Vol.2013-OS-126 No.14
2013/7/31

GPU の完全仮想化

鈴木 勇介[1,a)]　加藤 真平[2,b)]　山田 浩史[3,c)]　河野 健二[1,d)]

概要：Graphics processing units (GPUs) は最先端のメニーコア計算デバイスとなっていて，その最も大きな利点は超並列計算による性能のスケーラビリティである．しかしながら， GPU は仮想化の機能を欠いており，特定の用途への利用は制限されている．GPU をシステムにおける第一級の計算資源とするために，本研究では， Xen を用いた GPU の仮想化を提案する．特に，既存の多くのプログラミングフレームワークを利用すべく，本研究では GPU の完全仮想化を実現する．我々のマイクロベンチマークの評価の結果，GPU ワークロードが本質的に時間が掛かるとしても，GPU 仮想化のオーバーヘッドは低く抑えられることがわかった．

キーワード：GPGPU, 仮想マシン技術

1. Introduction

Graphics processing units (GPUs) are becoming very powerful platforms in the parallel computing market, embracing the concept of many-core compute devices. Application domains of GPUs are increasingly spread ranging from embedded systems to supercomputers. Examples include autonomous vehicles [1], software routers [2], encrypted networks [3], storage and file systems [4], and a plenty of scientific applications. Such a rapid growth of GPUs is encouraged by recent advances in the programming language. CUDA and OpenCL are particular instances of the programming language that facilitated general-purpose computation on GPUs, *a.k.a.*, GPGPU.

Due to emergence of GPGPU programming, GPUs are becoming more and more generalized for compute applications. Notably the peak performance of GPUs in the current state of the art exceeds 3 Tela FLOPS, integrating more than 1,500 cores on a single chip, which is nearly equivalent of 19 times that of traditional microprocessors such as Intel Core i series.

Assuming that this performance trend continues, the GPU may turn into an on-chip supercomputer platform in the future. Instead of building a cluster of wimpy nodes, cloud computing services could be provided with just a single node employing the GPU. What is currently lacking in this scenario is a reliable platform of virtual machines (VMs) to isolate client accesses to GPU resources. Although first-class GPU resource management in multitasking environments has been studied recently [5], [6], [7], system support for GPU virtualization is limited to the programming runtime level and paravirtualization [8], [9], [10], [11], [12], [13], [14]. Albeit a long history of virtualization technology, an integration of GPUs still remains an open problem.

Contribution: This paper presents Gxen, an open architecture of GPU virtualization using Xen [15]. Gxen is distinguished from previous work in that we support "full" virtualization in the hypervisor layer. There is no need to modify device drivers and runtime libraries. We provide the design and implementation of Gxen, disclosing details of GPU virtualization mechanisms. We also demonstrate that the overhead imposed on full virtualization is not very significant given that GPU workload is time consuming by nature. To the best of our knowledge, this is the first piece of work that answers the question: *why not fully virtualize GPUs?*

Organization: The rest of this paper is organized as follows. Section 2 presents the system architecture of Gxen. Section 3 describes the status and assumption of our

[1] 慶應義塾大学
[2] 名古屋大学
[3] 東京農工大学
[a)] yusuke.suzuki@sslab.ics.keio.ac.jp
[b)] shinpei@is.nagoya-u.ac.jp
[c)] hiroshiy@cc.tuat.ac.jp
[d)] kono@ics.keio.ac.jp

ⓒ 2013 Information Processing Society of Japan

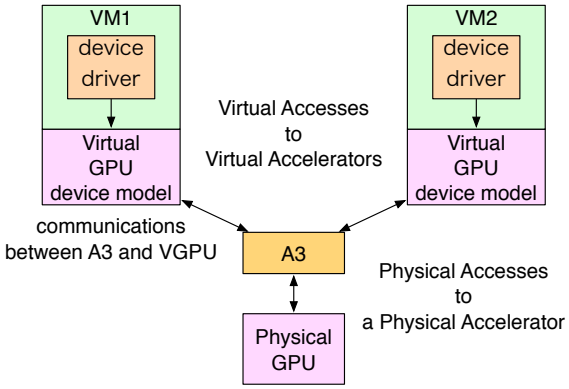

図 1 Conceptual architecture of Gxen.

prototype implementation. Section 4 provides a brief performance evaluation using a microbenchmark. Section 5 discusses related work, and this paper concludes in Section 6.

2. System Architecture

Figure 1 illustrates the conceptual architecture of Gxen. The device interface, *a.k.a.*, the device model, of the virtual GPU is provided with each VM where a native GPU device driver is running. Any access to the virtual GPU is managed by the hypervisor and is routed to the device model. Hence direct accesses to the physical GPU are not allowed in Gxen. The device model in turn sends the access information to the resource manager, called *Aggregator of Accesses to Accelerators* (A3) in this paper, which arbitrates accesses to the physical GPU and manages its status.

A3 is a primary component of Gxen. It aggregates all accesses to the physical GPU while managing the virtual GPUs. The communication between the device model and A3 is based on a client-server model. A3 virtualizes resources of the physical GPU and assigns them to each virtual GPU. Accesses to the virtual GPU are aggregated in A3. While arbitrating these accesses, A3 modifies them to manage the status of the physical GPU. This architecture allows a single physical GPU to be used as multiple logical GPUs among multiple clients. As a result, the physical GPU is successfully multiplexed among VMs.

The main objective of A3 is the virtualization and separation of (i) the device memory, *a.k.a.*, the video memory (VRAM), and (ii) the hardware channels of the GPU. A3 is also required to handle memory-mapped I/O (MMIO) accesses with respect to the PCIe base address registers (BARs). To enhance the performance of virtual GPUs, A3 uses Intel Virtualization Technology for directed I/O (VT-d) [16]. Switching the entries of VT-d accordingly, the guest system physical memory space of an appropriate VM is implicitly targeted for the direct memory access (DMA) from the physical GPU at runtime. Thus, the physical GPU is accessed by no more than one VM at the same time.

2.1 Translation of Physical GPU Addresses

The GPU incorporates a VRAM on board as a fast high-bandwidth device memory. This VRAM provides continuous physical memory addresses. Since a guest device driver assumes that it occupies this memory space starting at address "0x0", we must translate guest device physical addresses into host device physical addresses transparently.

A3 splits the VRAM of the physical GPU into multiple pieces for the virtual GPUs. It also reserves a particular region of the VRAM for A3 itself, which is to be used for such data sets that must reside in the VRAM for GPU virtualization. Any data access to the virtual GPU specified by guest device physical addresses are translated to the corresponding host device physical addresses by A3. For example, most GPUs support such I/O functionality that allows the CPU to read and write the VRAM space directly specifying the device physical address. To multiplex this type of direct read and write access, which does not use hardware-oriented DMA commands, the translation of physical GPU addresses is an essential task of A3.

2.2 Shadowing of GPU Page Tables

GPU contexts run in their own virtual memory space and physical addresses are not visible to programmers. In fact, CUDA does not provide an API to control physical addresses of the GPU. When accessing data, virtual addresses are translated to the corresponding physical addresses through GPU page tables. In case of NVIDIA GPUs, GPU page tables are provided as part of *GPU channels*. Almost all hardware resource primitives for the GPU are managed based on these GPU channels according to the NVIDIA's design. In other words, the isolation of GPU contexts is achieved through the GPU channels.

A3 creates shadow GPU page tables in the reserved VRAM area, translating guest device virtual addresses to host device physical addresses. By design, a device driver needs to flush TLB caches every time a GPU page table is updated. A3 detects this operation and accordingly updates the corresponding GPU shadow page table.

An issue of concern is raised when malicious guest VMs exist in the system. If some malicious guest device driver creates a real GPU page table configured to point to

ⓒ 2013 Information Processing Society of Japan

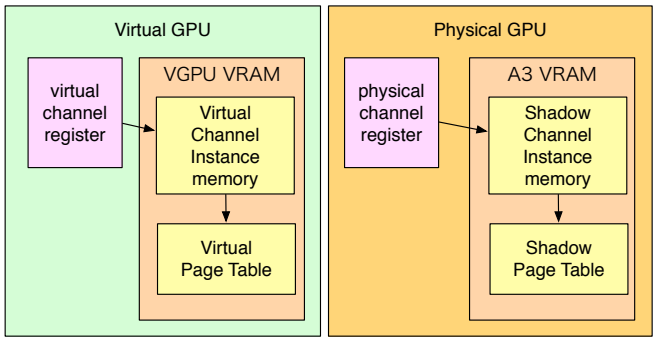

図 2 Shadow channel RAMIN and shadow page tables.

the VRAM area allocated to other VMs, the isolation of virtual GPUs is violated, because A3 cannot detect direct accesses from GPU channels (contexts). It should also be noted that real GPU page tables are referenced from the special VRAM area called the instance memory (RAMIN). Since the RAMIN is part of the VRAM, it can also be altered by malicious VMs.

In order to overcome this issue, A3 creates the *shadow channel RAMIN* in the reserved VRAM area. The interaction of the shadow channel RAMIN and the shadow page tables is illustrated in Figure 2. The shadow channel RAMIN is designed to reference the shadow page tables, while it is referenced through GPU channel registers. By design, these GPU channel registers are mapped onto the channel-based PCIe address space and they can never be accessed from channels.

2.3 Separation of PCIe BARs

The GPU allocates MMIO regions to the PCIe BARs as a means of allowing the CPU to directly read and write the VRAM. They often serve for GPU control registers and dynamic memory windows, called apertures. The apertures are used to make the specified areas of the VRAM visible and accessible to the CPU through the special GPU page tables. Note that this specification is more dependent on hardware vendors. For simplicity of description, we herein restrict our attention to NVIDIA GPUs.

By design, each PCIe BAR is exclusive for a physical GPU. Therefore A3 virtualizes the PCIe BARs and publishes them to virtual GPUs. When the system is loaded, we also scan the special GPU page tables preconfigured for the apertures to reconstruct them in the reserved VRAM area. Remember that data accesses to the apertures are managed by the GPU page tables. Unfortunately multiplexing of the apertures requires switching of the page tables even for a single byte access. Such a switching operation incurs a significant performance penalty largely due to the overhead of TLB flushes. It

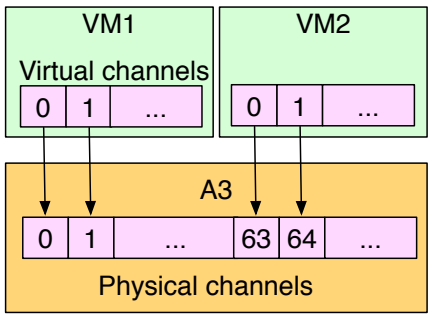

図 3 Mapping of guest and host channels.

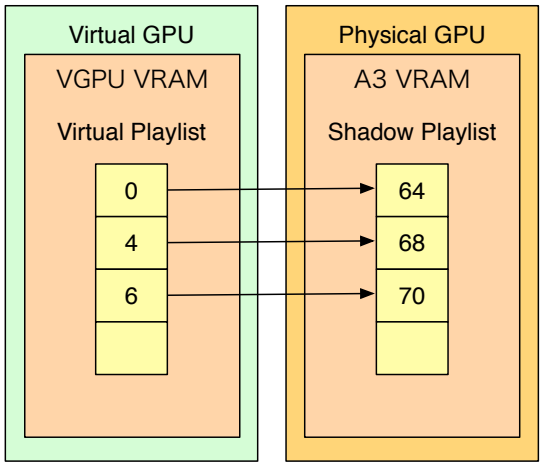

図 4 Shadowing of the playlist.

is also partly impossible to multiplex the apertures because the device driver may use these areas for the management of GPU channels, which should never be evicted. We find that the virtualization of PCIe BARs needs a way of data accesses that does not use the GPU page tables. Fortunately NVIDIA GPUs as well as most other GPUs support direct VRAM accesses based on physical address offsets. Therefore A3 hooks data accesses to the apertures and translates them to host device physical addresses in order to use the above direct VRAM access mechanism.

2.4 Separation of GPU Channels

The number of GPU channels is limited in hardware and they are numerically indexed. The device driver assumes that these indexes start from zero. Since the same index number cannot be assigned to different GPU channels, the separation of GPU channels must be supported to multiplex VMs.

A3 divides physical GPU channels into the same number of virtual channels as VMs. Real indexes of GPU channels are hidden from VMs but virtual indexes are assigned to their virtual channels. Mapping of real and virtual indexes is managed by A3 as shown in Figure 3.

When a GPU channel is used, it must be activated through the *playlist* where the active channel index num-

ⓒ 2013 Information Processing Society of Japan

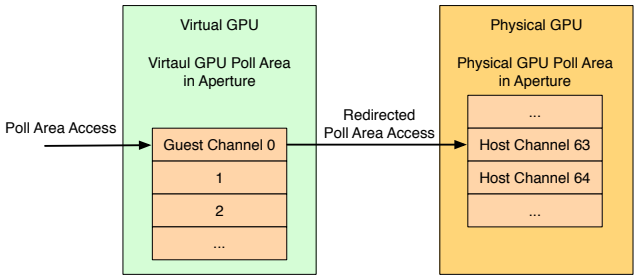

図 5 Mapping of the Poll Area.

ber is written. In virtualization, we use virtual indexes instead of real indexes. Therefore the index number written to the playlist must be translated accordingly. To support this translation, A3 provides the shadow playlist as shown in Figure 4. Each virtual GPU is assigned with one shadow playlist, which is created in the reserved VRAM space. When the virtual GPU is selected for run, the corresponding playlist is used as the physical playlist.

2.5 Poll Area Mapping

The GPU operates based on commands issued from the CPU. There is a command buffer prepared for each GPU channel, which has a fixed size of a continuous region on the virtual memory space allocated to the specified aperture. It is mostly used to control the channel.

This command buffer may be architecture-dependent. Assuming that we use NVIDIA GPUs, the command buffer is mapped to a particular continuous region of PCIe BAR1 aperture and is entitled with "Poll Area". Since we can change the location of this Poll Area by modifying the value of a GPU register, the memory space cannot be assigned for virtual GPUs statically. To successfully execute GPU requests from VMs, we need to handle each VM's operations related to this area.

A3 maps the physical Poll Area to an appropriate channel at runtime. An overview of this mapping is depicted in Figure 5. A3 prepares a page table that maps virtual-to-physical addresses for the Poll Area, and sets it to a GPU register. By performing the channel translation described the previous section, each GPU request is executed appropriately.

3. Implementation

We implement Gxen using Xen 4.2.0. The underlying operating system is based on Linux Kernel v3.6.5. In the Xen environment, the Dom0 VM uses Ubuntu 12.10 while DomU VM uses Fedora 16. We assume NVIDIA GPUs and an open-source device driver called Nouveau, which is provided as part of the mainline Linux kernel. Gdev [5] is used as the CUDA runtime driver providing a GPGPU programming environment. Note that Gxen will not require modifications at all to Linux Kernel, Nouveau, and Gdev. This is a primary advantage of full virtualization.

Limitation: The current prototype of Gxen does not complete the implementation of shadow page tables yet. A small piece of code is manually added to Nouveau to make it work with Gxen but it will be removed in the near future. Note that this manual modification is a minor factor and does not influence performance. We also still partly work in progress for multiplexing of VMs. In particular, GPU channels are not automatically switched among VMs. Instead we run VMs in a pre-configured order for the experiment.

表 1 List of Gdev benchmarks.

Benchmark	Description
MADD	NxN matrix addition
MMUL	NxN matrix multiplication

4. Evaluation

We now provide a basic performance evaluation for Gxen using microbenchmarks. The experimental results encourage the GPU to be fully virtualized if applications deal with a practical data size.

4.1 Experimental Setup

We ran CUDA applications by using Gdev [5] as CUDA runtime. For comparison, we also measure their performance in native Linux (*Native*), and Xen PCI Passthrough (*Pass-through*) where the underlying GPU is exposed to a VM via the PCI-passthrough function. We first measured typical matrix calculation programs shown in Table 1.

We used a machine that has Intel Xeon E5-2470 processor and NVIDIA Quadro 6000. We ran Linux 3.6.5 and Nouveau module as the GPU device driver.

4.2 Results

The results are shown in Fig. 6 and 7. The x-axis represents execution time normalized by the result of Native, and the y-axis is the matrix size. The figure reveals that overhead incurred by Gxen gets negligible. Since the main task of Gxen is to intercepts guest kernel GPU operations such as PCIe BARs access and page table updates, the execution of matrix calculation is not almost affected. When the matrix size is bigger, the execution times in Gxen is almost the same as ones in the others. The execution time in Gxen is shorter than the others one when the matrix

ⓒ 2013 Information Processing Society of Japan

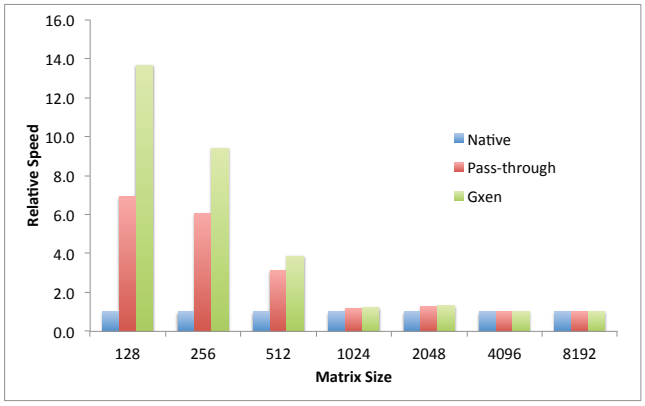

図 6 Relative speed of MADD benchmark.

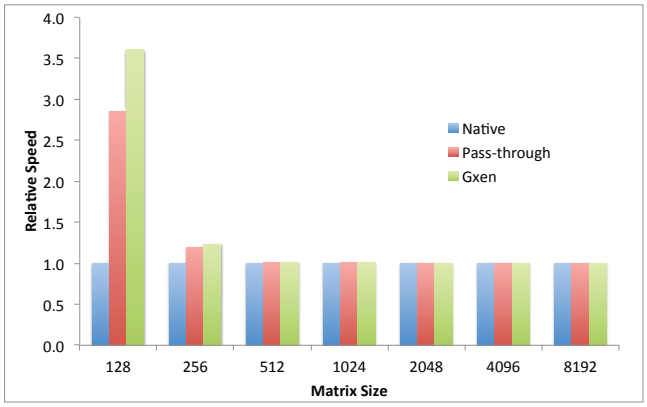

図 7 Relative speed of MMUL benchmark.

表 2 Detailed times of MADD with $N = 128$.

	Native	Pass-through	Gxen
Initialize (ms)	20.14	14.57	7710.24
Kernel (ms)	3.65	0.53	0.27
Finalize (ms)	1.93	0.36	221.74
Total (ms)	25.72	15.46	7932.25

sizes are smaller, due to mysterious behavior of the GPU.

Table. 2 exhibits an analysis of the execution of MADD whose N is 128. *Initialize* is time for initiating a GPU context, *Kernel* is the GPU kernel execution time, *Finalize* is time for finalizing a GPU context, and *Total* is the total execution time. This result reveals that Gxen takes more time for *Initialize* and *Finalize* than Native and Pass-through. Because the VM sends many requests to GPU in initiating and finalizing a GPU context, Gxen routes the requests from the device model to A3, and from A3 to the physical GPU. This overhead gets more smaller or negligible as time for Kernel is longer in cases where N is longer. We believe that this overhead is not a serious problem since typical GPGPU workloads perform their calculation for a long time, which means time for Kernel is typically long.

5. Related Work

To execute GPU applications on VMs, some approaches support APIs for GPU computing in the VMs, hook the APIs, and forward the requests to the real GPU like a proxy. GViM [8], vCUDA [9], and Pegasus [10] provide NVIDEA's CUDA programing API in the VMs so that GPGPU applications running on them can be executed. rCUDA [11] also supports the API and sends the requests to remote GPUs. VMGL [12] provides OpenGL API inside VMs and Tien et al. present a way to provide OpenCL API using KVM [13]. In these systems, we cannot use "as-is" existing software stack in VMs; we have to insert into VMs special modules that forward the requests of GPU applications to the real GPUs. Since our approach fully virtualizes GPUs, we can build our own software stack on the VMs.

VMware SVGA II[14] para-virtualizes GPUs. By loading a special device driver, we can execute GPU applications inside VMs. Since our approach is to virtualize GPUs fully, we can load any device driver, which means we can use any software stack we want.

There are several approaches to efficiently manage resource of GPUs. TimeGraph [6] schedules GPU commands at the device-driver-level in a prioritized manner. Gdev [5] is a kernel-level mechanism that provides a GPU scheduling scheme to logically partition a physical GPU into multiple GPUs. GPUfs [17] allows GPU code to access the host's file systems by providing POSIX-like API. While these approaches focus on the efficient management of GPUs, our approach focuses on the virtualization of GPUs. We can combine our approach with these ones to efficiently virtualize GPUs.

6. Conclusion

We have presented Gxen, an open architecture of GPU virtualization. We provided an open-source prototype implementation of Gxen and its overhead evaluation. It turned out that full virtualization of the GPU adds some performance penalty to the initialization of GPU channels, but the execution times of GPU contexts are not very affected. To the best of our knowledge, this is the first piece of work that achieved full virtualization of the GPU.

In future work, we will complete the remaining pieces of coding related to the shadow page table and multiplexing of VMs. We will also allow multiple GPU channels

ⓒ 2013 Information Processing Society of Japan

assigned to different VMs to run simultaneously, while this paper assumed that VMs access the GPU exclusively. This simultaneous executions of VMs on the GPU increase the total throughput of the system.

参考文献

[1] McNaughton, M., Urmson, C., Dolan, J. M. and Lee, J.-W.: Motion planning for autonomous driving with a conformal spatiotemporal lattice, *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, pp. 4889–4895 (2011).

[2] Han, S., Jang, K., Park, K. and Moon, S.: PacketShader: a GPU-accelerated software router, *ACM SIGCOMM Computer Communication Review*, Vol. 40, No. 4, pp. 195–206 (2010).

[3] Jang, K., Han, S., Han, S., Moon, S. and Park, K.: SSLShader: cheap SSL acceleration with commodity processors, *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, USENIX Association, pp. 1–1 (2011).

[4] Sun, W., Ricci, R. and Curry, M.: GPUstore: Harnessing GPU Computing for Storage Systems in the OS Kernel (2012).

[5] Kato, S., McThrow, M., Maltzahn, C. and Brandt., S.: Gdev: First-Class GPU Resource Management in the Operating System, *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*, pp. 401–412 (2012).

[6] Kato, S., Lakshmanan, K., Rajkumar, R. and Ishikawa, Y.: TimeGraph: GPU Scheduling for Real-Time Multi-Tasking Environments, *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*, pp. 17–30 (2011).

[7] Rossbach, C. J., Currey, J., Silberstein, M., Ray, B. and Witchel, E.: PTask: Operating System Abstractions to Manage GPUs as Compute Devices, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, pp. 223–248 (2011).

[8] Gupta, V., Gavrilovska, A., Schwan, K., Kharche, H., Tolia, N., Talwar, V. and Ranganathan, P.: GViM: GPU-Accelerated Virtual Machines, *Proceedings of the 3rd ACM Workshop on System-level Virtualization for High Performance Computing (HPCVirt '09)*, pp. 17–24 (2009).

[9] Shi, L., Chen, H., Sun, J. and Li, K.: vCUDA: GPU-accelerated high-performance computing in virtual machines, *Computers, IEEE Transactions on*, Vol. 61, No. 6, pp. 804–816 (2012).

[10] Gupta, V., Schwan, K., Tolia, N., Talwar, V. and Ranganathan, P.: Pegasus: Coordinated Scheduling for Virtualized Accelerator-based Systems, *Proceedings of the 2011 USENIX Annual Technical Conference (ATC '11)*, pp. 31–44 (2011).

[11] Duato, J., Pena, A. J., Silla, F., Mayo, R. and Quintana-Orti, E.: rCUDA: Reducing the number of GPU-based accelerators in high performance clusters, *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, IEEE, pp. 224–231 (2010).

[12] Lagar-Cavilla, H. A., Tolia, N., Satyanarayanan, M. and de Lara, E.: VMM-Independent Graphics Acceleration, *Proceedings of the 3rd ACM International Conference on Virtual Execution Environments (VEE '07)*, pp. 33–43 (2007).

[13] Tien, T.-R. and You, Y.-P.: Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine, *Software: Practice and Experience* (2012).

[14] Dowty, M. and Sugerman, J.: GPU virtualization on VMware's hosted I/O architecture, *ACM SIGOPS Operating Systems Review*, Vol. 43, No. 3, pp. 73–82 (2009).

[15] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and Art of Virtualization, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp. 164–177 (2003).

[16] Abramson, D.: Intel virtualization technology for directed I/O, *Intel technology journal*, Vol. 10, No. 3, pp. 179–192 (2006).

[17] Silberstein, M., Ford, B., Keidar, I. and Witchel, E.: GPUfs: Integrating a File System with GPUs, *Proceedings of the 18th International Conference on Architectural Support for Programmng Languages and Operating Systems (APLOS '13)*, pp. 485–498 (2013).

ⓒ 2013 Information Processing Society of Japan