软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.006068]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

利用跨虚拟机零下陷通信的加速器虚拟化框架*

李鼎基[1], 麋泽羽[1], 吴保东[2], 陈逊[2], 赵永望[3], 丁佐华[4], 陈海波[1]

[1](上海交通大学 软件学院, 上海 200240)

[2](北京市商汤科技开发有限公司, 北京 100080)

[3](浙江大学 网络空间安全学院, 浙江 310007)

[4](浙江理工大学 信息学院, 浙江 310018)

通讯作者: 陈海波, E-mail: haibochen@sjtu.edu.cn

摘 要: 人工智能技术的长足发展对于云计算的算力提出了更高的要求,云服务提供商在数据中心内添置了拥有大量并行计算单元的加速器,这些加速器需要与已有的虚拟化平台相结合以进行计算资源的划分.当前主流的加速器虚拟化方案是通过 PCI 透传的方式,但是该方式不支持细粒度的资源划分;部分特定型号的加速器还支持了时分复用的方案,通过硬件与虚拟机监视器配合划分计算资源和时间片,但是该方案可移植性差,对于任何新型加速器的适配都要重新开发,固定的资源划分策略也导致可扩展性有限;另有基于 API 转发的方案,通过分离式驱动的模式将虚拟机的请求转发给后端驱动处理,而转发通信的过程中存在着性能瓶颈.本文提出了 **Wormhole**,一种基于 C/S 架构的、支持跨虚拟机快速代理执行的加速器虚拟化框架,旨在为上层用户提供高效透明的加速器 API 转发虚拟化的同时保障多用户间的强隔离性.本框架利用硬件虚拟化技术,允许 CPU 控制流在虚拟机间快速切换而不触发任何下陷,大幅降低了虚拟机间通信带来的虚拟化性能开销.实验结果表明,**Wormhole** 的原型系统相较于具有代表性的开源虚拟化方案 GVirtuS 在经典模型的训练测试中能够有高达 5 倍的性能提升.

关键词: 虚拟化;加速器;人工智能;代理执行;虚拟机间通信

中图法分类号: TP311

中文引用格式: 李鼎基,麋泽羽,吴保东,陈逊,赵永望,丁佐华,陈海波.利用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). http://www.jos.org.cn/1000-9825/6068.htm

英文引用格式: Li DJ, Mi ZY, Wu BD, Chen X, Zhao YW, Ding ZH, Chen HB. Accelerator virtualization framework based on inter-VM exitless communication. Ruan Jian Xue Bao/Journal of Software, 2020,31(10) (in Chinese). http://www.jos.org.cn/1000-9825/6068.htm

Accelerator Virtualization Framework based on Inter-VM Exitless Communication

LI Ding-Ji[1], MI Ze-Yu[1], WU Bao-Dong[2], CHEN Xun[2], ZHAO Yong-Wang[3], DING Zuo-Hua[4], CHEN Hai-Bo[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, Zhejiang Sci-Tech University, Hangzhou 310018, China)

* 基金项目: 广东省重点领域研发计划(2020B010164003); 国家杰出青年科学基金(61925206); 上海市科委高技术支持计划(19511121100)

Foundation item: Key-Area Research and Development Program of Guangdong Province (2020B010164003); The National Science Fund for Distinguished Young Scholars (61925206); The HighTech Support Program from Shanghai Committee of Science and Technology (19511121100)

收稿时间: 2020-02-10; 修改时间: 2020-04-04; 采用时间: 2020-05-09; jos 在线出版时间: 2020-06-10

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 another 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 accelerator virtualization with API forwarding while ensuring strong isolation between multiple users. By leveraging hardware virtualization feature, the framework minimizes performance degradation through exitless cross-VM control flow switch. Experimental results show that **Wormhole**'s prototype system can achieve up to 5 times performance improvement over the classic open-source virtualization solution such as GVirtuS in the training test of the classic model.

Key words: Virtualization; Accelerator; Artificial Intelligence; Delegated Execution; Inter-VM Communication

随着以深度学习[1] 为代表的计算密集型应用大量兴起,仅仅依靠 CPU 提供的算力已经显得捉襟见肘,开发者们纷纷将目光转向了 CPU 之外具有更强大计算能力的设备作为加速器.传统的通用加速器主要以 GPGPU 为主,而工业界也为了特定的目的开发出了以 TPU(tensor processing unit)[2] 为代表的专用加速器.数据中心是云计算(cloud computing)[3] 时代的核心基础设施,其中的虚拟化平台是云服务能够高效运行的基础保证.近年来,越来越多的人工智能服务提供商倾向于将其应用部署在云系统中,使得各类加速器需要被整合进已有的虚拟化平台中,因此有关加速器虚拟化的需求也应运而生[4] .虽然目前市面上已经出现了若干种加速器虚拟化的解决方案,但是这些方案在现实场景中的应用仍然存在着各类限制和挑战.

目前主流的加速器虚拟化方式是 PCI 透传(passthrough)的方式,以 Intel VT-d[5] 为代表的 I/O 设备虚拟化方案会将加速器设备直通到客户虚拟机中.这种虚拟化方式绕开虚拟机监视器(virtual machine monitor, 简称 VMM)的干预把加速器全权交由客户虚拟机管理,虽然获得了与裸金属(bare-metal)环境几乎无异的性能,但是无法细粒度地虚拟化给多台客户虚拟机共享使用.因此该类虚拟化方案使虚拟化平台失去了计算资源弹性分配的能力,较差的可扩展性让虚拟机监视器不能灵活地在多台虚拟机之间动态调度计算资源.

一些加速器制造商也提供了以 Nvidia Grid[6] 和 gVirt[7] 为代表的虚拟化方案,通过让设备驱动程序与加速器配合以达到硬件资源的划分与时分复用.这些方案在虚拟机监视器协助下对于加速器的特定操作进行干预,其余操作则与 PCI 透传方式类似,将加速器的运行时间公平地分配到给各个客户虚拟机.然而对于目前的加速器而言,时分复用的方案的可用性并不高:一方面,现在成熟的时分复用方案不仅需要在软件层面配合特定的驱动程序,硬件层面上也对加速器的型号有着严格的限制[8] ,导致大部分普通加速器无法使用此虚拟化功能.另一方面,该类虚拟化方案可移植性差，在每个新型的加速器出现时都需要重新进行针对性的开发,而资源划分策略无法自由的调整也导致可扩展性差。

另外也有以 rCUDA[9] 和 GVirtuS[10] 为代表的基于 API 转发方式的虚拟化方案.该类方案均建立在分离式的驱动模型[11] 基础上,在动态链接库层面进行虚拟 vGPU 的抽象.该模型将设备驱动划分为前端驱动(frontend driver)和后端驱动(backend driver)两部分,其中后端驱动程序扮演服务器的角色,并将从前端驱动程序接收到的请求转换为实际与底层硬件设备交互的驱动程序调用.由于需要进行前后端驱动程序间的通信,还涉及到数据的序列化、内存的额外拷贝等操作,相较于原生的非虚拟化方案,性能会有较大程度的损失,具体损失程度主要取决于通信部分的性能以及所传输数据量的大小.

为了解决目前主流的加速器虚拟化面临的问题,本文基于硬件虚拟化技术,提出了一种针对加速器的虚拟化框架,目的是:1)针对加速器提供方便可用的多租户虚拟化方案,提高硬件资源利用率;2)保证用户间强隔离性与安全性;3)尽可能降低虚拟化带来的额外性能开销.为了满足目的 1,本文的虚拟化框架放弃了 PCI 直通的虚

拟化方式,选择了基于 API 转发方法的 C/S 架构,参考了成熟的 I/O 设备虚拟化方案,通过前后端的分离来支持多租户;为了满足目的 2,本文将虚拟机作为基本保护域以保证多租户场景下的强隔离性[12] ,而不是直接在物理主机的操作系统上使用诸如 Kubernetes[13] 的容器编排系统[14] ;硬件虚拟化技术的 VMFUNC 功能允许应用在非特权模式下切换扩展页表,为了满足目的 3,本文借助该特性实现了 CPU 控制流在虚拟机间零下陷地快速切换,重点优化了 API 转发过程中虚拟机间通信流程的性能,尽可能地将性能损失减到最小.经过实践证明,本文实现的**Wormhole**原型系统很好地解决了目前加速器虚拟化方案所面临的问题,在模型训练测试中相较于目前开源的类似方案取得了 1.4 到 5 倍的性能提升.

本文做出了如下贡献:

1) 基于 API 转发的方法,提出了一种针对加速器的、基于跨虚拟机代理执行的虚拟化框架.
2) 提出了一种基于硬件虚拟化技术的无下陷的虚拟机间通信加速机制.
3) 在 KVM 上,将虚拟化框架应用于常见的 Nvidia GPU,实现并支持了主流深度学习训练框架 Caffe.
4) 扩展并优化了开源 API 转发框架 GVirtuS,使其同样支持深度学习框架,经过对比测试表明,本文提出的虚拟化框架相较于优化后 GVirtuS 的性能有很大提升.

1 背景知识

1.1 虚拟机系统简介

基于虚拟机的虚拟化方案,是通过为客户操作系统提供一套完整的虚拟硬件资源来实现虚拟化的.这种对于物理硬件进行虚拟化的方案具有很多优点,例如对于客户操作系统透明,并且允许运行未经修改的操作系统,具有很强的兼容性[13] .由于每一个虚拟机都拥有其独立的操作系统、函数库、存储资源等,所以从安全性的角度来看,虚拟机的隔离性相当出色[15] .

当前数据中心内主流的虚拟化方案是以 KVM 为代表的基于主机操作系统的虚拟化架构.如图 11 所示,对于 CPU 的虚拟化,虚拟机监视器将每个物理 CPU 划分为一个或多个虚拟 vCPU 分配给不同的客户虚拟机使用,所有的 vCPU 都会由虚拟机监视器统一管理和调度;内存的虚拟化方面,现代虚拟机系统会为每个虚拟机分配一个扩展页表(extended page table,简称 EPT),通过限制地址翻译来保证不同的虚拟机处于不同的地址空间中[16] ;I/O 的虚拟化通常会复用主机操作系统中的原生驱动程序,当虚拟机内发起 I/O 请求后会经过虚拟机监视器转发给主机操作系统上的 I/O 代理模块(如 QEMU)进行处理.

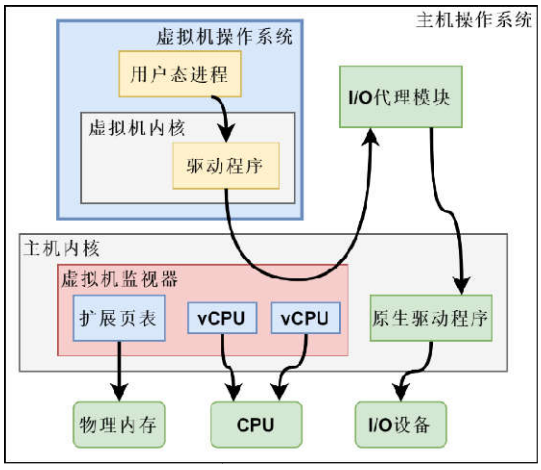

Fig.1 Architecture of host-based virtualization

图 1 基于主机操作系统的虚拟化架构图

1.2 硬件虚拟化技术—VMFUNC

为了提高虚拟化系统的性能,以 Intel 为代表的各大 CPU 制造商纷纷在其 CPU 产品上添加了硬件虚拟化技术[17] ,其中针对内存的硬件虚拟化已经基本替代了原本的影子页表(shadow page table,简称 SPT)[18] 方式,成为了默认的内存虚拟化方式.现代虚拟机系统中,客户虚拟机想要访问物理内存,需要经过两级地址翻译:第一级是根据虚拟机内的页表(page table,简称 PT)进行客户虚拟地址(guest virtual address,简称 GVA)到客户物理地址(guest physical address,简称 GPA)的翻译,第二级是根据虚拟机监视器配置的扩展页表进行客户物理地址到主机物理地址(host physical address,简称 HPA)的翻译.

从 Intel 的第四代 CPU 开始,CPU 指令集中加入了 VMFUNC[19] 这一硬件指令,允许客户虚拟机能够在非特权模式(non-root mode)执行一些特定的虚拟机相关操作而不会引发任何下陷到虚拟机监视器的行为.截至目前,VMFUNC 指令仅仅支持扩展页表指针(EPT pointer,简称 EPTP)切换这一功能,该功能允许客户端虚拟机在非特权模式由一条 VMFUNC 指令执行 EPTP 的切换.为了防止切换到错误的内存区域,需要预先在虚拟机监视器中为对应客户虚拟机配置 EPTP 列表(EPTP list),应用程序只能从该列表中选择合法的 EPTP,而一个 EPTP 列表中最多可以容纳 512 个 EPTP.

VMFUNC 指令由于其无下陷的特点相较于传统方法有着很大的性能优势,在该指令出现之前,如果一个客户虚拟机试图完成更改 EPTP,需要由非特权模式切换到特权模式修改 VMCS 中 EPTP 域的值,然后再由特权模式返回到非特权模式,其中仅仅非特权模式到特权模式的切换就需要花费不少于 300 个 cycle.相比之下,一条 VMFUNC 指令可以完成与上述整个流程同样的功能,在开启了 VPID(virtual processor identifier)功能的情况下只需要花费 134 个 cycle.VMFUNC 也因此被很多已有工作[20] [21] 所采用.

2 现有工作分析

根据上文所述的背景,本文选择了 API 转发的方法作为 **Wormhole** 的基础方法以兼顾性能和可用性.为了能够针对性地解决现有基于 API 转发的虚拟化方案中存在的关键问题,本章节调研了目前公开的相关工作并进行了深入的测试与分析.

2.1 客户端与服务端交互模式的问题

在 API 转发方法下,服务端进程与客户端进程的交互模式有多种选择.

1) 以 GVirtuS 为代表的方案选择了 Host-VM 的模式,即将服务端进程放置在主机操作系统中,客户端进程放置在虚拟机或容器中.该模式下,加速器的管理与主机操作系统耦合,主机操作系统的内核需要分饰两角,不仅要扮演客户虚拟机的管理者角色,还要负责运行加速器的驱动程序.这样既打破了单一功能原则(single responsibility principle),又只能同时支持一个版本的加速器驱动,给整个系统的运维带来了困难.例如,当操作系统不支持动态地升级驱动程序时,可能需要重启使新版本驱动生效,但是正在运行着的客户虚拟机显然是无法避免受到影响的,在云服务提供商看来这是无法接受的.
2) 以 vCUDA[22] 为代表的的方案选择了 VM-VM 的模式,服务端进程与客户端进程分别放置在不同的客户虚拟机中,可以选择多种跨虚拟机的通信方式进行数据交换.传统的基于网络或共享内存的通信方式由于系统调度等因素造成了较大的性能损失,而近来有一些工作[21] 已经提出了虚拟化环境下的快速通信方式,但是对于具有较复杂操作系统(如主流的 Linux 系统)的虚拟机无能为力.此外随着加速器种类以及配套软件的不断更新,这类方案也未能提出适合的配套进化措施.通信模块的性能问题

现有的 API 转发方案的主要问题表现在性能方面,而主要的性能损失来自于通信模块.从目前了解到的基于 API 转发的各类系统来看,通信方式主要分为以下几种:

1) TCP/IP 通信方式[23] .由于需要经过多次的内存拷贝,造成了大量的额外开销:以主流的 Linux 操作系统利用套接字(socket)进行单向 TCP 传输为例,用户态进程在发送数据时需要先将待发送的数据拷贝到内核中缓冲区中,然后内核中的 TCP 栈会利用本地网卡将数据发送给目标地址.从上述流程中可以

看出,为了在两个进程间拷贝一段数据,TCP/IP 通信方式引入了两次额外内存拷贝,如果考虑 I/O 虚拟化,额外内存拷贝的次数可能会翻倍,以常见的 Virtio 方式[24] 为例,一次单向 TCP 通信会增加两次虚拟机到主机内核缓冲区的内存拷贝.除此之外,在服务端进程等待客户端请求时,CPU 会陷入睡眠或调度,等到网卡收到数据后会发送中断唤醒 CPU,这些异步的操作也会带来不小的延迟.

2) 共享内存(shared memory)方式.该通信方法通过在服务端进程和客户端进程之间建立一段共享的内存映射,消除了额外内存拷贝开销.然而单纯的共享内存并未提供在数据拷贝完成后的通知机制,目前常见的通知机制是在共享一个信号量(semaphore)作为能否修改共享内存的标志.由于双方需要主动地轮询信号量,这样会导致 CPU 花费大量时间在没有实际作用的轮询和调度上,造成较高的延迟.

3) 远程直接内存访问(remote direct memory access,简称 RDMA)方式.RDMA 允许一台服务器直接访问另一台服务器上的内存而无需任意一方的操作系统参与[25] ,意味着可以支持零拷贝(zero-copy)从而降低延迟以及 CPU 的性能损失,RDMA 的驱动会直接复用用户态进程的内存来进行传输且完全无需 CPU 参与.基于 RDMA 的通信方式有着优秀的性能与较低的延迟,然而由于需要购置专用的 RDMA 网卡并且安装专用的驱动,在运维方面会产生一笔较大的成本,降低了该类方案的可用性.

Table 1 List of APIs frequently called during Neuron Layer testcase

表 1 Neuron Layer 测试中被高频调用的 CUDA API 列表

CUDA API 名称	调用次数
cudaMemcpy	290,122 次
curandSetPseudoRandomGeneratorSeed	268,537 次
curandSetGeneratorOffset	268,536 次
cudaLaunch	142,394 次
cudaPeekAtLastError	127,246 次

本章节以 Caffe 中 Neuron Layer 测试为例,对于 API 转发模式的虚拟化方案进行了一系列测试分析,该测试用例在一次运行过程中调用了超过 110 万次、共计 40 种不同的 CUDA API,表 11 列出了调用频率前 5 的 API 明细.测试之前本文说先对于 GVirtuS 做了扩展和优化,将其以 VM-VM 模式部署在两台处于同一物理服务器的虚拟机中,通信模块以共享内存作为参数拷贝的载体,TCP/IP 作为参数拷贝完成的通知机制.

Fig.2 Analysis of the average time cost of Neuron Layer in GVirtuS

图 2 GVirtuS 中 Neuron 测试用例平均耗时占比分析

接下来本文对于 Caffe 中 Neuron Layer 测试用例的 API 转发流程进行了耗时分析,将虚拟化时的一次 CUDA API 调用流程划分为以下三个部分:1)额外内存拷贝耗时,包含数据与共享内存间序列化与反序列化的

时间开销;2)通知机制耗时,包含虚拟机之间通知对方共享内存可用的时间开销;3)原生 API 执行耗时,其中只有原生 API 执行耗时是不可避免的有效工作时间,剩余的耗时均属于通信模块的额外性能开销.经过测试,各部分平均耗时占比如图 22 所示,测试中 API 调用的总时间为 283,750,417,817 个 cycle,可以看到额外性能开销占整个流程耗时的 88%以上(250,528,017,067 cycle),其中额外内存拷贝耗时仅占不到 1%(1,910,231,774 cycle),通知机制的耗时占比高达 87%(248,617,785,293 cycle),因此现有虚拟化系统中的通知机制是造成性能问题的最大瓶颈,也是本文针对进行优化的重点.

2.2 CPU利用率的问题

在原生物理服务器的环境下,调用加速器的过程中一个进程的 CPU 有效运行时间 $T_{有效}$可以主要划分为两部分:1)在用户态运行应用程序功能的时间,2)在内核态运行驱动程序与加速器交互的时间,其余时间由于没有产生真正有用的结果均可以视作无效运行时间 $T_{无效}$.因此在加速器虚拟化情况下,衡量一个虚拟化框架性能和效率的重要指标就是虚拟化后 CPU 的有效运行时间占总运行时间的比例 $T_{有效} / (T_{有效} + T_{无效})$.

在理想情况下,虚拟化后的比例和原生物理环境下的比例相同,即虚拟化没有造成额外性能开销.本小节所述的基于 API 转发的方案均使用的是主动式的交互方式,不论是通过网络还是共享内存通信,都需要服务端与客户端双方的 CPU 协助完成,而实际有效运行时间只有客户端 CPU 执行应用程序以及服务端 CPU 执行驱动程序的时间.为降低调度导致的延迟,正常情况下服务端与客户端都会被绑定在不同的物理 CPU 上,如图 33 所示,虚拟化时一次转发调用会耗费的 CPU 资源会是原生物理环境的 2 倍以上.

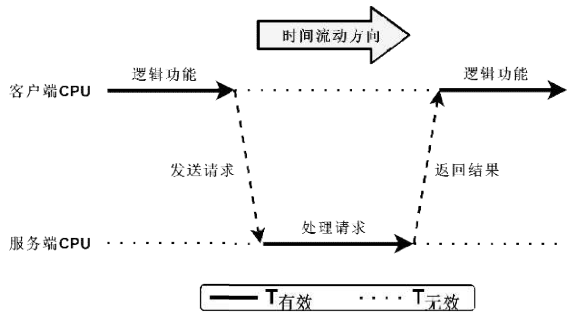

Fig. 3 CPU effective time in a virtualized environment

图 3 虚拟化环境下的 CPU 有效时间示意图

本小节使用了 Linux 操作系统内置的 time 命令,利用上一小节中的测试用例在原生物理环境与 GVirtuS 虚拟化环境下分别测量了 CPU 利用率.如表 22 所示,原生物理环境下只占用了 1 个 CPU,利用率高达 97.28%,而 GVirtuS 环境下占用了 2 个 CPU 且利用率分别仅为 46.61%和 38.52%.

Table 2 CPU utilization during Neuron Layer testcase

表 2 Neuron Layer 测试过程中的 CPU 利用率

	总时间	用户态执行时间	内核态执行时间	CPU 利用率
原生物理环境	6.947 s	4.494 s	2.264 s	97.28%
GVirtuS 后端	94.705 s	11.833 s	32.306 s	46.61%
GVirtuS 前端	93.990 s	6.498 s	29.706 s	38.52%

3 Wormhole 设计

Wormhole 加速器虚拟化框架的目标是面向数据中心的现实场景,在保证用户间强隔离性与安全性的前提下,针对加速器提供可用性高、性能好、支持多租户的高效虚拟化方案.本文利用虚拟机作为前后端驱动程序

的保护域,结合已被广泛应用的硬件虚拟化技术,改进现有虚拟化方案的不足,实现了一个灵活通用、易于维护、高性能的加速器虚拟化框架.图 44 展示了 **Wormhole** 的架构设计以及一次 API 转发调用的流程,本章将围绕该图的设计与调用流程示例进行详细展开.

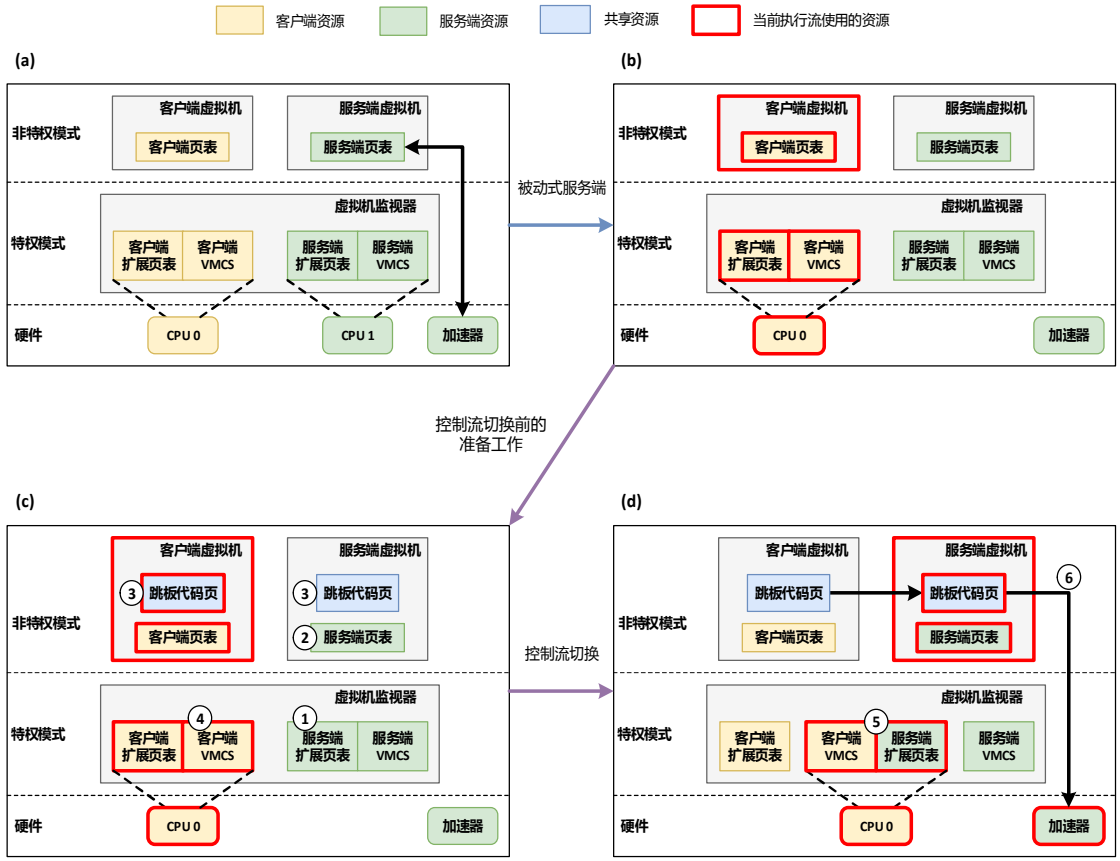

Fig.4 Architecture and invocation example of Wormhole

图 4 **Wormhole** 架构设计及调用流程示例

3.1 基于虚拟机的被动式服务端

Wormhole 的设计选择了一种基于虚拟机的被动式服务端(passive server VM)模式,相较于现有的交互模式有着性能以及灵活性方面的优势.所谓被动式服务端,就是服务端虚拟机在没有待处理的用户请求时不会主动占用任何 CPU 资源.

在本文的设计中,所有加速器会通过 PCI 直通的方式透传给专门用于管理的服务端虚拟机,从而实现加速器管理与主机操作系统解耦.服务端虚拟机在初始化阶段与一个普通的客户虚拟机别无二致,也可以拥有自己的 CPU,如图 44(a)所示,此时的服务端虚拟机独占 1 号 CPU,正在准备所需的通信接收模块等运行环境.在所有预先配置任务完成之后,服务端虚拟机会主动陷入类似快照的冻结状态,如图 44(b)所示,此时的服务端虚拟机已经不再拥有 CPU 资源,其原有的 1 号 CPU 资源可以被释放给其他的客户端虚拟机使用,这样就消除了不断等待客户端虚拟机的请求造成的服务端 CPU 资源浪费.

根据测试数据可知,现有加速器虚拟化方案的通信模块花费了大量时间在互相等待对方,例如服务端 CPU 在运行后端驱动程序等逻辑时,客户端 CPU 在结果返回之前一直处于空闲状态.因此在 **Wormhole** 中,完全可以利用发送完请求的客户端 CPU 在服务端虚拟机中代理执行来消除 CPU 资源的浪费,使得被动式的服务端虚拟

机变得可行.本设计提高了虚拟化后的 CPU 使用效率,节约下来的 CPU 资源可以分配给更多的客户虚拟机使用,解决了 2.2 节提到的问题.

在同一个物理服务器上可以同时存在多个被动式服务端虚拟机,每个服务端虚拟机可以被分配到不同数量的加速器.同时得益于虚拟机的隔离,不同的服务端虚拟机可以安装不同版本的加速器驱动程序以及配套的计算库等,以适配不同用户的需要,这也意味着如果物理服务器上接入了多种异构加速器,不同的加速器也可以经由不同的服务端虚拟机进行隔离,不会出现互相干扰的情况.本设计使得虚拟化框架能够非常容易的接纳快速更迭的不同种类加速器,解决了 2.1 节提到的问题.

3.2 基于控制流切换的主动式跨虚拟机通信

由于 **Wormhole** 的被动式服务端释放了自己的 CPU 资源,虚拟机间的通信必须由客户端采取主动式通信的方式配合服务端,因此本文提出了代理执行的方法,允许客户端执行流主动地进入服务端虚拟机继续执行.为了能够确保代理执行的正确性,要求 CPU 在不同虚拟机中运行时处于正确的地址空间中、能够访问到正确的指令和数据.在此基础上,必须进一步解决 2.2 节提到的性能问题才能达到 **Wormhole** 的高性能目标.因为虚拟化额外开销与下陷的次数紧密相关[26] ,为了获得优异的跨虚拟机通信性能,本文通过预先配置实现了控制流切换过程中的零虚拟机下陷.

本小节将以图 4 中一次代理执行的流程为例介绍本设计的核心思想,在图 4(b)中服务端虚拟机在初始化完毕后向虚拟机监视器注册了自己的服务端信息,之后客户端虚拟机被允许在该服务端虚拟机中代理执行,在图 4(c)中客户端虚拟机首先进行了一系列控制流切换前的准备工作,准备完成后切换执行流进入服务端虚拟机,实现了如图 4(d)所示的代理执行.得益于本文的设计, 图 4(c)和图 4(d)中与控制流切换相关的操作均会在非特权模式中完成,从而不会触发任何虚拟机下陷.一次完整的代理执行流程的调用可以被划分为 6 个步骤:

首先,客户端虚拟机发起与服务端虚拟机配对的请求,会下陷到虚拟机监视器中添加一些内存映射.下列①-③步属于一次性的预先配置,此阶段虽然会主动触发虚拟机下陷,但是并不在跨虚拟机通信的关键路径上,后续的代理执行不需要重复这些操作,配置完成之后的控制流切换过程会保持零下陷.

① 虚拟机监视器会在服务端虚拟机的扩展页表中,将客户端进程 CR3 的值映射到服务端进程页表的 HPA.本步骤的目的是为后续的 VMFUNC 指令实现虚拟机地址空间的正确切换做准备.对于虚拟机的二级地址翻译机制而言,VMFUNC 指令的现有功能只有切换控制第二级地址翻译的扩展页表,而第一级地址翻译依靠客户端进程 CR3 指向的页表,所以需要在 VMFUNC 指令的前后对应正确的页表以完成地址空间的变更.在本步骤添加映射后,客户端进程 CR3 的值在可以在客户端虚拟机的扩展页表中被翻译为客户端进程的页表内容,而同样的值可以在服务端虚拟机的扩展页表中被翻译为服务端进程的页表内容,保证了 VMFUNC 指令前后地址空间的正确翻译.同时本步骤使得第⑤步在虚拟机间切换地址空间时,仅在虚拟机用户态执行一条 VMFUNC 指令就可以达到 CR3 替换的等价效果.

② 虚拟机监视器会在服务端虚拟机的页表中,将客户端虚拟机 LSTAR MSR(model specific register,简称 MSR)的 GVA 值映射到服务端虚拟机系统调用入口代码页的 GPA.本步骤的目的是当执行流处于服务端虚拟机的地址空间中时,能够在用户态应用程序发起系统调用后正确地执行系统调用.现代操作系统大都使用 SYSCALL 指令进行系统调用,在执行 SYSCALL 指令时 CPU 会根据 LSTAR MSR 的值跳转到系统调用的入口.默认配置下在虚拟机中修改 LSTAR MSR 会触发虚拟机下陷由虚拟机监视器完成操作,因此本步骤使用添加映射的方式避免了后续控制流切换过程中造成虚拟机下陷.在添加了上述映射后,代理执行过程中发起系统调用时可以凭借客户端 LSTAR MSR 的值访问到服务端系统调用的入口,从而正常的与内核进行交互.

③ 虚拟机监视器中预先存放了一份跳板代码,配对时会将两个虚拟机高地址空间的某个相同的 GVA 映射到这份跳板代码页.本步骤的目的是保证虚拟机地址空间切换前后 CPU 指令流能够正确过渡.CPU 的程序计数器(program counter)是根据虚拟地址获取当前指令的,当 VMFUNC 指令执行完

后程序计数器的值会增加对应的长度,下一条指令已经处于服务端虚拟机的地址空间中.本步骤在两个虚拟地址空间中的相同位置提供了相同的指令,因此在地址空间切换前后CPU执行的仍然是连续的正确指令.

然后,客户端虚拟机返回到用户态,调用跳板代码页提供的接口,开始执行控制流切换相关的代码.下列④-⑥步需要在每次代理执行过程中重复,所以要求每个步骤都是零下陷以提高跨虚拟机通信的性能.

④. 在切换地址空间之前,客户端进程需要临时修改FS.base和GS.base两个MSR的值为服务端虚拟机中对应的MSR的值.本步骤的目的是保证在代理执行过程中,段寄存器的访问机制在服务端虚拟机中可以正常工作.现代操作系统中有大量数据需要经过段寄存器机制进行访问,它们的基地址存储在对应虚拟机的VMCS中的某些域中,不会随着地址空间的变化而改变,如果不做对应的调整,在代理执行期间就会根据错误的地址访问到错误的数据.本步骤在预先替换好了正确的MSR的值,这样一来在地址空间切换前后段寄存器访问机制取得的地址均为合法地址.虽然本步骤采用了替换MSR的值而非添加映射的方法,但是虚拟机监视器默认对于虚拟机内读写FS.base和GS.base两个MSR的操作是不做拦截的,所以依然不会引发任何的虚拟机下陷.

接下来,客户端虚拟机的执行流真正地切换到了虚拟机地址空间中.

⑤. 继续执行跳板代码页中的VMFUNC指令以切换地址空间,从下一条指令开始,CPU上仍然是客户端虚拟机的VMCS,但其中的扩展页表指针已经指向服务端虚拟机的扩展页表.本步骤真正进入了服务端虚拟机的地址空间中,当前生效的数据和资源如图中红色边框的区域所示.此时CPU的程序计数器指向共享的跳板代码页中的下一条指令,可以正确地继续执行余下的代码.得益于第①步添加的映射,在切换至目标地址空间时,无需像传统方案一样显式地修改CR3寄存器的值,从而规避了虚拟机调用特权指令而造成的下陷开销.

最后,客户端虚拟机进入服务端的应用程序中开始代理执行.

⑥. 获取跳板代码页的代理执行入口地址,跳转到服务端虚拟机中的后端处理程序中,通过原生的驱动程序与加速器进行交互.本步骤终于将原本运行在客户端虚拟机中的控制流顺利地切换到了服务端虚拟机中开始了代理执行,得益于上述5步的准备工作,后续的系统调用、中断等与加速器交互必需的复杂操作均可以正常的进行.

当代理执行的任务完成后需要从服务端虚拟机返回到客户端虚拟机中,此时倒序执行④-⑥步的逆向操作即可,在此不再赘述.按照本小节的设计,通过初始化阶段的一次性预先配置,后续频繁的跨虚拟机控制流切换操作并不会触发任何一次虚拟机下陷,为代理执行的快速高效提供了保障.

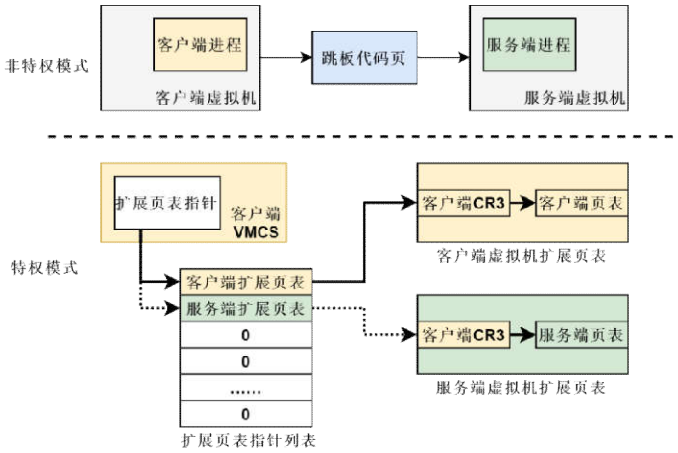

Fig.5 Switch virtual machine address space through VMFUNC

图 5 VMFUNC 指令切换虚拟机地址空间原理图

3.3 本设计的技术点

通过总结上一小节中的 6 步操作,本文提出了以下技术点:

1） 跨虚拟机地址空间的执行流快速切换.

主动式的跨虚拟机通信需要在执行流切换前后满足两点:切换前后地址翻译机制的正确以及切换前后 CPU 指令流的过渡.原本的一条 VMFUNC 指令只负责切换 GPA 到 HPA 的映射,因此需要有另外的措施负责切换到对应的页表.以一次客户端切换到服务端的地址空间为例,一种符合直觉的想法是在 VMFUNC 指令的前后修改 CR3 寄存器的值达到切换页表的目的,但是细看之下就会发现无法达到目的:程序计数器是根据虚拟地址获取当前指令的,如果 VMFUNC 前一条指令更换了页表,那么 CPU 在执行下一条指令时由于 GVA 到 GPA 映射的变化,实际上硬件看到的已经是一份无效的页表而导致运行错误;反之,如果试图在 VMFUNC 后一条指令更换页表,由于在 VMFUNC 指令执行完成后 GPA 到 HPA 的映射发生变化,实际访问到的下一条指令内容也不再是对于 CR3 的修改,同样会引发异常.

Wormhole 通过将第①步中 CR3 映射的添加与第⑤步中硬件虚拟化技术相结合,在不下陷到虚拟机监视器的情况下,实现了如图 5 所示的一条 VMFUNC 指令可以同时完成页表和扩展页表的切换,保证了切换前后地址翻译机制的正确,一条指令完成多项操作也大幅降低了跨虚拟机通信的开销.同时因为代理执行的特点,客户端虚拟机与服务端虚拟机均运行在同一个物理 CPU 上,避免了跨核中断(inter pocessor interrupt,简称 IPI)带来的高昂开销.

针对切换前后 CPU 指令流的过渡,**Wormhole** 在第③步操作中提供了一份共享跳板代码页以及一个代理执行专用的栈,代码页包括了 VMFUNC 指令以及一些上下文的保存逻辑,将它们映射到双方虚拟机的页表中相同的 GVA.这样客户端进程在需要进行代理执行时可以跳转到这份代码页,在地址空间切换成功后,由于在服务端进程在相同的地址共享这份代码页,程序计数器可以无缝过渡到 VMFUNC 的下一条指令继续执行,也不会污染双方进程原本的栈的内容.另外还需要服务端进程提前向虚拟机监视器注册代理执行函数的入口,跳板代码在控制流切换完成后会跳转到被注册的函数入口,正式开始在服务端进行加速器的访问操作.

2） 支持在控制流切换后系统调用、中断等复杂操作的正确处理.

一些以 SkyBridge[21] 为代表的前序工作同样使用了类似的代理执行思想,但是这些工作只是针对一些基于微内核的较简单的操作系统,面向的场景也仅仅是同一个操作系统中进程间的代理执行.在微内核的设计理念中,内核部分的代码量很小,通常只会保留几个最基础的管理功能,大量传统宏内核中的功能(例如设备驱动程序)被移出了内核态,作为一个专门的用户态进程提供相应的功能.因此在微内核场景下大多数功能不会在内核中完成,例如 I/O 相关的功能会由用户态的驱动程序负责,而中断发生后也会转交给用户态的特定进程进行中断处理.

然而目前在数据中心内部,主流的依然是基于宏内核的 Linux 操作系统.出于性能考虑,宏内核中集成了包括设备驱动、中断处理和资源管理在内的各类复杂功能,因此用户态应用程序在运行时会比较频繁地与内核进行交互.大量紧耦合的复杂功能带来了错误隔离方面的一些不便,宏内核中存在大量涉及段寄存器的非常规内存访问机制,如果发生一些数据访问的错误往往会造成例如系统崩溃等无法挽回的后果,所以在代理执行的过程中保证与内核交互的正确性是异常重要的.

现代操作系统大都使用 SYSCALL 指令进行系统调用, CPU 会根据 LSTAR MSR 的值跳转到系统调用处理函数的入口,所以 **Wormhole** 在第②步操作中在服务端虚拟机的地址空间中建立从客户端虚拟机 LSTAR MSR 到服务端虚拟机系统调用入口代码页的映射,保证可以通过客户端 LSTAR MSR 的值找到正确的系统调用入口.

出于权限分离与安全的考虑,宏内核中的用户态与内核态使用的包括栈在内的一系列内存结构是不同的,所以在用户态进程下陷到内核时需要保存用户态的上下文并切换到内核专用栈或中断专用栈,例如在 Linux 内核中这些栈顶的地址存储在一些 per-CPU(每个 CPU 一个)的变量中.在 AMD64 平台的 Linux

内核中,这些 per-CPU 的变量使用 GS 段寄存器来进行访问,它们的基地址存储在专门的 MSR 中,不会随着地址空间的变化而改变.**Wormhole** 通过第④步操作保证了控制流切换前后 GS 段内存访问机制的正确性.同样地,第④步操作也保证了 FS 段内存访问机制的正确性,例如 Linux 操作系统在 AMD64 平台上会利用段寄存器机制通过 FS:0x28 来访问一个特殊的“哨兵”(sentinel)值用于检查栈缓冲区溢出(stack buffer overflow)情况.

3） 控制流切换过程中零虚拟机下陷.

控制流切换在整个通信流程的关键路径上,而每次 API 转发调用都会有两次虚拟机间的控制流来回切换,因此要想尽量降低虚拟化额外开销则必须尽量缩短其所消耗的时间.虚拟化额外开销与虚拟机下陷的次数是强相关的,为了尽可能的消除特权模式和非特权模式间的切换耗时,**Wormhole** 的设计保证了控制流切换过程中不会主动触发任何的虚拟机下陷,所有配置操作均在非特权模式下完成.

在以 KVM 为代表的虚拟化平台上,非特权模式下对于 CR3 以及 LSTAR MSR 的修改操作会默认下陷到特权模式中,因此 **Wormhole** 在第①、②步中采用了添加映射的方式,虽然在地址空间切换的前后虚拟机看到的寄存器的值保持不变,但是借助地址翻译机制实际访问到了正确的物理内存区域.这样既实现了零下陷的特性,又达到了与执行特权操作同样的效果.非特权模式下对于 FS.base 和 GS.base 的 MSR 修改默认不会造成下陷,并且段寄存器的访问机制涉及的内存页数量较多、范围较广,不太适合采用在服务端地址空间添加映射的方法,故在第④步中发起虚拟机系统调用来临时更换两个 MSR 中的值.

4） 支持在控制流切换后扩展页表缺页的正确处理.

在扩展页表技术的支持下,现代虚拟化系统对于虚拟机的内存分配请求采用了惰性分配策略,即在客户虚拟机最初申请一块内存时只在其页表内添加 GVA 到 GPA 的映射,直到改内存区域第一次被访问时触发扩展页表缺页错误才会由虚拟机监视器在扩展页表中补充 GPA 到 HPA 的映射.在代理执行期间如果发生了虚拟机下陷,从虚拟机监视器的视角看来当前的下陷仍然来自客户端虚拟机,默认会根据 VMCS 中的相关信息对于客户端虚拟机执行下陷处理函数,而实际上造成这次下陷的原因来自于服务端虚拟机中,所以必须将处理对象从客户端虚拟机变更为服务端虚拟机.

Wormhole 的设计是,当发生了扩展页表缺页导致的下陷后,让虚拟机监视器判断是否是在代理执行过程中发生的缺页错误.如果是,则提取客户端虚拟机 VMCS 中缺页错误有关的参数,对于服务端虚拟机执行缺页处理函数,将 GPA 到 HPA 的映射添加到服务端虚拟机的扩展页表中.

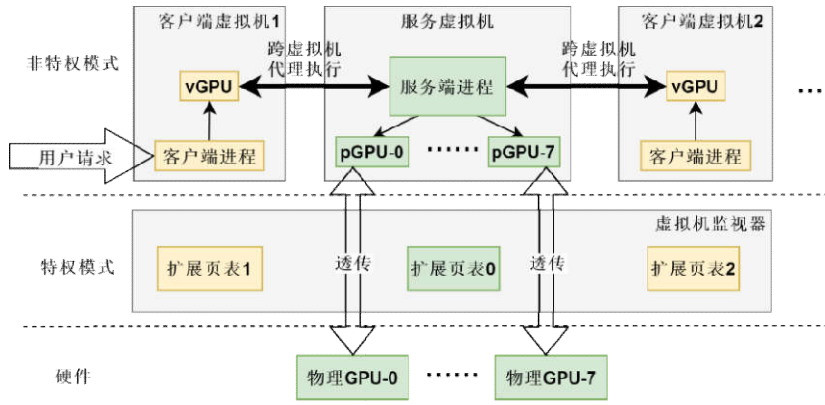

Fig.6 Architecture of prototype system

图 6 原型系统架构图

4 Wormhole 原型系统实现

为了验证本加速器虚拟化框架的设计,本文在 Intel 的 x86-64 平台上,基于主流的 Linux 操作系统以

QEMU-KVM 作为虚拟化平台,按照上文的设计对于常用的 NVIDIA GPU 实现了一个加速器虚拟化的原型系统,支持了 CUDA 9.0 版本.原型系统架构如图 6 所示,具体实现细节如下.

4.1 支持CUDA调用的API转发基础框架

目前主流的深度学习框架与 GPU 类加速器的交互主要通过调用 NVIDIA 公司推出的 CUDA[27] 统一计算 API.本文在收集了被使用到的 CUDA API 后发现,主流的深度学习框架用到的科学计算库主要有 cudart、cuBLAS、cuDNN 和 cuRAND,应用程序通过动态链接的方式调用这些计算库中的 CUDA API.

本文在实现时,将整个虚拟化系统的基础框架划分为前端模块、通信模块以及后端模块,其中前端模块放置在客户端虚拟机中,后端模块放置在服务端虚拟机中,通信模块用于在两个虚拟机之间实现代理执行.

(1) 在前端模块中,本系统为所有收集到的 CUDA API 按照官方文档实现了相同函数原型的桩函数,分别封装在对应计算库同名的桩函数库中(如 libcudart.so,libcudnn.so 等).这些桩函数库将通过修改环境变量 LD_LIBRARY_PATH 的方式代替客户端虚拟机中原生的计算库,实现对于应用程序 CUDA 调用的拦截.

(2) 在后端模块中,本系统首先使用 dlopen 预载用到的 CUDA 计算库,接着向虚拟机监视器注册后端处理函数的代理执行入口,然后会 fork 出与 CPU 核数等量的子进程以尽可能多地支持并发的前端请求,每个子进程可以对应一个前端模块.最后后端模块会进入冻结状态,等待来自前端的代理执行.

(3) 通信模块的具体实现将在本章节余下部分详细描述.

为了减少前后端通信的次数以降低通信开销,本系统对于常用的 CUDA 核函数(kernel)的转发方法做了与 GVirtuS、vCUDA 等前序工作类似的批处理(batching)优化.一次 CUDA 核函数的调用实际上依次分别调用了 3 种 CUDA API(一次 cudaConfigureCall → 一次或多次 cudaSetupArgument → 一次 cudaLaunch),因为只有最后的 cudaLaunch 是真正与设备交互的显式同步点,所以每次拦截到前两种 CUDA API 时不用立即转发到后端,可以与最后的 cudaLaunch 一起批量地处理.

较新的 CUDA 版本支持了统一虚拟内存(unified virtual addressing,简称 UVA)的特性,例如 cublasSdot 的指针参数既可能指向主机内存也可能指向设备内存,在原生环境下需要 GPU 的驱动程序对于内存拷贝的方向(比如从设备拷贝到主机内存)进行判断,必须根据源地址段和目的地址段的内存类型(主机内存地址或设备内存地址)进行决定.在本虚拟化系统中,由于 GPU 的驱动程序与应用程序处在不同的地址空间中,客户端虚拟机中的应用程序转发来的地址无法由服务端虚拟机中的驱动程序进行正确的检查和判断,因此本系统基于区间树(interval tree)实现了一套高效的 GPU 设备地址的追踪模块,对于诸如 cudaMalloc 等分配设备内存的 API 返回的设备内存区间进行记录,在拦截到使用了 UVA 特性的 CUDA API 后,使用追踪模块查询传入的内存地址参数所属的内存类型,然后依据具体情况进行处理.

按照 **Wormhole** 的设计,在服务器虚拟机启动前,需要主机操作系统将 GPU 设备通过 PCI 直通的方式实现透传,启动后需要配置好驱动程序和科学计算库等.双方虚拟机均需要下陷注册自己的相关信息,本系统修改了 KVM 中 CPUID 的下陷处理函数,在收到注册请求后会根据虚拟机的类型和请求完成相应的操作.

4.2 用于支持控制流切换的内存映射与用户态接口

包括跳板代码页在内的共享内存映射是主动式跨虚拟机通信的关键,本系统在 KVM 的 CPUID 处理函数中根据初始化时的虚拟机下陷指令,为服务端与客户端虚拟机按顺序预先配置了以下映射:

(1) **客户端 CR3、LSTAR 在服务端虚拟机中的映射.**这两种映射的添加是后续跳板代码页执行过程中零下陷的基础.本系统首先在服务端虚拟机下陷时的 VMCS 中读取到服务端 CR3 的值后,利用软件模拟的方式遍历服务端虚拟机的扩展页表,翻译得到服务端进程页表在主机物理内存中的 HPA.然后在客户端虚拟机下陷时的 VMCS 中读取到客户端 CR3 的值,再次遍历服务端虚拟机的扩展页表添加从客户端 CR3 到服务端页表 HPA 的映射.类似地,首先在服务端虚拟机下陷时的读取到服务端 LSTAR MSR 的值,利用服务端页表进行地址翻译获取对应的 GPA,然后在客户端虚拟机下陷时获取

客户端 LSTAR MSR 的值作为新的 GVA,接着在服务端页表中建立从该 GVA 到服务端系统调用代码页 GPA 的映射.

(2) **共享内存.**客户端虚拟机中前端模块拦截到的 CUDA API 参数需要转发给服务端虚拟机中的后端模块,存在大量 API 需要进行主机与设备间的内存拷贝.为了达到跨虚拟机传参的效果,本系统让 KVM 分配一块足够大的内存作为传参用共享内存,在服务端进程与客户端进程的高地址空间预留了一段起始 GVA,然后在 KVM 中添加双方应用程序页表与扩展页表的映射,使得 CPU 在切换地址空间前后可以通过预留的 GVA 访问到同一块物理内存.

(3) **过渡用共享栈.**在控制流从客户端进程切换到服务端进程的过程中有一段中间过渡期,为了不污染服务端进程与客户端进程原有的栈结构,本系统在 KVM 中分配了 16 个大小为 4KB 的页内存,映射到了双方进程的高地址空间中的相同 GVA,保证控制流切换前后栈结构的可用.

(4) **处理函数的指针数组在客户端虚拟机中的映射.**控制流从跳板代码页跳转到服务端进程地址空间时需要明确目标函数的位置,本系统在 KVM 中为每个服务端虚拟机中维护了一个函数指针数组,在服务端进程初始化时会下陷到 KVM 将所有可用的处理函数虚拟地址存入该数组.类似地,为了过渡时能够在用户态访问处理函数的指针数组,本系统也将其映射到了客户端进程的高地址空间中.

(5) **跳板代码页.**在上述准备工作完成后,本系统在 KVM 中存放了一份跳板代码页,其对用户态暴露了 delegate_to_server 函数接口,参数包括服务端虚拟机的偏移量和后端处理函数的偏移量,使得客户端虚拟机中的通信模块通过调用 delegate_to_server 切换到服务端虚拟机中对应的后端处理函数.跳板代码页被映射到了双方应用程序的高地址空间中的相同 GVA,客户端进程将该地址强制转换为函数指针后即可按照函数调用的方式调用 delegate_to_server 接口.跳板代码的逻辑是:a)在被客户端应用程序调用时,先将当前所有的寄存器压栈来保存上下文,然后保存将当前的栈指针并替换为过渡用的临时栈,最后发起 arch_prctl 系统调用替换 FS.base 和 GS.base MSR;b)调用 VMFUNC 指令切换到服务端进程的地址空间,此时完成了一次跨虚拟机通信;c)接着检查参数的合法性后从函数指针数组中读取后端处理函数的地址,将地址作为函数指针间接跳转进入后端处理函数执行;d)待后端处理函数返回后,调用 VMFUNC 指令切换回到客户端进程的地址空间;e)发起 arch_prctl 系统调用设置客户端进程的 FS.base 和 GS.base MSR,恢复为客户端进程原生的栈结构,然后从栈中恢复代理执行前的上下文.

4.3 服务端虚拟机的冻结

完成初始化后,服务端虚拟机的后端模块会从用户态进入冻结状态,此后服务端虚拟机的 CPU 资源可以释放给其他客户虚拟机使用,内存与 I/O 资源仍要保留供代理执行使用.要从用户态进入冻结状态的原因是在代理执行时客户端虚拟机的控制流也是从用户态切换而来,如果在内核态冻结则会导致内核的专用栈等数据结构被污染,代理执行过程中会造成内核的崩溃等严重错误.幸运的是,CPUID 指令在用户态与内核态都会无条件触发虚拟机下陷,因此后端模块会在用户态调用 CPUID 传递冻结指示参数下陷到 KVM 中,KVM 在收到冻结请求后会在 CPUID 处理函数中设置该虚拟机的冻结标志.在每个虚拟机的 vCPU 试图执行 VMRESUME 恢复运行之前,KVM 会检查该虚拟机的冻结标志,如果为真则拦截其 vCPU 并主动进入调度以释放 CPU 资源.

4.4 代理执行时的扩展页表缺页处理

使用 QEMU-KVM 虚拟化平台时,每个客户虚拟机从主机操作系统的角度来看,本质上都是一个可以利用 KVM 内核模块进行加速的用户态 QEMU 进程[28] ,所以每个虚拟机的内地址空间本质上都是对应 QEMU 进程的地址空间.一个正常运行的虚拟机如果触发了扩展页表的缺页错误,CPU 会发生原因为 EPT violation 虚拟机下陷,KVM 会根据缺页错误的 GPA 在当前 QEMU 进程的地址空间中进行处理.处理过程中,Linux 内核会首先依据名为 current 的 per-CPU 变量来获取当前 CPU 上运行着的进程描述符(task_struct 结构体),其中存放着与当前 QEMU 进程绑定的内存描述符(mm_struct 结构体),然后利用 Linux 内核的内存管理相关函数为该内存描述

符分配实际的物理内存,最后给扩展页表补充 GPA 到 HPA 的映射.

代理执行时如果触发了扩展页表缺页错误,下陷后 KVM 识别到的当前进程身份仍然是客户端虚拟机的 QEMU 进程,因此 KVM 会在客户端 QEMU 进程的地址空间中分配新的内存并向客户端虚拟机的扩展页表中添加映射.而实际上缺页错误发生在服务端虚拟机地址空间中,正确的操作应该是给服务端的 QEMU 进程分配新的内存并添加扩展页表映射.因此本系统修改了 KVM 的 EPT violation 处理函数,在发生缺页错误下陷后会首先判断当前是否正在代理执行.如果是,则会暂时将当前 current 变量存储并替换为初始化时记录的服务端 QEMU 进程的进程描述符,这样在内存分配和扩展页表映射时 KVM 的操作对象均为服务端虚拟机,待完成后再将 current 变量恢复.由于服务端虚拟机一直处于冻结状态,所以上述操作不会有数据冲突(data race)的风险.

4.5 丞待完善的部分CUDA特性

对于 CUDA 绑定内存特性(pinned memory)等由于驱动的闭源性未能支持,对于 CUDA 多流(stream)操作等异步 API 暂时会被转化为同步版的 API 调用,因此在 Host 和 Device 间的内存拷贝性能有一定影响.不过未完全实现的部分与本文的设计是正交的,并不会妨碍证明本加速器虚拟化框架设计带来的大幅性能提升.

5 系统评测

为了测试原型系统的性能,本文使用了一台支持 VMFUNC 硬件虚拟化特性的 Intel Haswell-E 消费级服务器作为测试平台,该测试平台的主要软硬件配置见表 33.本章节按照测试的粒度的从小到大主要分为三个部分,将 PCI 直通的虚拟化方案作为最高性能的基准线(baseline).

Table 3 Testbed configuration

表 3 测试平台配置信息

名称	Model/Version
CPU	Intel i7-5930K@3.5GHz (6 核 12 线程)
内存	40GB DDR4 2133MHz
GPU	NVIDIA Quadro GV100 (32GB HBM2)
操作系统	Ubuntu 19.10 (Eoan Ermine)
内核版本	Linux Kernel 4.19.56

为了对比目前公开可用的 GPU 虚拟化解决方案,本文选取了具有代表性的开源方案 GVirtuS 作为对照,由于最新的 GVirtuS 支持的 CUDA API 仍然非常有限,本文对 GVirtuS 的源码进行了补充使其支持了与本原型系统同样多的 CUDA API.此外,由于 GVirtuS 的通信模块对于 VM-VM 模式下的虚拟化仅支持 TCP/IP 方式,在 API 转发时额外的内存拷贝开销较大.本文为 GVirtuS 的通信模块添加了与原型系统相同的共享内存方式,消除了两个系统在内存拷贝开销上的差异,既提升了 GVirtuS 系统的性能,又保证性能测试的公平性.

过去的大部分原型系统仅仅支持了小部分的 CUDA API,选用的测试用例与现实场景中的应用程序相差较大,无法全面地反映出系统真实的性能表现.本文选取了流行的 Caffe[29] 作为基准测试程序,Caffe 是一个用 C++编写的深度学习框架,由于其清晰、高效的优点在深度学习领域[30] 中被广泛使用.

5.1 微型基准测试(Microbenchmark)

为了验证 **Wormhole** 的设计对于 2.1 小节提出的通信性能问题的提升,本小节对原型系统中 Neuron Layer 单元测试中所有 CUDA API 的转发流程作了时间拆分分析(time breakdown),调用过程中各部分的时间开销及占用总时间的百分比如图 77 所示.测试中 API 调用的总时间为 23,778,309,322 cycle,可以看到额外性能开销占整个流程耗时的比例从 GVirtuS 的 88%降到了 20%(4,660,728,884 个 cycle),其中额外内存拷贝耗时占 3.65%(866,946,457 个 cycle),系统调用修改 FS.base 和 GS.base MSR 的耗时占比 10.64%(2,530,932,006 个 cycle),余下包括 VMFUNC 在内的控制流量切换操作的耗时占比 5.31%(1,262,850,421 个 cycle).

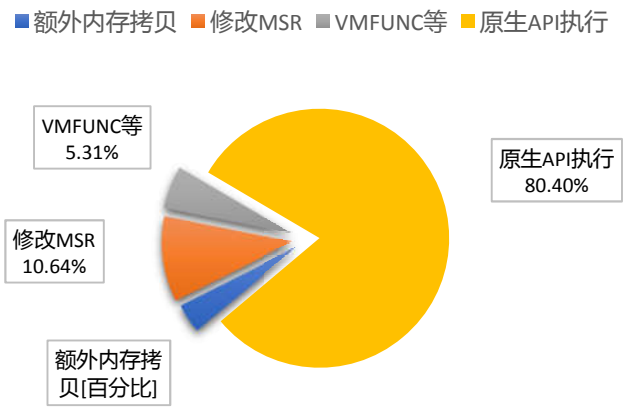

Fig.7 Analysis of the average time cost of Neuron testcase in the prototype system

图 7 原型系统中 Neuron 测试用例平均耗时占比分析

从绝对时间开销来看,本次测试中 GVirtuS 基于 TCP/IP 的通知机制消耗了 248,617,785,293 个 cycle,而 **Wormhole** 原型系统中的控制流切换机制仅消耗了 4,660,728,884 个 cycle.为了更细粒度地验证本设计在 3.3 小节提出的“跨虚拟机地址空间的执行流切换”(下称“快速切换”)和“执行流切换过程中零虚拟机下陷”(下称“零下陷”)两个技术点的效果,本小节增加了一次测试,在该测试中关闭了“零下陷”功能,即每次执行流切换前后主动下陷到虚拟机监视器更换 MSR.结果表明 API 调用的总时间为 29,638,851,151 个 cycle,而通信模块占用了 9,432,980,220 个 cycle,意味着“快速切换”技术点将 TCP/IP 通信开销降低了 2 个数量级,而“零下陷”技术点在进一步将执行流切换的开销降低了 50%.综上所述,本设计大幅优化了虚拟化带来的额外耗时,从微观角度证明了 **Wormhole** 的设计相较于以 GVirtuS 为代表的现有设计大幅降低了虚拟化带来的额外开销.

Table 4 The improvement of CPU utilization during Neuron Layer testcase

表 4 Neuron Layer 测试中的 CPU 利用率提升效果

	总时间	用户态执行时间	内核态执行时间	CPU 利用率
原生物理环境	6.947 s	4.494 s	2.264 s	97.28%
GVirtuS 后端	94.705 s	11.833 s	32.306 s	46.61%
GVirtuS 前端	93.990 s	6.498 s	29.706 s	38.52%
Wormhole	**12.145s**	**9.474s**	**2.661s**	**99.92%**

从 CPU 利用率来看,如表 4 所示，**Wormhole** 只占用了 1 个 CPU 的总时间 12.145s,其中用户态有效时间 9.474s、内核态有效时间 2.661s,利用率高达 99.92%,远高于 GVirtuS 环境下 2 个 CPU 的 46.61%和 38.52%,甚至要优于 PCI 直通方案的 97.28%,证明了 **Wormhole** 的设计相较于现有方案大幅提升了 CPU 的利用率.

5.2 神经网络层单元测试

本小节将使用 Caffe 官方提供的多种神经网络层的单元测试用例,从宏观角度体现 **Wormhole** 在不同神经网络层测试中的获得的性能提升,采用测试用例自带的时间统计工具来衡量各个测试的用时,时间单位为毫秒(millisecond),评价性能高低的标准是耗时越短性能越好.

本小节选取了现实场景中一些重要的神经网络层作为单元测试用例,主要有以下几类:计算机视觉[31] 领域常用的图像处理网络层:1)卷积层(下称 CONV)和逆卷积层(下称 DECONV);2)自然语言处理[32] 领域常用的循环网络层:循环网络层(下称 RNN)和长短期记忆网络层(下称 LSTM);3)深度神经网络中常用的规范化(normalization)网络层:批规范化网络层(下称 BN);4)深度神经网络中常用的激活(activation)网络层:各类激活函数网络层,包含 ReLU、Sigmoid、TanH 等常见激活函数,统称为神经元网络层(下称 Neuron).

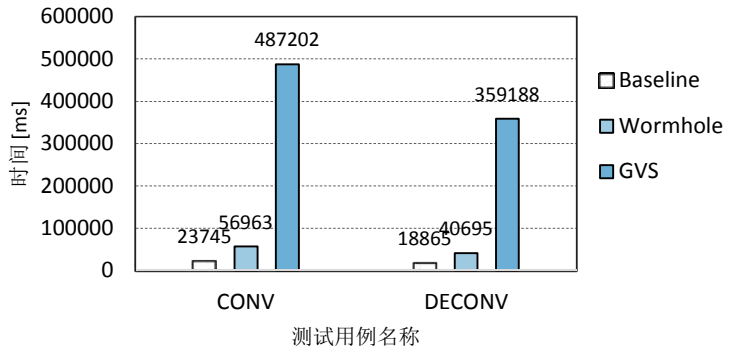

Fig.8 Performance comparison of image processing layer

图 8 图像处理层性能对比

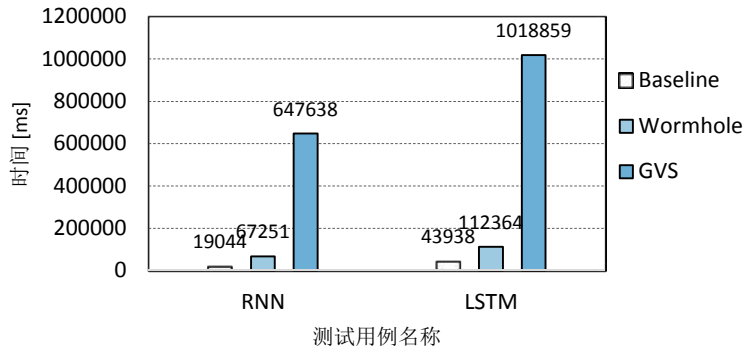

Fig.9 Performance comparison of recurrent network layer

图 9 循环网络层性能对比

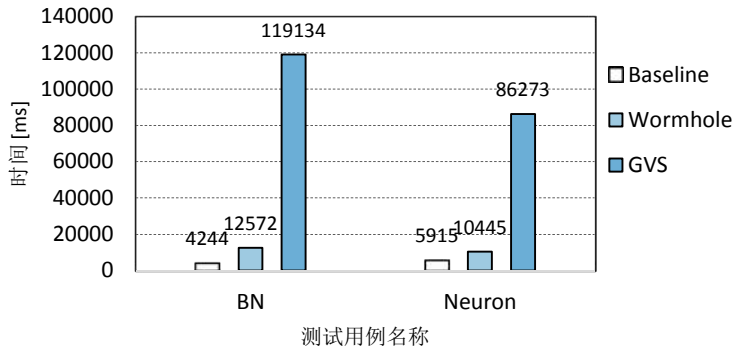

Fig.10 Performance comparison of normalization and activation layer

图 10 规范化层、激活层性能对比

测试结果与对比如图 8、图 9、图 10 所示,图中 Baseline 代表 PCI 直通虚拟机方式下的理想性能,Wormhole 为本原型系统的性能,GVS 为优化后的 GVirtuS 系统的性能:

在图像处理网络层方面,本原型系统相较于优化后的 GVirtuS 系统,在 CONV 测试中性能提升达到了 88.31%,在 DECONV 测试中性能提升达到了 88.67%.

在循环网络层方面,本原型系统相较于优化后的 GVirtuS 系统,在 RNN 测试中性能提升达到了 89.62%,在 LSTM 测试中性能提升达到了 88.97%.

在规范化层和激活层方面,本原型系统相较于优化后的 GVirtuS 系统,在 BN 测试中性能提升达到了 89.45%,在 Neuron 测试中性能提升达到了 87.89%.

5.3 经典模型训练测试

本小节选取了 AlexNet[33] 和 LeNet[34] 进行完整的深度学习模型训练测试,用以评测 **Wormhole** 以及本原型系统在完整的真实工作负载下的性能表现,采用深度学习框架自带的吞吐量统计工具作来衡量训练过程中的吞吐量,单位为迭代次数/秒(iter/s),评价性能高低的标准是吞吐量越大性能越好.同样地,本小节以 Baseline 代表 PCI 直通方式下的理想性能,Wormhole 为本原型系统的性能,GVS 为优化后的 GVirtuS 系统的性能:

LeNet 诞生于 1994 年,是最早的卷积神经网络之一,并且推动了深度学习领域的发展.本小节使用 MNIST[0] 作为数据集,批处理大小(batch size)为 100,基于 Caffe 进行 10000 次迭代训练.如图 11 所示,本原型系统相较于优化后的 GVirtuS 系统吞吐量提升了 5 倍.

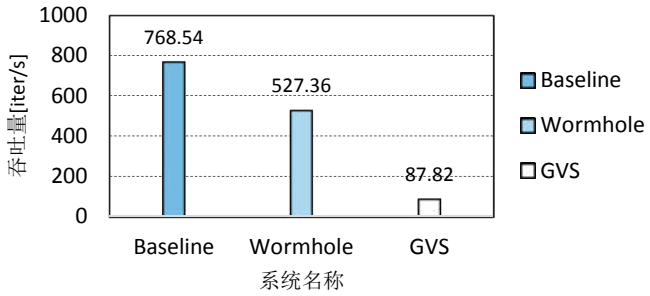

Fig.11 Performance comparison of LeNet training

图 11 LeNet 训练性能对比

AlexNet 于 2012 年被提出,首次在 CNN 中成功应用了 ReLU、Dropout 和 LRN 等,可以算是 LeNet 的一种更深更宽的版本,是现代深度 CNN 的奠基之作.本小节为消除大规模存储设备 I/O 造成的影响以方便测试,选用了模拟数据(dummy data)作为数据集,批处理大小为 64,基于 Caffe 进行 1800 次迭代训练.如图 12 所示,本原型系统相较于优化后的 GVirtuS 系统吞吐量提升了 1.4 倍.

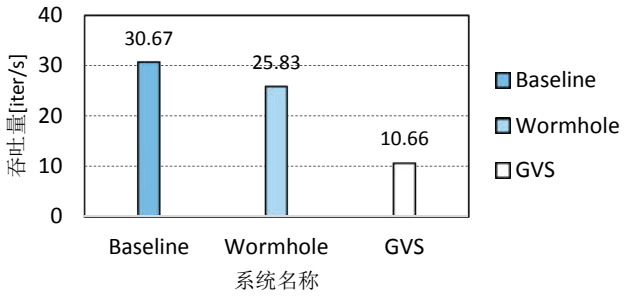

Fig.12 Performance comparison of AlexNet training

图 12 AlexNet 训练性能对比

6 讨论与展望

6.1 虚拟化额外开销

从系统评测章节可以看出,虽然本加速器虚拟化框架的设计相较于现有的解决方案有很大的性能提升,但是对比 PCI 直通方案的理想性能仍有一定的差距.在时间拆分分析部分,不难看出原型系统中通信模块的耗时仍然较多,其中耗时最长的操作是对于 FS.base 和 GS.base 两个 MSR 的修改.这两个操作存有优化的余地,理论上可以通过在服务端虚拟机地址空间中添加内存映射来回避写 MSR 的操作.由于涉及到的内存页较多,映射时必须收集到所有可能通过段寄存器访问的内存区域,因此本文会将这种优化作为未来工作继续研究.

6.2 安全性分析

虽然虚拟机之间有着很强的隔离性,但是本文所提到的代理执行设计允许应用程序在用户态实现虚拟机地址空间的切换,所以可能造成潜在的安全隐患,本小节就使用本加速器虚拟化框架后的安全性进行了分析.本文将虚拟机监视器以及服务端虚拟机视为可信部分,假设恶意用户只可能通过客户端虚拟机发起攻击,攻击对象可以是服务端虚拟机,也可以是同一物理服务器上的其他客户端虚拟机.本文分析了以下攻击方式:

(1) VMFUNC 非法切换攻击.一个恶意用户可以在自己控制的客户虚拟机内自定义包含 VMFUNC 指令的应用程序,该程序可以不使用 **Wormhole** 提供的跳板机制,从而自行指定参数试图将控制流切换到其他非服务端的客户虚拟机,造成敏感数据的泄露等.**Wormhole** 通过限定每个客户端虚拟机的 EPTP 列表,第 0 项为当前扩展页表,第 1 项为目标服务端虚拟机的扩展页表,其余 510 项强制填充无效地址 0,使客户端虚拟机在执行地址空间切换时只有两种选项:切换到自己或绑定的服务端虚拟机.如果恶意应用程序传给 VMFUNC 指令参数大于 1 则会发生下陷被虚拟机监视器拦截,因此不会对其他虚拟机造成影响.

(2) 非法后端函数跳转攻击.一个恶意用户可能试图篡改映射到客户端虚拟地址空间的后端处理函数指针数组中的地址,例如将函数指针指向一些可能会泄露敏感数据的函数来劫持控制流.**Wormhole** 在映射后端处理函数指针数组时,在扩展页表中将数组所在内存页的读写权限设为了只读(read-only),如果发生试图修改后端处理函数指针数组的行为将会触发虚拟机下陷,虚拟机监视器会捕捉并阻止后续行为.

7 总结

本文面向时下流行的云端深度学习场景,针对目前缺乏可用性好、方便高效、易于维护的加速器虚拟化方案的现状,提出了 **Wormhole**,一套基于硬件虚拟化技术的加速器虚拟化框架,为各类云服务提供商开发可定制化、易于更新的加速器虚拟化系统提供了支持.**Wormhole** 加速器虚拟化框架以 API 转发为基础,以虚拟机为隔离保护域,创新性地提出了被动式服务端虚拟机的抽象以及跨虚拟机快速代理执行的各项技术,在保证了用户间强隔离性的前提下,实现了高硬件资源利用率、低虚拟化开销的加速器虚拟化,并在主流的 QEMU/KVM 平台上实现了针对 NVIDIA GPU 的原型系统.测试结果表明,**Wormhole** 可以方便地部署在消费级服务器上,对比扩展优化后的 GPU 虚拟化代表性方案 GVirtuS 有大幅性能提升,验证了本加速器虚拟化框架的有效性.

References:

[1] Zhang ZK, Pang WG, Xie WJ, Lv MS, Wang Y. A Survey of Deep Learning Research for Real-Time Applications. Ruan Jian Xue Bao/Journal of Software, 2020,31(9). https://www.jos.org.cn/1000-9825/5946.htm (in Chinese).

[2] Norman P. Jouppi, Cliff Young, Nishant Patil, etc. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17). Association for Computing Machinery, New York, NY, USA, 1–12. DOI:https://doi.org/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 (in Chinese).

[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(04):959-992 (in Chinese).

[5] "Intel platform hardware support for I/O virtualization". intel.com. 2006-08-10. Archived from the original on 2007-01-20. Retrieved 2014-06-07.

[6] Herrera, A. (2014). NVIDIA GRID: Graphics accelerated VDI with the visual performance of a workstation. Nvidia Corp, 1-18.

[7] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A full GPU virtualization solution with mediated pass-through. In Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14). USENIX Association, USA, 121–132.

[8] GRID Virtual GPU User Guide. (2017, June 1). Retrieved Febrary 10, 2020, from: https://docs.nvidia.com/grid/4.3/grid-vgpu-user-guide/index.html

[9] J. Duato, A. J. Peña, F. Silla, R. Mayo and E. S. Quintana-Ortí, "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters," 2010 International Conference on High Performance Computing & Simulation, Caen, 2010, pp. 224-231. DOI: 10.1109/HPCS.2010.5547126

[10] Raffaele Montella, Giulio Giunta, Giuliano Laccetti, Marco Lapegna, Carlo Palmieri, Carmine Ferraro, Valentina Pelliccia, Cheol-Ho Hong, Ivor Spence, and Dimitrios S. Nikolopoulos. 2017. On the Virtualization of CUDA Based GPU Remoting on ARM and X86 Machines in the GVirtuS Framework. Int. J. Parallel Program. 45, 5 (October 2017), 1142–1163. DOI:https://doi.org/10.1007/s10766-016-0462-1

[11] François Armand, Michel Gien, Gilles Maigné, and Gregory Mardinian. 2008. Shared device driver model for virtualized mobile handsets. In Proceedings of the First Workshop on Virtualization in Mobile Computing (MobiVirt '08). Association for Computing Machinery, New York, NY, USA, 12–16. DOI:https://doi.org/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 (in Chinese).

[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(09):1429-1438 (in Chinese).

[14] Wu S, Wang K, Jin H. Research Status and Prospect of Operating System Virtualization. Ji Suan Ji Ji Shu Yu Fa Zhan/Computer Technology and Development, 2019,56(01):58-68 (in Chinese).

[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 (in Chinese).

[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(02):481-494 (in Chinese).

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

[18] Keith Adams and Ole Agesen. 2006. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international conference on Architectural support for programming languages and operating systems (ASPLOS XII). Association for Computing Machinery, New York, NY, USA, 2–13. DOI:https://doi.org/10.1145/1168857.1168860

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

[20] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-enforced Intra-domain Isolation. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). Association for Computing Machinery, New York, NY, USA, 1607–1619. DOI:https://doi.org/10.1145/2810103.2813690

[21] Zeyu Mi, Dingji Li, Zihan Yang, Xinran Wang, and Haibo Chen. 2019. SkyBridge: Fast and Secure Inter-Process Communication for Microkernels. In Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19). Association for Computing Machinery, New York, NY, USA, Article 9, 1–15. DOI:https://doi.org/10.1145/3302424.3303946

[22] L. Shi, H. Chen, J. Sun and K. Li, "vCUDA: GPU-Accelerated High-Performance Computing in Virtual Machines," in IEEE Transactions on Computers, vol. 61, no. 6, pp. 804-816, June 2012. 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(01):110-116 (in Chinese).

[24] Rusty Russell. 2008. Virtio: towards a de-facto standard for virtual I/O devices. SIGOPS Oper. Syst. Rev. 42, 5 (July 2008), 95–103. DOI:https://doi.org/10.1145/1400097.1400108

[25] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA efficiently for key-value services. In Proceedings of the 2014 ACM conference on SIGCOMM (SIGCOMM '14). Association for Computing Machinery, New York, NY, USA, 295–306. DOI:https://doi.org/10.1145/2619239.2626299

[26] Xiantao Zhang, Xiao Zheng, Zhi Wang, Hang Yang, Yibin Shen, and Xin Long. 2020. High-density Multi-tenant Bare-metal Cloud. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20). Association for Computing Machinery, New York, NY, USA, 483–495. DOI: https://doi.org/10.1145/3373376.3378507

[27] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable parallel programming with CUDA. In ACM SIGGRAPH 2008 classes (SIGGRAPH '08). Association for Computing Machinery, New York, NY, USA, Article 16, 1–14. DOI:https://doi.org/10.1145/1401132.1401152

[28] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05). USENIX Association, USA, 41.

[29] Yangqing Jia, Evan Shelhamer, Jeff Donahue, etc. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. In Proceedings of the 22nd ACM international conference on Multimedia (MM '14). Association for Computing Machinery, New York, NY, USA, 675–678. DOI:https://doi.org/10.1145/2647868.2654889.

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

[31] Forsyth, D. A., & Ponce, J. (2002). Computer vision: a modern approach. Prentice Hall Professional Technical Reference.

[32] Gerhard Weikum. 2002. Foundations of statistical natural language processing. SIGMOD Rec. 31, 3 (September 2002), 37–38. DOI:https://doi.org/10.1145/601858.601867

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet classification with deep convolutional neural networks. Commun. ACM 60, 6 (May 2017), 84–90. DOI:https://doi.org/10.1145/3065386

[34] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998. DOI: 10.1109/5.726791

[35] L. Deng, "The MNIST Database of Handwritten Digit Images for Machine Learning Research [Best of the Web]," in IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141-142, Nov. 2012. DOI: 10.1109/MSP.2012.2211477

附中文参考文献:

[1] 张政馗,庞为光,谢文静,吕鸣松,王义.面向实时应用的深度学习研究综述.软件学报,2020,31(9). https://www.jos.org.cn/1000-9825/5946.htm

[3] 张晓丽,杨家海,孙晓晴,吴建平.分布式云的研究进展综述.软件学报,2018,29(7):2116-2132.

[4] 高强,张凤荔,王瑞锦,周帆.轨迹大数据:数据处理关键技术研究综述.软件学报,2017,28(04):959-992.

[12] 张玉清,王晓菲,刘雪峰,刘玲.云计算环境安全综述.软件学报,2016,27(06):1328-1348.

[13] 余倩倩,董明凯,陈海波.虚拟环境下硬件事务内存辅助的同步机制.计算机科学与探索,2017,11(09):1429-1438.

[14] 吴松,王坤,金海.操作系统虚拟化的研究现状与展望.计算机研究与发展,2019,56(01):58-68.

[15] 刘宇涛,陈海波.虚拟化安全:机遇,挑战与未来.网络与信息安全学报,2016,2(10):17-28.

[16] 黄啸,邓良,孙浩,曾庆凯.基于硬件虚拟化的安全高效内核监控模型.软件学报,2016,27(02):481-494.

[19] 刘维杰,王丽娜,谈诚,徐来.基于 VMFUNC 的虚拟机自省触发机制.计算机研究与发展,2017,54(10):2310-2320.

[23] 张宏莉,方滨兴,胡铭曾,姜誉,詹春艳,张树峰.Internet 测量与分析综述软件学报,2003(01):110-116.