qCUDA: GPGPU Virtualization for High Bandwidth Efficiency

1st Yu-Shiang Lin
Information and Communications Research Lab
Industrial Technology Research Institute
HsinChu, Taiwan
YuShiangLin@itri.org.tw

2nd Chun-Yuan Lin
Dept. Computer Science and information Engineering
Chang Gung University
Taoyuan, Taiwan
cyulin@mail.cgu.edu.tw

3rd Che-Rung Lee
Dept. Computer Science
National Tsing Hua University
HsinChu, Taiwan
cherung@cs.nthu.edu.tw

4rd Yeh-Ching Chung
School of Science and Engineering
Chinese University of Hong Kong
Shenzhen, China
ychung@cuhk.edu.cn

***Abstract*—The increasing demand for machine learning computation contributes to the convergence of high-performance computing and cloud computing, in which the virtualization of Graphics Processing Units (GPUs) becomes a critical issue. Although many GPGPU virtualization frameworks have been proposed, their performance is limited by the bandwidth of data transactions between the virtual machine (VM) and host. In this paper, we present a virtualization framework, qCUDA, to improve the performance of compute unified device architecture (CUDA) programs. qCUDA is based on the virtio framework, providing the para-virtualized driver and the device module for performing the interaction with the API remoting and memory management methods. In our test environment, qCUDA can achieve above 95% of the bandwidth efficiency for most results by comparing it with the native. Also, qCUDA has the features of flexibility and interposition. It can execute CUDA-compatible programs in the Linux and Windows VMs, respectively, on QEMU-KVM hypervisor for GPGPU virtualization.**

***Index Terms*—GPGPU Virtualization, GPU, CUDA, QEMU, KVM, Virtio, Para-virtualization, API Remoting**

I. Introduction

Graphics Processing Units (GPUs) play an important role to contribute to the recent success of deep learning. The architecture of GPUs usually equip with thousands of cores that can perform embarrassingly parallel tasks is especially suitable for the computation of neural networks and image processing. Many studies have shown that with GPUs, the work of model training [1] or inference [2] can be accelerated significantly. As a result, more and more cloud providers, such as Amazon EC2 [3], start to support the GPU instances as the demand for AI work increases.

One of the challenges to enable the GPU in cloud for infrastructure-as-a-service (IaaS) is virtualization, which can split hardware resources into several VMs to give features of sharing, isolation and on-demand. Comparing to the direct hardware access by the traditional client-server connection, running on isolated multiple virtual machines (VMs) under the hypervisor enlarges to facilitate more secure and higher utilization [4]. However, most architecture and system software of GPUs are not open to the public, which makes efficient software-based virtualization difficult.

The current solutions for GPU virtualization used in cloud environments are supported by GPU hardware vendors. For instance, NVIDIA GRID K1/K2 [3] supports the hardware virtualization, which can effectively divide a physical GPU into multiple virtual GPUs. The other product-level GPU virtualization is Intel GVT-g [5], under the code name XenGT (gVirt) and KVMGT executing on Intel processor-graphics. Both NVIDIA GRID solutions and Intel GVT-g use the "*mediated pass-through*". Although each guest in mediated pass-through has a complete stand-alone virtual GPU, either its discharge is bounded to the specific GPU vendor or only with closed-source that could be rigid, awkward and lack of interposition [6].

To improve the interposition and flexibility of mediated pass-through, more and more API remoting methods for general-purpose GPU (GPGPU) virtualization have been proposed, such as GViM, GVirtuS, vCUDA, rCUDA, mrCUDA, and virtio-CL [7]–[12]. Most API remoting methods focused on virtualizing GPGPU for solving non-graphical HPC problems, through which the interaction of wrapping the GPGPU APIs as a "front-end "in the guest and mediating all accesses to the GPU as a "back-end "in the host. To facilitate the description of the GPU located in the API remoting method, we define it as the "local "or "remote "GPU. The local GPU is a dedicated or an integrated device connecting with the host in the same node via a peripheral device interface or an on-chip bus, and the remote GPU is linked to the host via the intermediate connection with different nodes.

However, in API remoting methods, considerable overhead could occur from the transmissions between the front-end and back-end. For instance, rCUDA, one of the most popular API remoting methods for GPGPU virtualization, although can

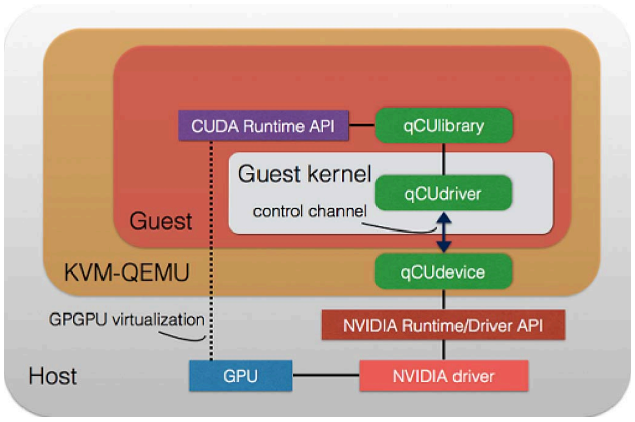

Fig. 1: The system architecture of qCUDA.

achieve approximatively native performance under InfiniBand (IB) interconnection [13]–[17], performs poorly for local GPU virtualization environment. To show that, we launch rCUDA in a VM on QEMU-KVM hypervisor and connect it to the local GPU with only one GbE interface. We measure the bandwidth between the VM/native and local GPU. In the native environment, the average bandwidth was 1745.4 MB/sec and 4890.4 MB/sec for pageable memory and pinned memory respectively. For rCUDA, the average bandwidth for the local GPU is only 61.2 MB/sec and 273.3 MB/sec for pageable memory and pinned memory respectively.

To reduce the overhead in this kind of environment mentioned above, we propose an alternative approach to GPGPU virtualization: qCUDA. qCUDA supports the features of the API remoting method on the KVM-based hypervisor with para-virtualization, such as accessing the GPU routed via the front-end/back-end module. In comparison to the prior work, qCUDA shows its noticeable flexibility and interposition that the framework of qCUDA could perform on Linux and Windows in VM. The VM executes the binary file with the CUDA runtime APIs and GPU code from which is compiled by NVIDIA CUDA compiler (nvcc); furthermore, qCUDA has low-overhead interactions between the front-end and back-end, is leveraged by the virtio framework [18] which provides the ring buffer to the high efficient control channel.

This paper is organized as follows. In Section II, system concepts and design of qCUDA for system components, library interposition, control channel, memory management, NCVM & CVM and pinned host memory are described. The experimental results and discussion are shown in Section III. Section IV and Section V give the related work and conclusion respectively.

II. System Architecture and Design

A. System Components

Figure 1 shows the system architecture of qCUDA. The framework of qCUDA has three components: qCUlibrary, qCUdriver and qCUdevice; their functions are described as follows:

- *qCUlibrary* - The interposer library in VM (guest OS) provides CUDA runtime access, the interface of memory allocation, qCUDA command (qCUcmd), and passing the qCUcmd to the qCUdriver.
- *qCUdriver* - The front-end driver is responsible for the memory management, data movement, analyzing the qCUcmd from the qCUlibrary, and passing the qCUcmd by the control channel which is connected to the qCUdevice.
- *qCUdevice* - The virtual device as the back-end is responsible for receiving/sending the qCUcmd through the control channel. The virtual device depends on gaining the qCUcmd to active related operations in the host, including to register GPU binary, convert guest physical addresses (GPA) into host virtual addresses (HVA), and handle the CUDA runtime/driver APIs for accessing the GPU.

Due to the interaction among components of qCUDA, the VM (guest) which obtains the profit via GPGPU virtualization for accessing the GPU by using CUDA runtime APIs.

B. Library Interposition

In the guest VM, the qCUlibrary provided the wrapper functions that intercepted dynamic memory allocation of CPU code and CUDA runtime APIs. To achieve the zero-copy between the host and guest, qCUDA replaced the dynamic memory allocation function for the policy of converting the address. For directly passing and replacing CUDA runtime APIs from the guest to host, qCUDA created the buffer with 384-bits size to store the qCUcmd structure. The qCUcmd as the argument was passed by the *ioctl* instruction to the qCUdriver; the qCUdriver analyzed the qCUcmd from the qCUlibrary and communicated with the qCUdevice by the control channel.

The qCUcmd was packaged with the unique function IDs, function parameters and GPU binary. The GPU binary, named fat binary (fatbin), which was compiled from the PTX code by the CUDA driver. To query the GPU binary, the qCUlibrary found the pointer, `fatCubin`, pointing to `__fatBinC_Wrapper_t`, which was an intermediate structure of the `nvFatBinSegment` section of the x86 ELF executable. One member of `__fatBinC_Wrapper_t` points to `computeFatBinaryFormat_t`, which was the structure containing the members of pointers, pointing to the fatbin and the size of fatbin. Therefore, the fatbin as the GPU binary can be passed to host without any modification, then in the host, depended on different function IDs and parameters which were loaded from the qCUcmd, the correct processing of GPU would be completed.

C. Memory Management

In the past studies, the API remoting methods for GPGPU virtualization required extra data copies between the guest and host [8]–[10]. In this work, qCUDA provided efficient memory management for data movements between the guest and GPU. qCUDA eliminated the extra data copies between the guest and host; while the guest was created by the host in the same node, the allocated memory from the guest can be seen in the host. The central concept was that the QEMU process allocated the guest physical memory with the malloc() or mmap() function call; a contiguous GPA can be shifted as an HVA by the QEMU process. qCUDA used this mechanism of converting an address to establish efficient memory management for accessing the same pages in a real machine.

Figure 2 illustrates the flow of memory management among the interactions of qCUDA components.

1) The user application dynamically allocates or releases memory in the guest VM.
2) The qCUlibrary makes an allocation hook to change the original behavior of dynamic memory allocation in the CPU code.
3) The qCUdriver implements the mmap() function that allocates several 4MB chunks, each containing the pages within the contiguous GPA.
4) The qCUdriver maps the guest virtual address (GVA) of each chunk to its corresponding guest physical address (GPA).
5) The memory logger maintains a doubly linked list to record each malloc() function call from the user application.
6) Control channel. (Sec. 2D).
7) Map GPA to HVA with NCVM/CVM. (Sec. 2E).
8) Handle the functions of pinned memory mapping. (Sec. 2F).
9) Trigger the functions of GPU memory.
10) Access the GPU memory.

In step 5, the related information of each malloc() function call is packaged in a group G_i inserting into the doubly linked list L: $L = \{G_0, G_1, ..., G_{n-1}\}$ and $G \in L$; $G_i = \{B_i, S_i, E_i, F_i, P_i\}$, where B_i is the number of allocated chunks; S_i and E_i are the start/end positions of the contiguous GVA, respectively; F_i is the flag data; P_i is the GVA of head pointer of adjacent pages and P_{ij} represents the GPA converting from one of the allocated pages.

The mmap() function of qCUdriver maps P_{ij} to corresponding GVA region, so the memory is allocated from a user space of guest need not copy from user space to kernel space. If the user application executes the free() function, the qCUlibrary deliveries the parameter ptr of free() function to the qCUdriver; ptr is a pointer which points to an allocated GVA region with malloc() function call. In the qCUdriver, it could find out which G_i satisfies $ptr \in [S_i, E_i]$, and removes G_i from L. Unless the specific functions are triggered for converting GPA to HVA, P_{ij} can be only accessed from the guest.

The functions for converting GPA to HVA include cudaMemcpy(), cudaMemcpyAsync() and cudaHostRegister(); when the user application executes one of these functions, the qCUdriver converts P_i to GPA $P_i^{'}$; the qCUdriver incorporates $P_i^{'}$ into the qCUcmd and deliveries it to the qCUdevice. The qCUdevice convertes P_i^i to the HVA P_i^h, which denotes a pointer that can be read from the host and P_i^h pointes to P_{i0}.

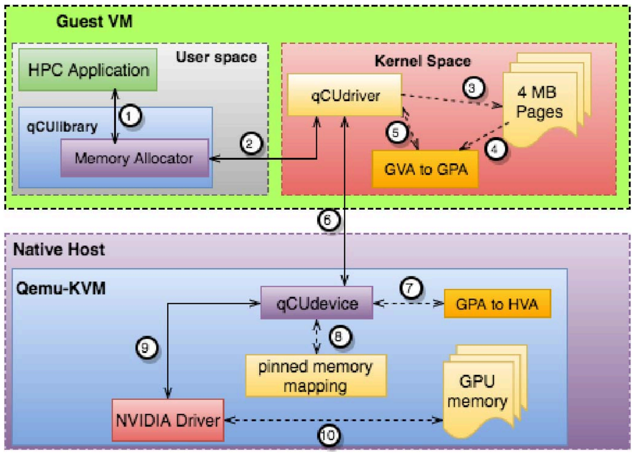

Fig. 2: The flow of memory management in qCUDA.

Therefore, P_{ij} can be queried by the qCUdevice and converted it to the corresponding HVA P_{ij}^h. Actually P_{ij}^h and P_{ij} map to the same physical frames. With the converting process ($P_{ij} \to P_{ij}^h$), the qCUdevice can access the memory region where is pointed by P_{ij}^h; furthermore, by this converting process, it can achieve the features of zero-copy in CUDA since the pointer can be passed and indicated to the same memory region sharing with the guest and host.

D. Control Channel

qCUDA based on the para-virtualization technique [19] that took the qCUdriver as a hook interface for the guest-host communication. qCUDA leveraged the virtio framework as the control channel (Figure 1); each CUDA runtime API function took only two transactions of the qCUcmd at most, which was penetrated by the control channel. qCUDA provided the *qcu_send_cmd* interface, which merely exploited the qCUcmd to communicate the guest-host via the virtio framework. Although each CUDA runtime API function call could accompany some virtio latency, the virtio latency was the tiny proportion of the execution time; we will discuss this in Section III.

E. NCVM & CVM

In the host, P_{ij}^h represented that has the same physical frames mapping with P_{ij}; the qCUdevice directly scanned P_{ij}^h where pointed to the non-contiguous virtual memory; the content of memory would be iteratively queried with different chunks, we call it as the non-contiguous virtual memory mapping (NCVM). For instance, the qCUdevice needs to transfer the data to/from the GPU; the transferred data from/to the memory where P_{ij}^h pointed, and it would be split into several pieces with a chunk size (4MB) for iteratively cudaMemcpy()/cudaMemcpyAsync() function call due to the NCVM. In addition to NCVM, the qCUdevice can map each P_{ij}^h to contiguous virtual memory, named the contiguous virtual memory mapping (CVM). If qCUDA adopted CVM, it only required the pointer which pointed to the contiguous virtual memory and size of data transfers; the cudaMemcpy()/cudaMemcpyAsync() function call can be executed only once. We will compare the performance of qCUDA with NCVM and qCUDA with CVM in Section III.

F. Pinned Host Memory

By default, memory allocations (C library function malloc()) are pageable, which cannot be accessed directly by GPU. It is essential to move the pageable data to pinned (page-locked) memory for a DMA copy. When a data transfer occurred from pageable host memory to device memory, the CUDA driver must allocate the temporary pinned buffer and copy the host data to the pinned buffer. Therefore, a pageable data allocation could cause a copy twice when a data transfer from the host to the device. To avoid the cost of the data copy from the pageable memory, allocating pinned memory in CUDA runtime APIs by using cudaMallocHost() or cudaHostAlloc() function call, and deallocating pinned memory by using cudaFreeHost() function call.

In qCUDA, it cannot directly use the cudaMallocHost()/ cudaHostAlloc() function call to allocate pinned memory since the pinned memory must be seen in the host. Hence, the qCUlibrary intercepted the pinned memory allocation and provided the host-register function. The central concept of host-register was that qCUDA registered the memory regions, converting from the guest to host, and in the host, the qCUdevice can directly use these pinned host regions for data transfers by using DMA.

III. Experimental Results and Discussion

The host as native testing was conducted on a supermicro 2028GR-TRH server with Xeon E5-2600, 64 GB of memory, four slots of PCI-E 3.0 X16, and Intel I350 GbE controller running Linux OS (Ubuntu 14.04.3). The GPU device was the NVIDIA Tesla K20m GPU with 2496 CUDA streaming processors and 5 GB DDR5 RAM. The hypervisor was based on QEMU 2.4.0, KVM of the Linux kernel for version 3.19.0-25-generic on 64-bit and the virtio framework for para-virtualized network device [20].

In our qCUDA framework, each VM (guest) was configured with two vCPU cores, 4 GB of RAM, 32 GB of qcow2 image format running Ubuntu 14.04.3. We installed the CUDA Toolkit 8.0 on the host. The guest was also introduced the CUDA Toolkit 8.0 except the NVIDIA drivers since we would install the third-party packages such as qCUDA or rCUDA for indirectly forward related commands to the host. The implementation of qCUDA on Linux VM can be entirely transplanted to the Windows VM environment, as long as replacing the corresponding APIs from the Linux system call to Windows. This paper didn't show the results of Windows VM, because to compare the overheads between the APIs of Windows and Linux are not the focus of this paper. The results would be very similar between Windows and Linux VM due to the same mechanism of memory management.

Many API remoting methods proposed in the past studies, such as GvirtuS and vCUDA (mentioned in the Section I and Section V), but most of them lacked source or were not able to operate correctly in our computing environment; only rCUDA can perform on our computing environment, basing on its client-server architecture between VM and host. Therefore, we adopted the rCUDA v15.07 as our subject for comparisons. Although fairness was concerned to compare with rCUDA only focus on the same computing environment, we emphasized that in our local GPU virtualization environment should be a shortcut to reduce virtualization overheads. Moreover, our work also explored how to improve API remoting in local GPU virtualization.

Due to limited space, this paper only showed the results of the benchmarks of data movements which caused the main bottleneck in GPGPU computing. There were the same performance outcomes whatever the guest or host launches the GPU kernel function. Both of them also delivered the GPU binary to the device. In our measurement, we gave two benchmarks of memory copy: bendwidthTest and simpleStreams. Furthermore, we compared qCUDA with rCUDA and native in detail. To avoid obscuring the comparisons among qCUDA, rCUDA, and native, we all used the Ubuntu 14.04.3 as the guest OS of GPGPU virtualization.

A. Bandwidth of Data Transaction

The data movements between the host memory and the device memory of GPU were through PCI-E communication, which was a performance bottleneck of whole GPGPU computing. In native, we measured the memory bandwidth from bandwidthTest; the data transactions from the host to device (H2D) or the device to host (D2H). The average bandwidth by allocating host pageable memory and host pinned memory were 1745.4 MB/sec and 4890.4 MB/sec, respectively, where the range of testing datasets were from 32 KB to 1 GB size of data movements.

Figures 3 and 4 illustrate the experimental results. The former and the latter allocate the pageable memory and pinned memory in the host, respectively. The *KERNEL_COPY* in Figures 3 and 4 represent the data movement without the memory mapping (GVA to GPA) from user space to kernel space in the guest, but copies the data from user space to kernel space, and vice versa. Figures 3(a) and 4(a) show the memory bandwidth results of the Y-axis. Figures 3(b) and 4(b) show the bandwidth efficiency is computed by the formula:

$$\frac{\text{bandwidth of qCUDA}}{\text{bandwidth of native}} \times 100\%$$

of the Y-axis. The X-axis of Figures 3 and 4 is the testing datasets of size per transfer; the line graph and bar chart correspond with the results of Y-axis. The bar chars display the bandwidth with size per transfer, and the line graphs show the percentage of bandwidth efficiency for the ratios to native.

It can be seen that the best performance of GPGPU virtualization is qCUDA with *NCVM*, and the second-best is qCUDA with *CVM*. Since the performance by qCUDA with *KERNEL_COPY* has extra copies between user space and kernel space in the guest, it is weaker than qCUDA with

NCVM/*CVM* in bar chars. Although qCUDA with *CVM* does not have the extra iteratively function calls as qCUDA with *NCVM*, it could take many minor page faults due to the first time that the qCUdevice accessed the shared pages. Also, the minor page faults in qCUDA with *CVM* cause the reason of poor performance than that of qCUDA with *NCVM* in most cases (it could be a tradeoff issue due to the extra iterations and minor page faults. But in most cases, qCUDA with *NCVM* are better than qCUDA with *CVM*).

In Figure 3, the general trend appears to be increased in qCUDA; when the size per transfer achieves 1 GB, the bandwidth efficiency can reach 90.23% in the H2D with *NCVM* and 98.45% in the D2H with *NCVM*. In Figure 4, the bandwidth efficiency has risen above 95% after 128 KB size per transfer with *NCVM* and remained stable. Figures 3 and 4 also show that the performance is weaker during the transactions of smaller size, but the line graphs of qCUDA with *NCVM* in Figure 4 upturn more rapidly than those in Figure 3. Since qCUDA with *NCVM* in pinned memory allocations can asynchronous copy several chunks with 4 MB size for the H2D/D2H, it can enhance the utilization of PCI-E communication bandwidth to cover some overheads of virtualization.

As seen from Figures 3 and 4, the average bandwidth of rCUDA in pageable memory allocations and pinned memory allocations are 61.2 MB/sec and 273.3 MB/sec, respectively. The average bandwidth in qCUDA can achieve 1358.72 MB/sec in pageable memory allocations and 4463.66 MB/sec in pinned memory allocations.

Figure 5 shows the evidence of smaller size per transfer that the time of transaction decreases in bandwidthTest. From Figure 5, the ratios of the overheads of bandwidthTest, including the qCUlibrary call, qCUdriver call, virtio latency and memory copy of H2D/D2H in the qCUdevice with *NCVM*. We can see that the virtio latency is the main reason causing the extra overhead of virtualization in qCUDA with a smaller size per transfer. Therefore, the efficiency is susceptible to the system's initialization or regular overheads of virtualization in qCUDA during the smaller time of data transfers. The bandwidth measurements by rCUDA did not appear to be well as qCUDA in our testing, as reported in [21].

B. Multiple Streams

The CUDA functions from which the host code issued the runtime APIs can be overlapped for performing simultaneously; for instance, the functions included CUDA kernel, cudaMemcpyAsync H2D and cudaMemcpyAsync D2H. The stream denoted a sequence of CUDA functions that execute on the GPU in order. Multiple streams may run concurrently with CUDA functions for overlapping kernel execution and data transfers. We adopted the benchmark simpleStreams to measure the performance of single and multiple streams. It can be anticipated to improve whole performance when data transfers and kernel launch overlapping execution by multiple streams.

TABLE I: Elapsed time of multiple streams.

Array Size, MBs	Non-Streamed			4 Streams		
	Elapsed Time, msec			Elapsed Time, msec		
	Native	rCUDA	qCUDA	Native	rCUDA	qCUDA
64	14.71	626.5	14.88	10.42	597.2	10.46
128	29.41	1223.49	29.69	20.86	1215.93	20.9
192	44.01	1847.18	44.54	31.06	1804.23	31.22
256	58.91	2454.42	59.39	41.64	2388.11	41.61
320	73.32	3060.34	74.22	51.75	3005.96	51.98
384	87.99	3613.95	89.05	62.1	3585.74	62.17
448	102.66	4285.77	103.93	72.35	4224.14	72.42
512	118.12	4883.1	118.63	83.77	4839.5	82.77
576	131.98	5478.32	133.56	93.23	5399.98	93.07
640	146.64	6006.64	148.35	103.73	5963.65	103.38
704	161.29	6436.04	163.26	114.09	6464.54	113.51
768	175.96	7377.23	178.06	124.48	7335.32	123.8
832	190.63	7877.15	192.85	134.86	7895.53	134
896	205.29	8693.09	207.78	144.97	8610.51	144.34
960	219.93	9364.58	222.63	155.46	9263.99	154.65
1024	235.73	9942.98	237.38	167.42	9853.03	164.61

Table I shows the results of simpleStreams. We executed the simpleStreams by the option "–use_cuda_malloc_host"which internally called cudaHostRegister() function. While simpleStreams simultaneously copied the input array to GPU and did some functions for initializing the input array in the kernel function, asynchronous array copied from the GPU to host. As shown in Table I, the elapsed time of qCUDA in all datasets is very close to native. The low latency involving pinned memory in qCUDA takes the credit for better performance than rCUDA in our test environment. In Table I, we find that some results are not only very close to native but are even better than native. For instance, when the array size is greater than 512 MB in 4 streams, qCUDA has a slight gap of elapsed time over native (the differences of elapsed time are about 0.16 to 2.81 ms). The reason for these results could be referred to as the NCVM strategy since the data is split into small chunks and each chunk is transferred asynchronously from the H2D/D2H, in multiple streams, there are opportunities to increase the bandwidth utilization when the data transfer overlapped with kernel launch. The different sizes of chunks and the iteration times for asynchronously CPU-to-GPU data movements to change the overhead is a tradeoff issue [22]. The results show that the default configuration (4 MB chunk size) of qCUDA can achieve robust performance.

IV. Related Work

Prior studies have attempted to adopt the API remoting methods for GPGPU virtualization, which can execute HPC applications with CUDA or OpenCL on the VM, such as GViM, GVirtuS, vCUDA, rCUDA, mrCUDA, and virtio-CL [7]–[12]. Each of these studies achieved the features of GPGPU virtualization within their defined scopes. Most ideas of these studies are similar, instead of GViM [7], others with channels between the back-end and front-end are TCP/IP based communicators. GViM performs on the Xen-based hypervisor, supports CUDA 1.1 runtime APIs on the VM and installs NVIDIA driver in Dom0. The XenStore [23] is used to establish event channels between both domains connecting the front-end and back-end. GViM is similar to our

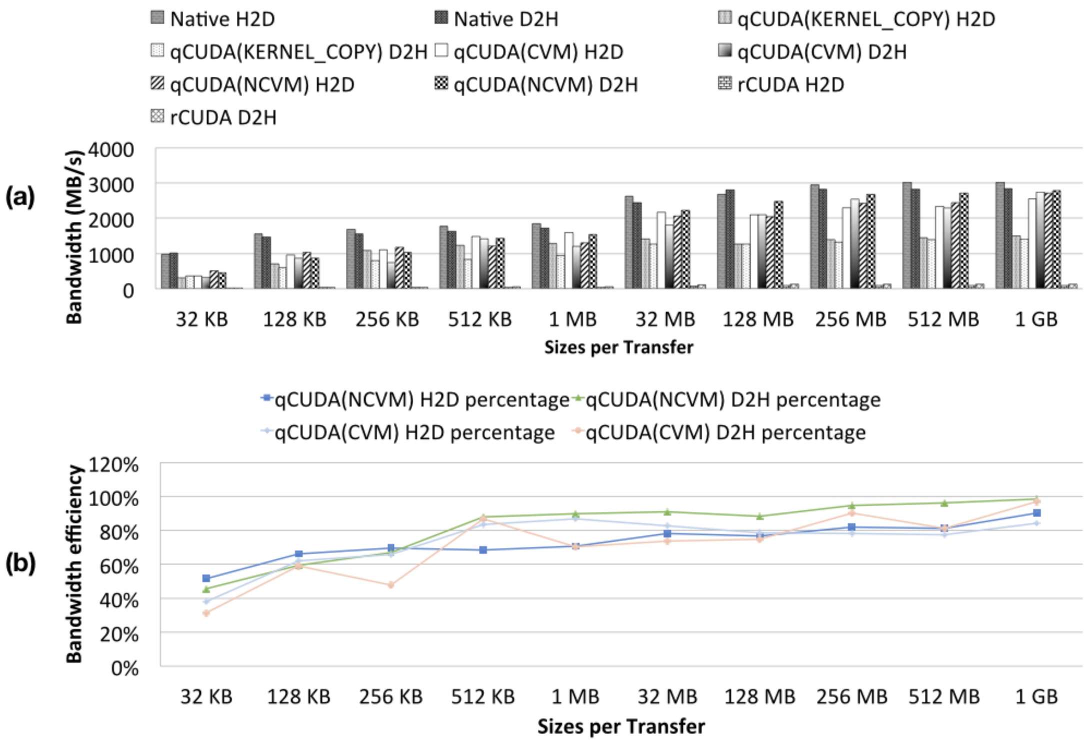

Fig. 3: bandwidthTest using pageable memory.

work that it calls mmap() function replacing malloc() function in a user space of VM and then uses XenStore to share a page directory structure within the front-end and back-end. Although GViM can also remap the page by the back-end for accessing in the host without extra copies, the approach in GViM is similar to the CVM we proposed. According to our memory management, the NCVM we offer for reducing the minor page faults. GVirtuS [8] is based on the QEMU-KVM hypervisor supporting CUDA 3.x runtime APIs in VM and NVIDIA driver in the host. Their communicator in the client-server architecture can be replaced by several subclasses such as TCP/IP, Unix sockets, VMSocket, and VMCI. vCUDA [9] is based on the Xen hypervisor supporting CUDA 3.2 runtime APIs and installs the NVIDIA driver on Dom0. They proposed the VMRPC for transferring data between the client and server via TCP/IP communications.

rCUDA [10], [13]–[17], [24] supports CUDA runtime APIs in the VM, a KVM-/Xen-based hypervisor and the NVIDIA driver in the host. rCUDA exhibits the low latency overhead through the IB for client-server interconnections; it also is the only GPGPU virtualization framework can be performed on newer CUDA runtime APIs and executed in our testing environment. Unfortunately, if the VM and remote GPU in the same heterogeneous computing node with the powerless physical ethernet interface, which the lack of high-speed and hardware virtualization support, could cause a considerable latency by using rCUDA. mrCUDA [11] is a middleware that incorporates the rCUDA framework for migrating the GPGPU resources to the local.

virtio-CL [12] is similar to our work that they leverage the virtio framework [18] to construct the connections between the front-end and back-end; it based on the QEMU-KVM hypervisor, supports the OpenCL runtime APIs in the VM and the NVIDIA driver in the host. However, virtio-CL cannot directly share the memory pages or regions between the VM and host; it could cause the extra copies between them; The source code with openCL in the VM could be modified to suit the virtio-CL. Furthermore, the features of zero-copy of which sharing the same memory region between the VM and hypervisor cannot be supported on virtio-CL. Besides virtio-CL, GVirtuS and vCUDA also seem not to provide the features of zero-copy. addition to these API remoting methods in GPGPU virtualization as mentioned above, DS-CUDA, GridCUDA, VOCL, SnuCL, and dOpenCL [25]–[29] are also API remoting methods via the client-server based on the TCP/IP communication with CUDA/OpenCL runtime APIs. However, these API remoting methods don't emphasize that if it exists the VM handling by the hypervisor.

In addition to the API remoting methods in GPGPU virtualization, the device emulation and mediated pass-through could

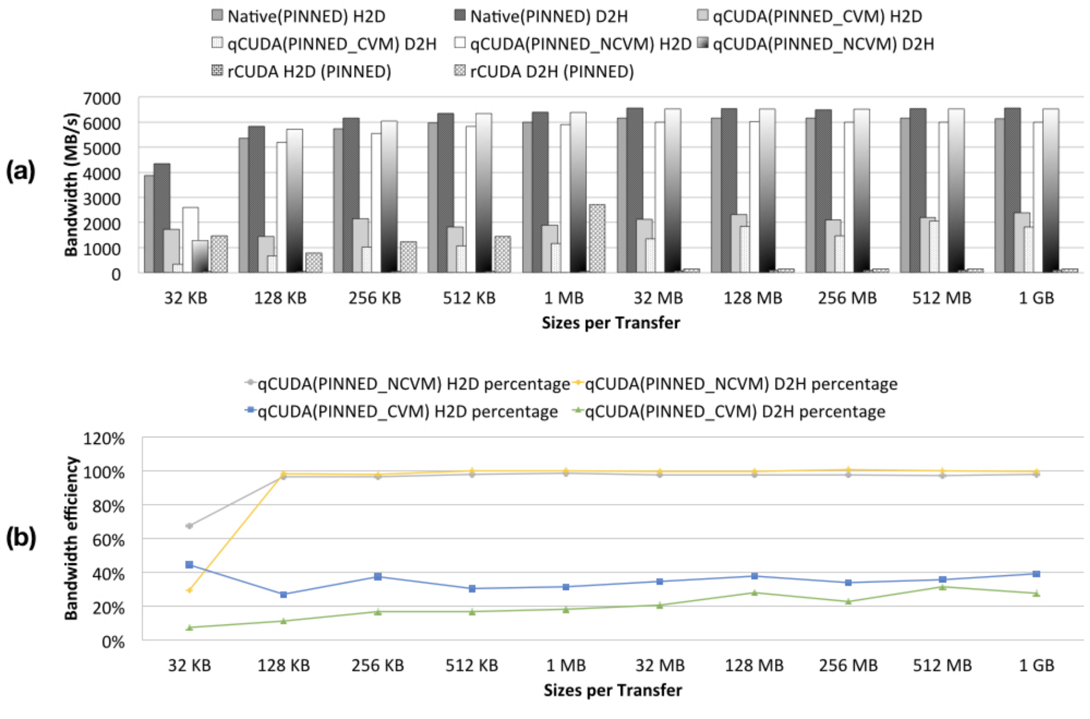

Fig. 4: bandwidthTest using pinned memory.

also provide GPGPU virtualization. GPUvm [30] proposed the concept of the aggregator in the hypervisor involving the emulation of some GPU hardware resources. GPUvm is based on Xen hypervisor [30] running the nouveau module as GPU device driver and using Gdev [31] as the CUDA runtime APIs. The proposed resource management providing with both full- and para-virtualization approaches for GPU virtualization and design a GPU access aggregator for isolating and multiplex multiple VMs accessing the GPU resources, memory areas, PCI-E BARs, and GPU channels by GPU shadow page tables and fair-share scheduler among VMs. However, executing the nouveau driver and emulating with GPU resources could hardly bring full performance in the NVIDIA GPU. G-KVM [32] is a similar framework on KVM for full GPU emulation.

The NVIDIA GRID K1/K2 [3] and Intel GVT-g [5] are the product level for mediated pass-through. Although these commercial packages can divide a GPU into several virtual GPUs and take advantage of full virtualization, it must be executed in some specific GPU hardware and hypervisor. Therefore, their flexibility and interposition are weaker than those of qCUDA. Besides, as the product level solution mentioned above, Windows VM also can be supported on qCUDA.

V. Conclusion and Future Work

In this paper, we proposed a robust API remoting framework, qCUDA, to improve the performance of GPGPU virtualization. qCUDA is based on the virtio framework for para-virtualization on QEMU-KVM hypervisor. qCUDA provides the ways for eliminating the times of data movements and reduces the latency between the VM and GPU by the memory management methods. Comparing to the prior work, it doesn't require any specific network interface, such as InfiniBand or SR-IOV support. Therefore, in the usual heterogeneous architecture, qCUDA offers a low latency solution for data transfer between GPU and VM. In the bandwidthTest, most results compared with native achieved above 95% of the bandwidth efficiency; furthermore, when the data transfer size is greater than 128 KB, the bandwidth efficiency in the average of H2D and D2H achieved 97.47% and 99.51%, respectively.

We have released the beta version of qCUDA (https://github.com/coldfunction/qCUDA) that could promote the research community to study or improve our work to apply more applications. In the future, we will increase the GPGPU coverage in our framework which gives more GPGPU functionality, schedule the computing resource of multiple GPUs, and incorporate a middleware, rapidly migrating the GPGPU resources from the remote to local. Then it can be expected to run more CUDA applications and

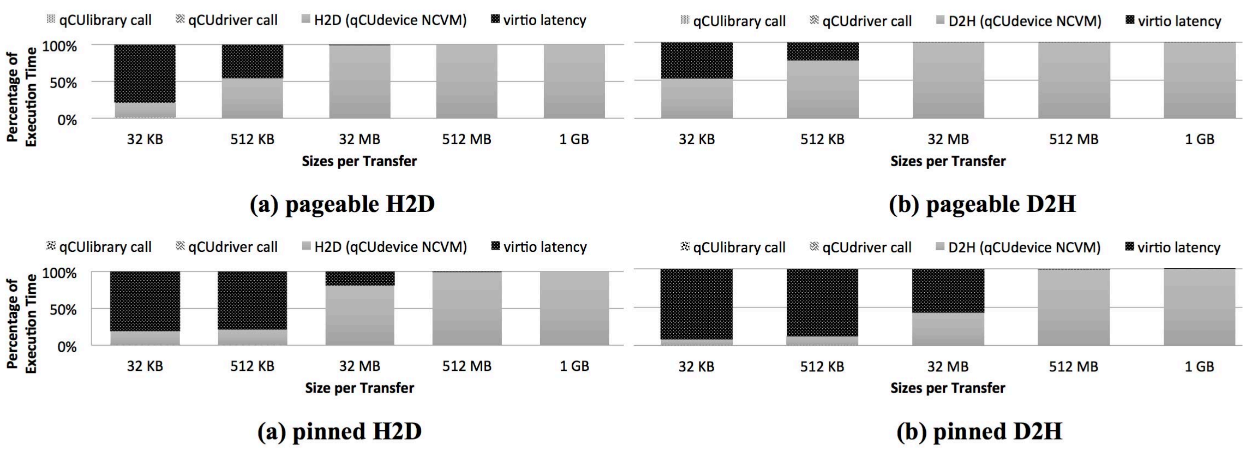

Fig. 5: The overhead ratios of bandwidthTest.

being a more powerful solution of GPGPU virtualization in the heterogeneous cloud environment.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[2] C.-H. Chen, C.-R. Lee, and W. C.-H. Lu, "Smart in-car camera system using mobile cloud computing framework for deep learning," *Vehicular Communications*, vol. 10, pp. 84 – 90, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214209616301814

[3] *Visual Computing Leadership from NVIDIA, Nvidia.com*, Available: http://www.nvidia.com/page/home.html.

[4] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," *In Proc. OASIS'04*, pp. 3–7, 2004.

[5] *Intel Graphics Virtualization Update, Intel Software*, Available: https://software.intel.com/en-us/blogs/2014/05/02/intel-graphics-virtualization-update.

[6] M. Dowty and J. Sugerman, "Gpu virtualization on vmware's hosted i/o architecture," *ACM SIGOPS OSR*, vol. 43, no. 3, p. 73, 2009.

[7] V. Gupta, A. Gavrilovska, K. Schwan, H. Kharche, N. Tolia, V. Talwar, and P. Ranganathan, "Gvim: Gpu-accelerated virtual machines," *In Proc. HPCVirt '09*, pp. 17–24, 2009.

[8] G. Giunta, R. Montella, G. Agrillo, and G. Coviello, "A gpgpu transparent virtualization component for high performance computing clouds," *In Proc. Euro-Par*, pp. 379–391, 2010.

[9] L. Shi, H. Chen, J. Sun, and K. Li, "vcuda: Gpu-accelerated high-performance computing in virtual machines," *IEEE Trans. Comput*, vol. 61, no. 6, pp. 804–816, 2012.

[10] J. Duato, A. Pena, F. Silla, R. Mayo, and E. Quintana, "rcuda: Reducing the number of gpu-based accelerators in high performance clusters," *In Proc. HPCS*, pp. 224–231, 2010.

[11] P. M. A. Nomura and S. Matsuoka, "mrcuda: Low-overhead middleware for transparently migrating cuda execution from remote to local gpus," *presented at the SC15. Conf, Austin, U.S.*, 2015.

[12] T. Tien and Y. You, "Enabling opencl support for gpgpu in kernel-based virtual machine," *Softw. Pract. Exper.*, vol. 44, no. 5, pp. 483–510, 2012.

[13] F. Pérez, C. Reaño, and F. Silla, "Providing cuda acceleration to kvm virtual machines in infiniband clusters with rcuda," *DASI*, pp. 82–95, 2016.

[14] C. Reaño, F. Silla, G. Shainer, and S. Schultz, "Local and remote gpus perform similar with edr 100g infiniband," *In Proc. Middleware Industry'15*, no. 4, 2015.

[15] C. Reaño and F. Silla, "A performance comparison of cuda remote gpu virtualization frameworks," *In Proc. CLUSTER*, pp. 488–489, 2015.

[16] A. Peña, C. Reaño, F. Silla, R. Mayo, and E. Q.-O. J. Duato, "A complete and efficient cuda-sharing solution for hpc clusters," *Parallel Computing*, vol. 40, no. 10, pp. 574–588, 2014.

[17] C. Reaño, R. Mayo, E. S. Quintana-Ortí, F. Silla, J.Duato, and A. J. Peña, "Influence of infiniband fdr on the performance of remote gpu virtualization," *In Proc. CLUSTER*, pp. 1–8, 2013.

[18] R. Russell, "virtio: towards a de-facto standard for virtual i/o devices," *ACM SIGOPS OSR*, vol. 42, no. 5, pp. 95–103, 2008.

[19] L. Youseff, R. Wolski, B. Gorda, and C. Krintz, "Paravirtualization for hpc systems," *In Proc. ISPA'06*, pp. 474–486, 2006.

[20] G. Motika and S. Weiss, "Virtio network paravirtualization driver: Implementation and performance of a de-facto standard," *COMPUT STAND INTER*, vol. 34, no. 1, pp. 36–47, 2012.

[21] J. Duato, A. Pena, F. Silla, J. Fernandez, R. Mayo, and E. Quintana-Orti, "Enabling cuda acceleration within virtual machines using rcuda," *In Proc. HiPC'11*, pp. 1–10, 2011.

[22] S. Che, J. Sheaffer, and K. Skadron, "Dymaxion: optimizing memory access patterns for heterogeneous systems," *In Proc. SC'11*, no. 13, 2011.

[23] *XenStore - Xen, Wiki.xen.org*, Available: https://wiki.xen.org/wiki/XenStore.

[24] C. Reaño, A. J. Peña, F. Silla, R. Mayo, E. S. Quintana-Ortí, and J. Duato, "Cu2rcu: towards the complete rcuda remote gpu virtualization and sharing solution," *In Proc. HiPC*, pp. 1–10, 2012.

[25] M. Oikawa, A. Kawai, K. Nomura, K. Yasuoka, K. Yoshikawa, and T. Narumi, "Ds-cuda: a middleware to use many gpus in the cloud environment," *In Proc. SCC*, pp. 1207–1214, 2012.

[26] T.-Y. Liang and Y.-W. Chang, "Gridcuda: A grid-enabled cuda programming toolkit," *In Proc. WAINA*, pp. 141–146, 2011.

[27] S. Xiao, P. Balaji, Q. Zhu, R. Thakur, S. Coghlan, H. Lin, G. Wen, J. Hong, and W. Feng, "Vocl: an optimized environment for transparent virtualization of graphics processing units," *In Proc. InPar*, pp. 1–12, 2012.

[28] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: an opencl framework for heterogeneous cpu/gpu clusters," *In Proc. ICS'12*, pp. 174–186, 2012.

[29] P. Kegel, M. Steuwer, and S. Gorlatch, "dopencl: towards a uniform programming approach for distributed heterogeneous multi-/many-core systems," *In Proc. IPDPSW*, pp. 341–352, 2012.

[30] Y. Suzuki, S. Kato, H. Yamada, and K. Kono, "Gpuvm: Gpu virtualization at the hypervisor," *IEEE Trans. Comput*, vol. 65, no. 9, pp. 2752–2766, 2016.

[31] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt, "Gdev: First class gpu resource management in the operating system," *In Proc. USENIX ATC'12*, pp. 37–37, 2012.

[32] H. Hsu and C. Lee, "G-KVM: A full GPU virtualization on KVM," in *2016 IEEE International Conference on Computer and Information Technology (CIT)*, Dec 2016, pp. 545–552.