International Journal of Software and Informatics, ISSN 1673-7288
http://www.ijsi.org, ijsi@iscas.ac.cn, +86-10-62661048
IJSI, 2021, 11(2): 169–193, doi:10.21655/ijsi.1673-7288.00248

International Journal of Software and Informatics

Research Article

Accelerator Virtualization Framework Based on Inter-VM Exitless Communication

Dingji Li (李鼎基)[1], Zeyu Mi (糜泽羽)[1], Baodong Wu (吴保东)[2], Xun Chen (陈逊)[2], Yongwang Zhao (赵永望)[3], Zuohua Ding (丁佐华)[4], Haibo Chen (陈海波)[1]

[1] (School of Software, Shanghai Jiao Tong University, Shanghai 200240, China)

[2] (Sensetime, Beijing 100080, China)

[3] (School of Cyber Science and Technology, Zhejiang University, Hangzhou 310007, China)

[4] (School of Information Science and Technology, Zhejiang Sci-Tech University, Hangzhou 310018, China)

Corresponding author: Haibo Chen, haibochen@sjtu.edu.cn

Abstract The increasing deployment of artificial intelligence has placed unprecedent requirements on the computing power of cloud computing. Cloud service providers have integrated accelerators with massive parallel computing units in the data center. These accelerators need to be combined with existing virtualization platforms to partition the computing resources. The current mainstream accelerator virtualization solution is through the PCI passthrough approach, which however does not support fine-grained resource provisioning. Some manufacturers also start to provide time-sliced multiplexing schemes and use drivers to cooperate with specific hardware to divide resources and time slices to different virtual machines, which unfortunately suffer from poor portability and flexibility. One alternative but promising approach is based on API forwarding, which forwards the virtual machine's request to the back-end driver for processing through a separate driver model. Yet, the communication due to API forwarding can easily become the performance bottleneck. This paper proposes Wormhole, an accelerator virtualization framework based on the C/S architecture that supports rapid delegated execution across virtual machines. It aims to provide upper-level users with an efficient and transparent way to accelerate the virtualization of accelerators with API forwarding while ensuring strong isolation between multiple users. By leveraging hardware virtualization features, the framework minimizes performance degradation through exitless inter-VM control flow switch. Experimental results show that Wormhole's prototype system can achieve up to 5 times performance improvement over the traditional open-source virtualization solution such as GVirtuS in the training test of the classic model.

Keywords virtualization; accelerator; artificial intelligence; delegated execution; inter-VM communication

Citation Li DJ, Mi ZY, Wu BD, Chen X, Zhao YW, Ding ZH, Chen HB. Accelerator virtualization framework based on inter-VM exitless communication, *International Journal of Software and Informatics*, 2021, 11(2): 169–193. http://www.ijsi.org/1673-7288/00248.htm

This is the English version of the Chinese article "基于跨虚拟机零下陷通信的加速器虚拟化框架. 软件学报, 2020, 31(10): 3019–3037. doi: 10.13328/j.cnki.jos.006068".

Funding items: Key-area Research and Development Program of Guangdong Province of China (2020B010164003); National Science Fund for Distinguished Young Scholars (61925206); HighTech Support Program from Shanghai Committee of Science and Technology (19511121100)

Received 2020-02-10; Revised 2020-04-04; Accepted 2020-05-09; IJSI published online 2021-06-22

With the emergence of computationally intensive applications such as deep learning[1], the computing power provided by CPUs alone is no longer sufficient. Developers have used devices with greater computing power other than CPUs as accelerators. Traditional accelerators are mainly General-Purpose computing on Graphics Processing Units (GPGPUs), while industry has developed specialized accelerators represented by Tensor Processing Units (TPUs)[2] for specific purposes. Data centers are core infrastructure in the era of cloud computing[3], and the included virtualization platforms are foundation for efficient operation of cloud services. In recent years, more and more AI service providers have tended to deploy them in cloud systems, making it necessary for accelerators to be integrated into existing virtualization platforms. Thus, the need for accelerator virtualization has arisen[4]. Although several solutions for accelerator virtualization are available, there are still many limitations and challenges for their applications in practical scenarios.

Currently, the mainstream accelerator virtualization solution is the PCI passthrough approach. The I/O device virtualization solution represented by Intel VT-d[5] can pass through accelerators to guest Virtual Machines (VMs). This virtualization approach bypasses the intervention of Virtual Machine Monitors (VMMs) and enables accelerators to be totally managed by guest VMs. Although this approach achieves almost the same performance as that in a bare-metal environment, it cannot realize fine-grained resource provisioning for several guest VMs. Thus, this virtualization solution makes the virtualization platform lose the ability to allocate computing resources elastically, and the poor scalability makes it hard for VMMs to dynamically schedule computing resources among multiple VMs.

Some accelerator manufactures also offer virtualization schemes, represented by Nvidia Grid[6] and gVirt[7], to achieve division and time-division multiplexing of hardware resources by integrating device drivers and accelerators. These schemes intervene in specific operations of accelerators with the help of VMMs, while the other operations are similar to the PCI passthrough approach, in which the runtime of accelerators can be fairly distributed to each guest VM. However, the time-division multiplexing schemes do not have a high availability for current accelerators. On the one hand, existing mature time-multiplexing schemes not only require specific drivers at the software level, but also have strict limitations on the model of accelerators at the hardware level[8]. This results in most of common accelerators not being able to use this virtualization fuction. On the other hand, such virtualization solutions should be re-developed for each new accelerator due to their poor portability. Moreover, the resource partitioning strategy cannot be freely adjusted, which also leads to poor scalability.

There are also virtualization schemes based on API forwarding, represented by rCUDA[9] and GVirtuS[10]. These schemes are based on the separate driver model[11], which abstracts vGPU at the dynamic link library level. This model divides device drivers into two parts: front-end and back-end drivers. The back-end drivers play the role of servers and convert the requests received from front-end drivers into drivers that interact with the underlying hardware devices for calling. Since this approach requires communication between front-end and back-end drivers and involves operations such as serialization of data and additional memory copies, there is a significant performance loss compared with original non-virtualization schemes. The exact loss depends on the performance of the communication part and the amount of transferred data.

To solve the problems in the current mainstream accelerator virtualization approaches, this paper proposes an accelerator virtualization framework based on hardware virtualization technology. This framework aims to:

(1) provide a convenient and available multi-tenant virtualization solution for accelerators to improve hardware resource utilization;

(2) ensure security and strong isolation among users;

(3) minimize extra performance overhead introduced by virtualization.

To satisfy the first objective, the proposed virtualization framework abandons the PCI passthrough virtualization approach and chooses the C/S architecture based on the API forwarding method. The proposed method refers to mature I/O device virtualization schemes and supports multi-tenancy by separating front-ends and back-ends. To satisfy the second objective, the paper takes VMs as the basic protection domain to ensure strong isolation in multi-tenant scenarios[12], instead of using container orchestration systems such as Kubernetes[13] directly on operating systems of physical hosts[14]. The VMFUNC function of hardware virtualization technology allows applications to switch extended page tables in the non-root mode. To satisfy the third objective, this paper implements fast exitless CPU control flow switching among VMs with the help of this feature and focuses on optimizing the performance of communication flow among VMs during API forwarding to minimize the performance degradation as much as possible. It has been proved that the Wormhole prototype system is a good solution to the problems faced by current accelerator virtualization schemes, and has achieved a 1.4–5 times performance improvement in model training tests compared to similar open-source solutions.

This paper makes the following contributions:

(1) According to the API forwarding approach, an accelerator virtualization framework based on inter-VM delegated execution is proposed.

(2) An exitless inter-VM communication acceleration mechanism based on hardware virtualization technology is proposed.

(3) The virtualization framework is applied to common Nvidia GPUs on KVM to implement and support the mainstream deep learning training framework Caffe.

(4) This framework extends and optimizes the open-source API forwarding framework GVirtuS, making it support deep learning frameworks as well. Comparison tests reveal the performance of the proposed virtualization framework is greatly improved compared with the optimized GVirtuS.

1 Background

1.1 Introduction to VMs

VM-based virtualization solutions realize virtualization by providing a complete set of virtual hardware resources for guest operating systems. This type of solutions for virtualizing physical hardware has many advantages. For example, they are transparent to guest operating systems and allow running unmodified operating systems with a high degree of compatibility[13]. Since each VM has its own independent operating system, function libraries and storage resources, it has good isolation from the perspective of security[15].

The current mainstream virtualization solution in the data center is the host operating system-based virtualization architecture represented by KVM. As shown in Figure 1, for CPU virtualization, the VMM divides each physical CPU into one or more vCPUs for different guest VMs. All vCPUs are managed and scheduled by the VMM. For memory virtualization, modern VM systems allocate an Extended Page Table (EPT) to each VM to ensure that different VMs are in different address spaces by restricting address translation[16]. The virtualization of I/O usually reuses the native drivers in the host operating system. When an I/O request is initiated within a VM, it is forwarded by the VMM to an I/O agent module (e.g., QEMU) in the host operating system for processing.

1.2 Hardware virtualization technology VMFUNC

To improve the performance of virtualized systems, major CPU manufactures, represented by Intel, have added hardware virtualization technology to their CPU products[17]. The hardware

virtualization for memory has largely replaced the original Shadow Page Table (SPT)[18] approach and became the default memory virtualization method. In modern VM systems, a guest VM needs to go through two levels of address translation to access physical memory. The first level translates the Guest Virtual Address (GVA) to the Guest Physical Address (GPA) according to the Page Table (PT) in VMs. The second level translates the GPA to the Host Physical Address (HPA) on the basis of extended page tables configured by VMMs.

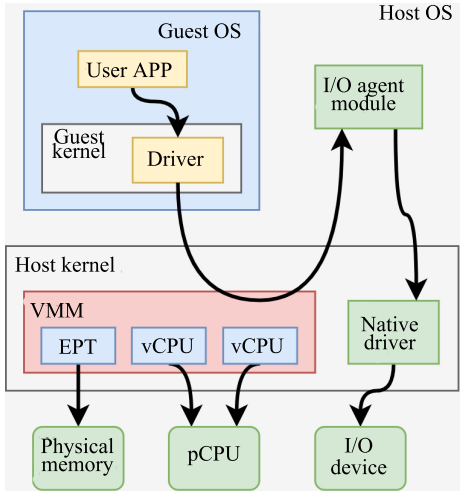


Figure 1 Architecture of host-based virtualization

From the Intel's fourth generation CPUs, the hardware instruction VMFUNC[19] was added to the CPU instruction set. It allows guest VMs to perform certain VM-related operations in the non-root mode without exiting into VMMs. Till now, the VMFUNC instruction only supports EPT Pointer (EPTP) switching, which allows a guest VM to perform the EPTP switching by a VMFUNC instruction in the non-root mode. For preventing from switching to a wrong memory area, it is necessary to pre-configure the EPTP list for corresponding guest VMs in the VMM. Applications can only select legal EPTPs from this list, and an EPTP list can contain up to 512 EPTPs.

The VMFUNC instruction has a significant performance advantage over traditional methods due to its exitless feature. Before this instruction has been developed, if a guest VM tried to modify an EPTP, it would switch from the non-root mode to the root mode to modify the value of the EPTP domain in Virtual Machine Control Structure (VMCS) and then return from the root mode to the non-root mode. The switch from the non-root mode to the root mode alone takes no less than 300 cycles. In contrast, a VMFUNC instruction can perform the same function as the entire process described above, while only taking 134 cycles with the Virtual Processor IDentifier (VPID) function enabled. VMFUNC has therefore been applied in many existing studies[20, 21].

2 Analysis of Existing Work

Based on the background described above, this paper selects the API forwarding approach as the fundamental method for Wormhole to balance performance and availability. To address the key issues of existing API forwarding-based virtualization solutions in a targeted manner, this section investigates the current related public studies and conducts in-depth testing and analysis.

2.1 Interactive mode between clients and servers

In the API forwarding method, there are various options for the interaction mode between server processes and client processes.

(1) The solution represented by GVirtuS selects the Host-VM model, in which server processes are placed in host operating systems and client processes are placed in VMs or containers. In this mode, the management of accelerators is coupled with host operating systems. The kernel of host operating systems needs to play two roles, not only as the manager of guest VMs, but also as the body for running drivers of accelerators. This not only breaks the single responsibility principle but also it only supports one version of acceleration drivers at the same time, making the operation and maintenance of the whole system difficult. For example, when operating systems do not support dynamic driver upgrades, it may requires a reboot to make the new version of drivers effective. However, the running guest VMs are inevitably influenced, which is unacceptable to cloud service providers.

(2) The approaches represented by vCUDA[22] select the VM-VM mode. The server processes and client processes are placed in different guest VMs, and many inter-VM communication methods can be selected for data exchange. The traditional network-based or sheared memory-based communication methods cause large performance degradation due to system scheduling and other factors. Some recent studies[21] have proposed fast communication methods in virtual environments, but they are not suitable for VMs with complex operating systems (such as the mainstream Linux system). Moreover, such schemes have failed to propose appropriate supporting evolutionary measures as the types of accelerators and supporting software are constantly updated.

The main problem of existing API forwarding solutions is reflected in performance, and the major performance loss comes from communication modules. From the current various API forwarding-based systems, the communication methods mainly include the following types:

(1) TCP/IP communication[23]: This method requires many times of memory copies, bringing about massive additional overhead. In the case of unidirectional TCP transmission using sockets in the mainstream Linux operating system, the user-mode processes send data by copying the data to be sent into buffer zones in the kernel, and then the TCP stacks in the kernel send the data to target addresses by local network cards. From the above flow, the TCP/IP communication method introduces two additional memory copies to copy a piece of data between two processes. In light of I/O virtualization, the number of additional memory copies may be doubled. With the common Virtio method[24] as an example, one unidirectional TCP communication adds two memory copies from VMs to host kernel buffers. Moreover, when a server process is waiting for guest requests, the CPU will fall into sleep or scheduling. Then network cards will send interrupts to wake up CPU when they receive data. These asynchronous operations can also cause significant delay.

(2) Shared memory: This communication method eliminates the extra overhead of copying memory by creating a shared memory mapping between server processes and client processes. However, shared memory alone does not provide a notification mechanism when data copies are complete. The common notification mechanism uses the ability to share a semaphore as an indicator to indicate whether the shared memory can be modified. Since both parties need to actively poll the semaphore, CPU spends substantial time on unnecessary polling and scheduling, resulting in serious delay.

(3) Remote Direct Memory Access (RDMA): The RDMA method allows one server to directly access memory on another server without the involvement of either operating system[25]. This means that this method can support zero-copy to reduce delay and performance loss of CPU. The RDMA driver directly multiplexes the memory of user-mode processes for transmission

without the involvement of CPU. The RDMA-based communication methods have excellent performance and slight delay. However, this method requires dedicated RDMA network cards and drivers, which leads to high maintenance costs, compromising its usability.

In this section, we test and analyze the API forwarding virtualization scheme with the Neuron Layer test in Caffe as an example. This testcase calls 40 different CUDA APIs for more than 1.1 million times in one running process. The top five most-frequently called APIs are listed in Table 1. Before testing, we first extend and optimize GVirtuS and deploy it in two VMs on the same physical server in the VM-VM mode. The communication module takes shared memory as the carrier for copying parameters and TCP/IP as the notification mechanism for completing parameter copies.

Table 1 List of APIs frequently called during Neuron Layer testcase

CUDA API name	Number of calls
cudaMemcpy	290,122
curandSetPseudoRandomGeneratorSeed	268,537
curandSetGeneratorOffset	268,536
cudaLaunch	142,394
cudaPeekAtLastError	127,246

Subsequently, we analyze the time cost of API forwarding processes for the Neuron Layer testcase in Caffe. One calling flow of CUDA API during virtualization is divided into the following three parts: (1) time consumption of additional memory copies, including the time cost of serialization and deserialization between data and shared memory; (2) time consumption of the notification mechanism, including the time cost of one VM notifying another VM that shared memory is available; (3) time consumption for executing native APIs. Among them, only the time consumption for executing native APIs is unavoidable and effective. The others belong to the extra performance overhead of communication modules. Through testing, the average time consumption of each part is shown in Figure 2. The total time for calling APIs in the test is 283,750,417,817 cycles. It reveals that the extra performance overhead accounts for more than 88% (250,528,017,067 cycles) of the overall process time. Particularly, the time consumption of additional memory copies accounts for less than 1% (1,910,231,774 cycles), and the notification mechanism accounts for 87% (248,617,785,293 cycles). It is clear that the notification mechanism in existing virtualization systems is a real bottleneck causing performance degradation and thus the focus of targeted optimization in this paper.

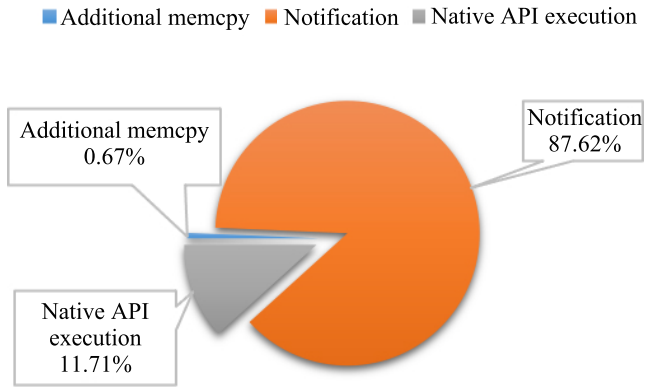


Figure 2 Analysis of the average time cost of Neuron Layer in GVirtuS

2.2 CPU utilization

In a native physical server environment, the effective CPU runtime T_{eff} of a process during calling accelerators can be mainly divided into two parts: (1) the time for running applications in the user mode; (2) the time for running drivers interacting with accelerators in the kernel mode. The rest of time can be regarded as ineffective running time T_{ineff} as it produces no useful results. Thus, in the case of accelerator virtualization, an important index for measuring the performance and efficiency of a virtualization framework is the ratio of the effective CPU runtime to the total running time after virtualization, which is denoted by $T_{\text{eff}}/(T_{\text{eff}} + T_{\text{ineff}})$.

In the ideal case, the ratio after virtualization is the same as that in the native physical environment, namely that virtualization causes no extra performance overhead. The API forwarding-based solutions described in this section all use the active interaction. Whether the communication is realized through networks or shared memory, the server CPU and the client CPU are both required. The actual effective runtime is only the time for the client CPU executing applications and the server CPU executing drivers. For slighter delay caused by scheduling, servers and clients are normally bound to different physical CPUs, as shown in Figure 3. The CPU resources consumed by one forwarding call during virtualization can be more than twice the resources consumed in the native physical environment.

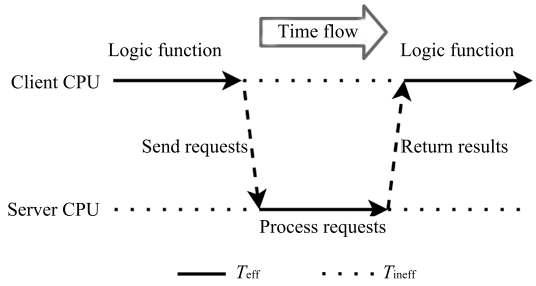


Figure 3 CPU effective time in a virtualized environment

With the testcase in the above section as an example, we measure the CPU utilization in the native physical environment and the GVirtuS virtualized environment using the built-in time command of the Linux operating system. As Table 2, the process in the native physical environment occupies only one CPU with a utilization ratio of 97.28%, while the process in the GVirtuS environment occupies two CPUs with utilization ratios only 46.61% and 38.52% respectively.

Table 2 CPU utilization during Neuron Layer testcase

Case	Total time (s)	User-mode execution time (s)	Kernel-mode execution time (s)	CPU utilization (%)
Native physical environment	6.947	4.494	2.264	97.28
GvirtuS back-end	94.705	11.833	32.306	46.61
GvirtuS front-end	93.990	6.498	29.706	38.52

3 Design of Wormhole

The accelerator virtualization framework, Wormhole, aims to provide an efficient virtualization solution with high availability, good performance, supporting multiple tenants for accelerators while ensuring strong isolation and security among users in practical scenarios of data centers. With VMs as the protection domain of front-end and back-end drivers, we improve existing virtualization solutions by combining the widely used hardware virtualization

technology and develop a flexible and versatile accelerator virtualization framework with high performance, which is also easy to be maintained. Figure 4 illustrates the architecture of Wormhole and the flow of one API forwarding call, which will be elaborated on in this section.

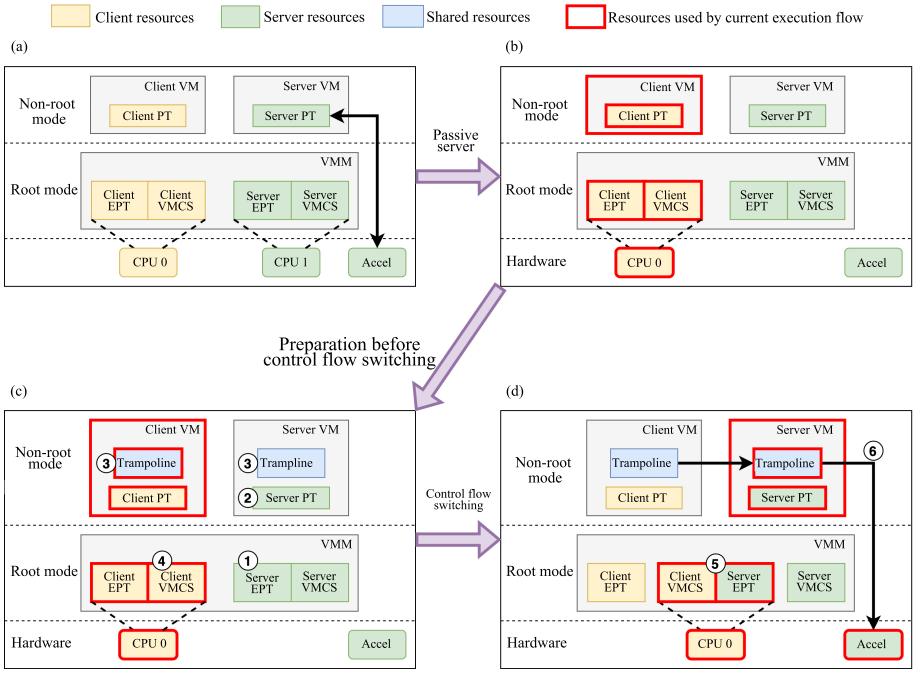


Figure 4 Architecture and invocation example of Wormhole

3.1 Passive server VM

The design of Wormhole adopts a passive server VM mode, which performs better and more flexibly compared with the existing interactive mode. The passive server VM mode means that server VMs do not actively consume any CPU resources when there are no user requests waiting to be processed.

In the design, all accelerators are passed to the server VM dedicated to management by the PCI passthrough method, thus decoupling accelerator management from the host operating system. The server VM is the same as a normal guest VM in the initialization stage, which can also have its own CPU, as shown in Figure 4(a). At this point, the server VM has exclusive access to CPU 1, which is preparing the required communication receiver module and other operating environments. After all the pre-configuration tasks are completed, the server VM will actively fall into a snapshot-like freezing state, as shown in Figure 4(b). At this moment, the server VM no longer has CPU resources, and its original CPU1 resources can be released for other guest VMs, eliminating the waste of server CPU resources caused by constantly waiting for requests from guest VMs.

According to the test data, the communication modules of existing accelerator virtualization solutions spend massive time waiting for each other. For example, while the server CPU is running back-end drivers and other logics, the client CPU remains idle until the results are returned. Therefore, Wormhole can eliminate the waste of CPU resources using the client CPU that has sent requests for delegated execution in the server VM, making a passive server VM feasible. This design improves the utilization of CPU after virtualization, and the saved CPU

resources can be allocated to more guest VMs, solving the problem mentioned in Section 2.2.

One physical server can include multiple passive server VMs at the same time, and each server VM can be assigned to a different number of accelerators. Meanwhile, due to the isolation of VMs, different server VMs can be installed with different versions of accelerator drivers and supporting computing libraries to fulfil needs of different users. It also means that if multiple heterogeneous accelerators are connected to a physical server, different accelerators can be isolated by different server VMs without interfering with each other. This design makes it very easy for the virtualization framework to accommodate different accelerators that are rapidly updating, addressing the problem mentioned in Section 2.1.

3.2 Active inter-VM communication based on control flow switching

As Wormhole's passive serve VM releases its CPU resources, communication between VMs must be initiated by the client to cooperate with the server through active communication. As such, this paper proposes a delegated execution method that allows the client execution flow to actively enter the server VM for continuous execution. In light of the correctness of delegated execution, it is required that the CPU is in the correct address space and has access to correct instructions and data when running in different VMs. On this basis, the performance problem mentioned in Section 2.2 must be addressed, so that the high-performance objective of Wormhole can be achieved. Since the extra virtualization overhead is closely related to exiting times[26], this paper realizes zero VM exit during control flow switching through pre-configuration to obtain excellent inter-VM communication performance.

This section will introduce the core idea of this design, by using the flow of one delegated execution shown in Figure 4 as an example. In Figure 4(b), the server VM registers its server information with the VMM after initialization. Then the guest VM is allowed to do delegated execution in the server VM. In Figure 4(c), the guest VM first performs a series of preparations for control flow switching. After the preparation work, the execution flow is switched to the server VM, enabling the delegated execution as shown in Figure 4(d). Thanks to the proposed design, the operations related to control flow switching in Figure 4(c) and Figure 4(d) are conducted in the non-root mode, without triggering VM exiting. The invocation of a complete delegated execution flow can be divided into six steps.

First, the guest VM sends a request of matching with the server VM, and this causes the guest VM to exit to the VMM to add some memory mapping. The following Steps 1–3 are one-off pre-configuration. Although this stage actively triggers the VM exiting, it is not on the critical path of the inter-VM communication. Therefore, subsequent delegated executions do not need to repeat these operations, and the control flow switching after the configuration will remain exitless.

Step 1. The VMM maps the value of the CR3 register associated with the client process to the HPA of the page table of the server process in the extended page table of the server VM. It aims to make preparations for the subsequent VMFUNC instruction to implement the correct switching of the VM address space. For the two-level address translation mechanism of VMs, the existing function of the VMFUNC instruction is only to switch the extended page table that controls the second level address translation. The first level address translation relies on the page table CR3 points to. Then the correct page tables should be matched before and after the VMFUNC instruction to change the address space. After mapping is added in this step, the value of the client process CR3 can be translated to the page table of the client process in the extended page table of the guest VM. The same value can be translated to the page table of the server process in the extended page table of the server VM, ensuring the correct translation of the address space before and after the VMFUNC instruction. Meanwhile, this step makes it

possible to achieve the equivalent effect of CR3 by executing only one VMFUNC instruction in the VM user mode when the address space is switched between VMs in Step 5.

Step 2. The VMM maps the GVA value of the LSTAR Model Specific Register (MSR) of the guest VM to the GPA of the code page at the system call entry of the server VM in the page table of the server VM. It aims to correctly execute system invocations after user-mode applications initiate them when the execution flow is in the address space of the server VM. Most modern operating systems use the SYSCALL instruction to invoke the system calls. When the system executes the SYSCALL instruction, the CPU jumps to the entry of system invocation according to the value of LSTAR MSR. By default, modifying the LSTAR MSR in VMs causes VMs to exit and VMMs complete the operation. Therefore, this step adds the mapping to avoid VMs to exit in the subsequent control flow switching process. After the above mapping is added, the entry of the server system invocation can be accessed with the value of the client LSTAR MSR when the system invocation is initiated during the delegated execution, so that it can interact with the kernel normally.

Step 3. A copy of a trampoline is pre-stored in the VMM, and the pairing operation will map an identical GVA in the high address space of two VMs to this trampoline page. The purpose of this step is to ensure that the CPU instruction flow can be correctly transitioned before and after the address space switching. The CPU program counter points to the current instruction based on the virtual address. When the VMFUNC instruction is executed, the value of the program counter will increase by the corresponding length and the next instruction is already in the address space of the server VM. This step provides the same instruction at the same location in both virtual address spaces, so the CPU still executes the continuous correct instruction before and after the address space switching.

Then the guest VM returns to the user mode and calls the interface provided by the trampoline page to start to execute the code related to the control flow switching. The following Steps 4–6 need to be repeated during each delegated execution, so each step is required to be exitless to improve the performance of inter-VM communications.

Step 4. Before the address space switching, the client process needs to temporarily modify the values of the two MSRs, FS.base and GS.base, to the values of the corresponding MSRs in the server VM. The purpose of this step is to ensure that the address mechanism based on segment registers works normally in the server VM during delegated execution. In modern operating systems, a large amount of data needs to be accessed through segment registers. Their base addresses are stored in some domains in the VMCS of the corresponding VM and do not change with the address space. Without the correct adjustment, wrong data will be accessed during the delegated execution due to the wrong address. This step replaces it with the correct MSR value in advance, so that the addresses obtained by means of segment registers before and after address space switching are legal. Although this step replaces the MSR values instead of adding a mapping, the VMM does not intercept the reading and writing of MSRs, FS.base and GS.base, in the VM by default, which still does not cause any VM to exit.

Then, the execution flow of the guest VM actually switches to the VM address space.

Step 5. From the next instruction, the VMCS of the guest VM is still on the CPU, but the extended page table pointer in it has already pointed to the extended page table of the server VM. This step actually enters the address space of the server VM, and the data and resources currently in effect are shown in the area with red border in Figure 4. At this point, the CPU program counter points to the next instruction in the shared trampoline page and can continue to correctly execute the remaining code. Thanks to the mapping added in Step (1), there is no need to explicitly modify the value of the register CR3 as in the conventional solution during the switching to the target address space. Thus, the VM can avoid the exiting overhead caused

by the invocation of privileged instructions.

At last, the guest VM enters the server application and starts the delegated execution.

Step 6. The guest VM obtains the entry address of the trampoline page of the trampoline page and jumps to the back-end processing program in the server VM. Then it interacts with accelerators through native drivers. This step finally switches the control flow that originally runs in the guest VM to the server VM to start the delegated execution. Through the above five steps, the subsequent system invocations, interrupts, and other complex operations, which are necessary to interact with accelerators, can be performed normally.

After the delegated execution, the execution flow should return from the server VM to the guest VM. At this point, the reverse operations of Steps 4–6 are executed in the reverse order, which we omit here. According to the design of this section, through the one-off pre-configuration in the initialization stage, the subsequent frequent inter-VM control flow switching operations will not cause any VM to exit, guaranteeing fast and efficient delegated execution.

3.3 Technical points of the design

By summarizing the six steps in the previous section, we propose the following technical points.

(1) Fast execution flow switching across VM address space.

Active inter-VM communication should satisfy two requirements before and after execution flow switching: the correct address translation mechanism and the transition of CPU instruction flow before and after switching. Originally, one VMFUNC instruction is only responsible for switching mapping from GPA to HPA, so an additional measure is required to switch to the corresponding page table. With the switching from the guest address space to the server address space as an example, an intuitive idea is to modify the value of the register CR3 before or after the VMFUNC instruction to switch the page table. However, this operation will not work. The program counter obtains the current instruction based on the virtual address. If the previous instruction of VMFUNC changes the page table, the CPU will actually face an invalid page table when it executes the next instruction due to the change in the mapping from GVA to GPA, resulting in runtime errors. Conversely, if we try to replace the page table after the VMFUNC instruction, the next instruction that is actually accessed is not related to the modification of CR3 as the mapping from GPA to HPA changes after the VMFUNC instruction. It will also cause exceptions.

Combining the mapping addition of CR3 in Step 1 with the hardware virtualization technique in Step 5, Wormhole realizes that a single VMFUNC instruction can simultaneously switch page tables and extended page tables without exiting to VMMs, as shown in Figure 5. This ensures the correct address translation mechanism before and after switching. In addition, Wormhole also significantly reduces the inter-VM communication overhead by using one instruction to complete multiple operations. Meanwhile, delegated execution enables the guest VM and the server VM to operate on the same physical CPU, avoiding the high overhead caused by Inter Processor Interrupt (IPI).

With regard to the transition of the CPU instruction flow before and after switching, Wormhole provides a shared trampoline page and a stack dedicated to delegated execution in Step 3. The code page contains the VMFUNC instruction and some contextual saving logics. They are mapped to the same GVA in the page tables of the guest VM and the server VM. In this way, the client process can jump to this code page when it needs delegated execution. After the address space is successfully switched, as the client process and the server process share this code page at the same address, the program counter can switch to the next instruction of VMFUNC without affecting the original stacks of both processes. In addition, it is necessary

for the server process to register the entry of the delegated execution function with the VMM in advance. The trampoline will jump to the entry of the registered function after the control flow is switched and officially start to access accelerators on the server.

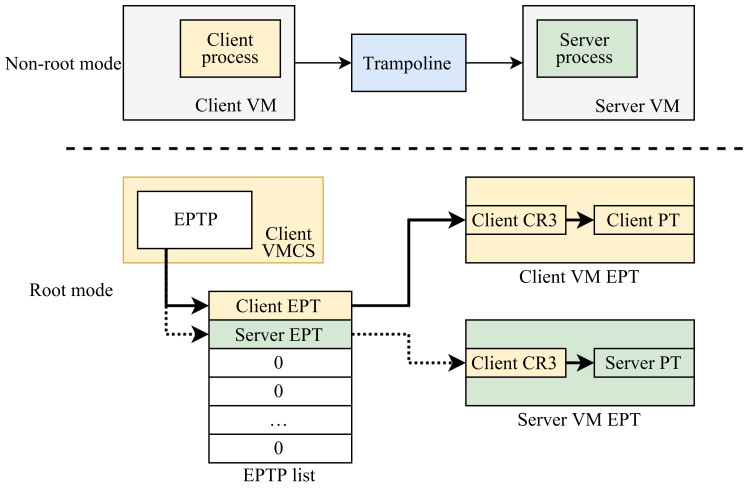


Figure 5 Switch VM address space through VMFUNC

(2) Correctly perform complex operations such as system invocations and interrupts after control flow switching.

Some preprocessing works, represented by SkyBridge[21], also follow similar ideas of delegated execution. However, these works are only for simple microkernel-based operating systems, and their working scenarios consider only delegated execution between processes in an operating system. In the design principle of microkernel, the kernel part only includes a small amount of code and usually keeps a few basic management functions. A large number of functions in traditional macro kernel (such as device drivers) are removed from the kernel mode, which are used as a specialized user-mode process. Accordingly, most functions are not performed in the kernel in a microkernel scenario. For example, functions related to I/O are handled by user-mode drivers, and interrupts are transferred to specific user-mode processes for handling after they occur.

However, the macro kernel-based Linux operating system is still the mainstream in data centers. For performance reasons, macro kernels integrate a variety of complex functions, including device drivers, interrupt handling and resource management. As a result, user-mode applications frequently interact with the kernels during their execution. Massive tightly coupled complex functions cause some inconvenience in error isolation. As there are many unconventional memory access mechanisms involving segment registers in macro kernels, data accessing errors can often lead to system crashes and other irreversible consequences. Therefore, it is extremely important to ensure correct interaction with the kernels during delegated execution.

Most modern operating systems depend on the SYSCALL instruction to make system calls. The CPU will jump to the entry of the system call processing functions based on the value of LSTAR MSR, so Wormhole creates a mapping from the client VM LSTAR MSR to code page of the system call entry of the server VM in the address space of the server VM in Step 2. This ensures correct system call based on the value of the guest LSTAR MSR.

For separation of privileges and security reasons, the stack structures and other memory structures used by the user mode and the kernel mode are different in macro kernels. Therefore,

when a user-mode process traps into kernels, we should save the user-mode context and switch to a kernel-specific stack or an interrupt-specific stack. For example, the addresses of these stack tops are stored in some per-CPU variables in the Linux kernel. In the Linux kernel on AMD64, these per-CPU variables are accessed by GS segment registers. Their base addresses are stored in dedicated MSRs and do not vary as the address space changes. Wormhole ensures the correctness of the GS-segment memory access mechanism before and after control flow switching by Step 4. Meanwhile, Step 4 also ensures the correctness of the FS-segment memory access mechanism. For example, the Linux operating system on AMD64 uses the segment register mechanism to access a special "sentinel" value by FS:0x28 to check stack buffer overflow.

(3) Exitless VM during control flow switching.

Control flow switching is on the critical path of the whole communication process, and each API forwarding call involves two control flow switches between VMs. Then, it is necessary to minimize the time cost of virtualization to minimize the additional overhead. The additional virtualization overhead is closely related to the exiting times of VMs. To minimize the time consumption of switching between the root mode and the non-root mode, Wormhole ensures that the control flow switching will not actively trigger any VM exit, and all configuration operations are performed in the non-root mode.

On the virtualization platform represented by KVM, the modification of CR3 and LSTAR MSR in the non-root mode will exit to the root mode by default. Therefore, Wormhole adds mapping in Steps 1 and 2. Although the value of the register seen by the VM remains the same before and after address space switching, the correct physical memory area is actually accessed by the address translation mechanism, enabling the exitless characteristic and the same effect as performing the privileged operations. The MSR modifications of FS.base and GS.base in the non-root mode will not cause any exiting by default. Moreover, the access mechanism of segment registers involves a large number of memory pages, and the mapping addition to the server address space is not suitable. Thus, the VM call is initiated in Step 4 to temporarily replace the values of two MSRs.

(4) Properly handle the page fault of extended page tables after control flow switching.

With the support of extended page tables, modern virtualization systems follow a lazy memory allocation policy with regard to the memory allocation requests from VMs. Specifically, when a guest VM initially requests a block of memory, only the GVA-to-GPA mapping is added to the page table. The GPA-to-HPA mapping is not added by the VMM to the extended page table until the page fault of the extended page table is triggered for the first time when this memory area is accessed. If a VM exit occurs during delegated execution in the server VM, the VMM still assumes this VM exit comes from the client VM. By default, the exiting processing function is executed for the guest VM according to relevant information in VMCS. In fact, the reason of causing this exiting comes from the server VM, so the processing object must be changed from the guest VM to the server VM.

When the page fault of extended page tables causes exiting, Wormhole asks the VMM to determine whether this fault occurs during the delegated execution. If it occurs, the parameters related to the page fault from the VMCS of the guest VM are extracted, and the page processing function is executed for the server VM. Then the GPA-to-HPA mapping is added to the extended page table of the server VM.

4 Prototype of Wormhole

To verify the design of the proposed accelerator virtualization framework, this paper implements a prototype system of accelerator virtualization for common NVIDIA GPU according to the above design, which supports CUDA 9.0. Based on the mainstream Linux

operating system with the QEMU-KVM as the virtualization platform, this prototype is implemented on the Intel x86-64 platform. The system architecture is illustrated in Figure 6, and the details of implementation are as follows.

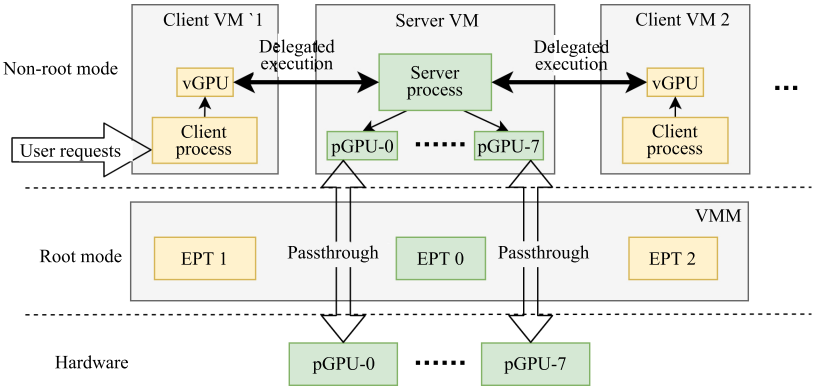


Figure 6 Architecture of prototype system

4.1 Basic API forwarding framework supporting CUDA calls

Currently, the mainstream deep learning frameworks interact with CPU accelerators mainly by calling CUDA[27] APIs developed by NVIDIA to calculate APIs. After collecting the used CUDA APIs, we find that the main scientific computing libraries used by the mainstream deep learning frameworks include cudart, cuBLAS, cuDNN and cuRAND. Applications call CUDA APIs in these libraries through dynamic links.

During implementation, the basic framework of the whole virtualization system is divided into front-end modules, communication modules and back-end modules. The front-end modules are placed in guest VMs; the back-end modules are arranged in the server VM; the communication modules are employed for delegated execution between two VMs.

(1) In the front-end module, the system implements stub functions with the same function prototypes for all collected CUDA APIs according to the official documentation, which are encapsulated in the stub function libraries (such as libcudart.so, libcudnn.so) with the same name as the corresponding computing libraries. These stub function libraries replace the native libraries in the guest VM by modifying the environment variable LD_LIBRARY_PATH to intercept CUDA calls of applications.

(2) In the back-end module, the system first uses dlopen to preload the used CUDA computing libraries and then registers the delegated execution entry of the back-end processing function with the VMM. Then the system forks subprocesses with the equal number of CPU kernels to support as many concurrent front-end requests as possible, and each subprocess corresponds to a front-end module. At last, the back-end module enters a frozen state and waits for delegated execution from the front-end.

(3) The implementation of the communication module will be elaborated on in the rest of this section.

To reduce the number of front-end and back-end communications, our system conducts batching optimization for the forwarding method of the common CUDA kernel, which is similar to the optimization for pre-processing work such as GVirtuS and vCUDA. One invocation of a CUDA kernel function actually calls three CUDA APIs in turn (one call of cudaConfigureCall $\rightarrow$ one or multiple calls of cudaSetupArgument $\rightarrow$ one call of cudaLaunch). Only the last cudaLaunch is the explicit synchronization point for actually interacting with the device. As

such, each time the first two CUDA APIs are intercepted, they do not have to be forwarded to the back-end immediately and can be processed in bulk with the last cudaLaunch.

The latest versions of CUDA support the Unified Virtual Addressing (UVA). For example, the pointer parameter of cublasSdot can point to both the host memory and the device memory. In the native environment, the GPU driver should determine the direction of the memory copy (e.g., from device to host memory), which must be confirmed according to the memory type (the host memory address or the device memory address) of source and destination segments. In this virtualization system, since the GPU driver and the application are in different address spaces, the address forwarded by the application of the guest VM cannot be properly checked and judged by the driver of the server VM. Therefore, this system implements an efficient tracking module for GPU device addresses based on the interval tree. This module records the device memory interval returned by the APIs that allocate device memory, such as cudaMalloc. After the CUDA APIs with the UVA feature are intercepted, the tracking module queries the memory type that the received memory address parameters belong to and then processes them according to specific cases.

According to the design of Wormhole, before the server VM starts, the host operating system enables passthrough of the GPU device by the PCI passthrough approach. After the server VM starts, the host operating system should configure drivers and scientific computing libraries. The two VMs both need to exit to register their related information. This system modifies the exiting processing function of CPUID in KVM and conducts corresponding operations according to the types of VMs and requests after receiving registration requests.

4.2 Memory mapping and user-mode interface for control flow switching

The shared memory mapping, including trampoline pages, is the key to active inter-VM communication. Based on the exiting instruction of VMs at initialization in the CPUID processing function of KVM, this system pre-configures the following mappings for server and guest VMs in order:

(1) The mapping of client CR3 and LSTAR in the server VM: The addition of these two mappings is the exitless basis during the execution of subsequent trampoline pages. This system first reads the value of the server CR3 in the VMCS when the server VM exits. Then it uses software simulation to traverse extended page tables of the server VM and translates them to get the HPA of the server process page table in the host physical memory. Next, the system reads the value of the client CR3 in the VMCS when the guest VM exits and traverses the extended page tables of the server VM again to add the mapping from the client CR3 to the HPA of the server page table. Similarly, the system first reads the value of LSTAR MSR at the server when the server VM exits and uses the server page table for address translation to obtain the corresponding GPA. Then it acquires the value of the LSTAR MSR at the client side as the new GVA when the guest VM exits and then creates a mapping from this GVA to the GPA of the server-system call page table in the server page table.

(2) Shared memory: The CUDA API parameters intercepted by the front-end module in the guest VM should be forwarded to the back-end module in the server VM, and a large number of APIs need to conduct the memory copy between the host and the device. To achieve inter-VM parameter transferring, this paper asks KVM to allocate a large enough memory as the shared memory for transferring parameters and reserves a section of the starting GVA in the high address space of the server process and the client process. Then the system adds mappings of application page tables and extended page tables of both server and client processes in KVM, so that the CPU can access the same physical memory through the reserved GVA before and after the address space switching.

(3) Shared stack for transition: There is a transition period when the control flow is switched from client processes to server processes. To avoid tainting the original stack structures of server processes and client processes, this system allocates 16 memory pages with a size of 4 KB in the KVM. They are mapped to the same GVA in the high address space of both processes to ensure the availability of stack structures before and after the control flow switching.

(4) Mapping of pointer arrays of processing functions in the guest VM: The system should specify the location of the objective function when the control flow jumps from the trampoline page to the address space of the server process. This system maintains a function array pointer for each server VM in the KVM, and the server VM exits to KVM during the initialization of the server process to store the virtual addresses of all available processing functions in the array. Similarly, to access the pointer array of processing functions in the user mode during the transition, this system maps it to the high address space of the client process.

(5) Trampoline page: When the above preparations are completed, the system stores a trampoline page in the KVM. It exposes the interface of the delegate_to_server function to the user mode, and the parameters include the offset of the server VM and the offset of the back-end processing function. This allows the communication module in the guest VM to switch to the corresponding back-end processing function in the server VM by calling delegate_to_server. The trampoline page is mapped to the same GVA in the high address space of both applications. The client process converts this address to a function pointer and then it can call the delegate_to_server interface in the same way as the function call. The logic of the trampoline is as follows:

(a) When the trampoline is called by the client application, the system first stacks all the current registers to save the context. Then the system saves the current stack pointer and replaces it with a temporary stack for transition, and finally launches the arch_prctl system call to replace the two MSRs, FS.base and GS.base.

(b) The system calls the VMFUNC instruction to switch to the address space of the server process, and an inter-VM communication is completed.

(c) After checking the validity of parameters, the system reads the address of the back-end processing function from the function pointer array and takes the address as the function pointer to jump indirectly to the back-end processing function for execution.

(d) After the back-end processing function returns, the system calls the VMFUNC instruction to switch back to the address space of the client process.

(e) The system initiates the arch_prctl system call to set the FS.base and GS.base of the client process and restores them to the native stack structure of the client process. Then it restores the context before the delegated execution from the stack.

4.3 Freezing of server VM

After the initialization, the back-end module of the server VM will enter the frozen mode from the user mode. After that, the CPU resources of the server VM can be released to other guest VMs, but the memory and I/O resources are still reserved for delegated execution. The reason for changing from the user mode to the frozen mode is as follows. During delegated execution, the control flow of the guest VM is also switched from the user mode. If the control flow of the guest VM is frozen in the kernel mode, the kernel's dedicated stack and other data structures will be tained and serious errors such as kernel crashes will be caused during delegated execution. Fortunately, the CPUID instruction unconditionally triggers VMs to exit in both the user mode and the kernel mode, so the back-end module will call CPUID in the user mode to transfer freezing indicator parameters to exit into KVM. KVM sets the frozen flag for the VM in the CPUID processing function after receiving the freezing request. Before each VM's vCPU tries to execute VMRESUME to resume operation, KVM will check the frozen flag of the VM.

If it is true, its vCPU will be intercepted and scheduling is actively initiated to release CPU resources.

4.4 Handling of page fault of extended page table during delegated execution

On the virtualization platform QEMU-KVM, each guest VM is essentially a user-mode QEMU process that can be accelerated by the KVM kernel module from the perspective of the host operating system[28]. Therefore, the address space in each VM is essentially that of the corresponding QEMU process. If a normally running VM triggers a page fault in the extended page table, the CPU will experience a VM exit due to an EPT violation, and the KVM will process it in the address space of the current QEMU process according to the GPA of the page fault. During the processing, the Linux kernel first obtains the process descriptor (the task_struct structure) running on the current CPU based on the per-CPU variable named current, which saves the memory descriptor (the mm_struct structure) bound to the current QEMU process. The actual physical memory is then allocated for this memory descriptor by the memory management related functions of the Linux kernel. Then the GPA-to-HPA mapping is added to the extended page table.

If a page fault of the EPT is triggered during delegated execution, the current process identity recognized by KVM after exiting is still the QEMU process of the guest VM. Therefore, KVM will allocate new memory in the address space of the client QEMU process and add the mapping to the extended page table of the guest VM. Actually, the page fault occurs in the address space of the server VM. The correct action is to allocate new memory to the server QEMU process and add the mapping of the extended page table. Thus, this system modifies the EPT violation processing function of KVM to determine if the delegated execution is performed currently when the exiting occurs due to the page fault. If delegated execution is performed, the current variable is temporarily stored and replaced with the process descriptor of the server QEMU process which is recorded during initialization. In this way, the object of KVM is the server VM during memory allocation and mapping of the extended page table. After that, the current variable is restored. As the server VM is in the frozen state, there is no risk of data race in the above operations.

4.5 Some CUDA characteristics urgently requiring improvement

The pinned memory characteristic of CUDA is not supported due to the closed source nature of drivers. Thus, asynchronous APIs such as CUDA multi-stream operations are temporarily converted to synchronous APIs for calling. This has some impact on the memory copy performance between the host and the device. However, the part that is not fully implemented is orthogonal to the proposed design, without impact on the verification of the performance improvement of the proposed accelerator virtualization framework.

5 System Evaluation

To test the performance of the prototype system, an Intel Haswell-E consumer server supporting VMFUNC hardware virtualization is adopted as the testbed. The software/hardware configuration of the testbed is shown in Table 3. This section is divided into three parts according to the test granularity, and the PCI passthrough virtualization solution is taken as the baseline of the highest performance.

For comparison with the available GPU virtualization solutions, this paper selects a representative open-source solution, GVirtuS, as the contrast. As the latest GVirtuS still supports a limited number of CUDA APIs, this paper supplements the source code of GVirtuS to make it support as many CUDA APIs as this prototype system. In addition, as the communication

module of GVirtuS only supports TCP/IP for virtualization in the VM-VM mode, it has large extra memory copy overhead during API forwarding. This paper adds the same shared memory approach to the communication module of GVirtuS as the prototype system, eliminating the difference in memory copy overhead between the two systems. This improves the performance of the GVirtuS system and ensures the fairness of the performance test.

Table 3 Testbed configuration

Item	Model/Version
CPU	Intel i7-5930K@3.5 GHz (6-core 12-thread)
Memory	40 GB DDR4 2133 MHz
GPU	NVIDIA Quadro GV100 (32 GB HBM2)
Operating system	Ubuntu 19.10 (Eoan Ermine)
Kernel version	Linux Kernel 4.19.56

Most of the previous prototype systems only support a small part of CUDA APIs, and the selected testcases are distinct from practical applications in real scenarios, so they cannot fully reflect the real performance of systems. This paper selects the popular Caffe[29] as the benchmark test program. Caffe is a deep learning framework written in C++, which has been widely used in deep learning since it is clear and efficient[30].

5.1 Microbenchmark test

To verify that Wormhole can improve the communication performance stated in Section 2.1, this section analyzes the time breakdown for the forwarding processes of all CUDA APIs in the Neuron Layer unit test of the prototype system. During invocation, the time overhead of each part and the percentage of the total time are shown in Figure 7. In the test, all the API calls consume 23 778 309 322 cycles. The percentage of the extra performance overhead in the whole time of process is reduced from 88% in GVirtuS to 20% (4 660 728 884 cycles). To be specific, the time consumption by additional memory copies occupies 3.65% (866 946 457 cycles); that by system calling and modifying the MSRs, FS.base and GS.base, occupies 10.64% (2 530 932 006 cycles); that by control flow switching, including VMFUNC, occupies 5.31% (1 262 850 421 cycles).

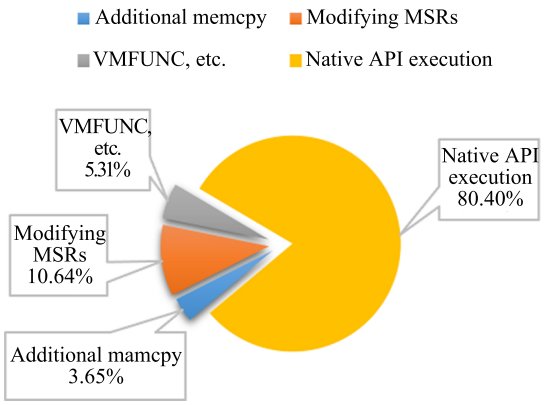


Figure 7 Analysis of the average time cost of Neuron testcase in the prototype system

In terms of absolute time overhead, the TCP/IP-based notification mechanism of GVirtuS consumes 248,617,785,293 cycles, while the control flow switching mechanism of the Wormhole prototype consumes only 4,660,728,884 cycles. This section adds a test to verify the effects of "execution flow switching across VM address space" (hereinafter referred to as fast switching)

and "no VM exit during execution flow switching" (hereinafter referred to as exitless) proposed in Section 3.3 at a finer granularity. In this test, the exitless function is turned off, namely that the execution flow actively exits to the VMM to replace MSRs before and after each execution flow switching. The results show that the total time consumed by calling APIs is 29,638,851,151 cycles, and the communication module takes up 9,432,980,220 cycles. This means that the fast switching technique reduces the TCP/IP communication overhead by two orders of magnitude, while the exitless technique further reduces the overhead of execution flow switching by 50%. Based on above, this design significantly reduces the additional time overhead consumed by virtualization. From a microscopic perspective, it has been proved that the Wormhole design significantly reduces the additional overhead of virtualization compared with existing designs implemented in GVirtuS.

As summarized in Table 4, with regard to CPU utilization, Wormhole occupies only one CPU for a total time of 12.145 s. The effective time of the user mode is 9.474 s, and the effective time of the kernel mode is 2.661 s. The utilization of CPU reaches 99.92%, much higher than the 46.61% and 38.52% of two CPUs in GVirtuS and even better than the 97.28% of the PCI passthrough solution. This proves that the Wormhole's design significantly enhances the CPU utilization compared with existing solutions.

Table 4 Improvement of CPU utilization during Neuron Layer testcase

Testcase	Total time (s)	User-mode execution time (s)	Kernel-mode execution time (s)	CPU utilization (%)
Native physical environment	6.947	4.494	2.264	97.28
GvirtuS back-end	94.705	11.833	32.306	46.61
GVirtuS front-end	93.990	6.498	29.706	38.52
Wormhole	12.145	9.474	2.661	99.92

5.2 Unit test of neural network layers

In this section, we use the unit testcases of various neural network layers provided by Caffe to show the performance improvement of Wormhole in different neural network layers from a macro perspective. The time statistics tool that comes with the testcases is used to measure the time (ms) consumed by each test. The performance is evaluated according to the criterion that the shorter time consumption indicates a better performance.

In this section, some neural network layers that are important in practical scenarios are selected as unit testcases, mainly including the following types:

(1) the common image processing network layers in the field of computer vision[31], such as the convolutional layer (hereinafter referred to as CONV) and the deconvolutional layer (hereinafter referred to as DECONV);

(2) the common recurrent network layers in the field of natural language processing[32], such as the recurrent neural network layer (hereinafter referred to as RNN) and the long short-term memory network layer (hereinafter referred to as LSTM);

(3) the common normalization network layers in deep neural networks, such as the batch normalization network layer (hereinafter referred to as BN);

(4) the common activation network layers in deep neural networks, such as activation function network layers, including ReLU, Sigmoid, and TanH, which are collectively called as neuronal network layers (hereinafter referred to as Neuron).

The test results and comparison results are shown in Figure 8–Figure 10. In the figures, Baseline represents the ideal performance of the PCI passthrough solution; Wormhole indicates the performance of this prototype system while GVS denotes the performance of the optimized GVirtuS system.

In terms of the image processing network layer, the performance of this prototype system is increased by 88.31% in the CONV test and by 88.67% in the DECONV test compared with the optimized GVirtuS system.

In regard to the recurrent network layer, the performance of this prototype system is increased by 89.62% in the RNN test and by 88.97% in the LSTM test compared to the optimized GVirtuS system.

In terms of the normalization and activation layers, the performance of this prototype system is increased by 89.45% in the BN test and by 87.89% in the Neuron test compared with the optimized GVirtuS system.

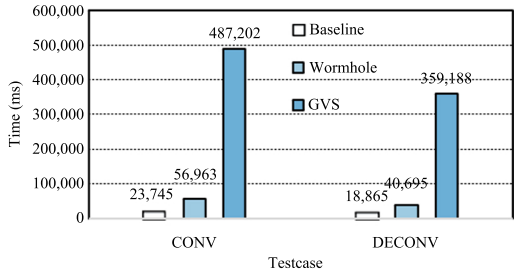


Figure 8 Performance comparison for the image processing layer

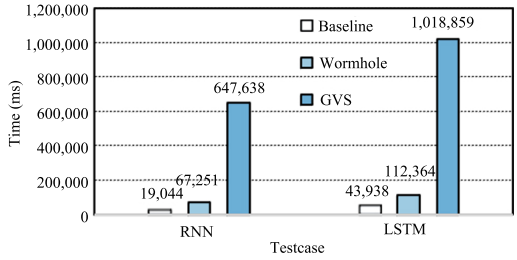


Figure 9 Performance comparison for the recurrent network layer

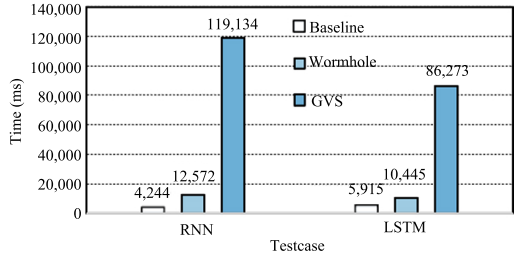


Figure 10 Performance comparison for the normalization and activation layers

5.3 Training test in classical models

In this section, we use AlexNet[33] and LeNet[34] for a complete training test of deep learning models to evaluate the performance of the prototype system of Wormhole under a complete real workload. The throughput statistics tool built in the deep learning framework is used to measure the throughput of the training process expressed in iterations per second (iter/s). A larger

throughput indicates a better performance. The meaning of Baseline, Wormhole and GVS is the same as in the previous section. Wormhole indicates the performance of the prototype system. GVS denotes the performance of the optimized GVirtuS.

Developed in 1994, LeNet is one of the first convolutional neural networks (CNNs) and has driven the progress in deep learning. This section uses MNIST[35] as the dataset with a batch size of 100 to conduct 10 000 iterations based on Caffe. As shown in Figure 11, the throughput of this prototype system is improved by five times compared with the optimized GVirtuS.

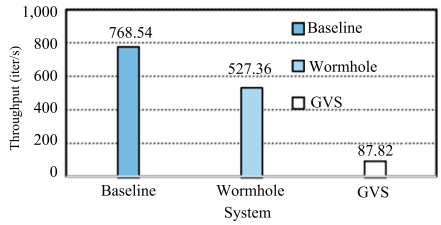


Figure 11 Performance comparison for the LeNet training

AlexNet was proposed in 2012 and successfully applied to ReLU, Dropout and LRN in CNNs for the first time. It can be considered as a deeper and wider version of LeNet, which is the foundation of modern deep CNNs. To eliminate the influence of large-scale storage device I/O to facilitate testing, we use the dummy data as the dataset with a batch size of 64 to conduct 1 800 iterations based on Caffe. As Figure 12, the throughput of the prototype system is improved by 1.4 times.

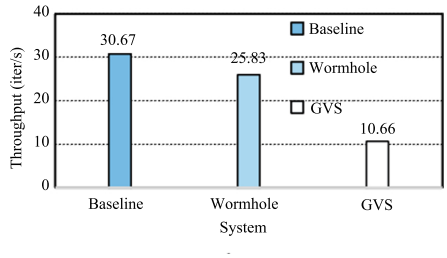


Figure 12 Performance comparison of AlexNet training

6 Discussion and Prospect

6.1 Additional overhead of virtualization

According to the system evaluation, although the performance of the proposed accelerator virtualization framework has been improved significantly compared with the existing solutions, it still falls short of the ideal performance of the PCI passthrough solution. The time breakdown analysis reveals the communication module in the prototype system consumes too many cycles. The modification of two MSRs, FS.base and GS.base, is the most expensive operation, which can be optimized. In theory, we can avoid the MSR writing operation by adding a memory mapping to the address space of the server VM. As a large number of memory pages are involved, the mapping must collect all the memory areas that may be accessed through segment registers. Hence, we will investigate this optimization in the future work.

6.2 Safety analysis

Although there is a strong isolation among VMs, the delegated execution stated in this paper allows applications to switch between VM address spaces in the user mode, so it may pose a potential security risk. This section analyzes the security of the proposed accelerator virtualization framework. In this paper, the VMM and the server VM are considered as reliable parts, and it is assumed that a malicious user can only initiate an attack through guest VMs. The attack target can be the server VM or other guest VMs on the same physical server. We analyze the following attack modes:

(1) Attack by illegally switching VMFUNC: A malicious user can customize an application that contains the VMFUNC instruction in a guest VM under their control. The application can determine parameters without using the trampoline mechanism provided by Wormhole. This application attempts to switch the control flow to other non-server guest VMs, causing leakage of sensitive data. Wormhole handles such attack by limiting the EPTP list of each guest VM, with Item 0 as the current extended page table and Item 1 as the extended page table of the target server VM, and the remaining 510 items are forced to be filled with the invalid address 0. This leaves the guest VM with only two options when performing address space switching: switching to itself or to the bound server VM. If the parameter that the malicious application passes to the VMFUNC instruction is bigger than 1, this application will exit and be intercepted by the VMM, without affecting other VMs.

(2) Attack by illegal back-end function jumping: A malicious user may try to tamper with the addresses in pointer arrays of back-end processing functions, which are mapped to the guest VM address space. For example, the user seizes the control flow by pointing function pointers to some functions that may leak sensitive data. When mapping the pointer arrays of back-end processing functions, Wormhole sets the read/write permissions of the memory page where the array is located to read-only in the extended page table. If there is an attempt to modify the pointer array of back-end processing functions, VMs are triggered to exit, and the VMM will capture and prevent subsequent operations.

7 Conclusion

Due to lack of usable, convenient, efficient and maintainable accelerator virtualization solutions, this paper proposed an accelerator virtualization framework Wormhole based on the hardware virtualization technology for popular cloud deep learning scenarios. It can support cloud service providers to develop customized accelerator virtualization systems that are easy to update. Based on API forwarding, the accelerator virtualization framework, Wormhole, innovatively introduced the abstraction of passive server VMs and fast inter-VM delegated execution with VMs as the isolation protection domain. It realized accelerator virtualization with high hardware resource utilization and low virtualization overhead while ensuring strong isolation among users. In addition, a prototype system for NVIDIA GPUs has been implemented on the mainstream QEMU/KVM platform. Test results demonstrate that Wormhole can be deployed on consumer servers conveniently. Compared with GVirtuS, a representative GPU virtualization solution expanded and optimized in this paper, Wormhole shows remarkable improvements in performance, verifying the effectiveness of this accelerator virtualization framework.

References

[1] Zhang ZK, Pang WG, Xie WJ, Lü MS, Wang Y. A survey of deep learning research for real-time applications. Ruan Jian Xue Bao/Journal of Software, 2020, 31(9): 2654–2677. http://www.jos.org.cn/1000-9825/5946.htm [doi: 10.13328/j.cnki.jos.005946]

[2] Jouppi NP, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. Proc. of the 44th Annual Int'l Symp. on Computer Architecture (ISCA 2017). New York: Association for Computing Machinery, 2017. 1–12. [doi: 10.1145/3079856.3080246]

[3] Zhang XL, Yang JH, Sun XQ, Wu JP. Survey of geo-distributed cloud research progress. Ruan Jian Xue Bao/Journal of Software, 2018, 29(7): 2116–2132. http://www.jos.org.cn/1000-9825/5555.htm [doi: 10.13328/j.cnki.jos.005555]

[4] Gao Q, Zhang FL, Wang RJ, Zhou F. Trajectory big data: A review of key technologies in data processing. Ruan Jian Xue Bao/Journal of Software, 2017, 28(4): 959–992. http://www.jos.org.cn/1000-9825/5143.htm [doi: 10.13328/j.cnki.jos.005143]

[5] Intel platform hardware support for I/O virtualization. 2006. http://www.intel.com.

[6] Herrera A. NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. White Paper, NVIDIA Corp., 2014. 1–18.

[7] Tian K, Dong YZ, Cowperthwaite D. A full GPU virtualization solution with mediated pass-through. In: Proc. of the 2014 USENIX Conf. on USENIX Annual Technical Conf. (USENIX ATC 2014). USENIX Association, 2014. 121–132.

[8] GRID Virtual GPU User Guide. 2020. https://docs.nvidia.com/grid/4.3/grid-vgpu-user-guide/index.html.

[9] Duato J, Peña AJ, Silla F, Mayo R, Quintana-Ortí ES. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. Proc. of the Int'l Conf. on High Performance Computing & Simulation. Caen, 2010. 224–231. [doi: 10. 1109/HPCS.2010.5547126]

[10] Montella R, Giunta G, Laccetti G, Lapegna M, Palmieri C, Ferraro C, Pelliccia V, Hong C-H, Spence I, Nikolopoulos DS. On the virtualization of CUDA based GPU remoting on ARM and X86 machines in the GVirtuS framework. Int'l Journal of Parallel Programming, 2017, 45(5): 1142–1163. [doi: 10.1007/s10766-016-0462-1]

[11] Armand F, Gien M, Maigné G, Mardinian G. Shared device driver model for virtualized mobile handsets. Proc. of the 1st Workshop on Virtualization in Mobile Computing (MobiVirt 2008). New York: Association for Computing Machinery, 2008. 12–16. [doi: 10.1145/1622103.1622104]

[12] Zhang YQ, Wang XF, Liu XF, Liu L. Survey on cloud computing security. Ruan Jian Xue Bao/Journal of Software, 2016, 27(6): 1328–1348. http://www.jos.org.cn/1000-9825/5004.htm [doi: 10.13328/j.cnki.jos.005004]

[13] Yu QQ, Dong MK, Chen HB. Memory-assisted synchronization mechanism for hardware transactions in a virtual environment. Ji Suan Ji Ke Xue Yu Tan Suo/Journal of Frontiers of Computer Science and Technology, 2017, 11(9): 1429–1438.

[14] Wu S, Wang K, Jin H. Research status and prospect of operating system virtualization. Ji Suan Ji Yan Jiu Yu Fa Zhan/Computer Technology and Development, 2019, 56(1): 58–68.

[15] Liu YT, Chen HB. Virtualization security: Opportunities, challenges and future. Wang Luo Yu Xin Xi An Quan Xue Bao/Chinese Journal of Network and Information Security, 2016, 2(10): 17–28.

[16] Huang X, Deng L, Sun H, Zeng QK. Hardware virtualization-based secure and efficient kernel monitoring model. Ruan Jian Xue Bao/Journal of Software, 2016, 27(2): 481–494. http://www.jos.org.cn/1000-9825/4866.htm [doi: 10.13328/j.cnki.jos.004866]

[17] Intel 64 and ia-32 architectures software developer's manual volume 3c. https://software.intel.com/en-us/articles/intel-sdm

[18] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII). New York: Association for Computing Machinery, 2006. 2–13. [doi: 10.1145/1168857.1168860]

[19] Liu WJ, Wang LN, Tan C, Xu L. VMFUNC-based virtual machine introspection trigger mechanism. Ji Suan Ji Yan Jiu Yu Fa Zhan/Computer Technology and Development, 2017, 54(10): 2310–2320.

[20] Liu YT, Zhou TY, Chen KX, Chen HB, Xia YB. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. Proc. of the 22nd ACM SIGSAC Conf. on Computer and Communications Security (CCS 2015). New York: Association for Computing Machinery, 2015. 1607–1619. [doi: 10.1145/2810103.2813690]

[21] Mi ZY, Li DJ, Yang ZH, Wang XR, Chen HB. SkyBridge: Fast and secure inter-process communication

for microkernels. Proc. of the 14th EuroSys Conf. 2019 (EuroSys 2019). New York: Association for Computing Machinery, 2019. 1–15. [doi: 10.1145/3302424.3303946]

[22] Shi L, Chen H, Sun J, et al. vCUDA: GPU-accelerated high-performance computing in virtual machines. IEEE Trans. on Computers, 2012, 61(6): 804–816. [doi: 10.1109/TC.2011.112]

[23] Zhang HL, Fang BX, Hu MZ, Jiang Y, Zhan CY, Zhang SF. Survey of Internet measurement and analysis. Ruan Jian Xue Bao/Journal of Software, 2003, 14(1): 110–116. http://www.jos.org.cn/1000-9825/20030117.htm

[24] Russell R. Virtio: Towards a de-facto standard for virtual I/O devices. SIGOPS Operating Systems Review, 2008, 42(5): 95–103. [doi: 10.1145/1400097.1400108]

[25] Kalia A, Kaminsky M, Andersen DG. Using RDMA efficiently for key-value services. Proc. of the 2014 ACM Conf. on SIGCOMM (SIGCOMM 2014). New York: Association for Computing Machinery, 2014. 295–306. [doi: 10.1145/2619239.2626299]

[26] Zhang XT, Zheng X, Wang Z, Yang H, Shen YB, Long X. High-density multi-tenant bare-metal cloud. Proc. of the 25th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2020). New York: Association for Computing Machinery, 2020. 483–495. [doi: 10.1145/3373376.3378507]

[27] Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. Proc. of the ACM SIGGRAPH 2008 Classes (SIGGRAPH 2008). New York: Association for Computing Machinery, 2008. 1–14. [doi: 10.1145/1401132.1401152]

[28] Bellard F. QEMU, a fast and portable dynamic translator. Proc. of the Annual Conf. on USENIX Annual Technical Conf. (ATEC 2005). USENIX Association, 2005. 41.

[29] Jia YQ, Shelhamer E, Donahue J, et al. Caffe: Convolutional architecture for fast feature embedding. Proc. of the 22nd ACM Int'l Conf. on Multimedia (MM 2014). New York: Association for Computing Machinery, 2014. 675–678. [doi: 10.1145/2647868.2654889]

[30] Le Cun Y, Bengio Y, Hinton G. Deep learning. Nature, 2015, 521: 436–444. https://doi.org/10.1038/nature14539

[31] Forsyth DA, Ponce J. Computer Vision: A Modern Approach. Prentice Hall Professional Technical Reference, 2012.

[32] Weikum G. Foundations of statistical natural language processing. SIGMOD Record, 2002, 31(3): 37–38. [doi: 10.1145/601858.601867]

[33] Krizhevsky A, Sutskever I, Hinton GE. ImageNet classification with deep convolutional neural networks. Communications of the ACM, 2017, 60(6): 84–90. [doi: 10.1145/3065386]

[34] Lecun Y, Bottou L, Bengio Y, Haffner P. Gradient-based learning applied to document recognition. Proc. of the IEEE, 1998, 86(11): 2278–2324. [doi: 10.1109/5.726791]

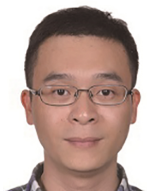
[35] Deng L. The MNIST database of handwritten digit images for machine learning research [Best of the Web]. IEEE Signal Processing Magazine, 2012, 29(6): 141–142. [doi: 10.1109/MSP.2012.2211477]

Dingji Li, bachelor. His research interests include the operating systems and virtualization systems.

Baodong Wu, Ph.D., engineer. His research interests include the high-performance computing, virtualization and container scheduling.

Zeyu Mi, Ph.D., assistant researcher, CCF student member. His research interests include the operating systems and system virtualization and safety.

Xun Chen, engineer. His research interests include the GPU virtualization and AI cloud scheduling systems.



Yongwang Zhao, Ph.D., associate professor, doctoral supervisor. His research interests include the formalization methods and operating systems.

Haibo Chen, Ph.D., professor, doctoral supervisor, CCF professional member. His research interests include the operating systems, concurrent and distributed systems.

Zuohua Ding, Ph.D., professor, doctoral supervisor, CCF senior member. His research interests include the requirement engineering, adaptive software systems and formalization methods.