SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **44**:483–510
Published online 22 November 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2166

Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine

Tsan-Rong Tien and Yi-Ping You*,†

Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan

SUMMARY

The importance of heterogeneous multicore programming is increasing, and Open Computing Language (OpenCL) is an open industrial standard for parallel programming that provides a uniform programming model for programmers to write efficient, portable code for heterogeneous computing devices. However, OpenCL is not supported in the system virtualization environments that are often used to improve resource utilization. In this paper, we propose an OpenCL virtualization framework based on Kernel-based Virtual Machine with API remoting to enable multiplexing of multiple guest virtual machines (guest VMs) over the underlying OpenCL resources. The framework comprises three major components: (i) an OpenCL library implementation in guest VMs for packing/unpacking OpenCL requests/responses; (ii) a virtual device, called *virtio-CL*, that is responsible for the communication between guest VMs and the hypervisor (also called the VM monitor); and (iii) a thread, called *CL thread*, that is used for the OpenCL API invocation. Although the overhead of the proposed virtualization framework is directly affected by the amount of data to be transferred between the OpenCL host and devices because of the primitive nature of API remoting, experiments demonstrated that our virtualization framework has a small virtualization overhead (mean of 6.8%) for six common device-intensive OpenCL programs and performs well when the number of guest VMs involved in the system increases. These results indirectly infer that the framework allows for effective resource utilization of OpenCL devices. Copyright © 2012 John Wiley & Sons, Ltd.

Received 10 March 2012; Revised 13 October 2012; Accepted 27 October 2012

KEY WORDS: OpenCL; system virtualization; GPU virtualization; KVM; API remoting

1. INTRODUCTION

Heterogeneous multicore programming is becoming more important because it allows programmers to leverage the computing power of different heterogeneous devices while making optimal use of the specific computation strength of each device so as to improve performance. Some programming models have been proposed to provide a unified layer of heterogeneous multicore programming so that programmers do not need to consider hardware factors, such as differences in the memory organizations and synchronization between host and devices. Two well-known programming models, Compute Unified Device Architecture (CUDA) [1] and Open Computing Language (OpenCL) [2], are both designed as host–device models. CUDA, which was proposed by NVIDIA, makes CPUs the host and graphics processing units (GPUs) the devices. OpenCL was proposed by the Khronos group, and its strong support from the industry has made it the standard for heterogeneous multicore programming. In contrast, OpenCL makes CPUs the host, and it can support various architectures such as CPUs, GPUs, and digital signal processors as devices. These programming models can help programmers to focus on high-level software design and implementation.

*Correspondence to: Yi-Ping You, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan.
†E-mail: ypyou@cs.nctu.edu.tw

Unified heterogeneous programming models simplify the development process for programmers, but resource management issues remain in multiuser systems because a user may not always have immediate access to the available resources of heterogeneous devices. System virtualization represents a good solution to improving resource utilization by providing an environment that supports the simultaneous execution of multiple guest operating systems (OSs) and supporting hardware resource management to optimally share physical resources among OSs. The CPU performance and memory virtualization have improved significantly in the past few years, thanks to the support of hardware-assisted virtualization, but I/O capabilities and performance remain the weak points in system virtualization. Many difficulties are encountered when virtualizing GPU devices because of their architectures both being closed and tending to change markedly with each generation, which limits the ability of guest OSs to utilize GPU resources, especially for general-purpose computing on a GPU (GPGPU). There is currently no standard solution for using CUDA/OpenCL in system virtualization, but there has been little research [3–5] into enabling CUDA support, and their evaluations show that virtualization overheads were reasonable.

We hypothesized that combining the OpenCL programming model with system virtualization would improve the utilization of heterogeneous devices. Enabling OpenCL support in system virtualization provides the benefit of automatic resource management in the hypervisor [also called the virtual machine (VM) monitor], which means that programmers do not have to consider resource utilization issues and also that resources are allocated fairly. Such an approach also has other benefits such as cost reduction and easier migration of executing environments due to the management scheme provided by a VM. Our aim was to build an OpenCL support into a system VM and to describe and test an environment that future studies could use to share hardware resources of heterogeneous devices both fairly and efficiently.

In this paper, we present our methodologies for enabling OpenCL support in a system VM. The ability to run OpenCL programs in a virtualized environment is supported by developing an OpenCL virtualization framework in the hypervisor and building a VM-specific OpenCL run-time system. We present *virtio-CL*, an OpenCL virtualization implementation based on the virtio framework [6] in Kernel-based Virtual Machine (KVM) [7], and evaluate the semantic correctness and effectiveness of our approach by comparing it with native execution environments. To our best knowledge, this study is the first to provide support for running OpenCL programs in a system-virtualized environment and to use the virtio framework for GPU virtualization.

The remainder of this paper is organized as follows. Section 2 introduces the basics of OpenCL and system virtualization, and Section 3 introduces the system design and implementation of OpenCL support in KVM. A performance evaluation is presented in Section 4. Section 5 introduces related work, and conclusions are drawn in Section 6.

2. BACKGROUND

This section provides the basic information needed to understand the remainder of the paper, including the fundamentals of OpenCL, system virtualization, GPU virtualization, and the framework underlying this study (the KVM). For I/O virtualization, we focus on current mechanisms used to virtualize GPU functionality.

2.1. Introduction to OpenCL

OpenCL is an open industry standard for the general-purpose parallel programming of heterogeneous systems. OpenCL is a framework that includes a language, API, libraries, and a run-time system to provide a unified programming environment for software developers to leverage the computing power of heterogeneous processing devices such as CPUs, GPUs, digital signal processors, and Cell Broadband Engine processors. Programmers can use OpenCL to write portable code efficiently, with the hardware-related details being dealt with automatically by the OpenCL run-time environment. The Khronos group proposed the OpenCL 1.0 specification in December 2008, whereas the current version, OpenCL 1.2 [8], was announced in November 2011.

DOI: 10.1002/spe

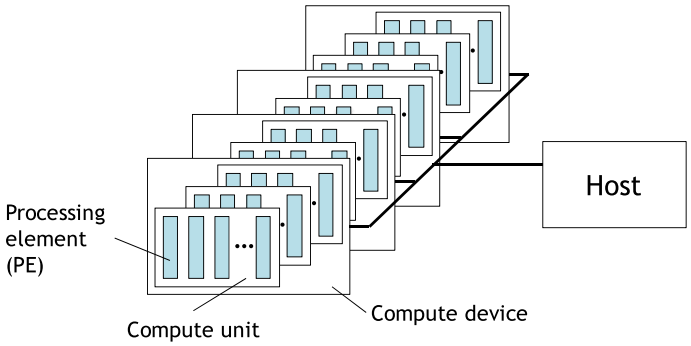

Figure 1. OpenCL platform model (adapted from [8]).

The architecture of OpenCL is divided into a hierarchy of four models: platform, execution, memory, and programming models. In this section, we briefly introduce the definition and relation between each model in the hierarchy. Detailed information about OpenCL can be obtained from the OpenCL Specification [8].

2.1.1. Platform model. Figure 1 defines the platform model of OpenCL, which includes a host connected to one or more OpenCL devices. An OpenCL device consists of one or more compute units, which further comprise one or more processing elements (PEs). The PEs are the smallest unit of computation on a device.

An OpenCL application is designed using a host–device model. The application dispatches jobs (the workloads that will be processed by devices) from the host to the devices, and the jobs are executed by PEs within the devices. The computation result will be transferred back to the host after the execution has completed. The PEs within a compute unit execute a single stream of instructions in a single-instruction multiple-data (SIMD) or single-program multiple-data (SPMD) manner, which is discussed in Section 2.1.4.

2.1.2. Execution model. An OpenCL program is executed in two parts: (i) host part executes on the host and (ii) the kernels execute on one or more OpenCL devices. The host defines a context for the execution of kernels and creates command queues to perform the execution. When the host assigns a kernel to a specific device, an index space is defined to help the kernel locate the resources of the device. We introduce these terms in the following paragraphs.

Context. The host defines a context for the execution of kernels that includes resources such as devices, kernels, program objects, and memory objects. Devices are the collection of objects of OpenCL devices that are capable of running data-parallel and task-parallel works. Kernels are the OpenCL functions that run on OpenCL devices. Program objects are the source codes or executables of kernel programs. Memory objects are visible to both host and OpenCL devices. Data manipulation by host and OpenCL devices is performed by memory objects. The context is created and manipulated using functions from the OpenCL API by the host.

Command queue. The host creates command queues to execute the kernels. Types of commands include kernel execution commands, memory commands, and synchronization commands. The host inserts commands into the command queue, which is then scheduled by the OpenCL run-time system. Commands are performed in one of two modes: in order or out of order. It is possible to define multiple command queues within a single context. These queues can execute commands concurrently, so programmers should use synchronization commands to ensure the correctness of the concurrent execution of multiple kernels.

DOI: 10.1002/spe

2.1.3. Memory model. There are four distinct memory regions: global, constant, local, and private memory. Global memory can be used by all work items (also called threads), constant memory is a region of global memory that is not changed during kernel execution, local memory can be shared by all work items in a work group (also called a thread block), and private memory can only be accessed and changed by a single work item. The host uses OpenCL APIs to create memory objects in global memory and to enqueue memory commands for manipulating these memory objects. Data are transferred between the host and devices by explicitly copying the data or by mapping and unmapping regions of a memory object. The relationships between memory regions and the platform model are described in Figure 2. OpenCL uses a relaxed consistency memory model, in that memory is not guaranteed to be consistent between different work groups. However, the consistency of memory objects that are shared between enqueued commands is ensured at a synchronization point.

2.1.4. Programming model. The OpenCL execution model supports data-parallel and task-parallel programming models. Data-parallel programming refers to the application of a sequence of instructions to multiple data elements. The index space defined in the OpenCL execution model is used to indicate a work item where data can be fetched for the computation. Programmers can define the total number of work items along with the number of work items to form a work group or only the total number of work items to specify how a unique work item will access data.

The OpenCL task-parallel programming model defines that a single instance of a kernel is executed independently of any index space. Users can exploit parallelism via the following three methods: (i) using vector data types implemented by the device; (ii) enqueuing multiple tasks; or (iii) enqueuing native kernels developed by a programming model that is orthogonal to OpenCL.

The synchronization occurs in OpenCL in two situations: (i) for work items in a single work group, a work group barrier is useful for ensuring the consistency, whereas (ii) for commands in the same context but enqueued in different command queues, programmers can use command-queue barriers and/or events to perform synchronization.

2.2. System virtual machine

System virtualization supports the simultaneous execution of multiple OSs on a single hardware platform, with the hardware resources being shared among these OSs. System VMs provide the benefits such as work isolation, server consolidation, safe debugging, and dynamic load balancing. The evolution of multicore CPUs has increased the usefulness of system virtualization.

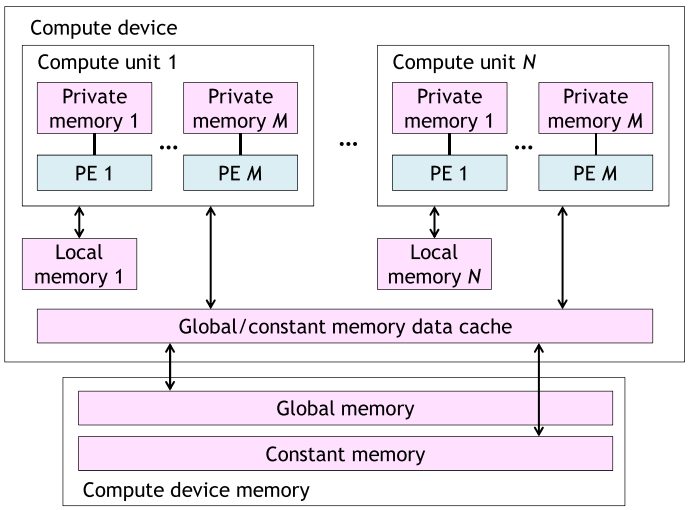

Figure 2. Conceptual OpenCL device architecture with processing elements (PEs), compute units, and compute devices (adapted from [8]).

DOI: 10.1002/spe

The hypervisor supports multiple OSs (called guest OSs) in a system VM running on a single hardware device by managing and allocating the underlying hardware resources among guest OSs, while ensuring that each guest OS does not affect the other guest OSs. The resources of the system VM are time shared in a similar manner to the time-sharing mechanisms of the OS. When the control switches from one guest to another guest, the hypervisor has to save the system state of the current guest and restore the system state of the incoming guest. The guest system state comprises the program counter, general-purpose registers, control registers, and other structures. Here, we focus on system virtualization on the Intel x86 (IA-32) architecture.

2.3. GPU virtualization

Supporting the functionality of OpenCL in VM environments requires GPUs to be virtualized, which has unique challenges because (i) they are extremely complicated devices, (ii) the hardware specification of GPUs is closed, and (iii) GPU architectures tend to change rapidly and dramatically across generations. These challenges make it nearly intractable to virtualize a GPU corresponding to a real modern design. Generally, three main approaches are used to virtualize GPUs: software emulation, API remoting, and I/O pass-through.

2.3.1. Software emulation. One way to virtualize a GPU is to emulate its functionality and to provide a virtual device and driver for guest OSs, which are used as the interface between guest OSs and the hypervisor. The architecture of the virtual GPU could remain unchanged, and the hypervisor would synthesize operations of the host graphics in response to requests from virtual GPUs. VMware has proposed VMware SVGA II as a paravirtualized solution for emulating a GPU on hosted I/O architectures [9]. VMware SVGA II defines a common graphics stack and provides 2D and 3D rendering with the support of OpenGL.

Because OpenCL supports different kinds of heterogeneous devices, it is extremely complicated to emulate such devices with different hardware architectures and provide a unified architecture for OpenCL virtualization. Thus, software emulation was considered to be inappropriate for the present study.

2.3.2. API remoting. Graphic APIs such as OpenGL and Direct 3D with heterogeneous programming APIs such as CUDA and OpenCL are standard, common interfaces. It is appropriate to target the virtualization layer at the API level, whereby the API call requests from guest VMs are forwarded to the hypervisor, which performs the actual invocation. Such a mechanism acts like a guest VM invoking a remote procedure call (RPC) to the hypervisor.

2.3.3. Input/output pass-through. Input/output pass-through involves assigning a GPU to a dedicated guest VM so that the guest VM can access the GPU in the same way as in the native environment. The hypervisor has to handle the address mapping of memory-mapped I/O (MMIO) or port-mapped I/O (PMIO) between virtual and physical devices, which can be achieved by either software mapping or hardware-assisted mechanisms such as Intel virtualization technology for directed I/O (Intel VT-d) [10]. The use of the hardware support can greatly improve the performance of I/O pass-through.

2.3.4. Comparing API remoting and input/output pass-through. A technical report by VMware [9] describes four primary criteria for assessing GPU virtualization approaches: performance, fidelity, multiplexing, and interposition. Fidelity relates to the consistency between virtualized and native environments, and multiplexing and interposition refer to the abilities to share GPU resources. Table I compares API remoting and I/O pass-through on the basis of these four criteria. I/O pass-through exhibits better performance and fidelity than API remoting because it allows direct access to the physical GPU and thus also its device driver and run-time library that are used in the native environment. In contrast, API remoting requires modified versions of the device driver and run-time library for transferring the API call requests/responses, and the data size of API calls is the key virtualization overhead. Although I/O pass-through is considered superior on the basis of the first two

DOI: 10.1002/spe

Table I. Comparing API remoting and input/output pass-through based on the four criteria.

	API remoting	Input/output pass-through
Performance	Δ	∨
Fidelity	Δ	∨
Multiplexing	∨	×
Interposition	∨	×

∨: best supported ; Δ: supported ; ×: not supported.

criteria, its fatal weakness is that it cannot share GPU resources across guest VMs. Resource sharing is an essential characteristic of system virtualization, and API remoting can share GPU resources on the basis of the concurrent abilities at the API level.

In this study, we chose API remoting for our OpenCL virtualization solution on the basis of the comparison in Table I because this technique not only provides OpenCL support in a VM environment but ensures resource sharing across guest VMs. The design and implementation issues are discussed in Section 3.

2.4. Kernel-based Virtual Machine

Kernel-based Virtual Machine is a full virtualization framework for Linux on the x86 platform combined with hardware-assisted virtualization. The key concept of KVM is turning Linux into a hypervisor by adding the KVM kernel module. The comprehensive functionality in a system VM can be adapted from the Linux kernel, such as the scheduler, memory management, and I/O subsystems. KVM leverages hardware-assisted virtualization to ensure a pure trap-and-emulation scheme of system virtualization on the x86 architecture, not only allowing the execution of unmodified OSs but also improving the performance in virtualizing CPUs and the memory management unit. The KVM kernel component has been included in the mainline Linux kernel since version 2.6.20 and has become the main virtualization solution in the Linux kernel.

2.4.1. Basic concepts of Kernel-based Virtual Machine. Kernel-based Virtual Machine is divided into two components: (i) a KVM kernel module that provides an abstract interface (located at /dev/kvm) as an entry point for accessing the functionality of Intel VT-x [11] or AMD-V [12] and (ii) the hypervisor process that executes a guest OS and emulates I/O actions with QEMU (Quick EMUlator) [13]. The hypervisor process is regarded as a normal process for the viewpoint of the host Linux kernel. An overview of KVM is provided in Figure 3.

Process model. The KVM process model is illustrated in Figure 4. In KVM, a guest VM is executed within the hypervisor process that provides the necessary resource virtualizations for a guest OS, such as CPUs, memory, and device modules. The hypervisor process contains N threads ($N \geq 1$) for virtualizing CPUs (vCPU threads), with a dedicated I/O thread for emulating asynchronous I/O actions. The physical memory space of a guest OS is a part of the virtual memory space in the hypervisor process.

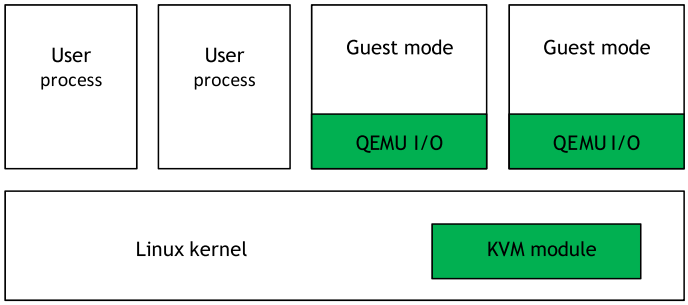

Figure 3. Kernel-based Virtual Machine (KVM) overview.

DOI: 10.1002/spe

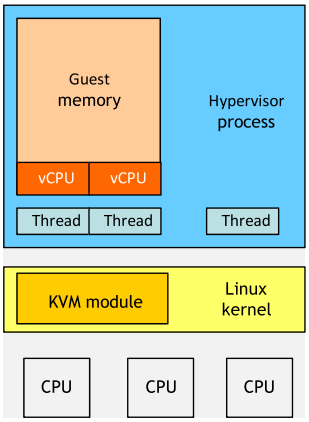

Figure 4. Kernel-based Virtual Machine (KVM) process model (adapted from [14]).

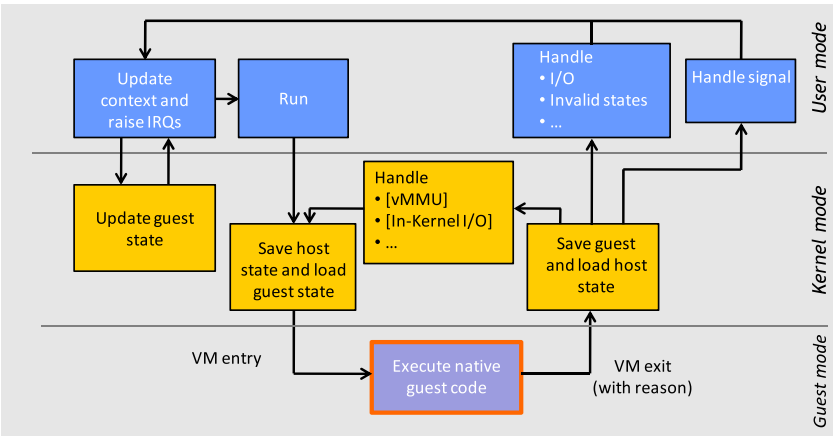

Figure 5. Kernel-based Virtual Machine (KVM) execution flow from the viewpoint of the vCPU (adapted from [14]).

Execution flow from the viewpoint of the vCPU. As illustrated in Figure 5, the execution flow of vCPU threads is divided into three execution modes: guest, kernel, and user modes. In Intel VT-x, the guest mode is mapped onto VMX nonroot mode, and both kernel and user modes are mapped onto VMX root mode.

A vCPU thread in guest mode can execute guest instructions as in the native environment except when it encounters a privileged instruction. When the vCPU thread executes a privileged instruction, the control will transfer to the KVM kernel module. This module first maintains the VM control structure of the guest VM and then decides how to handle the instruction. Only a small proportion of the actions is processed by the kernel module, including virtual memory management unit and in-kernel I/O emulation. In other cases, the control will further transfer to user mode, in which the vCPU thread performs the corresponding I/O emulation or signal handling by QEMU. After the emulated operation has completed, the context held by the user space is updated, and then the vCPU thread switches back to guest mode.

The transfer of control from guest mode to kernel mode is called a lightweight VM exit, and that from guest mode to user mode is called a heavyweight VM exit. The performance of I/O emulation is strongly influenced by the number of heavyweight VM exits because they cost much more than lightweight VM exits.

Input/output virtualization. Each virtual I/O device in KVM has a set of virtual components, such as I/O ports, MMIO spaces, and device memory. Emulating I/O actions requires access to such

DOI: 10.1002/spe

virtual components to be maintained. Each virtual device will register its I/O ports and an MMIO space handler routine when the device starts. When PMIO or MMIO actions are executed in the guest OS, the vCPU thread will trap from the guest mode to the user mode and look up the record of the allocation of I/O ports or MMIO spaces to choose the corresponding I/O emulation routines. Asynchronous I/O actions such as the arrival of response network packets or the occurrence of keyboard signals should be processed using the I/O thread. The I/O thread is blocked while waiting for new incoming I/O events and handles them by sending virtual interrupts to the target guest OS, or emulates direct memory access between virtual devices and the main memory space of the guest OS.

2.4.2. Virtio framework. The virtio framework [6] is an abstraction layer over paravirtualized I/O devices in a hypervisor. Virtio was developed by Russell to support his own hypervisor called *lguest*, and it was adopted by KVM for its I/O paravirtualization solution. Virtio makes it easy to implement new paravirtualized devices by extending the common abstraction layer. The virtio framework has been included in the Linux kernel since version 2.6.30.

Virtio conceptually abstracts an I/O device into front-end drivers, back-end drivers, and one or more 'virtqueues', as shown in Figure 6. Front-end drivers are implemented as device drivers of virtual I/O devices and use virtqueues to communicate with the hypervisor. Virtqueues can be regarded as memory spaces that are shared between guest OSs and the hypervisor. There is a set of functions for operating virtqueues, including adding/retrieving data to/from a virtqueue (`add_buf/get_buf`), generating a trap to switch the control to the back-end driver (`kick`), and enabling/disabling call-back functions (`enable_cb/disable_cb`), which are the interrupt handling routines of the virtio device. The back-end driver in the hypervisor retrieves the data from virtqueues and then performs the corresponding I/O emulation on the basis of the data from guest OSs.

The high-level architecture of virtio in the Linux kernel is illustrated in Figure 7. The virtqueue and the data-transfer function are implemented in `virtio.c` and `virtio_ring.c`, and there is a series of virtio devices such as `virtio-blk`, `virtio-net`, and `virtio-pci`. The object hierarchy of the virtio front-end is shown in Figure 8, which illustrates the fields and methods of each virtio object and their interrelationships. A virtqueue object contains the description of available operations, a pointer to the call-back function, and a pointer to a *virtio_device* that owns this virtqueue. A *virtio_device* object contains the fields used to describe the features and a pointer to a *virtio_config_ops* object, which is used to describe the operations that configure the device. In the device initialization phase, the virtio driver would invoke the `probe` function to set up a new instance of *virtio_device*.

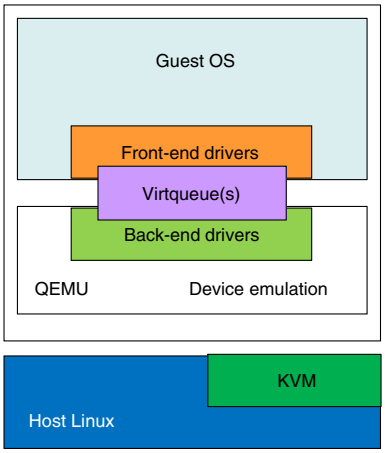

Figure 6. Virtio architecture in Kernel-based Virtual Machine (KVM).

DOI: 10.1002/spe

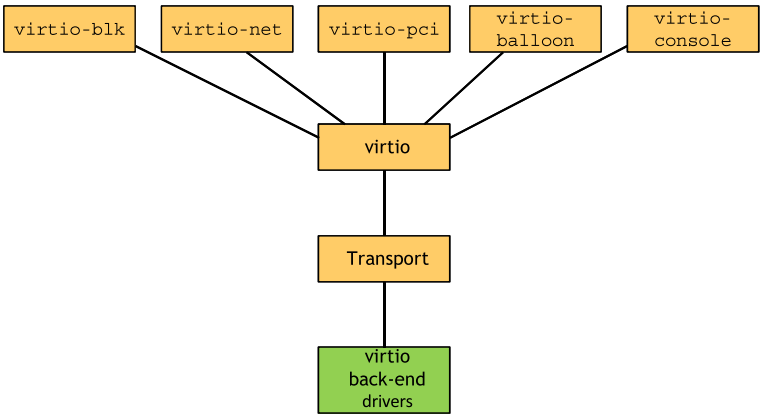

Figure 7. High-level architecture of virtio (adapted from [15]).

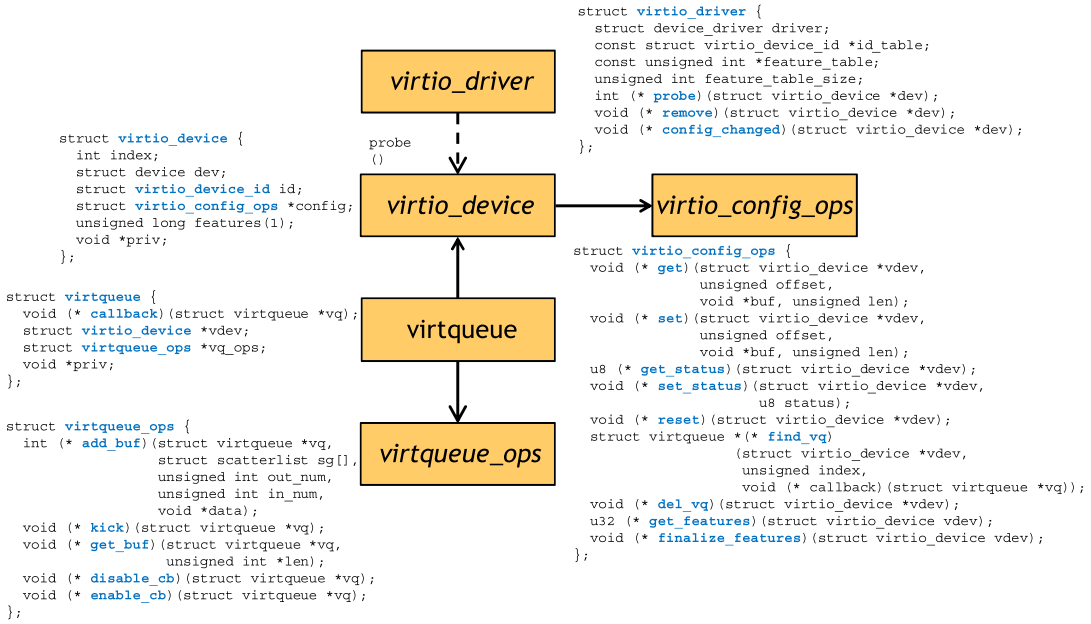

Figure 8. Object hierarchy of the virtio front-end (adapted from [15]).

In this study, we implemented our API remoting mechanism based on the virtio framework to perform the data communication for OpenCL virtualization. The design and implementation details are discussed in Section 3.

3. SYSTEM DESIGN AND IMPLEMENTATION

This section describes the software architecture and provides details of the design for enabling OpenCL support in KVM.

3.1. OpenCL API remoting in Kernel-based Virtual Machine

We first describe the framework of OpenCL API remoting in KVM, including the software architecture, the execution flow, and the relationships among the various software components.

3.1.1. Software architecture. Figure 9 presents the architecture of OpenCL API remoting proposed in this study, which includes an OpenCL library specific to guest OSs, a virtual device, called

DOI: 10.1002/spe

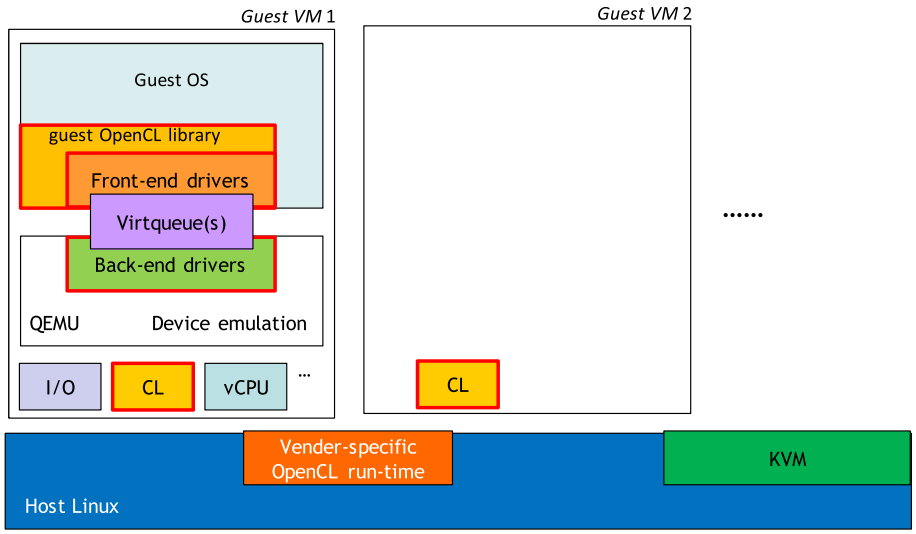

Figure 9. Architecture of OpenCL API remoting in Kernel-based Virtual Machine (KVM).

virtio-CL, and the CL thread. Multiple guest VMs in the virtualization framework can be modeled as multiple processes accessing the OpenCL resources in the native environment. The execution behaviors of these processes depend on the implementation of the vendor-specific OpenCL run-time system. The functions of each component (with bold, red borders) are described as follows:

Guest OpenCL library. The guest OpenCL library is responsible for packing OpenCL requests of user applications from the guest and unpacking the results from the hypervisor. In our current implementation, the guest OpenCL library is designed as a wrapper library and performs basic verifications according to the OpenCL specifications, such as null pointer or integer value range checking.

Virtio-CL device. The virtio-CL virtual device is responsible for data communication between the guest OS and the hypervisor. The main components of virtio-CL are two virtqueues for data transfer from the guest OS to the hypervisor and vice versa. The virtio-CL device can be further divided into the *front-end* (residing in the guest OS) and the *back-end* (residing in the hypervisor). The guest OS accesses the virtio-CL device by the front-end driver and writes/reads OpenCL API requests/responses via virtqueues using the corresponding driver calls. The virtio-CL back-end driver accepts the requests from the guest OS and passes them to the CL thread. Actual OpenCL API calls are invoked by the CL thread. The virtqueues can be modeled as device memory of virtio-CL from the viewpoint of the guest OS.

CL thread. The CL thread accesses vendor-specific OpenCL run-time systems in user mode. It reconstructs the requests, performs the actual invocation of OpenCL API calls, and then passes the results back to the guest OS via the virtqueue used for response transfer. Because the processing time differs between each OpenCL API call, it is better to use an individual thread to handle OpenCL requests than to extend the functionality of existing I/O threads in the hypervisor process because this will mean that the OpenCL APIs are independent of the functionality of the I/O thread.

An alternative approach to creating an CL thread to process OpenCL requests is to implement the actual invocation of OpenCL API calls in the handler of the vCPU thread and configure multiple vCPUs for each guest VM. However, both of these approaches require a buffer ($CLRequestQueue$, as shown in Figure 10) to store segmented OpenCL requests in case an OpenCL request is larger than the virtqueue for requests (VQ_REQ in Figure 10). In addition,

DOI: 10.1002/spe

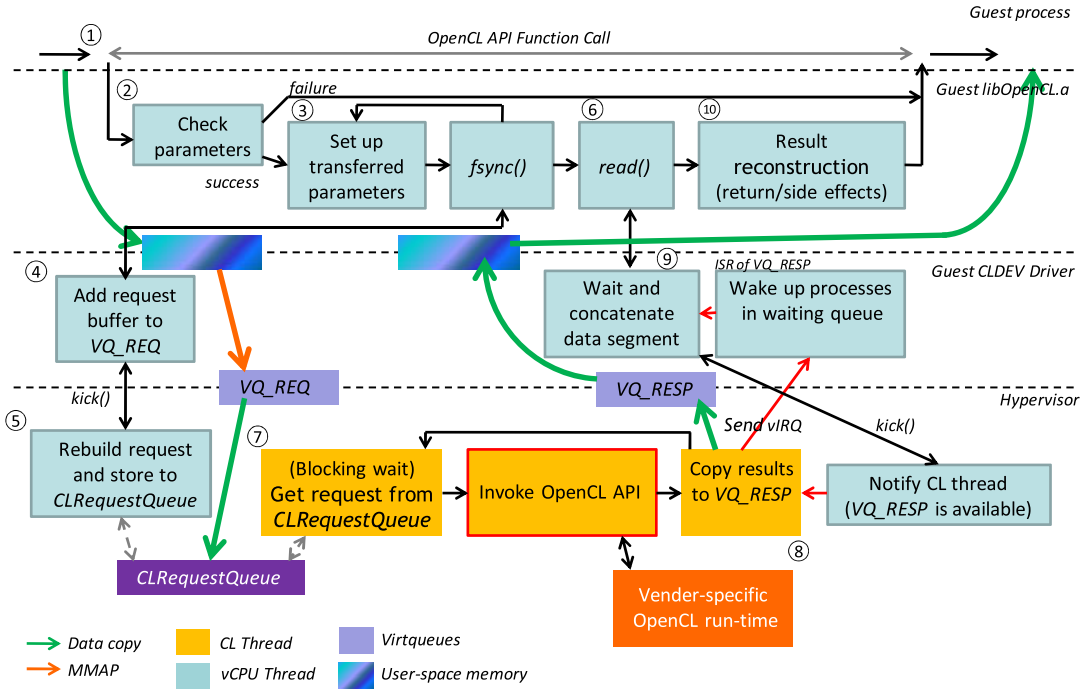

Figure 10. Execution flow of OpenCL API remoting. The circled numbers are explained in the text.

the latter approach has to handle the synchronization of the virtqueue for the response (VQ_RESP in Figure 10) between different vCPU threads, whereas the former approach handles the synchronization of $CLRequestQueue$ between the vCPU thread and the CL thread.

3.1.2. Execution flow. Figure 10 illustrates the execution flow of OpenCL API calls, which involves the following processing steps:

1. A process running in a guest OS invokes an OpenCL API function.
2. The guest OpenCL library (`libOpenCL.a`) first performs basic verifications of the parameters according to the OpenCL API specifications. If the verifications fail, it returns the corresponding error code to the user-space process.
3. After parameter verification, the guest OpenCL library sets up the data that need to be transferred and executes an `fsync()` system call.
4. The `fsync()` system call adds the data to the virtqueue for requests (VQ_REQ) and then invokes the `kick()` method of VQ_REQ to generate a VM exit. The control of the current vCPU thread is transferred to a corresponding handler routine of VM exit in user mode.
5. In user mode, the handler routine copies the OpenCL API request to *CLRequestQueue*, which is a memory space that is shared between vCPU threads and the CL thread. After the copy process completes, control is transferred back to the guest OpenCL library. If the data size of the request is larger than the size of VQ_REQ, the request is divided into segments, and the execution jumps to step 3 and repeats steps 3–5 until all the segments are transferred. If all of the data segments of the OpenCL request are transferred, the handler will signal to the CL thread that there is an incoming OpenCL API request, and the CL thread starts processing (see step 7).
6. After the request is passed to the hypervisor, the guest OpenCL library invokes a `read()` system call, which waits for the result data and returns after all of the result data have been transferred.
7. The CL thread waits on a blocking queue until receiving the signal from the handler routine of VM exit in step 5. The CL thread then unpacks the request and performs the actual invocation.

DOI: 10.1002/spe

8. After the OpenCL API invocation has completed, the CL thread packs the result data, copies them to the virtqueue for responses (VQ_RESP), and then notifies the guest OS by sending a *virtual interrupt* from the virtio-CL device.
9. Once the guest OS receives the virtual interrupt from the virtio-CL device, the corresponding interrupt service routine wakes up the process waiting for response data of the OpenCL API call, and the result data are copied from VQ_RESP to the user-space memory. Repeat steps 8 and 9 until all of the result data are transferred.
10. Once the `read()` system call returns, the guest OpenCL library can unpack and rebuild the return value and/or side effects of the parameters. The execution of the OpenCL API function has now been completed.

3.1.3. Implementation details. This section presents the implementation details of the proposed virtualization framework, including the guest OpenCL library, device driver, data transfer, and synchronization points.

Guest OpenCL library and device driver. The device driver of virtio-CL implements `open()`, `close()`, `mmap()`, `fsync()`, and `read()` system calls. `mmap()` and `fsync()` are used for transferring OpenCL requests, and the `read()` system call is used for retrieving the response data. The guest OpenCL library uses these system calls to communicate with the hypervisor.

Before a user process can start using OpenCL resources, it has to explicitly invoke a specific function, `clEnvInit()`, to perform the steps involved in resource initialization such as opening the virtio-CL device and preparing the memory storages for data transfer. An additional function, `clEnvExit()`, has to be invoked to release the OpenCL resources before the process finishes executing. Therefore, minor changes are required so that OpenCL programs can be executed on the virtualized platform: adding `clEnvInit()` and `clEnvExit()` at the beginning and the end of an OpenCL program, respectively. However, these changes can be made automatically by a simple OpenCL parser. An alternative approach that avoids such changes can be used, but it involves a complex virtualization architecture and greatly increases the overhead. The alternative implementation is discussed further in Section 3.2.5.

Synchronization points. There are two synchronization points in the execution flow: (i) in the CL thread that waits for a completed OpenCL request and (ii) in the implementation of the `read()` system call that waits for the result data processed by the CL thread. The first synchronization point is handled by the use of pthread mutexes and condition variables. When the data segments of an OpenCL request are not completely transferred to *CLRequestQueue*, the CL thread invokes `pthread_cond_wait()` to wait for the signal from the handler routine of VQ_REQ. The pseudocodes of the first synchronization point are presented in Figures 11 and 12. The second synchronization point works as follows: the `read()` system call is first placed in the waiting queue in the kernel space to be blocked. The blocked process will resume executing and will retrieve the response data segment from VQ_RESP after the virtual interrupt of virtio-CL is triggered. The pseudocodes of the second synchronous point are presented in Figure 13.

```
pthread_mutex_t clReqMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  clReqCond  = PTHREAD_COND_INITIALIZER;
...

pthread_mutex_lock( &clReqMutex );
if( /* OpenCL request is not ready */ ) {
  pthread_cond_wait( &clReqCond, &clReqMutex );
}

/* pop a request node from CLRequestQueue */
...
```

Figure 11. The first synchronization point (CL thread).

DOI: 10.1002/spe

```
pthread_mutex_lock( &clReqMutex );

/* Copy the request data segment to CLRequestQueue */

if( /* OpenCL request is ready */ ) {
  pthread_cond_signal( &clReqCond );
}

pthread_mutex_unlock( &clReqMutex );
...
```

Figure 12. The first synchronization point (virtual machine exit handler).

```
ssize_t cldev_read( struct file *filp, char __user *buf,
    size_t count, loff_t *f_pos ) {
  do {
    /* put the current process into the waiting queue */
    schedule();

    /* the process will be blocked until the response data
        segment is ready */

    copy_to_user( buf, kernel_buffer, size_data_seg );

    /* notify the hypervisor that VQ_RESP is available */

  } while( /* transfer is not completed */ )
}
```

Figure 13. The second synchronization point (`read()` system call).

The two synchronization mechanisms not only ensure data correctness among the data-transfer processes but also allow the vCPU resource to be used by other processes in the guest OS.

Data transfer. The green and orange arrows in Figure 10 represent the paths of data transfer. The green arrows indicate data copying among the guest user space, the guest kernel space, and the host user space. Because the data transfer of OpenCL requests is processed actively by the guest OpenCL library, data copying from the guest user space to the guest kernel space can be bypassed by preparing a memory-mapped space (indicated by the orange arrow). The data copying in the opposite direction cannot be eliminated because the guest OpenCL library is passively waiting for results from the hypervisor. Thus, there are two data-copy processes for a request and three data-copy processes for a response.

3.2. Implementation issues

This section discusses the issues related to supporting implementation of OpenCL in KVM, including the size of virtqueues and the data coherence between guest OSs and the hypervisor.

3.2.1. Size of virtqueues. A series of address translation mechanisms is used to allow both the guest OS and the QEMU part to access the data from virtqueues. On one hand, the size of virtqueues directly affects the virtualization overhead because larger virtqueues result in fewer heavyweight VM exits; on the other hand, because the total virtual memory space of the hypervisor process is limited (4 GB in 32-bit host Linux), the size of virtqueue is also limited.

In our framework, both VQ_REQ and VQ_RESP have a size of 256 kB (64 pages) according to the configurations of the existing Virtio devices: `virtio-blk` and `virtio-net`. `virtio-blk` has one virtqueue. The size of the virtqueue is 512 kB (128 pages), and `virtio-net` has two virtqueues, each of which is 1024 kB (256 pages). A request (or a response) will be partitioned into multiple 256-kB segments and then transferred sequentially if the data size of the request (or the response) exceeds the size of virtqueue.

DOI: 10.1002/spe

3.2.2. Signal handling. A user OpenCL process may experience a segmentation fault when calling OpenCL APIs. In the native environment, this would cause the process execution to terminate, but in our virtualization framework, this situation needs to be handled carefully by the CL thread to ensure that the hypervisor process does not crash. The CL thread has to build handler routines for signals such as `SIGSEGV` to recover the thread from such a signal and return the corresponding error messages to the guest OpenCL library.

3.2.3. Data structures related to the run-time implementation. Figure 14 lists the data structures related to the run-time implementation in OpenCL. These data structures are maintained by the vendor-specific OpenCL run-time system, and the users can only access them via the corresponding OpenCL functions. For example, when a process invokes `clGetDeviceIDs()` with a pointer to an array of `cl_device_id` as a parameter, the function fills each entry of the array with a pointer to the object of an OpenCL device. In our framework, the CL thread is used to access the actual OpenCL run-time system in user mode and consume the parameters provided by the guest process. However, the CL thread cannot access the array directly because it is in a virtual address space of the guest process. Thus, we have to construct a 'shadow mapping' of the guest array, whereby the CL thread allocates a 'shadow array' and maintains a mapping table between the array in the guest and the shadow array. Figure 15 illustrates an example of the shadow-mapping mechanism. When a guest process invokes `clGetDeviceIDs()`, the CL thread allocates a shadow array and creates a new entry in the mapping table that records the pointer type, the process identifier of the guest process, and the mapping between the host address and the guest address. The CL thread

```
/* /usr/local/cuda/include/CL/cl.h (NVIDIA OpenCL SDK) */

typedef struct _cl_platform_id *    cl_platform_id;
typedef struct _cl_device_id *      cl_device_id;
typedef struct _cl_context *        cl_context;
typedef struct _cl_command_queue *  cl_command_queue;
typedef struct _cl_mem *            cl_mem;
typedef struct _cl_program *        cl_program;
typedef struct _cl_kernel *         cl_kernel;
typedef struct _cl_event *          cl_event;
typedef struct _cl_sampler *        cl_sampler;
```

Figure 14. Data structures related to the OpenCL run-time implementation.

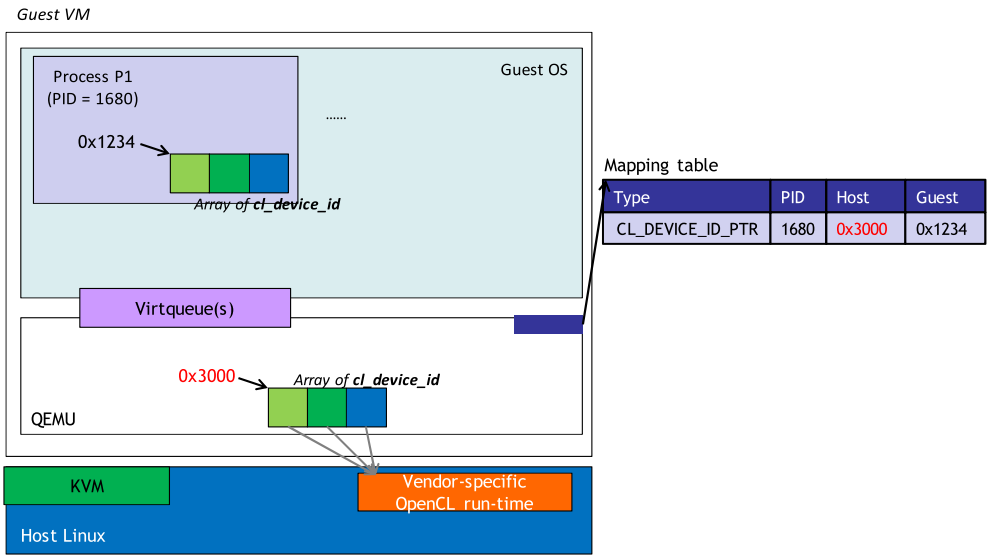

Figure 15. Shadow-mapping mechanism. KVM, Kernel-based Virtual Machine.

DOI: 10.1002/spe

```
cl_mem clCreateBuffer( cl_context   context,
                       cl_mem_flags flags,
                       size_t       size,
                       void *       host_ptr,
                       cl_int *     errcode_ret );
```

Figure 16. Prototype of `clCreateBuffer()`.

then invokes the actual OpenCL function with the shadow array and transfers all of the contents of the shadow array to the guest process after the function invocation completes to ensure that the `cl_device_id` array in the guest has the same contents as the shadow array.

When the guest process invokes an OpenCL function that uses a `cl_device_id` object, the value of this object can be directly used because it is a pointer to the host address space. When the guest process invokes an OpenCL function that uses an array of `cl_device_id` objects, the CL thread will look up the mapping table to find the address of the shadow array for the actual function invocation. After the guest process invokes `clEnvExit()`, the CL thread deletes the entries related to the process and releases the memory spaces recorded by the entries in the mapping table.

3.2.4. OpenCL memory objects. An OpenCL memory object (`cl_mem`) is an abstraction of global device memory that can serve as a data container for computation. A process can use `clCreateBuffer()` to create a memory object that can be regarded as a one-dimensional buffer. Figure 16 shows the prototype of `clCreateBuffer()`, where `context` is a valid OpenCL context associated with this memory object, `flags` is used to specify allocation and usage information (Table II describes the possible values of `flags`), `size` indicates the size of the memory object (in bytes), `host_ptr` is a pointer to the buffer data that may already be allocated, and `errcode_ret` is the field used to store the error-code information.

The buffer object can be allocated in either the OpenCL device or the OpenCL host memory, depending on the `flags` parameter. If the CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR option is selected, the buffer object would use the memory spaces pointed by `host_ptr` that resides in the virtual address space of the guest process. Although the memory spaces pointed by `host_ptr` belong to the OpenCL host memory, they can only be accessed by a set of corresponding OpenCL functions such as `clEnqueueReadBuffer()` and `clEnqueueWriteBuffer()`, and the behavior is undefined when directly accessing the memory spaces. Two difficulties are encountered when supporting the CL_MEM_USE_HOST_PTR or CL_MEM_ALLOC_HOST_PTR option in our virtualization framework, which is illustrated in Figure 17: (i) the CL thread cannot access this memory region directly and (ii) ensuring memory coherence between the guest process and the CL thread remains a complicated problem even when a shadow memory space is created in the CL thread. Possible solutions to the memory coherence problem are discussed in Section 3.2.5.

3.2.5. Enhancement of the guest OpenCL library. The current implementation of our OpenCL virtualization framework only fully supports the CL_MEM_COPY_HOST_PTR option, which allocates storage in the device memory and copies the data pointed by `host_ptr` to the device memory. The CL_MEM_USE_HOST_PTR and CL_MEM_ALLOC_HOST_PTR options are supported only if users follow the OpenCL specification to access the memory object by the corresponding OpenCL functions. Memory coherence cannot be guaranteed if the memory region was accessed by the OpenCL host program directly. Although the functions of `clCreateBuffer()` are not fully supported in the current virtualization framework, the framework can be used to virtualize most of the OpenCL operations on a CPU–GPU platform because buffer objects are generally accessed by OpenCL functions in response to operations thereon. The second drawback in our framework is that programmers have to add `clEnvInit()` and `clEnvExit()` function calls at the beginning and the end of an OpenCL program, respectively, as mentioned in Section 3.1.3, which means that the OpenCL programs are not natively portable.

DOI: 10.1002/spe

Table II. List of supported `cl_mem_flags` values (adapted from [8]).

`cl_mem_flags`	Description
CL_MEM_READ_WRITE	Specifies that the memory object will be read and written by a kernel. This is the default.
CL_MEM_WRITE_ONLY	Specifies that the memory object will be written but not read by a kernel. Reading from a buffer or image object created with CL_MEM_WRITE_ONLY inside a kernel is undefined.
CL_MEM_READ_ONLY	Specifies that the memory object is read-only when used inside a kernel. Writing to a buffer or image object created with CL_MEM_READ_ONLY inside a kernel is undefined.
CL_MEM_USE_HOST_PTR	Valid only if `host_ptr` is not null. If specified, it indicates that the application wants the OpenCL implementation to use memory referenced by `host_ptr` as the storage bits for the memory object. OpenCL implementations are allowed to cache the buffer contents pointed to by `host_ptr` in device memory. This cached copy can be used when kernels are executed on a device. The result of OpenCL commands that operate on multiple buffer objects created with the same `host_ptr` or overlapping host regions is considered to be undefined.
CL_MEM_ALLOC_HOST_PTR	Specifies that the application wants the OpenCL implementation to allocate memory from host accessible memory. CL_MEM_ALLOC_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive.
CL_MEM_COPY_HOST_PTR	Valid only if `host_ptr` is not null. If specified, it indicates that the application wants the OpenCL implementation to allocate memory for the memory object and copy the data from memory referenced by `host_ptr`. CL_MEM_COPY_HOST_PTR and CL_MEM_USE_HOST_PTR are mutually exclusive. CL_MEM_COPY_HOST_PTR can be used with CL_MEM_ALLOC_HOST_PTR to initialize the contents of the `cl_mem` object allocated using host accessible (e.g., PCIe) memory.

A possible approach for solving these problems is to substitute the guest OpenCL library with a new virtualization layer that can notify the hypervisor about the start/completion of a guest process and intercept the read/write events of the guest process so as to ensure memory coherence between the guest and shadow memory spaces. The virtualization layer is conceptually like a *process VM*. The disadvantage of introducing such a layer is the higher cost of ensuring memory coherence. Enhancing the guest OpenCL library constitutes one aspect of our future work.

4. EXPERIMENTAL RESULTS

Evaluations of the OpenCL virtualization framework proposed herein are presented in this section, including the environments tested, the experimental results obtained, and the discussions thereof.

DOI: 10.1002/spe

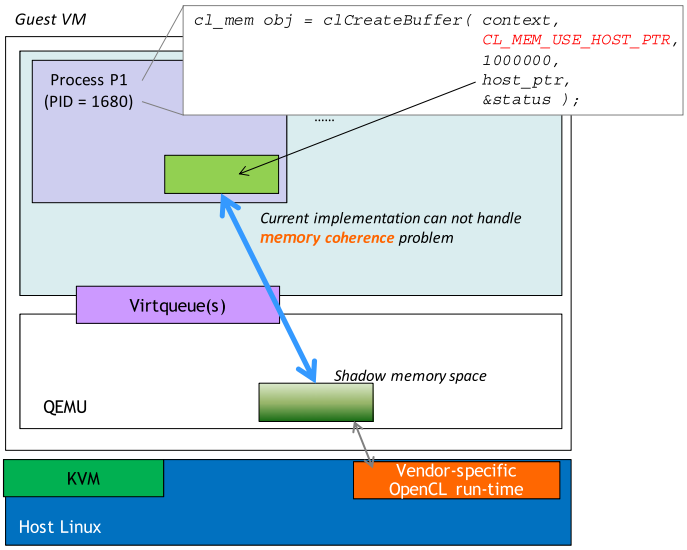

Figure 17. Memory coherence problem in `clCreateBuffer()`. KVM, Kernel-based Virtual Machine.

4.1. Environment

4.1.1. Testbed. All of the evaluations were conducted on a computer with an Intel Core i7-930 CPU (containing eight cores operating at 2.8 GHz), 8 GB of RAM, and two NVIDIA GeForce GTX 580 GPUs running the Linux 2.6.32.21 kernel with NVIDIA CUDA SDK version 3.2 (including NVIDIA OpenCL SDK). Our OpenCL virtualization framework was implemented in qemu-kvm-0.14.0 operating with kvm-kmod-2.6.32.21. Each guest VM in the virtualization framework was configured with one virtual CPU, 512 MB of RAM, and a 10-GB disk drive running the Linux 2.6.32.21 kernel. The size of virtqueues in virtio-CL was configured as 256 kB (64 pages).

As described in Section 3.1, the architecture of our OpenCL virtualization framework can be modeled as multiple processes accessing the OpenCL resources in the native environment. To realize the resource-sharing characteristics of our virtualization framework, we took advantage of NVIDIA *Fermi* GPUs that support 'concurrent kernel execution' and 'improved application context switching' [16]. Concurrent kernel execution allows multiple OpenCL kernels in an application to execute concurrently if the GPU resources are not fully occupied, and the optimized *Fermi* pipeline significantly improves the efficiency of context switches.

4.1.2. Benchmarks. The benchmarks we used to evaluate our proposed OpenCL virtualization framework were collected from the NVIDIA OpenCL SDK [17] and the AMD Accelerated Parallel Processing SDK [18], both of which contain a set of general-purpose algorithms in various domains. Table III summarizes these benchmarks, including their descriptions, the number of OpenCL API calls, and the amount of data transferred between the guest OS and the hypervisor process.

4.2. Evaluation

First, we evaluated the virtualization overhead of the proposed framework by measuring the execution time of OpenCL programs for two configurations:

- Native
  Benchmark programs execute on the host platform and access the vendor-specific OpenCL run-time system directly. This configuration represents the baseline in all experiments.
- 1-Guest
  Benchmark programs execute in a guest VM and acquire the functionality of OpenCL using our proposed virtualization framework. This configuration is used to evaluate the overhead in the virtualized environment.

DOI: 10.1002/spe

Table III. Parameters of the benchmark programs.

Benchmark	Source	Number of API calls	Data size (MB)	Description
FastWalshTransform	AMD	34	3.08	Generalized Fourier transformations
BlackScholes	NVIDIA	34	77.27	Modern option pricing techniques that are commonly applied in the finance field
MersenneTwister	NVIDIA	44	183.88	Mersenne twister random number generator and Cartesian Box–Muller transformation on the graphics processing unit
MatrixManipulation	AMD	53	48.76	An implementation of the matrix multiplication
ScanLargeArray	AMD	70	0.51	An implementation of scan algorithm for large arrays
ConvolutionSeparable	NVIDIA	38	72.75	Convolution filter of a 2D image with an arbitrary separable kernel
ScalarProduct	NVIDIA	37	0.81	Scalar product of set of input vector pairs

Table IV. Execution time of seven OpenCL benchmarks for the Native and 1-Guest configurations.

Benchmark	Native (seconds)	1-Guest (seconds)	Virtualization Overhead (seconds)	Virtualization Overhead (%)
FastWalshTransform	1.175	1.256	0.081	6.9
BlackScholes	2.330	2.424	0.094	4.0
MersenneTwister	8.078	11.603	3.525	43.6
MatrixManipulation	0.969	1.082	0.113	11.7
ScanLargeArray	0.939	0.976	0.037	3.9
ConvolutionSeparable	3.937	4.154	0.217	5.5
ScalarProduct	0.581	0.631	0.050	8.6

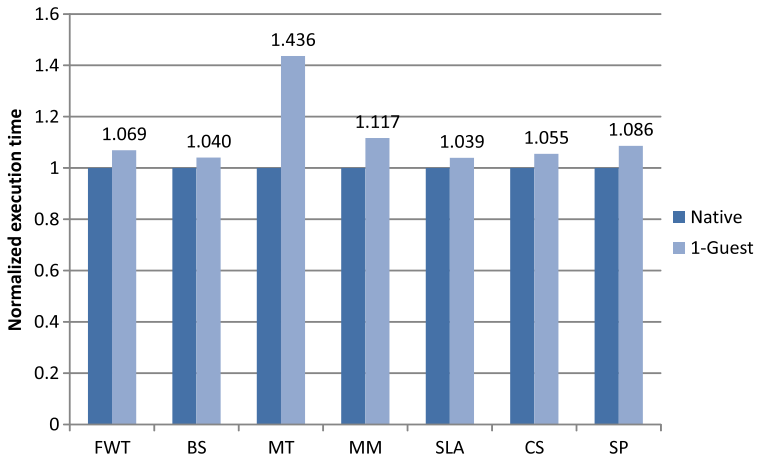

Figure 18. Normalized execution time for the Native and 1-Guest configurations. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

4.2.1. Virtualization overhead. Table IV summarizes the execution time of the seven OpenCL programs for the Native and 1-Guest configurations, and Figure 18 shows the time with the Native configuration normalized to 1. The overhead of executing the OpenCL programs in the virtualized platform ranged from 3.9% [for ScanLargeArray (SLA)] to 43.6% [for MersenneTwister (MT)], with the mean value of 12.0%. The overhead was only markedly higher than 10% for the MT benchmark program—the mean overhead was 6.8% if MT is excluded.

DOI: 10.1002/spe

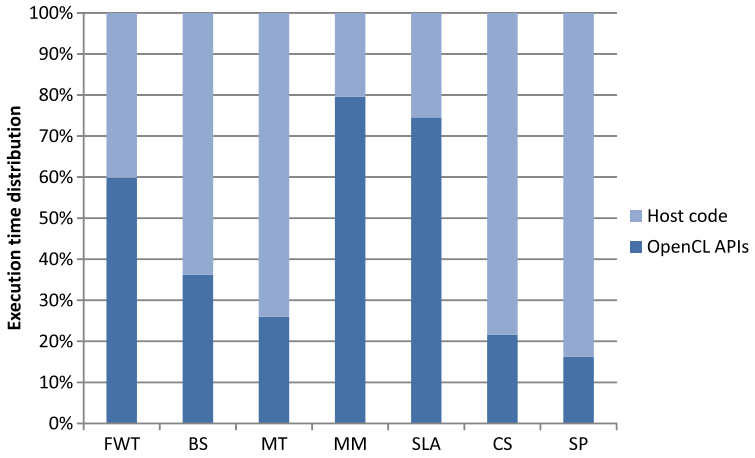

Figure 19. Ratios of execution time for OpenCL APIs and OpenCL host code. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

Figure 20. Ratios of virtualization overheads for OpenCL APIs and OpenCL host code. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

Overhead analysis. To identify the reason for the various overhead distributions, we first measured the execution time for OpenCL APIs and OpenCL host code in both configurations. Figures 19 and 20 show the ratios of execution time and virtualization overheads for OpenCL APIs and OpenCL host code, respectively. According to the analysis, MT is a CPU-intensive program, and the overhead of OpenCL host code (referred to as the CPU virtualization in KVM) dominates the performance loss. In addition, we analyzed the virtualization overhead of the OpenCL APIs. Table V and Figure 21 provide detailed breakdowns of the virtualization overhead, which is divided into three parts in our measurements: (i) data transfer from the guest VM to the hypervisor; (ii) data transfer from the hypervisor to the guest VM; and (iii) other overheads. We used the `gettimeofday()` function to measure the time required for data transfer from the guest VM to the hypervisor in the guest OpenCL library and the transfer time in the opposite direction in the CL thread. Other overheads represent the remainder of the period of the virtualization overhead that is incurred in executing the guest OpenCL library, synchronizing between the vCPU thread and CL thread, executing the CL thread, and other processes. The overhead caused by data transfer ranged from 78.3% and 94.1%, and the other overheads ranged from 5.9% to 21.7%. As expected, the data transfer dominates the OpenCL virtualization overhead because of the nature of API remoting.

DOI: 10.1002/spe

Table V. Detailed breakdown of the OpenCL virtualization overhead.

Benchmark	Overhead (seconds)	G to H* (seconds)	H to G† (seconds)	Others (seconds)
FastWalshTransform	0.020	0.009	0.008	0.002
BlackScholes	0.092	0.045	0.042	0.005
MersenneTwister	1.416	0.042	1.067	0.307
MatrixManipulation	0.106	0.059	0.036	0.011
ScanLargeArray	0.005	0.002	0.002	0.0003
ConvolutionSeparable	0.133	0.048	0.068	0.017
ScalarProduct	0.010	0.003	0.006	0.0010

*Guest to hypervisor.
† Hypervisor to guest.

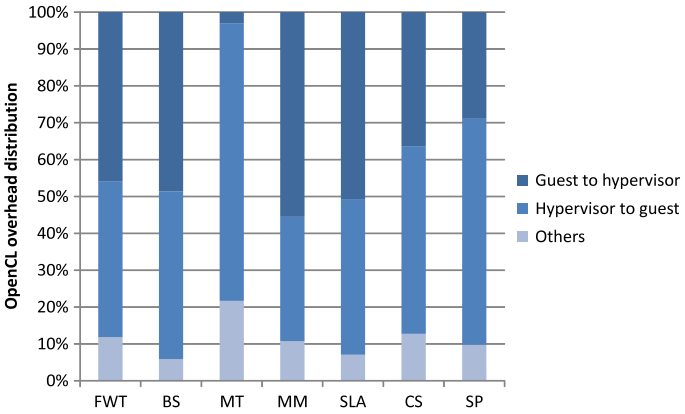

Figure 21. Breakdown of the virtualization overhead. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

Table VI. Profiling information of data transfer and virtual machine exits.

Benchmark	G to H (MB)	H to G (MB)	Total (MB)	No. of virtual machine exits
FastWalshTransform	3.02	0.05	3.07	744
BlackScholes	46.75	30.52	77.27	485
MersenneTwister	0.75	183.13	183.88	1541
MatrixManipulation	32.75	16.01	48.76	482
ScanLargeArray	0.50	0.01	0.51	132
ConvolutionSeparable	36.75	36.00	72.75	645
ScalarProduct	0.79	0.22	0.81	66

We ascertained the amounts of data transferred to and from the hypervisor and the number of VM exits to investigate the relationships among overheads and the aforementioned two factors. The profiling information is given in Table VI and Figure 22. In general, the transfer of greater amount of data between the host and the device in an OpenCL program results in more VM exits because the data-copy process between the guest and the hypervisor requires a series of VM exits: two data-copy processes from the guest to the hypervisor, and three data-copy processes from the hypervisor to the guest for the data transfer of an OpenCL API call. Thus, both VM exits and data transfer cause additional overhead. However, the relationship cannot be clarified by the profile information because each benchmark program exhibits its own specific behavior that results in different amounts of data transferred and numbers of VM exits.

Variable input size. To clarify the relationships among the amount of data transferred, the number of VM exits, and the OpenCL virtualization overhead, we investigated BlackScholes (BS)—which implements a mathematical model for financial options—with variable input sizes by adjusting the

DOI: 10.1002/spe

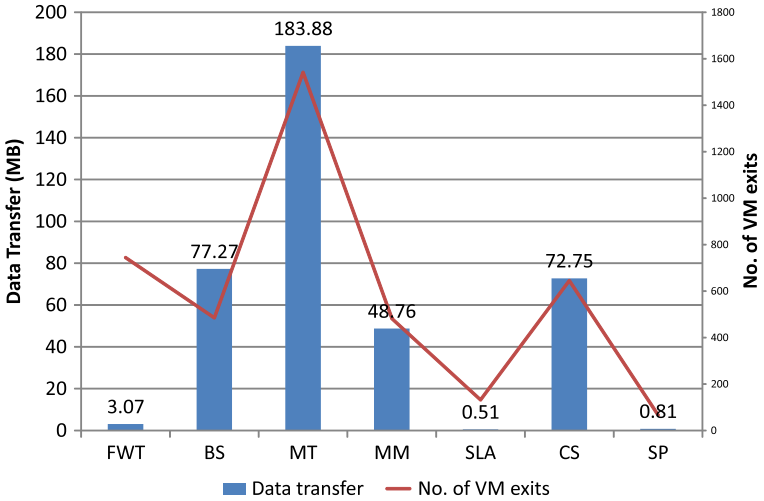

Figure 22. Profiling information of data transfer and virtual machine (VM) exits. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

Table VII. OpenCL virtualization overhead and profile information of BlackScholes with variable input size. `optionCount` is the variable that configures the input size.

`optionCount`	Native (seconds)	1-Guest (seconds)	Overhead (seconds)	Overhead (%)	Data size (MB)	No. of virtual machine exits
2,000,000	0.738	0.787	0.050	6.7	38.76	268
4,000,000	0.753	0.869	0.116	15.4	77.27	485
6,000,000	0.784	1.022	0.238	30.4	115.03	695
8,000,000	0.820	1.434	0.614	74.9	153.54	912
10,000,000	0.852	2.826	1.974	231.5	191.86	1124
12,000,000	0.877	4.281	3.404	388.1	229.81	1339
14,000,000	0.908	5.725	4.817	530.2	269.69	1557
16,000,000	0.946	7.267	6.320	667.8	306.07	1766

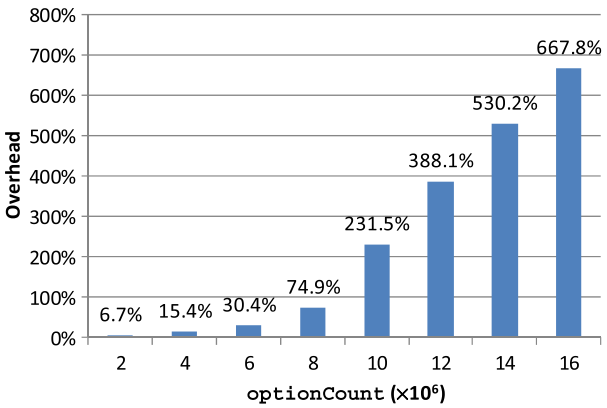

Figure 23. OpenCL virtualization overhead for BlackScholes with variable input size.

value of variable `optionCount` from 2×10^6 to 16×10^6. Table VII and Figures 23 and 24 summarize the evaluation results and the profiling statistics, including both the amount of data transferred and the number of VM exits. It can be seen that both of these parameters are proportional to the input data size, and the OpenCL virtualization overhead has a slightly more than proportional increase with respect to the input data size. Figure 25 provides a detailed breakdown of the virtualization overhead for BS with variable input size.

DOI: 10.1002/spe

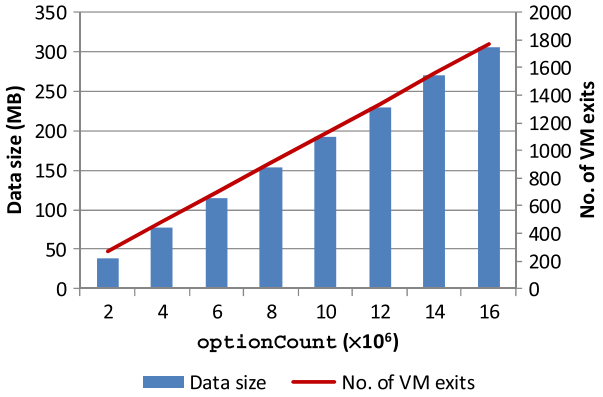

Figure 24. Profile information for BlackScholes with variable input size.

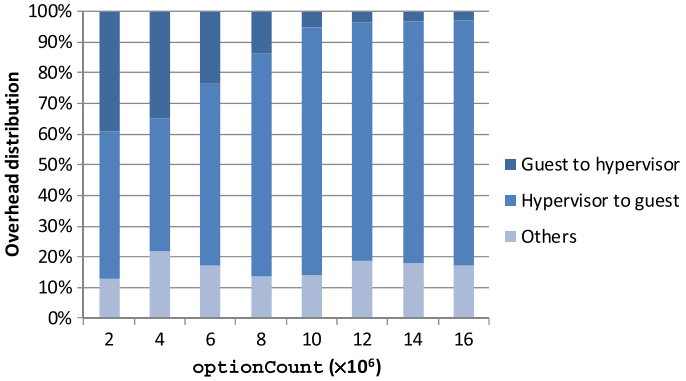

Figure 25. Breakdown of OpenCL virtualization overhead for BlackScholes with variable input size.

We observed that the contribution of other overheads was relatively stable, ranging from 12.9% to 22.0%, whereas the ratio for hypervisor-to-guest data transfer increased from 43% to about 80% when `optionCount` increased to 16×10^6, whereas the ratio for guest-to-hypervisor data transfer decreased. As mentioned in Section 3.1.2, data transfer from the guest to the hypervisor is performed actively by the vCPU thread and requires a series of VM exits, whereas the data transfer from the hypervisor to the guest is performed actively by the CL thread and requires a series of VM exits and virtual interrupts. This difference in the implementations of data transfer results in the overhead for hypervisor-to-guest data transfer increasing faster as the amount of transferred data increases.

In conclusion, the virtualization overhead in our framework is strongly affected by the amount of data transferred and the number of VM exits of OpenCL programs, and this characteristic is also obvious in other API remoting approaches.

4.2.2. Evaluation in multiple guest virtual machines. The following experiment was used to evaluate the scalability of our OpenCL virtualization framework. We executed multiple guest VMs, each of which ran an arbitrary benchmark program. There were two configurations: 2-Guest and 3-Guest, which represent scenarios of concurrently executing two and three OpenCL programs by different guest VMs, respectively. Figure 26 shows the normalized execution time of each benchmark in the Native, 1-Guest, 2-Guest, and 3-Guest configurations. We observed that the virtualization overhead was smaller in the 2-Guest configuration than that in the 1-Guest configuration and even than that in the Native configuration for most of the benchmarks: such as FastWalshTransform (FWT), BS, MatrixManipulation (MM), SLA, and ConvolutionSeparable (CS). The virtualization overhead in the 3-Guest configuration was larger than that in the 2-Guest configuration but still smaller than that in the 1-Guest configuration.

DOI: 10.1002/spe

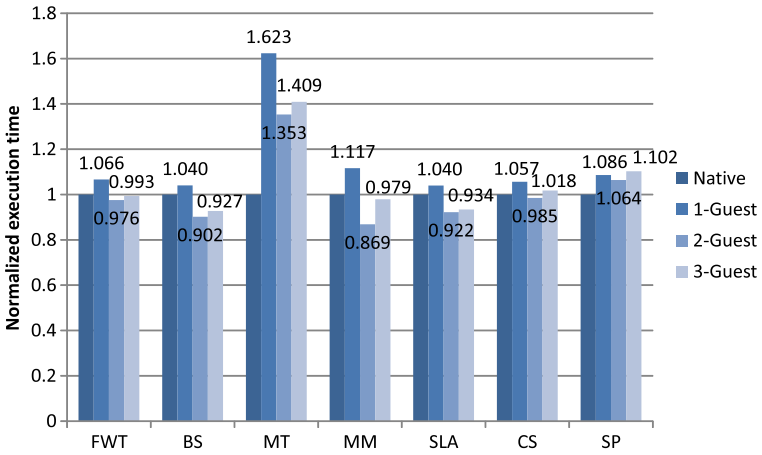

Figure 26. Normalized execution time for the Native, 1-Guest, 2-Guest, and 3-Guest configurations. FWT, FastWalshTransform; BS, BlackScholes; MT, MersenneTwister; MM, MatrixManipulation; SLA, ScanLargeArray; CS, ConvolutionSeparable; SP, ScalarProduct.

The experimental results shown in Figure 26 are surprising because we expected the virtualization overhead to be larger in the 2-Guest and 3-Guest configurations than in the 1-Guest configuration. To identify the reason for this, we determined the execution time of each OpenCL API function in the guest OpenCL library and performed in-depth analyses of the interaction among two or three OpenCL programs residing in two or three different VMs. We observed that the execution time of OpenCL API calls for resource initialization in one program decreased dramatically if the other program(s) had completed its(their) own resource initialization(s). Moreover, the difference in the execution time for these API calls almost matched the difference in overall execution time of the OpenCL program between the 1-Guest and 2-Guest configurations. This provided strong evidence that the abnormality of the 2-Guest and 3-Guest configurations outperforming the 1-Guest configuration is attributable to the implementation of the vendor-specific OpenCL run-time system, which may contain a resource pool for maintaining the OpenCL resource initialization.

Figure 27 illustrates the scenario of multiple guest VMs accessing the vendor-specific OpenCL run-time system. As mentioned in Section 3.1.1, the lifetime of a CL thread almost spans that of the hypervisor process that it belongs to. Once an OpenCL process in a guest VM starts executing,

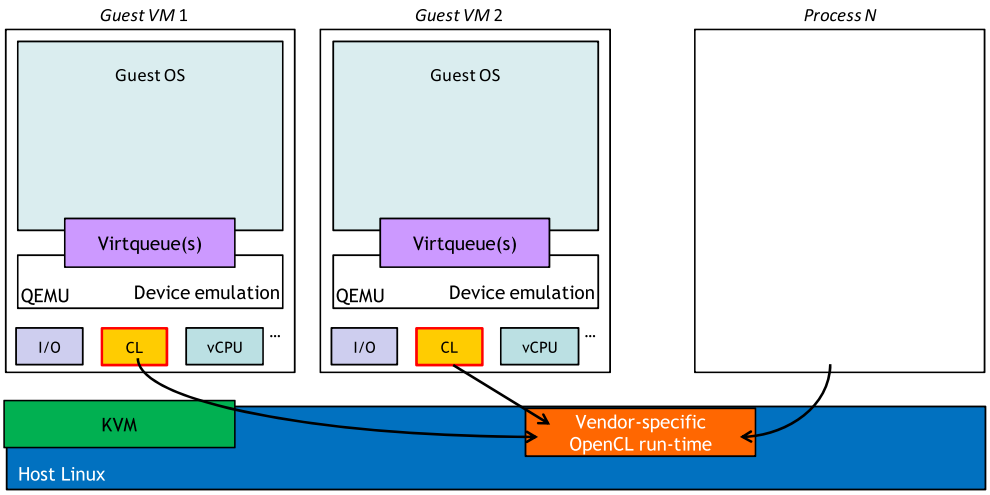

Figure 27. Scenario of multiple virtual machines accessing the vendor-specific OpenCL run-time system. KVM, Kernel-based Virtual Machine; I/O, input/output; OS, operating system.

DOI: 10.1002/spe

the CL thread in the guest VM accesses the vendor-specific OpenCL run-time system and interferes with the resource allocation scheme in the vendor-specific OpenCL run-time system, which somehow keeps the information about the underlying OpenCL resources even if the OpenCL resources are released by OpenCL API calls inserted by the programmers and the information are kept until the CL thread ends (i.e. until the guest VM is shut down). Consequently, the execution time of resource initialization-related API functions is nearly eliminated if a CL thread is active, and hence, the performance of OpenCL virtualization in the 2-Guest and 3-Guest configurations surpasses that in the 1-Guest configuration. In addition, without considering the influence of the resource initialization mechanism of the run-time implementation, the virtualization overhead increases slowly as the number of guest VMs increases. This implies that the virtualization framework improves the utilization of the underlying OpenCL resources without further intervention from programmers.

4.3. A brief comparison with related work

The literature contains reports of researches into API remoting for virtualization. GViM [3], vCUDA [4], rCUDA [19], and gVirtuS [5], which are discussed in detail in Section 5, are such researches whose objectives were conceptually similar to those of the present study. In brief, GViM, vCUDA, and rCUDA were designed to support CUDA virtualization in Xen-based VMs [20] for high-performance computing, and gVirtuS supports CUDA virtualization in KVM, whereas the present study enabled OpenCL support in KVM. In this section, we briefly compare virtualization overheads of GViM, vCUDA, rCUDA, gVirtuS, and the present approach. Although the comparison might not prove the superiority of our approach over the others because of differences in the evaluation configurations, such as the underlying hardware platforms, implementations of benchmark programs in CUDA and OpenCL, and input data sizes, the comparison still provides a rough estimate of the relative virtualization overheads among the five frameworks. Table VIII summarizes the input data size of the benchmark programs for the five frameworks.

Figure 28 shows the virtualization overhead of CUDA or OpenCL programs in GViM, vCUDA, rCUDA, gVirtuS, and our approach (note that some results are not available for GViM, vCUDA, rCUDA, and gVirtuS). The overhead of the benchmark programs in vCUDA ranged from 73.6% (for BS) to 427.8% (for MT), which is much more than the overhead in GViM, gVirtuS, and the present study. We believe that the high virtualization overhead in vCUDA and rCUDA is attributable to the scheme that they use for data transfer between the hypervisor and guests, which involves wrapping an API call as an RPC, whereas GViM, gVirtuS, and our framework use shared-memory-based approaches for this communication. The virtualization overheads in GViM, gVirtuS, and our framework were roughly similar: 4.0%, 6.9%, 11.7%, and 8.6% for BS, FWT, MM, and SP, respectively, in our framework; 25.0%, 14.0%, and 3.0% for BS, FWT, and MM, respectively, in GViM; and 22.4% and 11.1% for MM and SP, respectively, in gVirtuS. The five research frameworks are compared in more detail in Section 5.

It is worth noting that in Section 3.2.3, we mention that the proposed framework requires an increased level of indirection for pointers during the guest–host communication, but the evaluation results show that the proposed framework performed slightly better than other frameworks for most of the benchmarks; this is likely attributable to virtio-CL, the proposed shared-memory-based communication mechanism, being tailored for the underlying hypervisor.

Table VIII. Configurations of input data size in this study and the related works.

Benchmark	This study	GViM	vCUDA	rCUDA	gVirtuS
BlackScholes	16 MB	4 MB	16 MB	–	–
ConvolutionSeparable	36 MB	–	36 MB	–	–
FastWalshTransform	32 MB	64 MB	32 MB	N/A	–
MersenneTwister	22.89 MB	–	22.89 MB	–	–
ScanLargeArray	0.004 MB	–	N/A	–	–
MatrixManipulation	2 kB × 2 kB	2 kB × 2 kB	–	2 kB × 2 kB	2 kB × 8 kB
ScalarProduct	2 MB	–	–	–	2 MB

DOI: 10.1002/spe

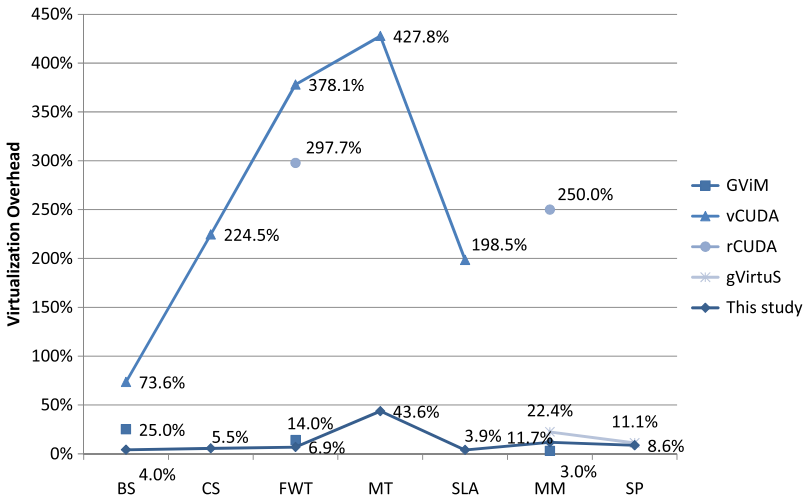

Figure 28. Comparison of virtualization overheads among different virtualization frameworks. BS, BlackScholes; CS, ConvolutionSeparable; FWT, FastWalshTransform; MT, MersenneTwister; SLA, ScanLargeArray; MM, MatrixManipulation; SP, ScalarProduct.

5. RELATED WORK

OpenCL has become an industrial standard in the field of heterogeneous multicore computing. Many industry-leading companies (e.g. AMD, Apple, IBM, Intel, and NVIDIA) have released their own OpenCL run-time implementations to help users access OpenCL functionality [17, 18, 21–23]. However, there are no standard solutions for supporting OpenCL execution in a virtualized system in either industry or academia. The present study represents the first research to support OpenCL in a system virtualization environment based on API remoting. Although the framework currently only supports GPU devices, it would be easy to extend it with API remoting to support more OpenCL devices.

Several researches have adopted API remoting as an approach for virtualization. Lagar-Cavilla *et al.* proposed VMGL, a solution for supporting the OpenGL graphics-acceleration API in Xen-based or VMware-based VMs with the functionality of *GPU multiplexing* and the suspension and resumption of execution [24]. VMGL virtualizes OpenGL API and uses an advanced OpenGL network transmission scheme called WireGL [25] to optimize the data transfer of OpenGL contexts. Weggerle *et al.* proposed VirtGL, which is an OpenGL virtualization solution for QEMU [26]. VirtGL implements OpenGL API remoting by adding a new QEMU virtual device, with data transfer occurring via the MMIO spaces of the virtual device. Gupta *et al.* proposed GViM, which is a CUDA virtualization solution for Xen hypervisor [3]. GViM uses the shared-memory-based I/O communication mechanism in the Xen hypervisor to implement data transfer between guest VMs and the dedicated VM that emulates I/O actions. GViM supports GPU multiplexing and implements a simple credit-based scheduling mechanism for the CUDA API. Shi *et al.* proposed a CUDA API remoting framework, called vCUDA, for the Xen hypervisor [4]. vCUDA uses an RPC scheme named XML-RPC [27] to perform data transfer between the hypervisor and guests. Those authors introduced a 'lazy RPC' mechanism to reduce the frequency of RPC and thus the virtualization overhead. Duato *et al.* proposed rCUDA [19], which is a framework for executing CUDA applications in cluster environments that contain CUDA devices in certain cluster nodes. rCUDA uses an RPC-based mechanism to perform data transfer among different cluster nodes. Giunta *et al.* proposed gVirtuS [5], which is a CUDA virtualization framework, and developed a guest–host communication mechanism, called vmSocket, which is independent of the underlying hypervisor and the communication technology (such as shared memory, QEMU virtual device, and TCP/IP).

Table IX compares the related work and our study. Our virtualization framework enables OpenCL support in KVM, whereas GViM and vCUDA allow CUDA programs to execute within Xen-based

DOI: 10.1002/spe

Table IX. Comparison of API-remoting-based virtualization frameworks.

Virtualization framework	Target API	Target VM	Guest–host VM Communication	Virtualization overhead	Main features
VMGL [24]	OpenGL (graphics purpose)	Xen, VMware, etc.	WireGL [25]	Incomparable*	GPU multiplexing Suspension and resumption
VirtGL [26]	OpenGL (graphics purpose)	QEMU	QEMU virtual device (based on shared memory)	Incomparable*	None
GViM [3]	CUDA (general purpose)	Xen	Shared ring for function calls	Low	GPU multiplexing Credit-based GPU scheduling
vCUDA [4]	CUDA (general purpose)	Xen	XML-RPC Lazy RPC	High	GPU multiplexing Suspension and resumption
rCUDA [19]	CUDA (general purpose)	Clusters (not system VM)	RPC	High	Remote execution in cluster environments
gVirtuS [5]	CUDA (general purpose)	KVM, VMware, Xen, etc.	vmSocket	Low	GPU multiplexing
This study	OpenCL (general purpose)	KVM	virtio-CL (based on shared memory)	Low	GPU multiplexing

*Incomparable since the virtualization framework is not for general-purpose computation.
KVM, Kernel-based Virtual Machine; RPC, remote procedure call.

DOI: 10.1002/spe

VMs. Unlike OpenCL, which supports different kinds of heterogeneous computing devices, CUDA is only supported on machines with NVIDIA GPUs. Our proposed OpenCL virtualization framework provides a solution for heterogeneous multicore computing in VMs. Both KVM and Xen are well-known frameworks in the system virtualization field. Xen supports paravirtualization and hardware-assisted virtualization, and KVM supports hardware-assisted CPU virtualization. Both KVM and Xen provide excellent performance in CPU virtualization relative to other virtualization frameworks. However, the architecture of I/O virtualization differs markedly between Xen and KVM. The I/O actions of guest VMs in Xen have to be intercepted and processed by a specific guest VM called the I/O domain (or Dom0), and thus, the I/O domain becomes the bottleneck of I/O virtualization and results in unpredictable I/O bandwidth and latency [28, 29]. In contrast, the I/O virtualization in KVM is processed in the same hypervisor process, and the architecture of I/O processing is much simpler. Therefore, we consider KVM to be a better platform for OpenCL virtualization than Xen.

6. CONCLUSION

This paper has described how to enable OpenCL support in KVM using API remoting. The proposed OpenCL virtualization framework allows multiplexing of multiple guest VMs over the underlying OpenCL resources, which provides for their better utilization. Owing to the characteristic that each VM in KVM is a process in Linux, virtualizing OpenCL programs in multiple guest VMs can be modeled as multiple processes accessing the OpenCL resources in the native environment. Although the virtualization framework only supports running OpenCL programs in GPGPU architectures in our current implementation, it would be easy to extend the API remoting scheme to support other OpenCL devices.

Our evaluations indicate that the virtualization overhead of the proposed framework is mainly due to data transfer and the synchronization between the KVM vCPU thread and our new added CL thread, which are directly affected by the amount of data to be transferred for an API call because of the primitive nature of API remoting and the limited size of the memory that is shared between the hypervisor and guest VMs. The experimental results show that the virtualization overhead was around 10% or less (mean of 6.8%) for six common GPU-intensive OpenCL programs. Furthermore, OpenCL resources are better utilized because of the framework supporting of OpenCL multiplexing.

ACKNOWLEDGEMENTS

This study was partially supported by the National Science Council of Taiwan under grants NSC-97-2218-E-009-043-MY3 and NSC-100-2218-E-009-011-MY3.

REFERENCES

1. CUDA: Compute Unified Device Architecture. Available at: http://www.nvidia.com/object/cuda_home_new.html/. [last accessed 10 March 2012].
2. OpenCL—the open standard for parallel programming of heterogeneous systems. Available at: http://www.khronos.org/opencl/. [last accessed 10 March 2012].
3. Gupta V, Gavrilovska A, Schwan K, Kharche H, Tolia N, Talwar V, Ranganathan P. GViM: GPU-accelerated virtual machines. In *Proceedings of the 3rd ACM Workshop on System-Level Virtualization for High Performance Computing*, HPCVirt '09. ACM: New York, NY, USA, 2009; 17–24, DOI: 10.1145/1519138.1519141.
4. Shi L, Chen H, Sun J. vCUDA: GPU accelerated high performance computing in virtual machines. *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, Rome, Italy, 2009; 1–11, DOI: 10.1109/IPDPS.2009.5161020.
5. Giunta G, Montella R, Agrillo G, Coviello G. A GPGPU transparent virtualization component for high performance computing clouds. In *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, EuroPar'10. Springer-Verlag: Berlin, Heidelberg, 2010; 379–391.
6. Russell R. Virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review* July 2008; **42**:95–103. DOI: 10.1145/1400097.1400108.
7. Kivity A, Kamay Y, Laor D, Lublin U, Liguori A. kvm: the Linux virtual machine monitor. *Proceedings of the 2007 Ottawa Linux Symposium,* OLS '07, Ottawa, Canada, 2007; 225–230.
8. Khronos Group. The OpenCL Specification. 1.2 edn, 2011.

9. Dowty M, Sugerman J. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Operating Systems Review* 2009July; **43**:73–82. DOI: 10.1145/1618525.1618534.
10. Intel Cooperation. Intel®virtualization technology for directed I/O. 1.3 edn, 2011.
11. Intel Virtualization Technology. Available at: http://www.intel.com/technology/virtualization/. [last accessed 10 March 2012].
12. AMD Virtualization. Available at: http://sites.amd.com/us/business/it-solutions/virtualization/Pages/virtualization.aspx. [last accessed 10 March 2012].
13. Bellard F. QEMU, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05. USENIX Association: Berkeley, CA, USA, 2005; 41–41.
14. Kiszka J. Architecture of the Kernel-based Virtual Machine (KVM), 2010. Technical Sessions of Linux Kongress.
15. Jones MT. Virtio: an I/O virtualization framework for Linux. Available at: http://www.ibm.com/developerworks/linux/library/l-virtio/. [last accessed 10 March 2012].
16. NVIDIA Corporation. NVIDIA's Next Generation CUDA™ Compute Architecture: Fermi™. 1.1 edn, 2009.
17. NVIDIA OpenCL SDK. Available at: http://developer.nvidia.com/opencl. [last accessed 10 March 2012].
18. AMD Accelerated Parallel Processing (APP) SDK. Available at: http://developer.amd.com/sdks/amdappsdk/pages/default.aspx. [last accessed 10 March 2012].
19. Duato J, lgual FD, Mayo R, Peña AJ, Quintana-Ortí ES, Silla F. An efficient implementation of GPU. Virtualization in High Performance Clusters. *Euro-par 2009,* Euro-Par '09, 2009, DOI: 10.1007/978-3-642-14122-5_44.
20. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03. ACM: New York, NY, USA, 2003; 164–177, DOI: 10.1145/945445.945462.
21. OpenCL, taking the graphics processor beyond graphics. Available at: http://developer.apple.com/technologies/mac/snowleopard/opencl.html. [last accessed 10 March 2012].
22. OpenCL Development Kit for Linux on Power. Available at: http://www.alphaworks.ibm.com/tech/opencl/. [last accessed 10 March 2012].
23. Intel®OpenCL SDK. Available at: http://software.intel.com/en-us/articles/opencl-sdk/. [last accessed 10 March 2012].
24. Lagar-Cavilla HA, Tolia N, Satyanarayanan M, de Lara E. VMM-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07. ACM: New York, NY, USA, 2007; 33–43, DOI: 10.1145/1254810.1254816.
25. Buck I, Humphreys G, Hanrahan P. Tracking graphics state for networked rendering. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '00. ACM: New York, NY, USA, 2000; 87–95, DOI: 10.1145/346876.348233.
26. Weggerle A, Schmitt T, Low C, Himpel C, Schulthess P. VirtGL—a lean approach to accelerated 3D graphics virtualization. *Cloud Computing and Virtualization 2010,* CCV '10, 2010.
27. XML-RPC Specification. Available at: http://www.xmlrpc.com/spec. [last accessed 10 March 2012].
28. Ongaro D, Cox AL, Rixner S. Scheduling I/O in virtual machine monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08. ACM: New York, NY, USA, 2008; 1–10, DOI: 10.1145/1346256.1346258.
29. Kim H, Lim H, Jeong J, Jo H, Lee J. Task-aware virtual machine scheduling for I/O performance. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '09. ACM: New York, NY, USA, 2009; 101–110, DOI: 10.1145/1508293.1508308.