Building a KVM-based Hypervisor for a Heterogeneous System Architecture Compliant System

Yu-Ju Huang
Department of Computer Science, National Chiao Tung University, Taiwan
gic4107@gmail.com

Hsuan-Heng Wu
Department of Computer Science, National Taiwan University, Taiwan
wuxx1279@gmail.com

Yeh-Ching Chung
Department of Computer Science, National Tsing Hua University, Taiwan
ychung@cs.nthu.edu.tw

Wei-Chung Hsu
Department of Computer Science, National Taiwan University, Taiwan
hsuwc@csie.ntu.edu.tw

Abstract

Heterogeneous System Architecture (HSA) is an architecture developed by the HSA foundation aiming at reducing programmability barriers as well as improving communication efficiency for heterogeneous computing. For example, HSA allows heterogeneous computing devices to share the same virtual address space. This feature allows programmers to bypass explicit data copying between devices, as was required in the past. HSA features such as job dispatching through user level queues and memory based signaling help to reduce communication latency between the host and other computing devices.

While the new features in HSA enable more efficient heterogeneous computing, they also introduce new challenges to system virtualization, especially in memory virtualization and I/O virtualization. This work investigates the issues involved in HSA virtualization and implements a KVM-based hypervisor that supports the main features of HSA inside guest operating systems. Furthermore, this work shows that with the newly introduced hypervisor for HSA, system resources in HSA-compliant AMD Kaveri can be effectively shared between multiple guest operating systems.

Keywords Heterogeneous System Architecture; HSA Virtualization; GPU Virtualization; KVM;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
VEE '16, April 02-03, 2016, Atlanta, GA, USA.
© 2016 ACM. ISBN 978-1-4503-3947-6/16/04...$15.00.
DOI: http://dx.doi.org/10.1145/2892242.2892246

1. Introduction

Heterogeneous architectures have become popular in recent years. Some computing tasks are more suited for GPGPU while others are a better fit for CPU or FPGA. In a heterogeneous computing system, power efficiency can be vastly improved when each job is dispatched to its most suited computing device.

The GPGPU programming model [1], as one example of heterogeneous computing models, allows programmers to dispatch computational kernels to GPU. The foremost GPGPU programming model, CUDA [2] and the older version OpenCL [3], (i.e. prior to OpenCL 2.0) see GPU as an I/O device so there must be data copying between CPUs and the GPU before a computation job can be launched. Such explicit data copying has caused significant overhead and programming inconvenience. Moreover, since GPU is viewed as an I/O device, every job to be executed has to go through a system driver. Such job dispatching mechanisms have resulted in context switch overheads between user mode and kernel mode.

HSA [4] is an architecture designed to address these inefficiencies and inconveniences. Two major goals of HSA are to (1) Reduce CPU/GPU communication latency such as data copying and jobs dispatching overhead, and (2) Decrease the heterogeneous computing programmability barrier such as the need for programmers to compress complicated data structures into a continuous memory region in order to copy it from CPU to GPU and to decompress it when the data is transferred back from the GPU. To achieve such goals, various features are defined as requirements for a HSA-compliant system. The HSA features and how these features are implemented on AMD Kaveri [5], our target platform, will be described in Section 2.

Table 1. Comparison of GPU programming in Non-HSA and HSA systems.

	Non-HSA systems	HSA systems
GPU job dispatching	Application calls system driver to store commands in GPU channel	Application stores AQL packets in user mode queue and kicks doorbell to signal GPU
GPU job finishing	GPU interrupts CPU	GPU notifies CPU via memory-based signals
GPU memory	Separate virtual address space between CPU and GPU	Shared virtual address space between CPU and GPU

This paper presents HSA virtualization and successfully implements a KVM-based [6] hypervisor. Since HSA is a new heterogeneous computing architecture, there has not been much research investigating system virtualization issues for this architecture. We analysed the features provided by HSA and figured out how to virtualize them so that processes inside guest OSes can also benefit from of it. Table 1 shows a comparison of a GPU programming model between non-HSA and HSA systems. These dissimilar behaviors require non-conventional GPU virtualization. Considering the GPU memory, non-HSA systems have a separate address space for the GPU. To virtualize the GPU so that it may be shared by multiple guest OSes, GPU memory must be virtualized for isolation and protection. In HSA systems, the virtual address is shared between CPU and GPU. All the addresses issued by the GPU are guest virtual addresses, which are the same as what the CPU issues. Thus a table translating guest virtual address to machine physical address is sufficient to virtualize the GPU memory in a HSA system. As for job dispatching, HSA supports user mode queues so that applications can store job attributes inside the queue and the GPU is able to access it as long as the address of the user mode queue is set to the GPU. To virtualize job dispatching in a HSA system, it would only be necessary to set the address of the user mode queue of guest process to the GPU during queue initialization, which is simpler than mapping the guest GPU channel to a physical GPU channel for every job dispatched in conventional GPU virtualization.

Our implementation also achieves GPU sharing, which allows processes of multiple guest OSes to share the same GPU in the HSA-compliant system. To the best of our knowledge, this is the first research pertaining to HSA virtualization and implementation of a hypervisor on a physical HSA-compliant machine. Furthermore, though the implementation is targeted on an AMD Kaveri machine, general analysis and insights about how to virtualize HSA features on other systems are provided. These analyses and insights can be applied to other architectures with similar features as well.

The rest of this paper is organized as follows. The techniques of how to virtualize various HSA features are presented in Section 3. Our implementation of the HSA-aware hypervisor are described in Section 4. Experiment results and performance evaluation are shown in Section 5. Section 6 discusses related work and Section 7 concludes.

This paper makes the following contributions:

- It investigates the issues involved in virtualizing HSA-compliant systems. It is the first successful implementation of a hypervisor that virtualizes various HSA features to support multiple guest OSes.
- This work achieves GPU sharing among multiple guest OSes and the host OS.
- It looks into issues on AMD IOMMU's two-level address translation mechanism, and provides insights and solutions to work around current hardware limitations.

2. Heterogeneous System Architecture

In this section we introduce more details about the HSA system architecture. We first describe the distinguishing features pertaining to job dispatching and execution in HSA, and we provide analyses on which features benefit from virtualization, given that some features can be simply achieved by hardware or runtime without the hypervisor. Then we explore our target platform, AMD Kaveri, and investigate how these features are implemented on the target machine. Before introducing HSA features, two specific terminologies must be explained.

Architected Queuing Language (AQL): A command interface for dispatching jobs between CPU and HSA devices. When an application wants to execute a job on an HSA device, an AQL packet is created and filled with the information of that job, such as the size of work-group, address of GPU kernel program, arguments and so on. AQL packets are stored inside application queues and the HSA device is able to access these queues to get the job descriptions and carry out the computation.

Doorbell: A signal used to notify the computing devices that there are jobs waiting to be executed. When applications want to perform computation on a HSA device, it first fills out the AQL packet with required information then activates the doorbell signal to notify the computing device. Doorbell is a notification of the existence of a job. The job's scheduled runtime is determined by the device.

2.1 HSA Features

There are many required features for an HSA-compliant system [4], we discuss those features that are related to the virtualization effort.

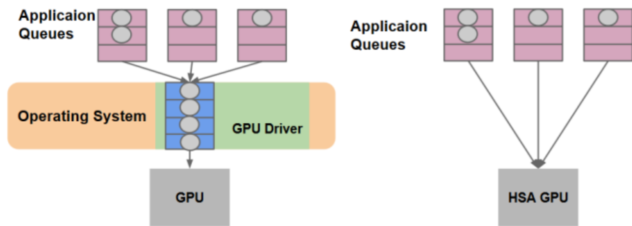

Figure 1. Job enqueuing steps in non-HSA (left) and HSA systems (right).

Shared virtual memory: Heterogeneous computing devices like GPU are integrated on the same bus and share the same virtual memory address space. Each process' virtual address is visible across CPU and other devices. Thus, computing devices can use the virtual address for computation directly without data copying between devices. This feature not only eliminates the data copying overhead but also reduces the programmer's burden to compress and decompress complex data structures as well.

I/O Page faulting: Prior to HSA, device DMA requires memory to be pinned and it could not be swapped out by the OS. This constraint is not practical in HSA since shared virtual memory across devices implies that devices may use all the memory space. If pinned memory is required, all process' address space must be pinned. Therefore, allowing I/O devices to generate page fault is necessary in HSA.

Cache coherence: Cache coherence is an important factor concerning a program's correctness. In HSA, all computing devices see the same memory space so keeping cache coherent is required, even though each computing device might have different cache systems.

User mode queuing: As Figure 1 shows, prior to HSA, every GPU job dispatched by applications must be passed through the GPU driver to be enqueued for execution. With user mode queuing, the GPU is aware of the address of the application queue and applications are allowed to dispatch jobs to the GPU directly without being trapped to a driver which reduces the latency of enqueuing jobs.

Memory-based signaling: This feature also aims at reducing the communication latency. An application assigns a memory address as a job-done listener and writes it into the AQL packet. With that address, a HSA device can directly signal the application or HSA runtime when a job is done rather than going through the traditional interrupt-based signalling.

The features listed above reduce both data communication latency and programing barriers. Moreover, the user mode queuing allows the GPU to access user level queues directly. Doorbell and memory-based signalling allow job dispatching and finishing to communicate without intervening with GPU drivers, which also reduces the CPU/GPU communication overhead as shown in Figure 2.

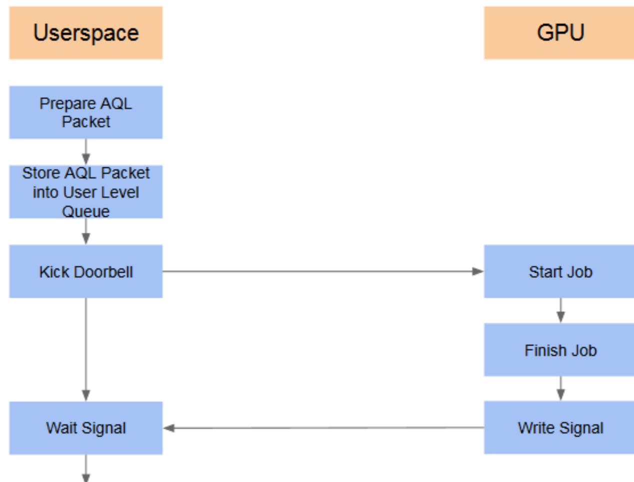

Figure 2. User mode queuing and memory based signaling eliminate GPU driver intervention.

2.2 AMD Kaveri Model

In this section, we describe how the five introduced features are implemented in the AMD Kaveri machine.

Shared virtual memory, I/O page faulting, and user mode queuing are co-implemented by the operating system and hardware. Cache coherence is implemented by hardware only in conventional processor designs. Memory-based signalling is achieved once shared virtual memory is realized. With shared virtual memory, the GPU is able to use the memory address designated for the signal (sent via AQL packet during job dispatching) to notify the user process when the dispatched job is finished (HSA runtime manages the function of waiting signals).

This work focuses on the virtualization of features co-implemented by OS and hardware, including the shared virtual memory, I/O page faulting, and user mode queuing.
There are two additional kernel modules in the OS which Kaveri runs on, the IOMMU [7] driver and kernel fusion driver (KFD) [8]. IOMMU is implemented for shared virtual memory and I/O page faulting. And it provides a mechanism called peripheral page request service (PPR) for fixing I/O page faults. KFD, on the other hand, is designed to support user mode queuing. These functions are discussed in more detail.

2.2.1 IOMMU in HSA

IOMMU is a hardware component designed to carry out address translation for I/O devices. On HSA-based systems, computing devices communicate with each other using the shared virtual memory. Since the addresses issued by computing devices are virtual, they must be translated into physical addresses. IOMMU carries out this address translation for computing devices like GPU in Kaveri. Furthermore,

since the virtual address space is shared between CPU and devices, the page table walked by IOMMU is the same as what is used by the CPU MMU.

When an application tries to use HSA computing devices, the driver gets the process page table and sets it to IOMMU. After proper configuration of the page table, computing devices are able to access process virtual address, as required by the shared virtual memory feature.

2.2.2 Peripheral Page Service Request (PPR)

Peripheral Page service Request (PPR) is introduced in AMD IOMMU to handle I/O page faulting in Kaveri. PPR is a mechanism that allows peripheral devices to issue requests to CPU for handling I/O page fault.

IOMMU performs a permission check and address translation when I/O devices attempt to access memory. If the page is not in the memory or the device does not have sufficient permission, IOMMU writes the faulting address, the faulting process ID and flags to PPR's log and issues a PPR interrupt to CPU. The PPR handler, which is called by the interrupt handler, reads PPR logs to find the corresponding process' memory control context. With the faulting address and memory control context, Linux API's get_user_pages can be used to grab the faulting page into memory. After the page fault is fixed, the PPR handler sends COMPLETE_PPR_REQUEST [7] command to IOMMU to finish this I/O page fault.

2.2.3 Kernel Fusion Driver (KFD)

KFD is implemented as a GPU driver for Kaveri to support the user mode queuing feature. The key point of this feature is to allow HSA computing devices, such as GPU, to know where the application queues are. One simple approach, as implemented in KFD, is to send the address of the application queue to GPU, and let the hardware manage the queue binding. Whenever an application wants to perform a computation on a computing device, it creates a user mode queue and sends the address of that queue to KFD. KFD, acting as an agent between application and GPU hardware, then writes the receiving address to GPU's configuration register. This process eventually binds user mode queue to GPU hardware. This queue binding process executes only once for each user mode queue. After queue binding, GPU knows the exact address of the application queue and is therefore able to access the queue without driver's intervention.

One more thing KFD does during queue initializing is remapping a doorbell from physical address to virtual address. Since the doorbell is a hardware signal, which is located in a fixed memory-mapped I/O (MMIO) region, a memory remapping has to be done so the user-space application can kick the doorbell directly. With this memory remapping mechanism, application programs would not be constrained by the OS for dispatching GPU jobs.

After these initialization, the application can kick the doorbell to notify the GPU to work and the GPU can get AQL packets from user level queues to execute jobs. The driver's interventions are eliminated in the job dispatching path, which effectively reduces the CPU/GPU communication latency.

3. Design of a HSA-aware Hypervisor

In this section, we discuss how to virtualize the three HSA features mentioned in Section 2.2, the shared virtual memory, I/O page faulting, and user mode queuing,

The focus of this work is on memory and I/O virtualization techniques in the hypervisor design since the shared virtual memory and I/O page faulting features are the main concerns. I/O virtualization helps to realize communication between guest application and the computing device such as the queue binding process.

We adopt the shadow page table approach to carry out memory virtualization and the VirtIO framework [9] to implement I/O virtualization. Modification is required for both KFD and IOMMU drivers. The system architecture and detailed implementation are presented in Section 4. Discussions about why shadow page table is chosen rather than the two-level address translation [10] and why KFD needs to be modified are also explained in Section 4.4 and 4.3 respectively.

3.1 Shared Virtual Memory Virtualization

As described in Section 2.2.1, IOMMU inside Kaveri supports shared virtual memory between the CPUs and the GPU. Kernel programs executing on GPU reside in virtual address space and IOMMU is responsible for translating the virtual addresses issued by GPU into physical addresses.

In a non-virtualized environment, IOMMU shares the same process page table with CPU MMU to conduct the virtual address to physical address translation. In a virtualized environment, however, the addresses issued by GPU are actually in the Guest Virtual Address (GVA) space, hence requiring IOMMU to translate the issued GVA into Machine Physical Address (MPA). Two common techniques used to construct this GVA-to-MPA translation are Shadow Page Table (SPT) and two-level address translation (such as the Extended Page Table approach used in Intel VT-x and Nested Page Table in AMD-V). In this work the shadow page table mechanism is adopted due to some limitations in Kaveri which will be discussed in Section 4.4 that forbids the use of the two-level address translation mechanism, despite its popularity.

Shadow page table has already been implemented in the original KVM code. SPT of each guest OS is constructed by the hypervisor as soon as the guest OS is started. CPU MMU walks the SPT to translate GVA to MPA during the guest OS

execution. The key to allow computing devices to support the shared virtual memory feature is to let IOMMU walk the same SPT used by the CPU MMU, where all necessary information for computing devices to perform GVA to MPA translation is provided.

3.2 I/O Page Faulting Virtualization

PPR, as described in Section 2.2.2, is a mechanism that I/O devices use to request page fault handling. PPR logs contain the faulting address and faulty process attributes.

When the GPU executes a guest process' kernel program, IOMMU walks its SPT to carry out the address translation. If a page does not exist in system memory or there is a permission violation, a PPR is issued and the faulty GVA is written into PPR logs. Our hypervisor will first notify the corresponding guest OS to get the faulty GVA as well as the process information, and ask the guest OS to fix the guest level page table. After the guest level page table is fixed, the SPT page fault handler is called to fix the SPT (which IOMMU actually walks). Finally, a finishing command is sent to IOMMU to complete the handling of a guest I/O page fault.

Shadow PPR and VirtIO-IOMMU are implemented in our hypervisor to construct the guest I/O page fault handler. These two modules will be presented in Section 4.2.

3.3 User Mode Queuing Virtualization

As described in Section 2.2.3, KFD sets the address of application queue to GPU device and remaps the doorbell from physical address to the process virtual address. After these initializations, the job dispatching can be conducted by kicking the doorbell.

With the user mode queuing feature, the virtualization of GPU job dispatching is simpler than normal GPU virtualization. Traditionally, GPU virtualization needs to map guest process queues to the hardware queue and is trapped every time a guest process dispatches a job to GPU.

For HSA, there are no hardware queues, only the user mode queues, so the hypervisor does not need to take care of the queue mapping. A hypervisor only needs to set the GVA of a guest application's queue to GPU and remaps the doorbell MPA back to GVA. Since the shadow page table has been set to IOMMU, GPU can therefore access GVA of guest application queues. After this queue binding, guest applications and the GPU can communicate through the application queue, doorbell and job-done signals without being limited to the hypervisor.

In this work a VirtIO-KFD module is implemented, which cooperates with the native KFD to manage guest application queue binding. The implementation details will be presented in Section 4.3.

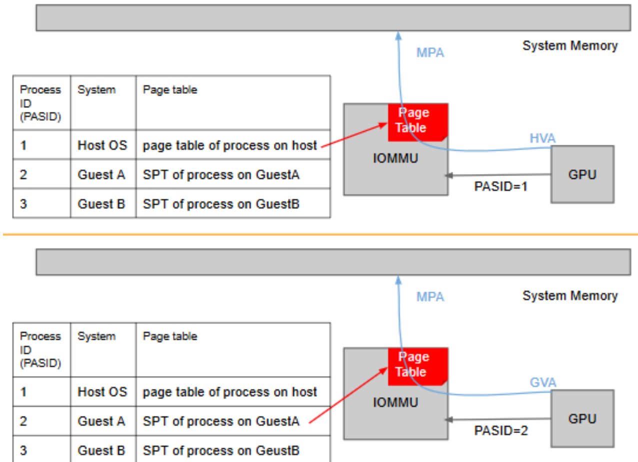

Figure 3. Illustration of GPU executing kernel programs from different systems.

3.4 GPU Sharing

For GPU sharing, two prerequisites have to be met: (1) All processes in multiple guest OSes should be able to dispatch jobs and (2) GPU kernel programs from processes of different guest OSes can be fairly executed, or at least behave like multiple host processes sharing a GPU. We will show how these two requirements are achieved in our design.

To begin, the user mode queuing feature implies the GPU will record information about the user mode queues and which processes they belong to. In a virtualized environment, multiple queues from different guest OSes are simply viewed as different queues on a normal host system. As long as the doorbell addresses are correctly remapped to guest processes' address space, applications from multiple guest OSes can notify the GPU about dispatching jobs.

When the doorbell is kicked, the GPU tries to access the application queue and gets AQL packets from it. An IOMMU page table walk on a properly set page table (the queue address binding to GPU is in virtual address space) allows the GPU to access the application queues. Also, since the SPTs belonging to each guest process are visible to IOMMU, kernel programs from processes of multiple guest OSes can be successfully executed. Figure 3 depicts the illustration of GPU sharing.

As described above, GPU can be shared between processes in different guest OSes and even the host OS, and the job scheduling is managed by the GPU hardware. In Section 5, the experiment shows the performance of multiple jobs dispatched simultaneously by processes from different guest OSes and from the host OS.

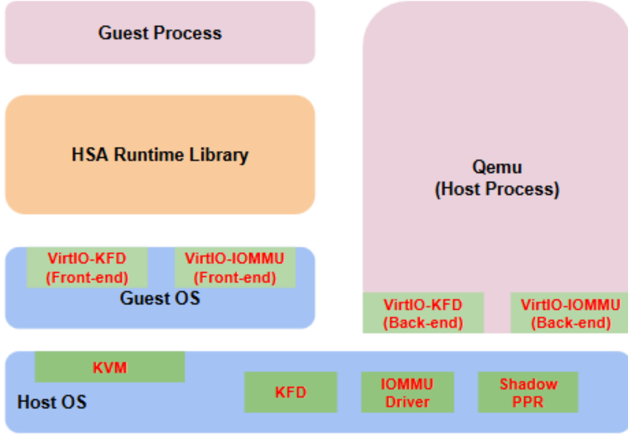

Figure 4. System architecture of our hypervisor implementation.

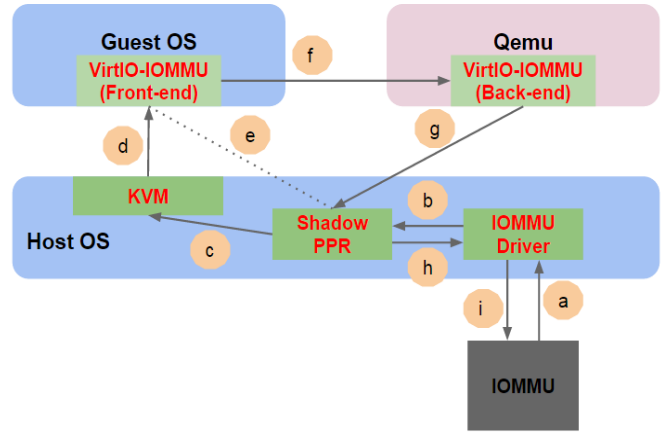

Figure 5. Flow of guest I/O page fault handling.

4. Implementation

Figure 4 shows the system architecture of our implementation. Several modules are modified or created for the hypervisor such that guest OSes can benefit from HSA. In this section, a brief overview of the role of each component is given. The implementations of the three necessary virtualized HSA features are then presented.

KFD acts as an interface between HSA runtime and HSA devices. We implemented a VirtIO-KFD module inside the guest OS. Guest applications and runtime see VirtIO-KFD as how they see KFD in a native environment. The VirtIO-KFD collaborates with native KFD to virtualize the user mode queuing feature.

KVM, the hypervisor that our implementation is based on, manages the SPTs for every processes in guest OSes. The IOMMU driver gets SPTs from KVM and sets it to IOMMU when guest processes are initialized in order to virtualize the shared virtual memory.

Shadow PPR is a newly created module that preserves PPR (peripheral page request) logs pertaining to the I/O page faults caused by guest processes' kernel programs and cooperates with VirtIO-IOMMU as the guest I/O page fault handler. Since PPR logs are stored in a MMIO region, the guest system cannot access it directly. Therefore, the shadow PPR is required for the I/O page faulting feature.

4.1 Shared Virtual Memory Virtualization

Shadow page tables are created and maintained by KVM. Basically, the page table structure is consistent between MMU and IOMMU (the second level page table, translating GPA to MPA, in two-level translation techniques is a little different but does not matter in our implementation since SPT is adopted). Both the page tables of host processes and the SPT use the same page table structure so the only thing that needs to be changed is to get the address of the shadow page table from KVM and pass it to the driver for setting to IOMMU.

We slightly modify KVM by creating an interface for IOMMU to acquire SPTs of guest processes that attempt to use GPU. Also the IOMMU driver is modified to query SPTs when the guest processes are initialized.

4.2 I/O Page Faulting Virtualization

The VirtIO-IOMMU and Shadow PPR are implemented in order to handle guest I/O page faults. We'll describe how these components are initialized and a detailed flow of guest I/O page fault handling follows.

When a guest OS is ready to boot up, Shadow PPR allocates a region inside the host kernel for storing PPR logs for guest I/O page faults. VirtIO-IOMMU also allocates a region and does memory mapping from this region to the Shadow PPR region inside the host kernel. After this memory mapping, VirtIO-IOMMU can access Shadow PPR region to get logs without any trap.

The guest I/O page fault handling flow is illustrated in Figure 5. (a) When PPR happens, CPU is interrupted by IOMMU and ends up calling the PPR handler. The handler then fetches PPR logs to obtain the faulting process attributes and the faulting address. (b) The process attributes will be used to identify whether this fault is caused by the guest process. If so, the PPR handler stores the logs into the log region of Shadow PPR. (c~d) Sends a virtual interrupt to the guest OS by the IRQFD [11] mechanism, which allows the host kernel module to send an interrupt to the guest OS via KVM. (e) The guest PPR handler implemented inside VirtIO-IOMMU will be called by the guest interrupt handler. It gets PPR logs from Shadow PPR. This part does not cause traps as previous described. Given these logs, Linux API get_user_pages will be used to fix the I/O page fault. (f~i) After fixing the page fault, a finishing command is sent to Shadow PPR via VirtIO-IOMMU back-end driver. Shadow

PPR will first call KVM to synchronize the shadow page table, since only the guest side page tables are fixed in previous steps, and then Shadow PPR calls the IOMMU driver to send the COMPLETE_PPR_REQUEST command to IOMMU to finish this guest I/O page fault.

4.3 User Mode Queuing Virtualization

All HSA related configurations, such as queue creation and destroying, are using Linux IOCTL commands passed to KFD and set to HSA devices. We implemented a VirtIO-KFD front-end driver to replace the original KFD inside a guest OS and receive commands sent from guest user-space. There are front-end and back-end drivers in the VirtIO framework. The front-end driver receives I/O requests from the guest OS and passes them to the back-end driver. Usually the back-end driver calls the host's driver to satisfy the guest's I/O requests.

The implementation of VirtIO-KFD is not complex: the IOCTL commands sent from the user-space will carry arguments, such as the address of application queues for a queue-creation command, are passed to the host KFD via the VirtIO framework. The host KFD, with our modification to accept commands sent from guest processes, will then set these configurations to the GPU hardware. The host KFD will also call the IOMMU driver to get the shadow page table of the related guest process from KVM and set it to IOMMU. Another major concern is the doorbell address mapping. Since the major advantage of user mode queuing is to eliminate the driver's intervention after process initialization, the doorbell address is memory remapped from MPA to GVA. This involves two memory mappings, from MPA to the host virtual address (HVA), which is conducted by the host KFD, and from GPA (can be obtained simply by a linear address translation from HVA) to GVA, which is carried out by VirtIO-KFD. With such efforts, the user mode queuing feature can be successfully virtualized.

Finally, our modification to the host KFD is described. In the original design, the host KFD assigns a unique process address space ID (PASID) to each process that uses it. PASIDs are tied up with page tables in IOMMU to achieve the shared virtual memory feature, as previously illustrated in Figure 3.

In our virtualized environment, it is the VirtIO-KFD back-end driver that calls the host KFD on the behalf of the guest processes, and only one VirtIO-KFD back-end process per guest OS gets an assigned PASID. However, there may be multiple guest processes in a single guest OS that tries to use HSA devices. This causes an asymmetric mapping problem between PASID and the guest processes. To solve this issue, a supplementary VM_CREATE_PROCESS command is appended to create PASIDs for guest processes.

Moreover, the host KFD uses the process memory control context, mm_struct in Linux, to identify the relation between process and PASID. Whenever the host KFD gets an IOCTL command, it gets the mm_struct of the demanding process to figure out the corresponding PAISD. Under this design, the guest application's PASID cannot be recognized since only the VirtIO-KFD back-end process is able to call the host KFD, and the VirtIO-KFD's memory control context will be obtained rather than that of the guest process. To fix this problem, we added a set of new IOCTL commands for the guest-process-related configurations. For instance, VM_CREATE_QUEUE and VM_SET_MEMORY_POLICY are the commands corresponding to CREATE_QUEUE and SET_MEMORY_POLICY commands in the original code. The guest process' memory control context will be carried with these newly created commands so the host KFD can obtain the PASIDs belonging to guest processes and bind it to GPU and IOMMU.

4.4 Issues about Implementing IOMMU Two-level Address Translation

In the early stage of this work, we planned to use two-level address translation instead of shadow page table for virtualizing the shared virtual memory feature. Two-level address translation is supported by hardware, yields lower latency in general, is more advanced in virtualization designs, and is more widely adopted by mainstream hypervisors. As for our target machine, the IOMMU in Kaveri supports two-level address translation as it has been supported in AMD-v, the hardware virtualization extension of AMD processors. So two-level address translation seems feasible and is the first, and may be the best, choice to implement this work.

However, some limitations in the Kaveri machine prevent this approach from working. In Kaveri there are two different paths for translating a GPU virtual address: the IOMMU, and the GPUVM, as illustrated in Figure 6. A basic difference between these two paths is that IOMMU is used to translate the user space address while GPUVM is used to translate kernel space address.

User space address translation can be comprehended easily. The user level queues and GPU kernel programs reside in the user address space. On the other hand, in the kernel space address, the memory queue descriptor (MQD) is allocated and manipulate by KFD inside the host kernel. The MQD is used for user mode queue binding. It contains attributes of user level queues, such as address of queue and size of queue and will be sent to GPU for bind user mode queues. In our implementation, the host KFD is responsible for creating MQDs for both the guest and host user queues since only the host KFD can communicate with GPU. There is also a possible device pass-through [12] like approach discussed later to let guest OS manipulates GPUVM and creates MQD itself.

During user mode queue initializing, the host KFD first fills the attributes of a user queue to MQD and then performs

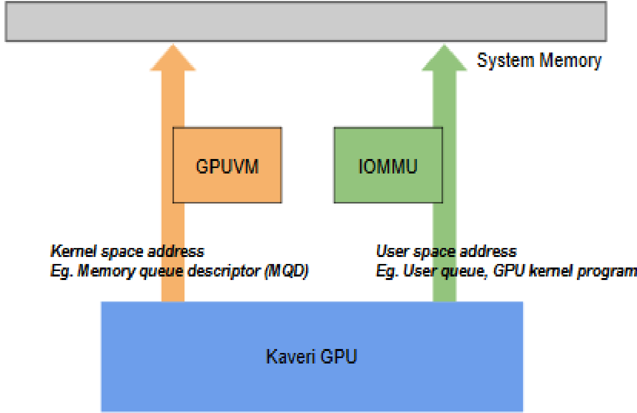

Figure 6. Kaveri GPU address translation components.

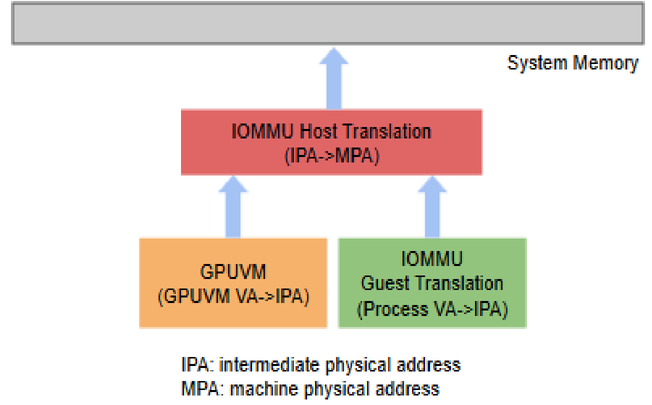

Figure 7. Kaveri GPU address translation path.

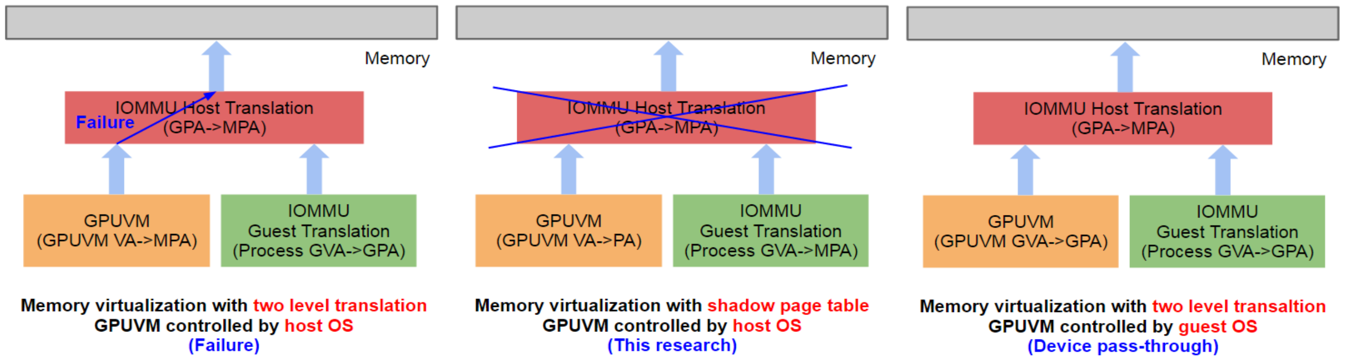

Figure 8. Comparison between different approaches

a memory mapping from the host kernel space address of MQD to GPUVM virtual address space and then sets the GPUVM virtual address (GPUVM VA) to GPU. Once GPU is kicked to execute, it tries to access the MQD corresponding to the process who kicks it and issues the GPUVM VA of that MQD. GPUVM hardware translates the GPUVM VA to a physical address so the GPU can access the MQD and get the address of a user level queue from it. After this, GPU is then able to access the user queue and get AQL packets for execution.

For implementing HSA virtualization with two-level address translation technique, the two-level translation of IOMMU must be enabled. Both of the outputs of GPUVM and IOMMU's first level translation go through the second level translation of IOMMU as shown in Figure 7. This means that if IOMMU two-level translation is enabled, both of the inputs of GPUVM and IOMMU are translated twice. It is reasonable that the input of IOMMU is translated twice, since the guest virtual address needs a GVA-to-GPA-to-MPA translation. For GPUVM, however, the input is GPUVM VA and it should only be translated into MPA with one level translation. If two-level translation is enabled, the GPUVM side translation will cause failures, and this is why two-level translation does not work in this implementation, as shown in the left-most figure of Figure 8.

The problem described above is caused by setting GPUVM VA of MQDs to GPU. However, if the guest OS is able to control GPUVM and create MQDs as shown in the rightmost figure of Figure 8, then the address of MQDs set to GPU are in the guest GPUVM virtual address space (GPUVM GVA), and it would need two-level address translation. The approach that lets the guest OS control the GPUVM is basically a device pass-through technique, but device pass-through is unsuitable for fair GPU sharing, so the shadow page table approach was adopted in this work. The implementation of device pass-through and the evaluation of its impact on performance are planned for future work.

5. Evaluation

In this section, we present the results of our HSA-aware hypervisor. The evaluation mainly focus on the performance comparison between native and guest's computation on GPU. The results are classified into queue initialization time and GPU kernel execution time. Overheads of VirtIO-KFD are measured by initialization time and the overheads of the shadow page table and guest I/O page fault handling are

Table 2. The set of benchmarks in our experiments.

Benchmark Name	Input Parameters
BinarySearch	array-length=100,000,000
FastWalshTransform	array-length=65536
BitonicSort	array-length=65536
FloydWarshall	nodes=3000
MatrixMultiplication (long)	matrix-a-height=5000 matrix-a-width=5000 matrix-b-width=5000
MatrixMultiplication (short) (only use in Section 5.4)	matrix-a-height=2000 matrix-a-width=2000 matrix-b-width=2000
MatrixTranspose	matrix-height=8192 matrix-width=8192
MonteCarloAsian	steps=512

measured by GPU kernel execution time. Furthermore, the performance of GPU programs dispatched simultaneously by processes from different guest OSes and the host process are also provided.

5.1 Experiment Configuration

The experiment hardware platform chosen for this experiment is AMD Kaveri A10-7850K APU, the first HSA-compliant machine, including the AMD steamroller processor with 4 CPU cores (running at 3.7Ghz), 8G system memory, and Radeon R7 GPU with 512 cores. Both the host and guest OS ran 64bit Ubuntu 14.04 LTS with a Linux 3.14.11 kernel released by HSA foundation and modified by us. The guest OSes were allocated with 1 VCPU and 4G system memory.

We used the AMD OpenCL SDK [13] as our test suite. The benchmarks and input parameters are listed in Table 2. The POCL-HSA [14] was adopted as an OpenCL runtime implementation that end up calling HSA runtime.

5.2 Queue Initialization Time

Figure 9 shows the time spent on HSA-related initialization and user mode queue creation, from the HSA runtime API hsa_init to hsa_queue_create. In this process, many attributes are sent to KFD and configured to GPU.

The performance drop of the guest system is around 30% in every benchmark. This is due to the propagation delay of KFD IOCTL commands from VirtIO-KFD front-end to back-end and then to the host KFD. This path also incurs overhead of VM world switch from the guest mode to the host mode. Moreover, the guest doorbell memory mapping from MPA to GVA takes more time than only MPA to HVA in the native scenario.

However, this initialization process only performs once for every user mode queue. As long as the queues exist, the application can dispatch jobs without paying such overhead.

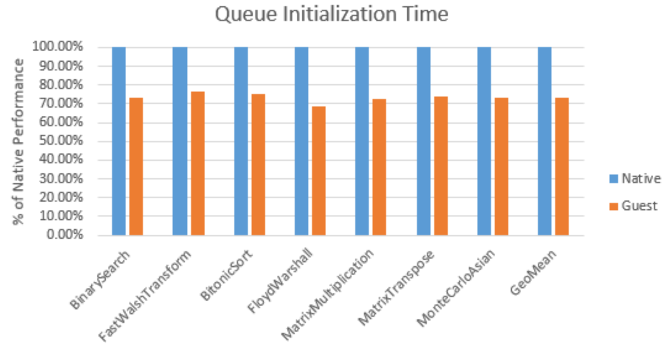

Figure 9. Performance comparison in queue initialization time, normalizing against the native scenario.

In comparison with GPU execution, the performance drop during initialization time would not be a great concern.

5.3 GPU Execution Time

In the shadow page table implementation, both the guest and the native GPU execution go through one level address translation in IOMMU. The factor that may cause performance difference is the I/O page fault handling. The GPU execution time and the I/O page fault handling time of every benchmark are presented in Table 3. Figure 10 shows the GPU virtualization performance normalized against the native run.

To analyze the performance overhead caused by PPR handling, Table 3 shows that native PPR time account for almost 0% of native GPU execute time and guest PPR time accounts for 0~5% for every benchmark. Though the guest PPR handling incurs more overhead than native PPR, the performance influence is still marginal. For Figure 10, it shows that the guest GPU execution achieves nearing 95% of native GPU performance in most benchmarks. The two anomalies, FastWalshTransform and BitonicSort, however, give around 88% of native performance. This is caused by the overhead of multiple job enqueuing and dispatching. Recall the GPU jobs execution flow in Figure 2. There is theoretically neither trap nor VM world switch during job dispatching and finishing because of the user mode queuing and memory-based signalling features. While taking a deeper look inside the job finishing situation, VM world switches may occur due to prolonged signal waiting which may make VM idle or exhaust the time slice allocated. As Figure 11 shows, these world switches do not affect the performance of GPU programs but do slow down the process when the guest application gets signalled and enqueues the next job.

In our test suite, FastWalshTransform, BitonicSort, FloydWarshall, and MonteCarloAsian enqueue and kick GPU many times while BinarySearch, MatrixMultiplication, and MatrixTranspose only activate once. The delay of the application which enqueues the next job is so negligible that

Table 3. GPU execution time and I/O page fault handling time.

Benchmark Name	GPU Execution Time (sec)		Number of I/O Page Fault		I/O Page Fault Handling Time (sec)		I/O Page Fault Handling Time of GPU execution time (%)	
	Native	Guest	Native	Guest	Native	Guest	Native	Guest
BinarySearch	0.011	0.011	0	0	0	0	0.00%	0.00%
FastWalshTransform	0.002	0.002	0	1	0	0.00004	0.00%	1.95%
BitonicSort	0.014	0.016	0	1	0	0.00014	0.00%	0.85%
FloydWarshall	16.094	16.603	75	4730	0.00037	0.30053	0.00%	1.81%
MatrixMultiplication	8.012	8.286	52	167	0.00027	0.00852	0.00%	0.09%
MatrixTranspose	0.502	0.538	114	366	0.00032	0.02485	0.06%	4.62%
MonteCarloAsian	17.458	18.342	6	113	0.00024	0.04454	0.00%	0.24%

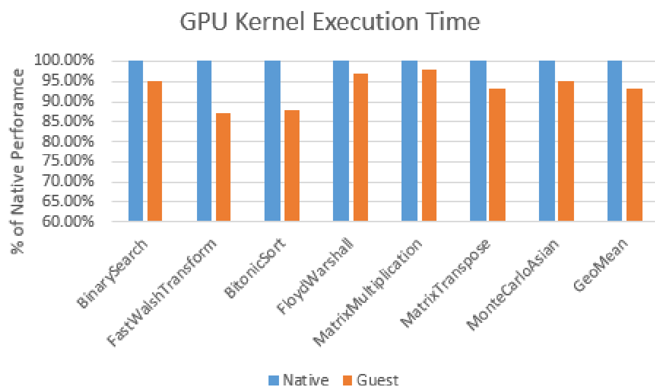

Figure 10. Performance comparison in GPU kernel execution time, normalizing against the native scenario.

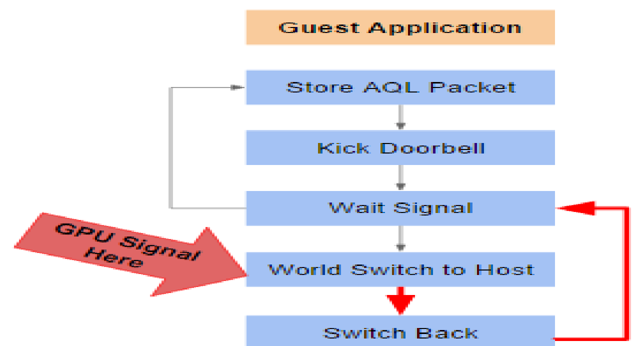

Figure 11. The delay (depicted in red arrow) of enqueuing next job caused by unintentionally world switch.

only the benchmarks with short execution time, FastWalshTransform and BitonicSort, may suffer from it. The overhead of delayed job enqueuing is amortized in the cases for benchmarks with longer execution time.

To sum up, the GPU job dispatched by guest processes on our hypervisor achieve around 95% of native GPU performance in the long-running benchmarks. Though some overhead incurs in short-running benchmarks, it can still achieve near 88% of native performance.

5.4 Multiple GPU Execution

In this subsection, we present multiple GPU execution in the following three scenarios, where the first two scenarios are conducted with process numbers of 1, 2 and 4: (1) All processes execute the MatrixMultiplication with same input parameters. (2) Same benchmark is used but with different input parameters. (3) Two processes execute a long-running and a short-running job respectively. We analyse the virtualization overhead in sharing a GPU in the first scenario and where the degree of sharing of GPU is evaluated in the last two scenarios.

Two process configuration groups are tested: (1) Combination of guest processes from different guest OSes and a host process (2) Mix of all host processes. Through the first configuration group we demonstrate that a GPU can be shared between multiple guest OSes and the host OS. The result of the second configuration group is used as a reference to compare the GPU performance across native and guest execution environments.

In Figure 12, 13 and Table 4, 5 the VM{N} means the process of the Nth guest OS and Host represents process of the host OS. The Native{N} also stands for the process of the host OS but it is in different groups with the Host bar.

The results of the first scenario are shown in Figure 12. It is observed that the GPU execution time scales up as the number of process increases. This corroborates that the GPU in Kaveri is able to compute multiple kernel programs simultaneously and fairly (some limits are discussed in later two experiments). As for the virtualization overhead, the relative performance drop between group 1 and group 2 is within 5%, which remains the same as the result in single kernel program execution scenario described in Section 5.3. This shows that there is almost no additional overhead in sharing GPU between multiple guests OSes based on our implementation.

Table 4. Multiple GPU execution in short MatrixMultiplication (with blue background color) and long MatrixMultiplication.

GPU Execution Time (sec)	Group 1				Group 2			
	VM1	VM2	VM3	Host	Native1	Native2	Native3	Native4
1 Process	0.55				0.53			
2 Processes	3.03	12.63			2.98	12.57		
4 Processes	5.31	5.27	24.81	24.49	5.13	5.06	23.85	23.81

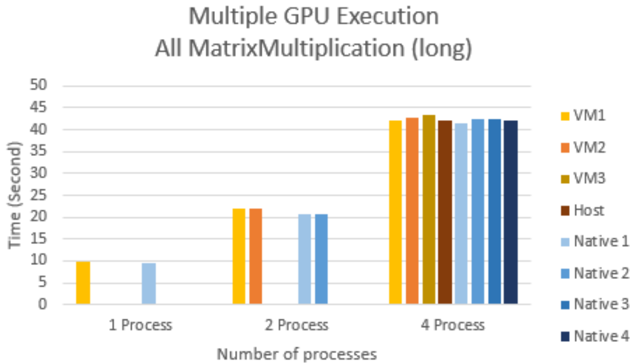

Figure 12. Multiple GPU execution time, all processes execute MatrixMultiplication with large input parameter.

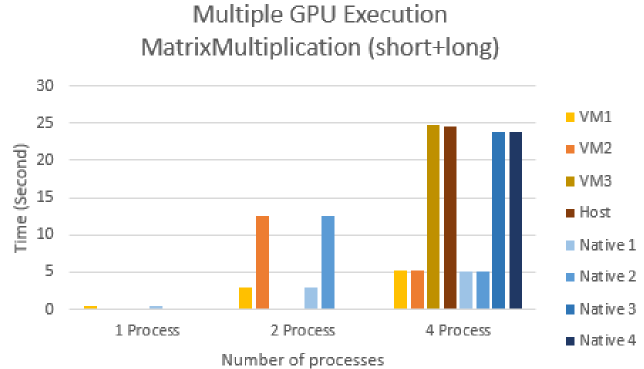

Figure 13. Multiple GPU execution time, left bars in one group are the execution time of short MatrixMultiplication, right bars in another group are that of long MatrixMultiplication.

Table 5. Multiple GPU execution time in MatrixTranspose (with blue background color) + MatrixMultiplication.

GPU Execution Time (sec)	Group 1		Group 2	
	VM1	VM2	Native 1	Native 2
1 Process	0.54		0.50	
2 Processes	11.35	10.74	11.29	10.70

In the second scenario, we first launched the MatrixMultiplication with large input parameters, and then launched that with small input parameters (the inputs are listed in Table 2). The result in Table 4 and Figure 13 shows that though a longer job is dispatched and executed beforehand, the shorter job can still be computed and finish earlier, meaning that the GPU computation power is indeed shared by multiple processes.

The last scenario is conducted by the launch of MatrixMultiplication, a longer job, followed by the launch of MatrixTranspose, a shorter job. As Table 5 shows, the GPU execution time of MatrixTranspose is about the time MatrixMultiplication runs plus the time MatrixTranspose runs. This means that the MatrixTranspose job is blocked by the previously launched MatrixMultiplication. From Table 5, we can also observe that the blocking of GPU job not only happens in configuration group 2 but group 1 as well, which means that the GPU hardware does not fully support fair job scheduling.

To conclude these experiments, the results show that the GPU in Kaveri seems able to compute jobs simultaneously with fair scheduling if the jobs are in the same kernel program. Otherwise, successive jobs will be blocked by previously dispatched jobs. The scheduling mechanism of the GPU in Kaveri is undocumented so we cannot verify its precise job scheduling policy. But regarding the goal of system virtualization, our implementation allows multiple guest OSes to share the GPU in a way identical to how multiple host processes behave with nearing no additional overhead. Moreover, we believe our work can be applied to other machine that supports full GPU job scheduling mechanism so that guest processes can get more benefits from GPU virtualization.

6. Related Work

Most of the GPU virtualization works are implemented in device pass-through and API forwarding. Device pass-through is a naïve approach that allows a guest system to access one dedicated GPU directly without modifying the guest GPU driver. The I/O virtualization hardware extension such as Intel VT-d [15] or AMD IOMMU [8] are required for the implementation of the pass-through approach. However, device pass-through suffers from an inability to share GPU in a fair manner. To solve this problem, Intel gVirt [16] and NVIDIA VGX [17] were recently proposed to not only allow virtual machines to directly access GPU, but also share it as well. These two approaches, however, require proprietary GPU information and additional hardware design so it is hard to be implemented by non-vendor developers.

API forwarding, on the other hand, modifies the guest runtime library to forward API calls to the hypervisor for further virtualization. The rCUDA [18], vCUDA [19], GViM [20] forward guest level CUDA APIs to the underlying simulation stack. The virtio-CL [21] is another API forwarding implementation for OpenCL. Xen3D [22] and VMGL [23] are implementations for OpenGL. The difficulty of API forwarding is its lack of fidelity, where the description of whether the features supported in virtualized and native environment should be consistent [24]. Since it is difficult to virtualize all APIs inside a guest system, to maintain the consistency is a great challenge.

Our implementation is a para-virtualization that forwards OS level commands only, without modification to the runtime library. Since OS level commands are used only during user mode queue initialization, it is simpler to virtualize GPU at this layer. Furthermore, indebted to user mode queuing, applications can forward their jobs to GPU directly without additional virtualization effort. GPUvm [25] provides full- and para-virtualization design that virtualize the GPU in hypervisor level. It virtualizes the GPU command channel and GART table so that GPU can be shared between multiple guest OSes. The virtualization of GART table requires guest GPU virtual address to be translated to GPU physical address. The main difference between our work and GPUvm is in virtual memory management: how page table and I/O page faults are handled. For page table, since HSA is a shared virtual memory architecture, the shadow page table is updated along with guest process execution. On the other hand, GPUvm's shadow page table is updated when data copy commands are sent. As for I/O page faults, GPUvm needs to scan the entire page tables upon TLB flush since it does not support I/O page faults, where our work does include a framework to support I/O page faults.

7. Conclusion

This work presents the concept, the design and a KVM-based implementation of HSA system virtualization. Though our implementation targets at the AMD Kaveri machine, we believe this work can be applied to other architectures with similar features like the shared virtual memory, the user level queues and the memory-based signals. Moreover, we demonstrate that the sharing of a GPU between multiple guest OSes and the host OS under the HSA compliant system can be accomplished with minor virtualization overhead. The results show that the performance of guest's kernel programs achieves near 95% of native GPU performance in most of the tested benchmarks, especially those with longer execution time.

As for future work, the device pass-through approach and performance comparison are planned. We will also port our work into the latest KFD version and run our hypervisor on the new HSA machine, AMD Carrizo, to measure the effort of applying our implementation to other HSA machines and conduct more investigation on GPU sharing issues.

References

[1] General-Purpose Computation on Graphic Hardware, http://gpgpu.org/.

[2] Nvidia CUDA, http://www.nvidia.com/object/cuda_home_new.html.

[3] J.E. Stone, D. Gohara, G. Shi, OpenCL: a parallel programming standard for heterogeneous computing systems, Comput. Sci. Eng. 12 (2010) 66–73.

[4] Heterogeneous System Architecture (HSA), http://www.hsafoundation.com/

[5] AMD Kaveri, http://www.amd.com/en-us/products/processors/desktop/a-series-apu.

[6] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In OLS 2007: Proceedings of the 2007 Ottawa Linux Symposium.

[7] AMD I/O Virtualization Technology (IOMMU) Specification, http://developer.amd.com/wordpress/media/2012/10/488821.pdf, 2012.

[8] Kernel Fusion Driver source code, https://github.com/HSAFoundation/HSA-Drivers-Linux-AMD.

[9] Rusty Russel. virtio: towards a de-facto standard for virtual I/O devices. In Operating Systems Review, 2008.

[10] AMD. AMD64 Virtualization Codenamed "Pacifica" Technology: Secure Virtual Machine Architecture Reference Manual, May 2005.

[11] IRQFD, https://lwn.net/Articles/329837/.

[12] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, Y. Jiang, Towards high-quality I/O virtualization. SYSTOR 2009.

[13] AMD OpcnCL APP SDK, http://developer.amd.com/tools-and-sdks/opencl-zone/.

[14] Portable Computing Language (pocl) for HSA, http://pocl.sourceforge.net/docs/html/hsa.html.

[15] D. Abramson, J. Jackson, S. Muthrasanallur, G. Neiger, G. Regnier, R. Sankaran, I. Schoinas, R. Uhlig, B. Vembu, and J. Wiegert. Intel virtualization technology for directed I/O. Intel Technology Journal, 10, August, 2006.

[16] TIAN, K., DONG, Y., AND COWPERTHWAITE, D. A full gpu virtualization solution with mediated pass-through. In Proc. USENIX ATC (2014).

[17] NVIDIA. NVIDIA GRID VGX SOFTWARE. http://www.nvidia.com/object/grid-vgx-software.html, 2014.

[18] DUATO, J., PENA, A. J., SILLA, F., MAYO, R., AND QUINTANA-ORTI, E. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In Proc. of IEEE Int'l Conf. on High Performance Computing Simulation (2010), pp. 224–231.

[19] SHI, L., CHEN, H., SUN, J., AND LI, K. vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines. IEEE Transactions on Computers 61, 6 (2012), 804–816.

[20] GUPTA, V., GAVRILOVSKA, A., SCHWAN, K., KHARCHE, H., TOLIA, N., TALWAR, V., AND RANGANATHAN, P. GViM: GPU-Accelerated Virtual Machines. In Proc. of ACM Workshop on System-level Virtualization for High Performance Computing (2009), pp. 17–24.

[21] Tien, Tsan-Rong, and Yi-Ping You. "Enabling OpenCL support for GPGPU in Kernel-based Virtual Machine." Software: Practice and Experience 44.5 (2014): 483-510.

[22] C. Smowton. Secure 3D graphics for virtual machines. In EuroSEC'09: Proceedings of the Second European Workshop on System Security. ACM, 2009, pp. 36-43.

[23] H. A. Lagar-Cavilla, N. Tolia, M. Satyanarayanan, and E. D. Lara. VMM-independent graphics acceleration. In Proc. VEE (2007), pp. 33-43

[24] M. Dowty and J. Sugerman. GPU Virtualization on VMware's Hosted I/O Architecture. ACM SIGOPS Operating Systems Review, 43:73–82, July 2009.

[25] SUZUKI, Y., KATO, S., YAMADA, H., AND KONO, K. GPUvm: why not virtualizing gpus at the hypervisor? In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (2014), USENIX Association, pp. 109–120.