU, not limited to specific GPUs. The reasons why it is so difficult are that GPUs have very complex architecture and most GPU vendors do not open source their drivers. So far, the only full GPU virtualization is GPUvm[12], [13], which is for Nvidia GPU on Xen hypervisor. GPUvm utilizes an open-source GPU driver Nouveau, the works of reversed engineering by envytools , and Gdev[14], a CUDA [15] driver runtime library for Nouveau, to emulate virtual GPU device model and uses an aggregator for isolating and scheduling among virtual machines.

In this paper, we provide a full GPGPU virtualization solution on KVM hypervisor[16], called G-KVM. Although inspired by GPUvm, G-KVM is not a simply porting of GPUvm from Xen to KVM, since those two hypervisors have fundamental differences in their system architectures. Xen [17] is an open sourced bared-metal hypervisor. Since directly runs on the hardware, Xen can manage the system memory and schedule the CPU tasks among multiple virtual machines, just like an operating system. On the other hand, Kernel-based virtual machine (KVM) is a hosted virtualization technique, which leverages the processor virtualization extension to accelerate the virtualization performance for CPU and I/O. To conquer those differences, we implemented additional APIs for KVM and integrated in QEMU device model and aggregator. The experiments show that G-KVM has better performance for MMIO operations than GPUvm on Xen hypervisor. For the compute-extensive experiments, execution time of G-KVM can achieve nearly 82% of native performance, which is similar to GPUvm. The performance scaling experiment shows that the performance of single machine with G-KVM can be scaled up to multiple virtual machines.

The rest of this paper is organized as follows. Section II introduces the required background knowledge about GPU and GPUvm. Section III describes the implementation of G-KVM. Section IV shows the experimental results. The last section presents conclusion and future works.

II. Background

This section gives a brief introduction on the system architecture of Nvidia GPU and GPUvm.

A. GPU architecture

With the help of open source drivers, researchers outside the GPU companies can know more details about the architecture of GPU. Envytools provides the detailed GPU hardware specification for Nvidia GPU, such as MMIO registers ranges, GPU architecture, etc. For GPGPU, Gdev[14] provides a CUDA [15] driver runtime library for Nouveau, and psnv driver using CUDA for GPGPU computing.

Figure 1 illustrates the overview of the Nvidia GPU architecture. First, PCI devices have a set of registers, whose the configuration space is mapped to memory. The operating system recognizes devices through the PCI configuration space, such as vendor ID, device ID, class code, etc. PCI Base-address register (BAR) is used to inform the device of the base-address of the mapped memory, which is called BAR memory. The BAR is configured by operating system or system firmware. The communication between operating system or device driver with I/O devices is usually by Memory-Mapped I/O (MMIO), which is also located on BAR memory.

The GPU commands are submitted through a special engine called PFIFO, which maintains multiple independent command queues, known as channels. A command queue is a ring buffer with the put and get pointers. A user program can use *ioctl*s to allocate channels, which are mapped to the user-space so that a user program can submit commands to the assigned channel. All accesses to channel control area are intercepted by PFIFO engine for execution. GPU driver uses a channel descriptor to store the settings for associated channel, such as pointers to command buffer, etc.

Figure 2 gives an overview of the GPU memory architecture. A GPU has its discrete on board memory, called GPU RAM. GPU RAM is usually mapped to system memory so that applications can use it as a normal system memory. Large data transfer between system RAM and GPU RAM can rely on DMA. Such mapping creates a virtual memory space for GPU. For the unified memory architecture (UMA), GPU shares the virtual address space with the CPU process. A page table maintains a mapping between the GPU virtual address to GPU physical address or system physical address with a tag in a page table entry indicating the type of the address. The information of page table is also stored in channel descriptors. Figure 3 illustrates how the GPU addresses are translated via GPU page table.

B. GPUvm

Figure 4 displays the system architecture of GPUvm, whose idea is to intercept the requests sent by guest GPU drivers and to handle it to avoid conflicts among multiple VMs. First, QEMU emulates GPU device models for each virtual machine that "has" a GPU. Since the GPU driver communicates with GPU devices through MMIO, all accesses to GPU can be intercepted by emulated IO region from QEMU. Next, all the intercepted requests are forwarded to the aggregatior, a manager for virtual GPUs. The aggregator uses GPU shadow page tables to isolate GPU memory and GPU shadow channels to isolate channel accessing. Finally, a scheduler in aggregator helps the arrangement of command execution from multiple VMs. The details of the aggregator, including memory isolation, channel isolation, and scheduling, are introduced below.

1) Memory isolation: GPUvm intercepts all MMIO accesses and forwards to the aggregator via inter-process calls (IPC). The MMIO area contains the device control and functionalities. Invalid accesses to the MMIO registers may affect the GPU status or cause host kernel panic.

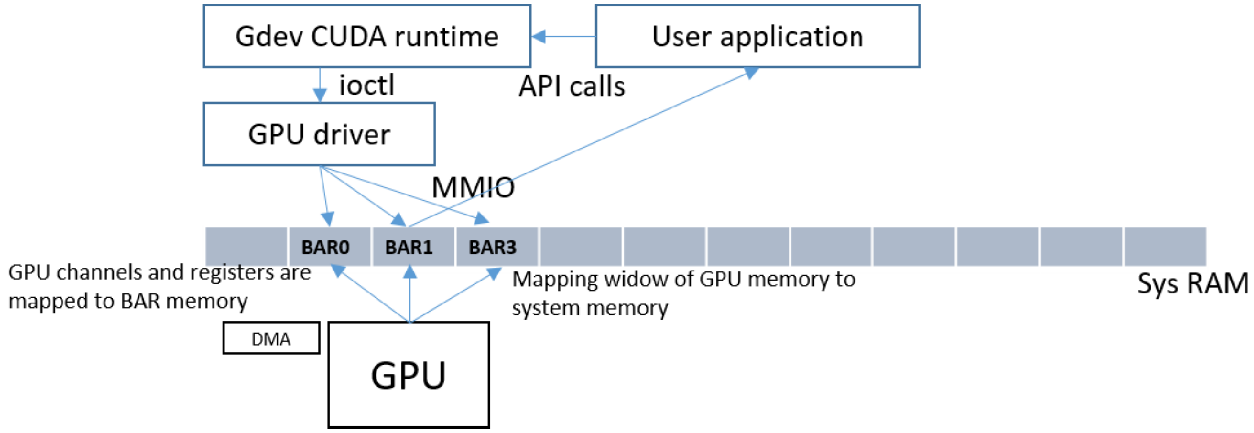

Fig. 1. Overview of GPU architecture

Figure 5 shows the memory system with shadow page table. The guest GPU driver updates GPU page table with guest address space. Since the guest driver thought it is run on a real GPU, the guest driver updates the page table with guest GPU physical address and guest system address. When the guest driver updates the page table, the PCI TLB flush is triggered by guest driver via MMIO registers located on BAR memory. GPUvm can intercept the requests and update the corresponding shadow page table entry to translate the guest physical address to host physical address.

2) Channel Isolation: The channel control region size is fixed in different chipset and the number of channels is limit to different GPUs. The channels should be assigned to virtual machines fairly and isolated among virtual machines. The guest device driver assumes that it runs on a real device, so the physical channels are hidden from guests. GPUvm manages

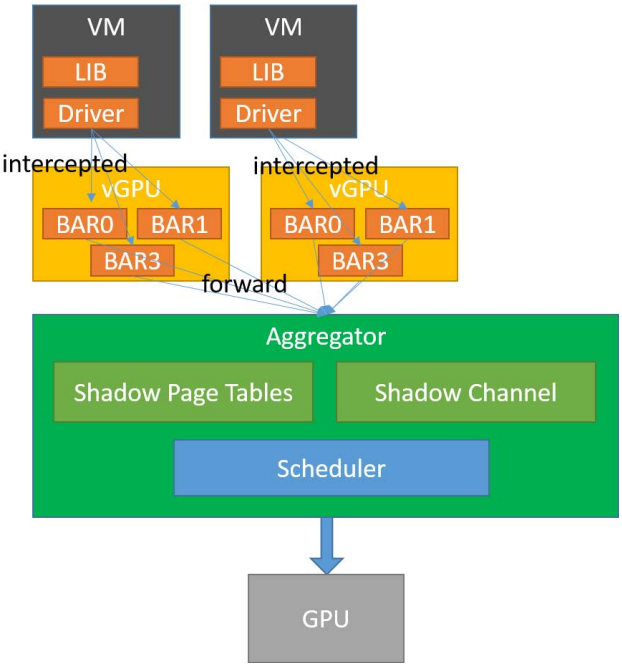

Fig. 4. Overview of GPUvm

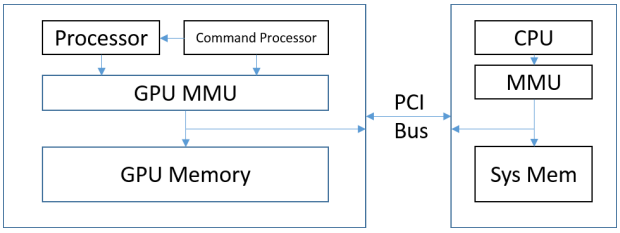

Fig. 2. Memory architecture of GPU

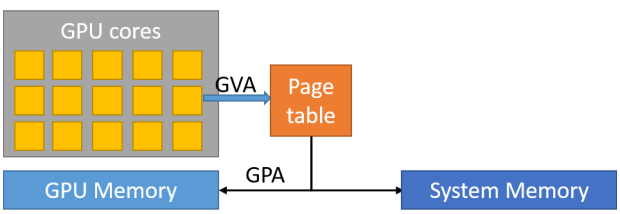

Fig. 3. Address translation on GPU

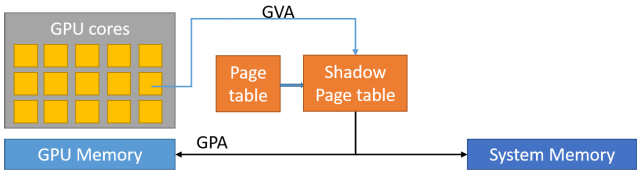

Fig. 5. GPU shadow page table

the mapping between physical and virtual channels. Whenever a guest accesses to virtual channel, the aggregator redirects it to the physical channel.

3) Scheduling: PFIFO internally schedules the GPU commands between channels by time-sharing between the channels. The aggregator maintains a separated context for each virtual machine. Each context has a dedicated queue used to save the commands submitted by guests. A GPU command is submitted to a channel mapped on the BAR memory. Since the guest BAR memory access is intercepted by the hypervisor, the GPU commands submission can be intercepted and added to a queue of context for scheduling. The scheduling algorithm used in GPUvm is BAND algorithm [14] which is designed for virtual GPU scheduling. Figure 6 shows the overview of scheduling architecture.

III. Implementation of G-KVM

Since Xen and KVM are two different types of hypervisors, the implementation of G-KVM is not a direct porting of GPUvm. Two major differences between G-KVM and GPUvm are (1) QEMU device model and (2) the design of aggregator.

A. QEMU-KVM Device Model

Although both Xen and KVM use QEMU for IO device virtualization, their QEMUs have different functions, APIs, and architectures. Therefore, the device model in QEMU needs to be redesigned.

For full virtualization, a device model is necessary to emulate the behavior of the emulated devices. For a PCI device, the device configurations, such as device ID and Vendor ID, are mapped to a memory region, called BAR memory, because it is pointed by the PCI Base-address register (BAR). G-KVM emulates the BAR memory using QEMU, which first adds the required memory regions to listeners. Whenever a memory region is accessed by the guest device driver, callback functions associated with BAR memory are invoked. Based on this mechanism, the accesses by the guest drivers can be easily intercepted and forwarded to the aggregator.

QEMU manages memory as memory regions, which could be classified to different types, such as RAM, MMIO, ROM, IOMMU, container, etc. The guest memory is managed as system memory regions, which are the subregions of the system memory. BAR memories are registered as memory regions and become a subregion of PCI memory region. QEMU uses ioctls APIs to KVM kernel modules to map guest physical address space to hypervisor virtual address space. On the other hand, the host address space can also be mapped to hypervisor virtual address space, which enables the guest to directly access to the host address space. Figure 7 illustrates the mapping among guest, hypervisor and host.

MMIO emulations are expensive to KVM, because they would cause privilege transitions that leads to heavyweight exits from kernel space to userspace. KVM supports coalesced MMIO, by which MMIO emulations can be batched to a ring buffer for later execution. Whenever a virtual machine is created via KVM_CREATE_VM called by QEMU, a MMIO ring buffer is allocated for the virtual machine. The coalesced MMIO region is configured via KVM_REGISTER_COALESCED_MMIO. KVM updates the MMIO range for coalescing. With the support of MMIO coalescing, the number of heavyweight exits is reduced. Figure

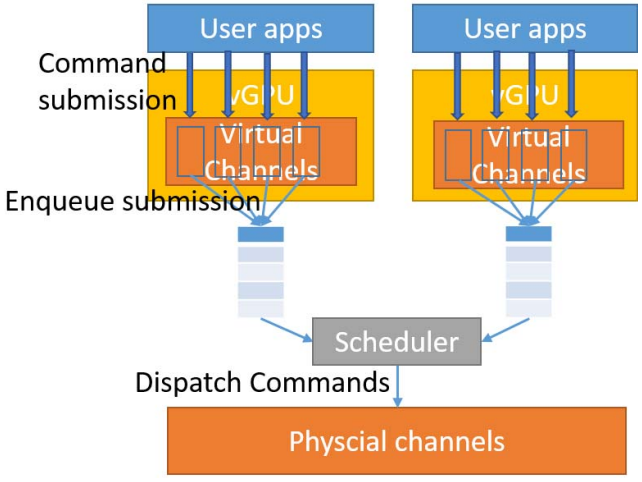

Fig. 6. Overview of GPU scheduing

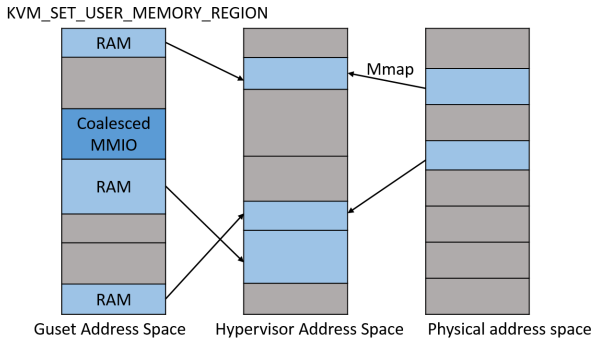

Fig. 7. PCI configuration space

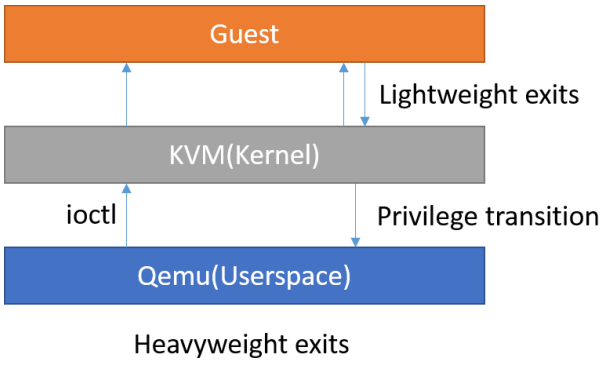

Fig. 8. QEMU-KVM I/O emulation

illustrates KVM coalesced MMIO.

B. Aggregator on KVM

The KVM module creates a VM file descriptor for each KVM-accerlerated virtual machine, by which QEMU can perform ioctl APIs to KVM module for virtual machine configuration, such as vCPU and memory region. Since a VM file descriptor is invisible to other processes, the file operation is only permitted by QEMU. The requests from aggregator are forwarded to QEMU via IPC for using the capabilities of KVM and QEMU. Figure 11 illustrates the implementation overview in our design. The guest driver updates the GPU page table with guest address space (guest system address, guest GPU physical address), so a translation between guest physical address and host physical address is required whenever guest update the guest GPU page table. The original KVM module does not provide a userspace ioctl API to allow translation between guest physical address to host physical address(GPA to HPA), so we added additional ioctl APIs to allow address translation from userspace. With those APIs, the aggregator can update the corresponding shadow page table entry with HPA via the translation API when the guest driver updates the guest GPU page table.

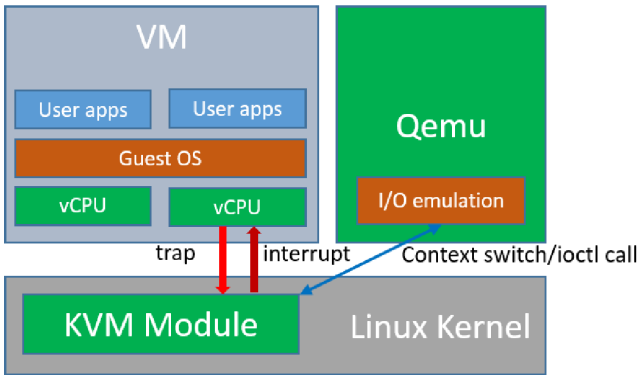

Fig. 9. QEMU-KVM I/O emulation

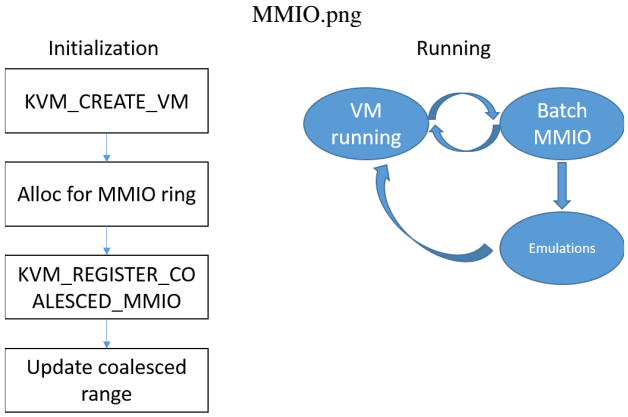

Fig. 10. KVM coalesced MMIO

TABLE I
API MAPPING BETWEEN XEN AND QEMU-KVM

Xen	QEMU-KVM
xc_domain_memory_mapping	memory_region_init_ram_ptr
	memory_region_skip_dump
	mmap
	memory_region_add_subregion
xc_domain_meory_unmapping	memory_del_subregion
xc_gfn_to_mfn	KVM_GFN_TO_PFN

In GPUvm, Xen hypervisor manages the whole system memory. Non-sensitive BAR memory is mapped to guest address space to avoid direct MMIO handling. In Xen, the memory mapping and unmapping between DOM0 and DOMU can be achieved via the APIs in Xen control library. On the other hand, KVM is just a Linux kernel module, which cannot map and unmap the memory by itself. So we implemented a similar mechanism for KVM to pass-through the requests of non-sensitive BAR memory. Figure 12 illustrates the overview of memory mapping in our implementation. When the aggregator requires a mapping between the guest and host system memory, the request is forwarded to QEMU via IPC since QEMU manage the guest system memory. We created subregions for BAR memory region via memory_region_init_io and add skip_dump flag to ignore the memory dump. The API in QEMU would allocate a KVM memory slot via KVM_SET_USER_MEMORY to create a mapping between guest address space and hypervisor address space. The host system memory region is mapped to hypervisor address space via simple system call mmap. After that, the memory region should be initialized with the pointer generated by mmap and added as a subregion to BAR memory. For unmapping, the correspond memory region is simply destroyed and freed. The differences of Xen and KVM's QEMU API are summarized in Table I.

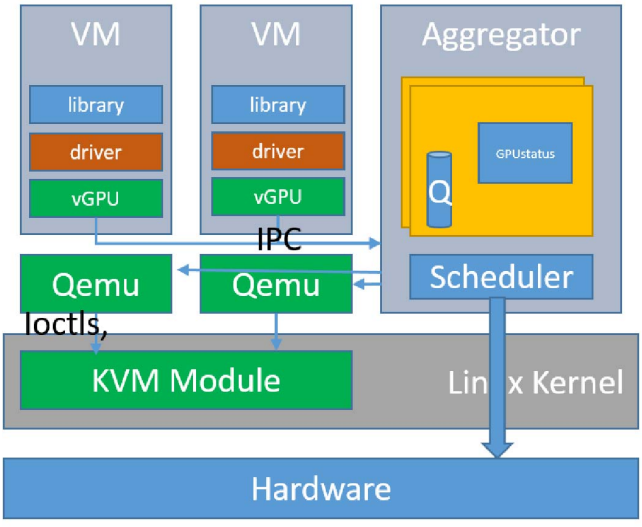

Fig. 11. Overview of our design

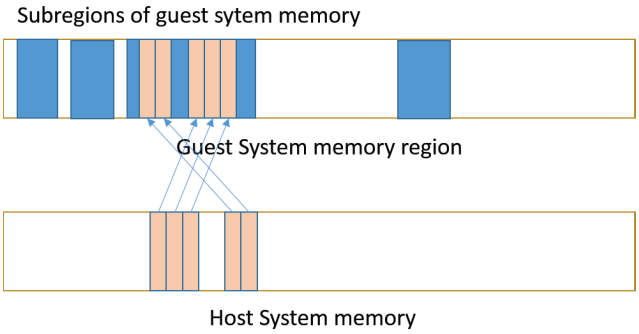

Fig. 12. Overview of memory mapping

TABLE II
EXPERIMENT ENVIRONMENT

	Host environment	Guest environment
Operating System	Ubuntu 14.04	Ubuntu 14.04
Linux Kernel	3.6.5	3.6.5
GPU Memory	1536 MB	512 MB
System Memory	8 GB	1 GB
QEMU	2.4.0	NA
Xen	4.2.0	NA
GPU	Geforce GTX 480	NA
CPU	Intel Xeon Processor W3565	NA

IV. EVALUATION

We have compared the performance of a set of GPU benchmark programs on GPUvm, G-KVM, and native GPU. Since GPUvm can be executed with memory map and without, we also compared their differences. The legend of "mapping" means using memory mapping for BAR memory, and the legend "no-mapping" indicates the version without BAR memory mapping.

Table II lists configurations of the experimental environment. The evaluated benchmark programs are listed in Table III. All experiments were run with the open-source driver Nouveau and gdev CUDA runtime library. The execution time is divided to nine stages. The meanings of execution stages are described in Table IV.

TABLE III
BENCHMARKS USED IN EXPERIMENTS

madd	Matrix addition in 4096x4096
mmul	Matrix multiplication in 4096x4096
hotspot	physical simulation
bfs	breadth first search
backprop	back propogation
lud	LU decomposition
loop	long loop compute without data

TABLE IV
MEANINGS OF EXECUTION STAGES

Init	timeto initialize context
MemAlloc	memory allocation time
DataInit	data initialization time
HtoD	memory copy host to device
kernConf	kernel configuration time
Exec	executing time
DtoH	memory copy device to host
DataRead	data read back time
close	context finalized

TABLE V
TIME ANALYSIS ON MATRIX ADDITION (MS)

	Native	Xen w/o map	Xen w/ map	KVM
Init	57.981	4682.603	678.984	1877.854
MemAlloc	0.841	89.976	8.955	37.949
DataInit	88.266	100.294	101.403	108.041
HtoD	65.069	210.731	210.658	226.503
KernConf	0.007	0.007	0.007	0.007
Exec	39.212	40.055	39.981	39.509
DtoH	32.403	40.860	40.761	35.007
DataRead	36.094	41.640	41.626	42.011
Close	3.205	635.351	595.364	742.767
Total	323.079	5841.520	1717.743	3109.652

For the performance analysis, we chose the matrix addition benchmark to reveal the overhead of MMIO emulation, and chose matrix multiplication benchmark for computation comparison. For performance scaling experiments, we repeatedly executed the matrix addition program thousands of times.

A. Matrix addition

The matrix in the addition benchmark is of dimension 4096. The result is listed in Table V, which shows the major overhead is on the initialization stage. The initialization time is 4.68 seconds in Xen-no-mapping version and 1.87 seconds in our implementation. For GPUvm memory mapped version, the initialization time is reduced to 0.6 seconds because it does not need MMIO emulations. The initialization time dominates total execution time in matrix addition experiment, which is nearly 80% in Xen-no-mapping version, 40% in Xen-mapping version and 60% in our implementation.

We added instrumented instructions to count the number of MMIO handling handled by aggregator. Table VI shows the number of MMIO handling in different benchmarks for GPUvm with and without memory mapping. The BAR memory mapping reduces a huge amount of MMIO emulations. Additionally, we found that almost 95% of MMIO handlings are triggered during initialization time. As mentioned before, MMIO emulations are expensive to KVM and Xen since MMIO emulations would generate trap-and-emulation, which causes execution context-switched to emulator and heavyweight exits. In the same number of MMIO emulations, the initialization time of version on KVM is better than on Xen. In our experiments, initialization time of G-KVM is nearly 3 seconds less comparing to Xen. KVM supports coalesced MMIO, so the number of heavyweight exits could be reduced. Whenever MMIO emulations are triggered, the emulations are first batched to a ring buffer allocated in KVM kernel space. Several emulations are batched for later executions.

B. Matrix multiplication

The matrix in the multiplication benchmark is of dimension 4096. We used it to identify virtualization overhead in the

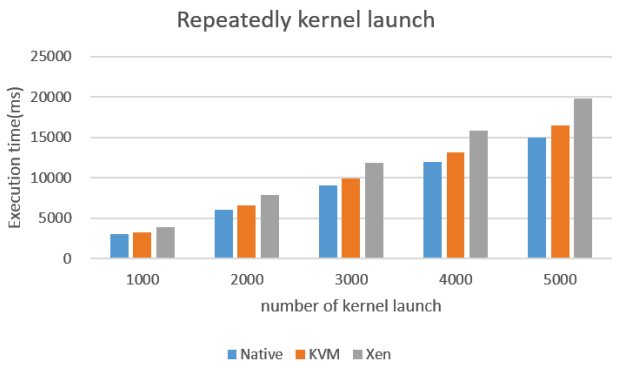

Fig. 13. Repeatedly launch kernel

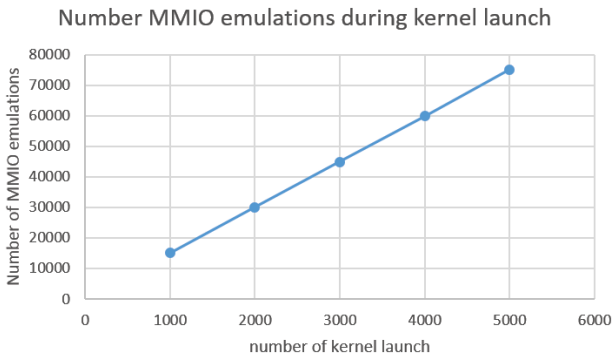

Fig. 14. Number of MMIO emulations

execution part. The result is listed in Table VII.

Comparing to the execution part, the result shows that our implementation can achieve 82% of native performance, which is similar to GPUvm on Xen. The time of HtoD is more than native in guest because whenever the guest GPU page table is updated, the aggregator would update the corresponding shadow page table entry.

C. Performance scaling

This experiment uses matrix addition program as the kernel, and repeatedly launches the kernel for executions on a bare machine and virtual machines. For the comparison of performance scaling, we conducted experiments with different number of kernel launches from 1000 times to 5000 times. It runs the benchmark on a physical machine, Xen guest, and KVM guest. Next, we prepared 2 virtual machine on KVM and 2 virtual machine on Xen and concurrently run the benchmark

TABLE VI
NUMBER OF MMIO HANDLES IN DIFFERENT BENCHMARK PROGRAMS

	w/o mem mapping	w/ mem mapping
madd	266986	1588
mmul	266985	1587
loop	257911	709
backprop	258956	987
lud	259095	1090
bfs	259935	1071
hotspot	258020	730

TABLE VII
TIME ANALYSIS ON MATRIX MULTIPLICATION

	Native	Xen w/o map	Xen w/ map	KVM
Init	46.382	4683.922	670.877	1585.75
MemAlloc	0.941	87.753	9.348	25.745
DataInit	88.722	99.309	99.956	102.210
HtoD	64.975	210.966	239.957	217.807
KernConf	0.007	0.007	0.007	0.008
Exec	19658.505	23783.904	23781.48	23780.597
DtoH	32.397	42.946	41.651	35.418
DataRead	35.112	41.645	41.660	42.441
Close	3.274	636.362	601.010	727.088
Total	19930.318	29586.820	25485.953	26517.066

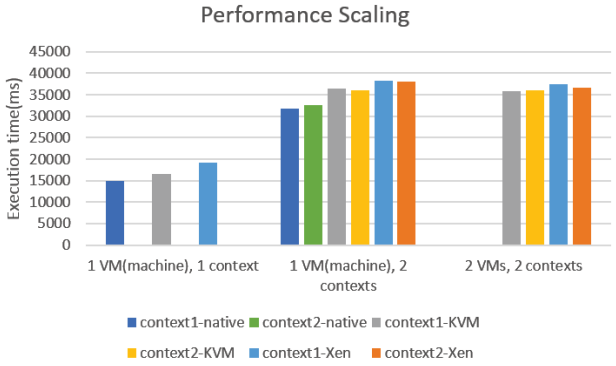

Fig. 15. QEMU-KVM I/O emulation

on KVM and Xen on different guests. For physical machine, we concurrently run this benchmark. Figure 13 shows that as number of kernel launch increases so does the execution time. KVM has better performance than Xen. As Figure 14 shows, launch kernel would cause MMIO emulations. As mentioned above, the MMIO emulation overhead on KVM is less than Xen. The result shows that with same number of MMIO emulations, KVM has better performance. Figure 15 shows that the performance for single VM can be scale up to multiple VMs.

V. CONCLUSION

In this paper, we present a full GPU virtualization G-KVM on KVM hypervisor and address the fundamental difference between Xen and KVM hypervisor. Experiments on KVM, Xen and bare machine show the difference overhead between KVM and Xen hypervisor. MMIO exits are expensive on Xen and KVM because of privilege transitions. MMIO emulation on KVM hypervisor has better performance than that on Xen hypervisor because of the support of coalesced MMIO on KVM.

In the future, we will continue the optimization of G-KVM. The first direction is using virtIO for para-virtualization. The second direction is to complete the BAR memory mapping

version to reduce MMIO emulations overhead for KVM. Last, we want to generalize the virtualization to more kinds of accelerators other than Nvidia's GPU.

ACKNOWLEDGMENT

This study is conducted under the Industrial Fundamental Technology Research Program in Electronic, Electrical and Software Fields (4/4) of the Institute for Information Industry which is subsidized by the Ministry of Economy Affairs and the project 105-2218-E-007-001- of Ministry of Science and Technology of the Republic of China.

REFERENCES

[1] C. Nvidia, "Programming Guide," 2008.
[2] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openaccfirst experiences with real-world applications," in *European Conference on Parallel Processing*. Springer, 2012, pp. 859–870.
[3] E. Amazon, "Amazon elastic compute cloud," *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.
[4] R. Hiremane, "Intel virtualization technology for directed i/o (intel vt-d)," *Technology@ Intel Magazine*, vol. 4, no. 10, 2007.
[5] A. I, "AMD I/O virtualization technology (iommu) specification," *AMD Pub*, vol. 34434, 2007.
[6] L. Shi, H. Chen, J. Sun, and K. Li, "vCUDA: Gpu-accelerated high-performance computing in virtual machines," *IEEE Transactions on Computers*, vol. 61, no. 6, pp. 804–816, 2012.
[7] J. Duato, A. J. Pena, F. Silla, R. Mayo, and E. S. Quintana-Ortí, "rCUDA: Reducing the number of gpu-based accelerators in high performance clusters," in *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. IEEE, 2010, pp. 224–231.
[8] T.-R. Tien and Y.-P. You, "Enabling OpenCL support for GPGPU in kernel-based virtual machine," *Software: Practice and Experience*, vol. 44, no. 5, pp. 483–510, 2014.
[9] N. G. NVIDIA and G. GPUs, "grid boards, nvidia corporation, 2013."
[10] A. Inc., "AMD multiuser GPU." [Online]. Available: http://www.amd.com/Documents/Multiuser-GPU-White-Paper.pdf
[11] P. Kutch, "PCI-sig SR-IOV primer: An introduction to SR-IOV technology," *Intel application note*, pp. 321 211–002, 2011.
[12] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "GPUvm: Why not virtualizing gpus at the hypervisor?" in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, Jun. 2014, pp. 109–120. [Online]. Available: https://www.usenix.org/conference/atc14/technical-sessions/presentation/suzuki
[13] ——, "GPUvm: Gpu virtualization at the hypervisor," *IEEE Transactions on Computers*, vol. 65, no. 9, pp. 2752–2766, Sept 2016.
[14] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First-class gpu resource management in the operating system," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 401–412.
[15] D. Kirk *et al.*, "NVIDIA CUDA software and gpu parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.
[16] 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, 2007, pp. 225–230.
[17] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 164–177.