GPU 虚拟化技术及其安全问题综述

吴再龙[1,2], 王利明[1], 徐 震[1], 李宏佳[1], 杨 婧[1]

[1] 中国科学院信息工程研究所 北京 中国 100093
[2] 中国科学院大学 网络空间安全学院 北京 中国 100049

摘要 人工智能与各行业全面融合的浪潮方兴未艾, 促使传统云平台拥抱以图形处理器(GPU)为代表的众核体系架构。为满足不同租户对于机器学习、深度学习等高密度计算的需求, 使得传统云平台大力发展 GPU 虚拟化技术。安全作为云平台 GPU 虚拟化应用的关键环节, 目前鲜有系统性的论述。因此, 本文围绕云平台 GPU 虚拟化安全基本问题——典型 GPU 虚拟化技术给云平台引入的潜在安全威胁和 GPU 虚拟化的安全需求及安全防护技术演进趋势——展开。首先, 深入分析了典型 GPU 虚拟化方法及其安全机制, 并介绍了针对现有 GPU 虚拟化方法的侧信道、隐秘信道与内存溢出等攻击方法; 其次, 深入剖析了云平台 GPU 虚拟化所带来的潜在安全威胁, 并总结了相应的安全需求; 最后, 提出了 GPU 上计算与内存资源协同隔离以确保多租户任务间的性能隔离、GPU 任务行为特征感知以发现恶意程序、GPU 任务安全调度、多层联合攻击阻断、GPU 伴生信息脱敏等五大安全技术研究方向。本文希望为云平台 GPU 虚拟化安全技术发展与应用提供有益的参考。

关键词 GPU 虚拟化安全; GPU 安全; GPU 虚拟化; 云计算安全; 安全需求
中图法分类号 TP309.1 **DOI 号** 10.19363/J.cnki.cn10-1380/tn.2022.03.03

GPU Virtualization Technology and Security Issues: A Survey

Wu Zailong[1,2], Wang Liming[1], Xu Zhen[1], Li hongjia[1], Yang Jing[1]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

Abstract The wave of the integration of artificial intelligence and various industries is emerging, prompting the traditional public cloud provider to embrace the Heterogeneous Computing System, especially Graphics Processing Unit (GPU), a many-core computing architecture who can provides several times floating point computing power of the Central Processing Unit(CPU). Meanwhile, to meet the demands of multi-tenant scenario for high-density computing, such as machine learning and deep learning, GPU virtualization technology can make multi-tenant sharing GPU possible, which has attracted great attention from academia and industries. However, there is still lack of a systematic exposition on the security of GPU virtualization that is the key to pratical applications. Thus motivated, we raise two fundemetnal questions of GPU virtualization security in public cloud: the potential security threats brought by typical GPU virtualization technology, as well as the security requirements of GPU virtualization and the evolution trends of security protection technology. To answer these two questions, we first illustrate the typical GPU architecture, the virtualization methods of GPU and their security mechanisms, and introduce the attack methods of side channel, covert channel and memory spill for existing GPU virtualization methods. Then, we digest the potential security threats to public cloud brought by GPU virtualization, and summarize the corresponding security requirements for GPU virtualization. Finally, we propose five research directions of the security of GPU virtualization, namely, collaborative isolation of computing and memory resources which can make sure the performance isolation between GPU tasks of mutliple tenants, GPU task behavior perception which can inspect the running malware on the GPU, secure scheduling of GPU tasks to ensure program and resource correspondence, multi-layer joint attack blocking, and GPU associated information desensitization. We hope this survey can provide some helpful references for the progress and application of the security technology of GPU virtualization in public cloud.

Key words GPU virtualization security; GPU virtualization; GPU security; cloud computing security; security requirements

1 引言

近年来, 随着人工智能与各行业全面融合的浪潮高涨, 以及图形处理器(Graphics Processing Unit, GPU)计算性能的稳步增长和编程工具的不断改进, 产业界越来越多的业务借助 GPU 的线程级并行框架

通讯作者: 王利明, 博士, 博士生导师, 正高级工程师, Email:wangliming@iie.ac.cn。
本课题得到国家重点研发计划(No. 2017YFB101000)资助。
收稿日期: 2019-09-19; 修改日期: 2019-11-21; 定稿日期: 2022-01-06

实现。这促使云平台尝试拥抱 GPU 等众核体系框架，以降低使用者自行购买、使用和维护 GPU 的成本。相较于高性能计算(HPC)集群中部署和使用 GPU 的方式，云平台中提供 GPU 服务需要充分考虑 GPU 资源虚拟化、租户间的安全隔离等问题。这使得“GPU 虚拟化技术给云平台引入的潜在安全威胁是什么”以及“GPU 虚拟化的安全需求及安全防护技术演进趋势是什么”成为云平台 GPU 虚拟化应用的两个基本问题。

虚拟化是云计算的核心技术之一，通过对软硬件资源的虚拟化，将基础资源离散化为可自由调度的“资源池”，实现资源按需配置。在传统的云平台中，虚拟化技术面向的是 x86 架构平台的计算资源(中央处理器, CPU)和存储资源。然而 GPU 虚拟化却困难重重，首先是 GPU 有其独有的计算资源和存储体系，使传统的虚拟化技术无法简单的在 GPU 上移植；其次, GPU 设备的硬件实现与虚拟化技术的标准化难以实现，当前独立 GPU 的设备提供商主要是英伟达(NVIDIA)和 AMD，他们的寡头竞争关系使合作难以实现；此外，现有的 GPU 驱动都是闭源的，限制了开源社区在 GPU 虚拟化上的工作。尽管业内有针对 Intel[1]和 ARM[2]集成 GPU 的虚拟化研究(如 gVirt[3])，但由于性能限制一直未广泛应用。我国太湖之光超算使用的神威处理器[4]也是类似于 GPU 的众核架构，但较少商用。GPU 虚拟化技术可分为面向图形用户界面的虚拟化技术和面向通用计算的虚拟化技术，本文的研究选择广泛使用的独立 GPU，并围绕面向通用计算的 GPU 虚拟化技术展开。

提供 GPU 服务已成为云平台发展的重要方向之一。亚马逊弹性计算云[5]于 2010 年率先提供了 GPU 服务，现在其基于英伟达的 V100、K80 等 GPU 提供多种系统镜像。国内最大的云平台服务商阿里云于 2018 年开始提供 GPU 服务[6]，现有 vGN5i, GN6 和 GA1 等系统镜像，分别使用英伟达 P4, V100, 和 AMD S7150 等 GPU。其中, vGN5i 基于轻量级 GPU 虚拟实例将公有云提供 GPU 服务的最小单位由单块物理 GPU 降至单个流式多处理器(Streaming Multi-processor, SM)，这提高了 GPU 的利用率并实现了租户间的安全隔离。与此同时，腾讯云、京东云、微软云等多家公有云平台都开始提供 GPU 服务。

然而，目前国内外对 GPU 虚拟化技术的安全问题研究较少。在云平台中引入 GPU 会带来安全威胁的演进：基于 GPU 上的特有资源，攻击可以在 GPU 的资源上发生，而传统的虚拟化隔离机制与云平台安全机制对此无能为力；同时, GPU 的加入使集群与节点的结构发生了变化，传统攻击可以借助这些变化演进其攻击行为。因此需要明确 GPU 虚拟化的安全需求及与之对应的安全技术研究方向。

在简要回顾 GPU 架构及 x86 平台的虚拟化方法(第 2 节)的基础上，为回答云平台中 GPU 虚拟化应用的两个基本问题，如图 1 所示，本文首先深入分析了多类型的 GPU 虚拟化方法以及其中的安全机制(第 3 节)；随后介绍了云平台中已有的多种 GPU 攻击方法(第 4 节)；在总结已有安全威胁的基础上，深入剖析了云平台中引入 GPU 后带来安全威胁的演进(第 5 节)；基于此，明确了 GPU 虚拟化的安全需求，并提出了 GPU 虚拟化的五大安全技术研究方向，最终给出 GPU 虚拟化的安全框架作为参考(第 6 节)。

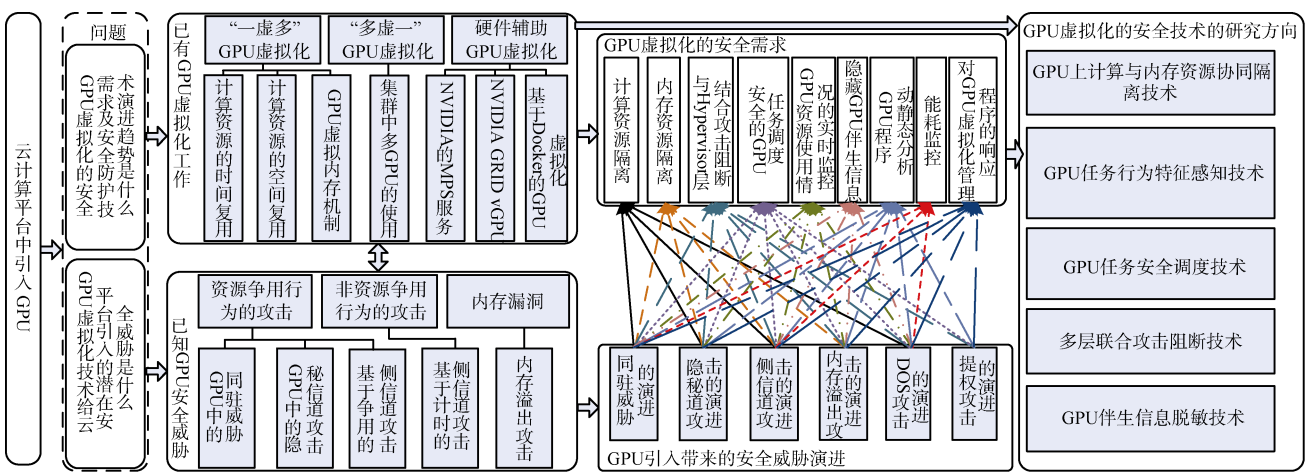


图 1　云平台中 GPU 虚拟化应用带来的安全威胁演进，及 GPU 虚拟技术的安全需求与安全技术

Figure 1　Security threat evolution brought by GPU virtualization, and corresponding security demand and technologys

2 GPU 体系结构及 x86 平台虚拟化

本节首先以英伟达①为主展示其历代 GPU 架构的硬件资源配置, 并概述 GPU 计算资源的组织及存储体系的结构。其次, 以 CUDA[7]为主讲述 GPU 通用计算编程模型。最后, 简要回顾 x86 平台上已有的虚拟化方法。

2.1 GPU 体系结构

CPU 为处理逻辑判断多、通用性且计算密度较低的计算任务, 被设计拥有用于指令预测、分支处理等工作的复杂的控制单元。GPU 早期被用于图形图像渲染, 故被设计为针对向量化计算的单指令多线程(SIMT)结构, 在后续发展中 GPU 上的计算能力与可编程性不断优化, 被应用于计算密集型的通用计算任务。图 2 展示了 CPU 与 GPU 架构对比。

表 1 中展示了支持 CUDA 通用计算的历代英伟达 GPU 架构的配置②。以 Pascal 架构 P100 GPU[9]为例, 如下图 3 所示其共有 56 个 SM, 提供 3584 个 CUDA 核(CUDA core)。GPU 中计算最小的单元为 CUDA 核, CUDA 核阵列、寄存器、shared memory、物理线程块(warp)③调度器等资源构成 SM。SM 上的资源为片上资源, 片下资源主要指全局内存及其上功能单元, 被所有 SM 共享。英伟达最新发布的 Volta 架构[10]的 GPU 提供了更高的深度网络训练能力, 在可编程性上, 其优化了虚拟内存及多进程服务(Multi-Process Service, MPS) [10]的实现。

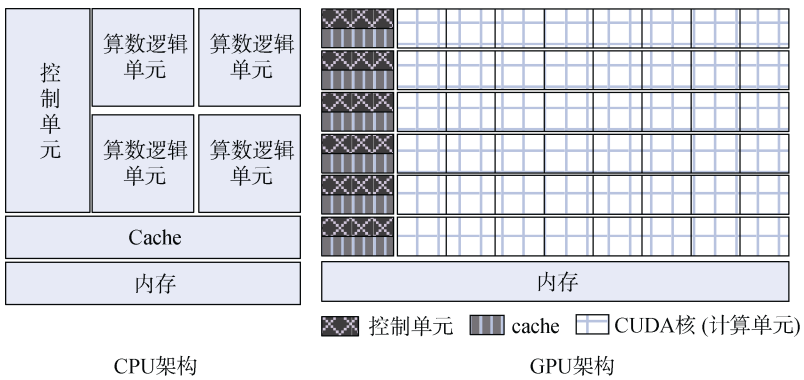


图 2 CPU 与 GPU 架构对比[8]

Figure 2 Architectural comparison: CPU vs. GPU [8]

表 1 历代 GPU 架构的配置

Table 1 Configuration of generations of GPU architectures

GPU 架构	Fermi[11]	Kepler[12]	Maxwell[13]	Pascal[9]	Volta[10]
SM 数量	16	15	24	56	80
CUDA 核数	512	2880	3072	3584	5120
单精计算能力	1331 GFlops	3950 GFlops	6840 GFlops	10600 GFlops	15.7 TFlops
双精计算能力	665.6 GFlops	1310 GFlops	210 GFlops	5300 GFlops	7.8 TFlops
SFUs/SM	4	32	32	16	16
Warp 调度器(单 SM)	2	4	4	2	4
Shared memory 大小(单 SM)	共 64 KB, 可 16/48 KB 或 48/16 KB 与 L1 cache 分割		96 KB, 单线程块 48 KB	64 KB, 单线程块 32 KB	最大 96 KB
L2 cache 大小	768 KB	1536 KB	3072 KB	4096 KB	6144 KB
并发 kernel 数	16	32	32	32	48
全局内存	6 GB	12 GB	12 GB	16 GB	32 GB
虚拟化支持演进		支持 Hyper-Q	引入统一地址	统一地址扩为 49 位	支持空间复用的 GPU 分割

① 以 AMD GPU 为平台研究虚拟化的工作较少, 故以 NVIDIA 为主展示。

② GPU 型号分别 C2090, K40, M40, P100, V100。

③ shared memory 可译为“共享内存”, 因在有些语境下会产生歧义, 故文中用 shared memory 表述。warp 是 CUDA 中特定名词, 故用英文表述。

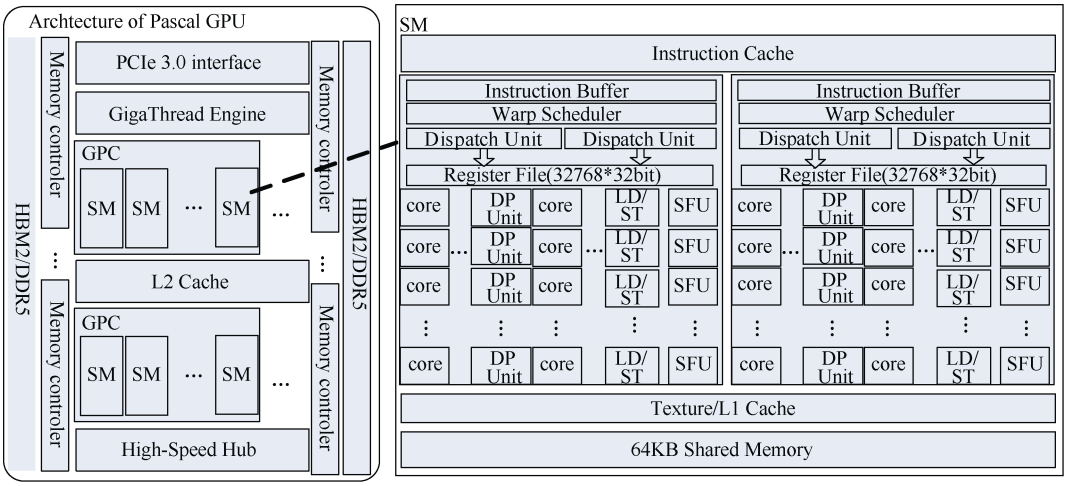


图 3　Pascal 架构

Figure 3　Architecture of Pascal GPU

AMD 最新的 GPU 架构为 Vega, 其同样提供强大的并行计算能力和良好的深度学习类业务的支撑。AMD 与英伟达的 GPU 在架构上是相似的, 以图 4 的 AMD HD7970 为例, 可看到 AMD 的 GPU 由多个计算单元(Compute Unit, CU)组成, 单个 CU 有局部内存(Local memory)、流处理器(PE)等组成, 这与英伟达 GPU 中组成基本相同。

2.2　GPU 编程模型及执行模型

GPU 上的主流通用计算编程模型有 CUDA 和 OpenCL[15], 二者十分相似, 表 2 展示了两者常用且等价的术语。CUDA 程序包含主机端程序和 GPU 内核程序(kernel①)两部分, 主机端程序为 CUDA 程序在主机端和设备端开辟内存空间, 并管理数据传输; kernel 负责 GPU 端并行处理数据, 在 CUDA 编程模型中通过设置线程空间(Grid)来组织 kernel 的并行执行, 每个 Grid 包含多个线程块(ThreadBlock), 线程块以 warp 为单位执行, 一个 warp 包含多个物理线程。相对 CPU 线程, GPU 线程更轻量级。每个线程有其私有的寄存器, 线程标识(ThreadID)等。同一线程块中的线程间共享 shared memory, 并可通过局部同步术语进行同步, 一个 SM 上可以开启一个或多个线程块。多个 SM 上的线程间通信可由全局内存实现, 使用其上的原子操作可完成 SM 间的线程同步。图 5 展示了 CUDA 程序中线程的组织方式。

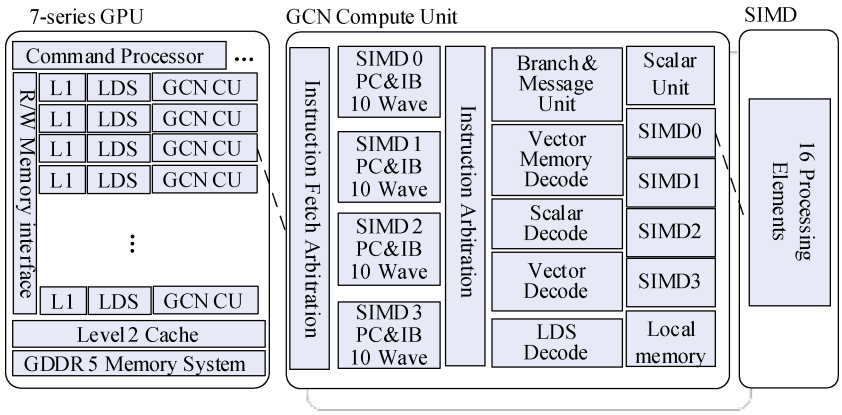


图 4　AMD GCN 架构[14]

Figure 4　Architecture of AMD GCN GPU[14]

① “kernel”为 GPU 程序的特定名词, 故全文用英文表述。

表 2　CUDA 与 OpenCL 等价术语对比
Table 2　Terms of CUDA vs. OpenCL

CUDA 术语	OpenCL 术语
grid	NDRange
threadBlock	workgroup
thread	workitem
warp	wavefront
global/device memory	global memory
shared memory	local memory
register	private memory
constant memory	constant memory
SM	CU
CUDA core	PE

OpenCL 程序也是由内存操作和 kernel 执行两部分, 线程组织方式和 CUDA 基本一致。OpenCL 可兼容 AMD 与英伟达的 GPU, 亦可支持 FPGA, DSP 等架构的编程。理论上可实现 GPU 上跨平台的功能迁移与性能迁移。

2.3　GPU 存储体系

GPU 上共有三个可编程存储空间, 寄存器、shared memory 及全局内存。寄存器和 shared memory 位于片上, 其中寄存器为线程私有, shared memory 被一个 SM 上的线程块所共有, 片上存储更靠近计算单元, 延迟低。全局内存位于片下, 访问速度低于片上存储器, 其通过 L1/L2 cache 提供对 CUDA 核的高速访问, 这类似于 CPU 上的内存与 cache 结构。此外, 全局内存中可以申请常量存储空间, 在 L1/L2 cache 中开辟专用的缓存空间。

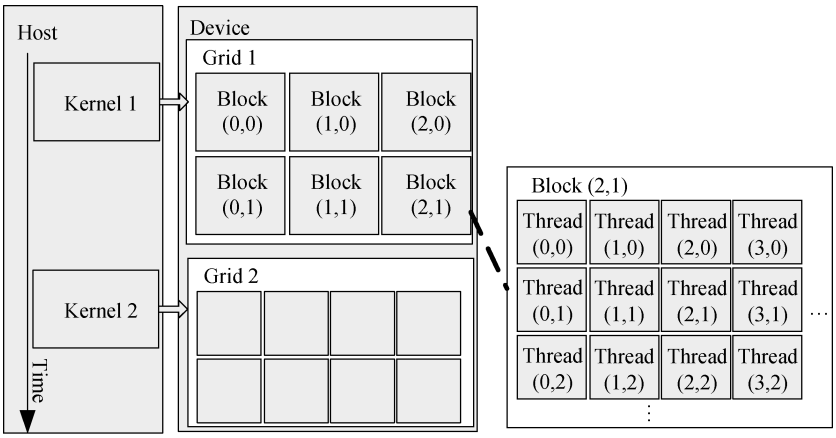


图 5　CUDA 程序中线程的组织方式[16]
Figure 5　Threads organization in CUDA applications[16]

对于 GPU 上的通用计算程序(如 CUDA 或 OpenCL 程序), CPU 端负责申请主机端内存及 GPU 端内存, 同时发送 kernel 到 GPU 端, 运行时系统(CUDA 运行时或 OpenCL 运行时)负责数据的拷贝(通过 PCI-e 总线或是 NV-Link[17])。

2.4　x86 平台虚拟化相关技术

系统虚拟化技术是指将一台物理机中的资源(如 CPU, 内存, I/O 设备等)进行虚拟化。多个虚拟机可以共享宿主机的硬件资源。宿主机通过搭载虚拟机监视器(如 Hypervisor)提供虚拟机运行环境, 具体而言, 通过 Hypervisor 控制硬件资源, 提供面向虚拟机的宿主机资源分割, 及虚拟机之间的资源隔离。图 6 展示了两种 Hypervisor 类型。

现有的 x86 平台虚拟化技术主要有面向计算资源 CPU 的虚拟化技术、内存的虚拟化技术和 I/O 设备的虚拟化技术。

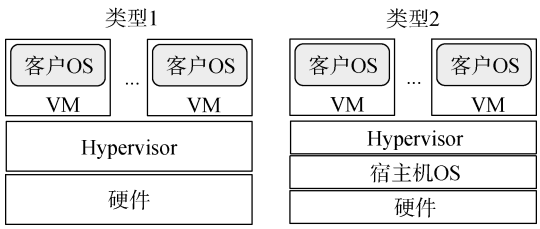


图 6　两种 Hypervisor 类型[18]
Figure 6　Two types of ypervisor [18]

CPU 虚拟化技术为每个虚拟实例提供一个或多个虚拟 CPU(vCPU), 多个 vCPU 分时使用 CPU。虚拟机监视器必须为多个 vCPU 合理分配时间片并维护所有 vCPU 的状态, 当一个 vCPU 的时间片用完后, 需保存当前 vCPU 的状态并调度下一个 vCPU 载入物理 CPU。x86 是主流的 CPU 指令集, 因其很多指令不支持虚拟化, 故在虚拟化实现中以指令翻译方式的不同划分为: (1)基于二进制翻译的全虚拟化, 其动

态扫描全部客户机操作系统的指令, 把不支持虚拟化的指令替换为可虚拟化的指令, 该方法实现复杂且效率低, VMware[19]是典型代表; (2)半虚拟化静态修改客户机操作系统源码, 在操作系统执行前替换掉不支持虚拟化的指令, 大大提高了虚拟化的效率, 以 Xen[20]为代表; (3)硬件辅助虚拟化修改 x86 硬件架构实现虚拟化支持, 例如 Intel VT-X[21], AMD SVM[22]技术, 其从芯片设计上提升了虚拟化的支持。

内存虚拟化完成宿主机的物理内存到虚拟机中的虚拟内存的转换, 由 Hypervisor 中内存管理单元(MMU)虚拟化支持。实现技术包括: (1)Xen 的直接页表技术, 其修改虚拟机的操作系统, 使 Hypervisor 与虚拟机共享页表, 通过段保护机制对地址空间进行分段确保了虚拟机与 Hypervisor 的隔离, 该方法实现较高的快表(TLB)效率, 但是需要 Hypervisor 的安全审计防止虚拟机对页表的恶意窜改; (2)虚拟 TLB 技术, 当虚拟机访问虚拟地址时会触发页错误, Hypervisor 对页错误进行分析, 将虚拟机访问的虚拟地址翻译成物理地址, 其效率低, 且只能对已访问的虚拟地址进行缓存; (3)影子页表技术, 虚拟机拥有自己的页表, Hypervisor 同时维护对应的转换表, 即影子页表, 虚拟机的每次访存都会通过影子页表完成, 效率低; (4)扩展页表技术, 通过硬件加强的方式将虚拟机的地址转换为物理内存, Hypervisor 不参与操作直接由硬件完成, 效率最高。图 7 展示了虚拟机内存与宿主机内存的关系。

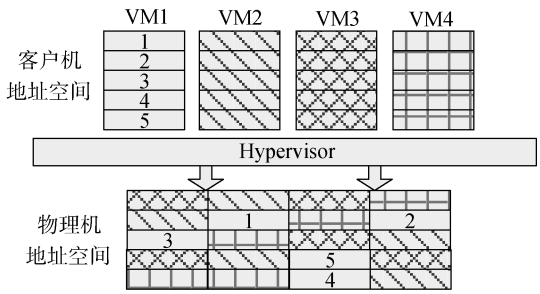


图 7 虚拟机内存与物理机内存的关系

Figure 7 Relationship between virtual memory and physical machine memory

I/O 虚拟化需要完成虚拟中断, 虚拟寄存器访问、虚拟直接内存存取 (Direct Memory Access, DMA)。实现方式包括: (1)软件模拟实现, 即软件模拟硬件设备, 所有的访问都需要对指令进行捕获和翻译, 效率较低; (2)半虚拟化实现, 其重新定义 I/O 架构, 使用分离驱动的方式分别实现前端驱动与后端驱动, 二者通过共享内存的方式来进行数据交换, 效率高, 被 Xen、KVM[23]采用; (3)设备直通方式, 即客户机操作系统使用未经修改的驱动直接访问物理设备, 这需要所有的 I/O 操作直接发往物理机, 同时中断必须被 Hypervisor 捕获, 且 DMA 需要重映射。

3 GPU 虚拟化技术分析

参考 x86 平台上成熟的虚拟化技术, GPU 虚拟化需要完成其上的计算资源和存储资源的虚拟化; 需要实现 GPU 的资源池化, 以此向多租户提供按使用付费的 GPU 服务。为此本文首先对比了 x86 平台已有虚拟化与 GPU 虚拟化的需求, 如表 3 所示。

表 3 x86 平台虚拟化与 GPU 虚拟化需求对比

Table 3 Requirements of virtualization of x86 vs. GPU

平台 虚拟化	x86 平台	GPU
计算资源虚拟化	1. 实现 x86 指令集对虚拟化的支持; 2. 实现 vCPU 并以时间复用使用 CPU。	1. 实现对 GPU 计算资源的有效切割; 2. 实现租户间的计算资源隔离与性能隔离。
内存资源虚拟化	1. 实现内存面向多租户的有效分割; 2. 实现多程序的内存空间的隔离与保护。	1. 实现显存面向多租户的有效分割; 2. 实现多程序的内存空间的隔离与保护; 3. 实现 GPU 虚拟内存机制对计算资源虚拟化的支持。
虚拟化管理平台	实现 Hypervisor。	实现可与 Hypervisor 兼容的 GPU 管理模块。

GPU 虚拟化在通用计算上的实现可分为“一虚多”和“多虚一”[24]两种技术。“一虚多”是指一块 GPU 上的资源被分享给多个虚拟实例; “多虚一”是指分布式集群中的多个计算节点上的多个 GPU 卡被统一管理, 虚拟实例可以使用集群中多个位置不同的 GPU。本节的研究首先介绍了当前“一虚多”的 GPU 虚拟化技术, 着重讲述了 GPU 上计算资源的虚拟化方法, 内存资源的虚拟化方法; 之后介绍了当前“多虚一”的 GPU 虚拟化方法; 最后介绍了 GPU 厂商提供 GPU 虚拟化产品的进展。

3.1 “一虚多”的 GPU 虚拟化技术

“一虚多”的 GPU 虚拟化技术需要解决对单 GPU 进行资源分割和其上租户间隔离的问题, 以有效提高 GPU 的资源利用率并确保租户间的安全隔离。具体而言, 需要实现对 GPU 计算资源虚拟化和内存资源的虚拟化。计算资源的虚拟化方法可分为基于时间复用的虚拟化技术和基于空间复用的虚拟化技术。内存资源的虚拟化技术可以通过多地址空

间的 TLB 技术、影子页表技术等方式实现。

3.1.1 GPU 时间复用与空间复用概述

GPU 以锁步(lockstep)的方式执行[25], 即 warp 中的每个物理线程同时执行相同的指令。当一个 warp 中的物理线程因访存等待执行时, GPU 调度其他 warp 执行来掩盖访存的延迟。GPU 以此方式提供高吞吐量的计算能力。随着 GPU 计算密度的逐渐增加, 单块 GPU 上可以提供 80 个 SM, 研究多程序间有效的计算资源复用方式是 GPU 虚拟化所必需的。

时间复用技术是指运行在同一 GPU 上的多个租户的 GPU 程序对 GPU 上的计算资源分时复用, 根据 GPU 上资源剩余情况开启多 kernel。这种方式简化了内存保护和资源调度。但带来资源浪费的问题, 如单个程序没有完全利用全部 GPU 资源, 而其他 kernel 会因剩余资源不足无法启动。除此之外, 时间分割的方式无法提供稳定的服务质量控制(Quality of Service, QoS), 会导致程序间资源分配的不公平。

空间复用技术以 SM 为单位向租户分配计算资源, 可以使多个租户同时拥有自己独占的 SM 资源, 实现多个 kernel 在 GPU 上同时执行。实现空间复用的技术包括: 基于软件实现的多 kernel 调度, 确保单 kernel 对其所分配的 SM 上物理资源的独占; 基于 kernel 合并, 实现 kernel 以 SM 为单位的资源调度。

3.1.2 GPU 时间复用技术

CUDA Hyper-Q[26]技术使多个 GPU 应用程序以时间复用的方式使用一个 GPU(Kepler 及之后架构的 GPU)。这提高了 GPU 利用率同时减少了 CPU 空闲时间, 并提高了可编程性。**GPU 程序按照其行为可以分为访存操作和 kernel 执行两部分, GPU 程序的任务级并发是充分利用两者的同步执行, 即用 kernel 执行来掩盖访存操作。**一个计算任务流(stream)会以队列形式组织 CPU 端输入的指令, 包括数据操作和 kernel 执行。每个计算任务流中的指令顺序执行, 而计算任务流之间的并发执行取决于资源在运行时的可用性。Hyper-Q 提供了对多个计算任务流的支持, 实现了 GPU 的时间复用。Hyper-Q 中对计算任务流的调度如图 8 所示。虽然较早的 Fermi 架构不支持 Hyper-Q, 但其最多可支持 16 个 kernel 的并发执行, 与 Hyper-Q 不同的是其只支持一个硬件任务流队列。

GPU 上多 kernel 并发执行受 GPU 资源的限制, 即每个 SM 上有限的寄存器、shared memory、线程数量和线程块数量制约了实际并发执行 kernel 的数量。若某个 kernel 的线程块使用了大量的寄存器和共享内存资源, 则造成单个 SM 上剩余资源不足够其

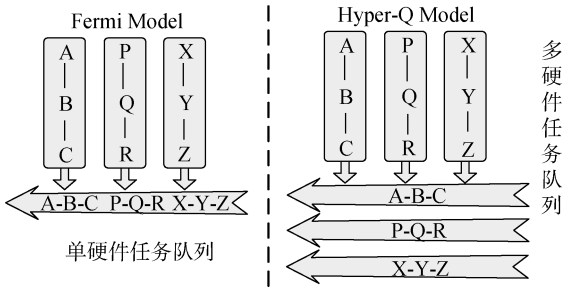


图 8 CUDA Hyper-Q 示意图

Figure 8 Schematic diagram of CUDA Hyper-Q

他 kernel 的线程块执行, 这时 kernel 的并发执行则无法实现, 直到运行中的 kernel 完成后释放资源后续 kernel 才能继续执行。可见, 按照剩余策略(leftover)并发执行 kernel 会导致了 GPU 资源的浪费。Pascal 架构之后的 GPU 支持基于资源抢占的时间复用, 但目前的这种虚拟化支撑技术并没有被广泛应用, 且额外的抢占行为加剧了 GPU 中资源的争用。

基于时间片分割的弹性 kernel 调度[27]方法是较早研究 kernel 并发机制的工作。文中以 Fermi 架构 GPU 为平台, 使用 Parboil2[28]库作为基准(benchmark)验证了 Fermi 平台上 kernel 的并发机制会带来超过 30%的系统资源浪费, 为此, 该工作中提出了弹性 kernel 机制、面向弹性 kernel 的资源分配机制、基于性能数据的弹性 kernel 并发策略和基于时间片的线程空间切换机制, 获得相对于 Fermi 平台原生并发机制 1.21 倍的系统吞吐量(System Throughput, STP)提升, 并将两程序的平均转换周期(Average Normalized Turnaround Time, ANTT)[29]提升了 3.37 倍。

弹性 kernel 机制。弹性 kernel 机制实现了 GPU 程序中线程空间的定义与硬件资源的分配之间的解耦合。弹性 kernel 由原始 GPU 程序的 kernel 转换得到, 转换后的弹性 kernel 需要使用与原始 GPU 程序不同大小的线程空间和线程块, 以保证运算结果一致。在 CUDA 编程模型中, 逻辑线程块和线程与硬件线程块和线程的映射机制是 1∶1 的映射, 弹性 kernel 需要使用 N∶1 的映射机制(线程重映射机制)实现面向线程空间的细粒度的资源分配。为此需采取如图 9 中代码段完成硬件资源标识(gridDim, blockDim 等)到逻辑资源标识的转换(gd, bd 等), 以获得弹性 kernel。如果原始 kernel 中用到了 shared memory 或者是同步指令, 为了保持程序语义, 不对原始 kernel 进行转换。对于 kernel 转换带来的寄存器使用量及计算次数的增加, 该工作认为可通过增加硬件支持解决此问题。

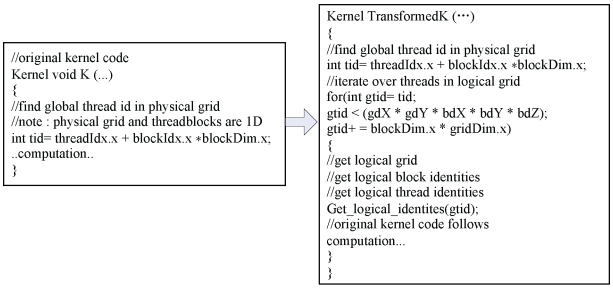


图 9　kernel 转换代码段[27]

Figure 9　Code of kernel transformation[27]

在获得弹性 kernel 后, 由于片上资源的限制, 需使用**面向弹性 kernel 的资源分配机制**设定弹性 kernel 的线程空间和线程块大小。之后, 使用**基于性能分析(profile)的弹性 kernel 的并发策略**实现多 kernel 的并发执行, 该工作基于 Parboil2 获得多种典型业务的性能分析数据, 在硬件资源的限制下制定多 kernel 并发执行的策略。最后, 使用弹性 kernel 实现了**基于时间片的线程空间切换机制**。

弹性 kernel 的方案提升了 GPU 资源的利用率并减少了 GPU 任务执行的等待时间, 相较于原生的 kernel 并发方案在性能上有了一定的提升, 为之后的研究提供了参考。但是, 该方法需要手动修改原始 GPU 程序的 kernel 代码, 没有考虑租户的安全性隔离问题, 因此无法在云平台中普及。Hyper-Q 方案虽支持多计算任务流的并发执行, 但其使用剩余策略容易导致 GPU 资源利用率低下, 且多个租户的 GPU 线程块会同时分享相同的 GPU 物理资源, 因此没有考虑在资源分享时实现安全隔离机制。综上, 可以看到 GPU 的时间复用方法存在诸多安全漏洞。

3.1.3　GPU 的空间复用技术

GPU 的空间复用技术以 SM 为单位向 kernel 分配计算资源, 可以使多个 kernel 同时拥有自己独占的 SM 资源, 实现 GPU 程序间在计算资源上的隔离, 有效应对因资源争用产生的安全漏洞问题。实现方法有 AccelOS[30], Pagoda[31]等。

Margiolas 等人[30]提出 AccelOS, 其基于 OpenCL 的封装与虚拟 NDrange 实现了 GPU 上资源的空间分割与 kernel 调度的机制, 并对 GPU 程序透明(无需对其上运行的 GPU 程序进行手动修改)。因其基于 OpenCL 实现, 具有跨平台性。该工作对标准 OpenCL 库进行了封装, 进而监控了 GPU 上执行的任务。同时, 因其实现了 GPU 片上计算资源以 SM 为单位在 kernel 间的分割, 故安全性较高。

AccelOS 的架构如图 10 所示, 其包括: **系统接口层(level 0)**, 用于与硬件平台连接, AccelOS 使用标准 OpenCL 调用 GPU 资源, 保证了 AccelOS 的兼容性; **AccelOS 核心层(level 1)**, 其以后台程序方式运行, 包含一个运行时系统和一个实时编译器, 运行时系统管理 GPU 物理资源并对 kernel 运行进行调度, 实时编译器对 kernel 代码进行转换并与调度库链接, 以支撑虚拟线程空间(NDRange)机制; **应用程序接口层(level 2)**提供 OpenCL 程序接口并监视程序运行过程, 通过对标准 OpenCL 库进行封装(得到 ProxyCL)实现对 OpenCL 程序执行指令与参数的记录, 获得程序的行为轨迹与 kernel 执行参数。

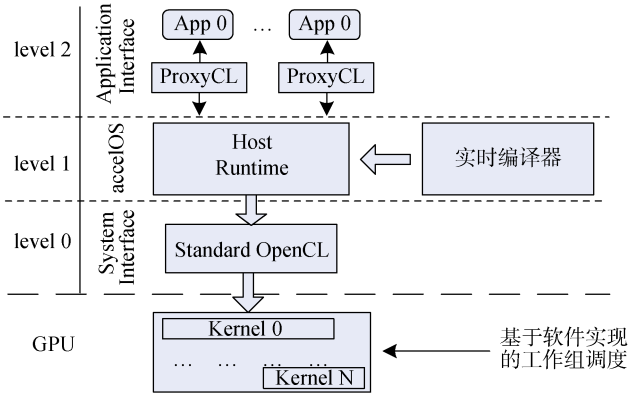


图 10　AccelOS 框架图[30]

Figure 10　Architecture of AccelOS[30]

AccelOS 中 kernel 调度如图 11(b)所示。AccelOS 中转换每个 kernel 的线程空间, 确保多个 kernel 有其独占的 CU(SM), 如图中的 4 个 kernel 被分配了相同大小的物理资源。标准 OpenCL 运行时系统为先到的 kernel 按其请求分配 GPU 物理资源, 后续 kernel 能否并发执行由 GPU 上的资源剩余情况决定, 这与 CUDA 的剩余策略一致, 如图 11(a)所示。

AccelOS 中使用了**虚拟 NDRange** 与**虚拟工作组**(Virtual group, VG)机制实现了基于空间复用的 GPU 分割方法与 kernel 的调度机制。图 12(a)展示在标准 OpenCL 环境下两个 GPU 任务并行执行时, GPU 中的工作组(workgroup, WG)和资源调度的情况。图中的 GPU 含有 4 个 CU, A 的 NDRange 包含 12 个工作组, B 的 NDRange 包含 8 个工作组。当 kernel A 先到达 GPU 时, 会占用全部的 CU, kernel B 会在 A 执行完成后继续执行。在 AccelOS 中虚拟 NDRange 机制如图 12(b)所示, kernel A 与 kernel B 被分配 2 个 CU, 在运行时系统对 kernel 转换后, GPU 程序的 NDRange 与工作组会被转换为虚拟 NDRange 与虚拟工作组, GPU 端实际分别分配给 kernel A 与 kernel B 各两个工作组, 每个工作组中实际执行的虚拟工作组数量与原始程序中工作组数量一致。由于 GPU 上的 CU

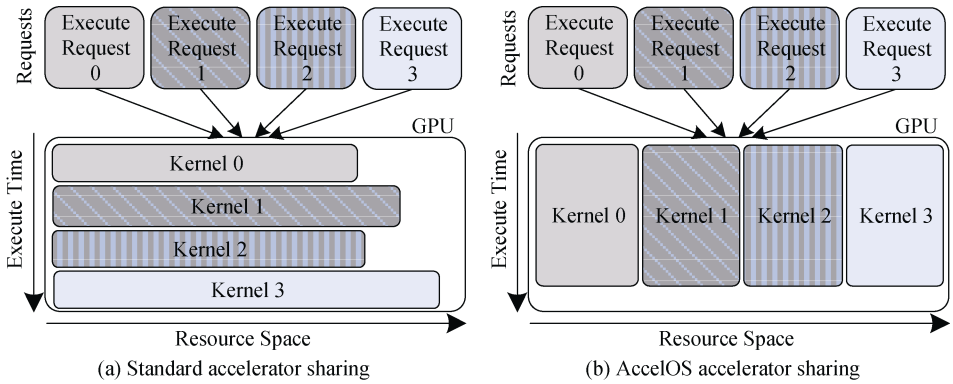


图 11　标准 OpenCL 与 AccelOS 的 kernel 调度对比[30]

Figure 11　Standard OpenCL vs. AccelOS in scheduler[30]

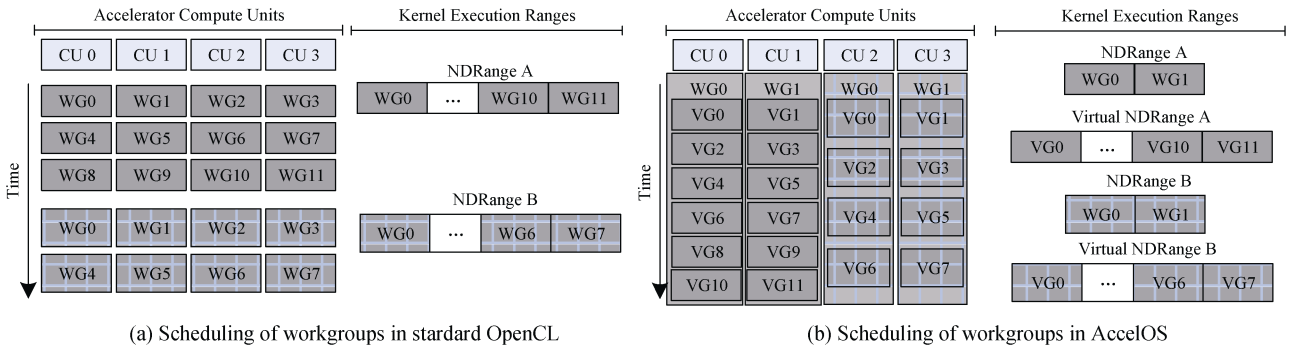


图 12　虚拟 NDRange 及虚拟 workgroup 的实现机制[30]

Figure 12　Implementation of virtual NDRange and virtual workgroup[30]

分配机制是将空闲的 CU 分配给先来的工作组, 当工作组数量设置为 CU 的数量时, 可实现工作组对 CU 的一对一占用, 之后用虚拟工作组填充工作组, 最终实现 GPU 计算资源的空间复用。虚拟 NDRange 会以队列的结构存储在 GPU 全局内存中。

AccelOS 使用实时编译器对原始 kernel 进行变换, 其过程类似于弹性 kernel 的转换。实时编译器首先基于 ProxyCL 对 GPU 程序执行进行静态分析, 判断如何对其 kernel 进行转换, 原始 kernel 转换后需要的工作组数量由运行时系统根据 GPU 剩余资源情况决定。最终转换后的 kernel 可支持虚拟 NDRange 机制带来的实际线程空间的尺寸变化。AccelOS 的运行时系统通过 ProxyCL 获取 OpenCL 程序的动态请求, 基于此对程序进行行为监控。运行时系统由 kernel 调度器、kernel 存储管理器组成, 前者集中管理 GPU 程序的资源请求, 基于此对 kernel 进行调度; 后者负责跟踪应用程序在 GPU 全局内存使用情况, 确保每个程序都被分配足够的内存空间。

综上, AccelOS 通过对 OpenCL 的封装实现对 GPU 程序的静态分析和动态行为监控, 并基于虚拟 NDRange 实现了 GPU 计算资源的空间复用, 较好的实现了多程序间的计算资源隔离, 安全性得到加强。

相较于 AccelOS, Tsung 等人[31]提出基于空间复用面向窄任务的 GPU 空间复用虚拟化方案 Pagoda。窄任务是指 GPU 线程启动数量小于 500, 且需要计算结果低延迟返回的任务。现有的 GPU 任务调度策略会导致执行窄任务时 GPU 资源利用率低下。为此, Pagoda 使用类操作系统(OS-like)的 kernel(名为“MasterKernel”)实现了一个运行时系统对 GPU 资源进行虚拟化。任务由 CPU 端生成后发送至 Pagoda 系统, 并通过 MasterKernel 以 warp 的粒度进行调度; 根据窄任务的特点, Pagoda 通过任务持续生成与执行机制降低任务生成和调度的延迟; 并实现了与之配套的 shared memory 管理和 warp 级同步机制。

基于 MasterKernel 的资源虚拟化。Pagoda 使用 MasterKernel 在 GPU 上持续运行并占用全部的片上资源, 这种方式与 AccelOS 中将工作组数量设为 CU 数量相似, 之后再将把资源分配给 Pagoda 上运行的任务。图 13 展示了 Titan X(Maxwell 架构[12] GPU)上的 Pagoda 框架。MasterKernel 通过在每个 SM 上开启 2 个线程块, 每个线程块被设置为 32 个 warp 的大小, 以此占据 SM 上可容纳的全部 warp。每个线程块叫做管理线程块(Master Thread Block, MTB), 其拥有 SM 中全部计算与存储资源, 并负责面向任务的资

源分配以及任务的调度管理。线程块中的每个线程占用 32 个寄存器。每个 MTB 中第一个 warp 叫做调度 warp, 其他 31 个 warp 叫做执行 warp。调度 warp 负责调度管理线程块中的执行 warp, 并管理 shared memory 和 warp 级局部同步。MasterKernel 中使用了两个数据结构, 任务表(TaskTable)和 warp 表(WarpTable)。前者负责任务的生成, 任务表中的每一项保存了一个任务信息, GPU 端的任务表根据 CPU 端的任务表实时更新。每个管理线程块有自己的 warp 表, warp 表中有 31 个位置保存执行 warp 的状态。Pagoda 基于任务表和 warp 表实现了**任务持续生成与执行机制,** 提升了任务启动与切换的效率。

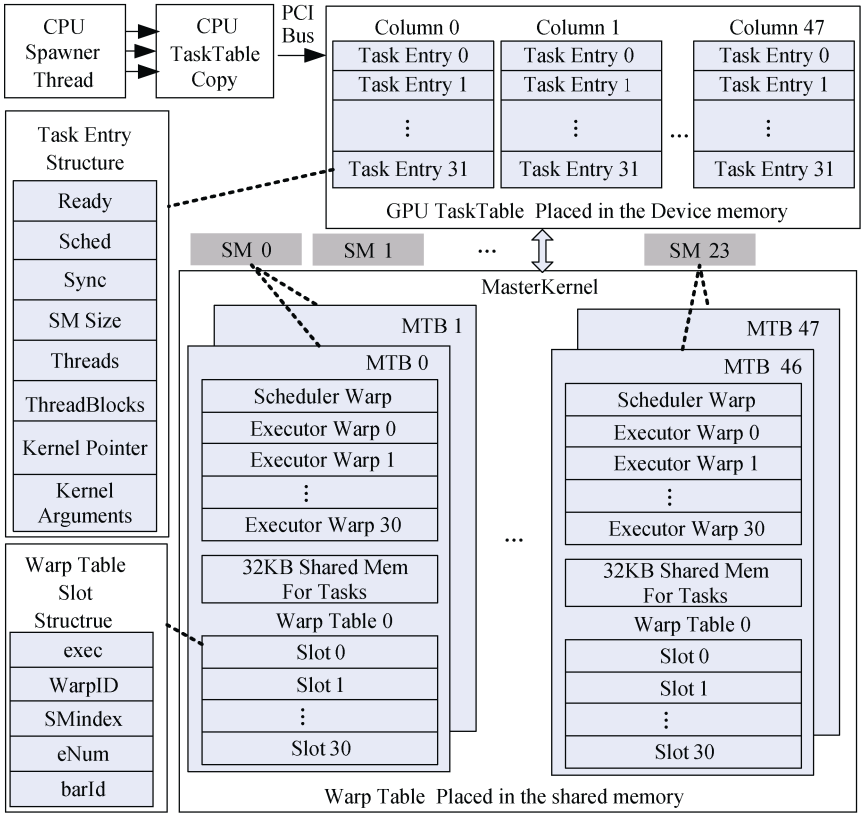


图 13 Pagoda 系统框架[31]

Figure 13 System architecture of Pagoda[31]

Pagoda 对空闲物理资源(warps, shared memory 等)进行管理, 将任务调度至其上运行。由于 Pagoda 中的任务都是在 MasterKernel 上执行, 故任务不能直接使用 shared memory, 为此全部 Shared memory 资源会以 2KB 为单位通过二叉树的结构进行管理。管理线程块开始执行时其持有全部 shared memory, 调度 warp 将 shared memory 的空间分配给任务, 并在其执行完成后回收。warp 级同步机制通过 CUDA 的汇编编程模型 PTX 实现, 同步操作会根据任务申请同步标识和线程标识来确定需要同步的 warp。

Pagoda 对于窄任务在 GPU 上的资源分配与调度提出了完整的方案, 提升了 GPU 物理资源的利用率, 但没有考虑安全隔离的问题且应用范围有限; 虽然提出了多个窄任务并发执行的方案, 但没有实现动态资源管理机制, 无法实时的启动和调度任务。

如上所述, 基于 GPU 上提供的众多 SM, 可以面向不同特点的任务实现计算资源的空间复用。空间复用技术可以实现计算资源面向多租户任务的隔离, 这可以有效的应对基于资源争用产生的安全漏洞。但仅实现计算资源的隔离并不足够安全, GPU 相对 CPU 有更多层级和复杂的存储体系, 多租户的计算任务在共用全局内存时, 必须拥有有效的 GPU 内存虚拟化机制, 实现 GPU 内存上的安全隔离。

3.1.4 GPU 虚拟内存技术

空间复用技术与支持多租户程序间隔离的 GPU 虚拟内存技术的结合, 才可以有效提升 GPU 资源在虚拟化中的安全性, 并提升 GPU 资源的利用率。GPU 虚拟内存需负责管理和保护每个 GPU 应用程序的地址空间。然而, 当前多数 GPU 虚拟内存的实现并不能支持多程序安全高效的并发执行。

CUDA 4.0[32]中首次支持统一虚拟地址。CUDA 6.0[32](Kepler 之后的架构可支持)引入了统一内存的

概念, 其创建了一个托管的内存池被 CPU 与 GPU 共享, 程序中 CPU 与 GPU 端的操作都可使用指针访问托管内存中的数据。但这种托管内存必须在 kernel 启动前同步, 统一内存的地址空间最大为 GPU 物理内存空间的大小。在 Pascal[13]架构中, 虚拟地址空间被扩大至 49 位足够覆盖主机端的 48 位虚拟地址空间, 进而支持不受限的统一地址上的访存。在 Volta 架构 GPU 上为 32 个并发地址空间提供了硬件支持。

Ausavarungnirun 等人[33]的工作中尝试使用可变页表的方法提高 GPU 的计算效率。在他们之后的工作 MASK[25]中, 提出了基于多地址空间的并发 kernel 执行框架, 试图减少 GPU 上并发程序间的干扰以及地址转换的开销。当前 GPU 虚拟内存的问题有: 缺少对多地址空间的内存保护; 会因单个 warp 的 TLB 丢失会导致多个 warp 的挂起, 产生很高的延迟; 多 SM 共享的 L2 cache 会因多程序的争用进一步提高 TLB 的丢失率; L2 cache 与全局内存中的数据请求与地址转换请求间因相互干扰会进一步增加 GPU 的计算延迟。为此, 其具体工作包括: 基于多程序地址空间的方式提供内存保护; 使用 TLB-Fill Tokens 技术减少多 SM 间共享的 L2 TLB(位于 L2 cache 上)上地址转换的丢失率; 使用对地址转换敏感的 L2 cache 绕过机制减少 L2 cache 中数据请求与地址转换请求之间的干扰; 使用地址空间敏感的内存调度器以减少全局内存上地址转换的开销。

多程序内存保护的增强。MASK 允许不同的 SM 拥有各自的地址空间。利用每个 SM 上的根页表寄存器(类似于 x86 架构中的 CR3 寄存器)设置每个 SM 的地址空间。每个 SM 的根页表寄存器中的值同时也被存放在 TLB 机制中的根页表(位于 cache 上)中, 以被页表遍历器使用。如果 SM 中的根页表寄存器中的值发生变化, SM 会保守的执行完正在运行的内存请求, 以保证计算的正确性。通过扩展 L2 TLB 中的地址转换项标识每项所对应的 SM。TLB 上的每次刷新操作只针对一个 SM, 刷新内容包括该 SM 的 L1 TLB, 以及共享 L2 TLB 中属于该 SM 的所有条目。

TLB-Fill Tokens 的机制如图 14 中①所示, TLB-Fill Tokens 使用时间窗口(10^6个 GPU cycle)和令牌(token)限制每个 SM 可向共享 L2 TLB 写入的 warp 的数量, 以此减少共享 L2 TLB 上单程序地址空间内的干扰。单程序所有的 warp 都可以探测共享 L2 TLB, 只有持有令牌的 warps 可以向其中写入数据。没有令牌的 warp 请求的页表项(PTEs)会被缓存在一个小的 TLB 旁路 cache(包含 32 个 TLB 项)中。由于单个程序的数据局部性, 其上不同 warp 有相似的 TLB 命中率, 因此没有令牌的 warp 会受益于持有令牌的 warp 将其所需的 TLB 项载入了共享 L2 TLB 中。

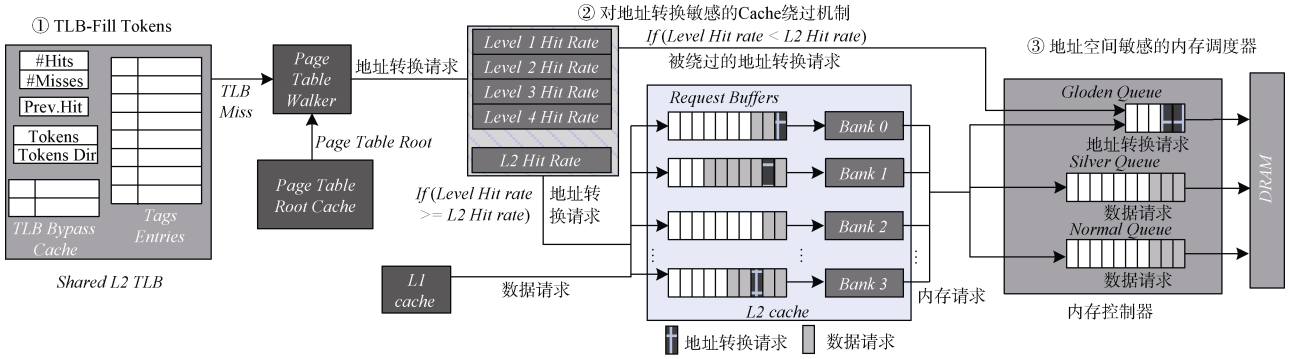


图 14 MASK 框架图[25]

Figure 14 Architecture of MASK[25]

对地址转换敏感的 cache 绕过机制如图 14 中②所示, 其负责寻找需被绕过处理的地址转换请求。MASK 采用 4 级页表机制(与主流 GPU 相似), 页表级别越低越可能被共享并在线程间重用, 因此命中率更高。地址转换请求在每级页表上生成相应的访存请求, 这会在 L2 cache 上产生排队等待的延迟。绕过机制通过对比 L2 cache 上数据请求与各级页表转换请求的命中率决定需要被绕过的地址转换请求。例如: 当某级的地址转换命中率低于数据请求命中率时, 该级页表的地址转换将绕过 L2 cache, 直接发送至全局内存中的内存调度器。

地址空间敏感的全局内存调度器如图 14 中③所示, 调度器使用三个独立的队列处理内存请求。黄金队列(Golden Queue)是长度较小的 FIFO 队列, 只用来处理地址转换请求, 其中的请求会被最先处理; 白银队列(Silver Queue)处理某一个特定程序的内存请求, 优先级次之; 普通队列(Normal Queue)处理来自其他程序的内存请求, 白银队列中的请求可以抢

占普通队列的请求优先执行, 以此防止普通队列中的请求占据大量内存带宽的情况发生。

MASK 实现了 GPU 上多多地址空间, 并对每个程序的内存段进行保护, 提高 GPU 整体的吞吐量和单指令处理周期(Instruction per cycle, IPC)。其工作被英伟达参考, 在 GRID vGPU[34]中应用。

除了 MASK 工作以外, **GPU 半虚拟化与全虚拟化相关工作也提出了其他的虚拟内存实现技术**。Suzuki 等人[35]基于 Xen 实现了 GPU 全虚拟化和半虚拟化方案 GPUvm。其使用了 GPU 影子页表, GPU 影子通道和虚拟虚拟 GPU 调度器等技术。这些资源管理技术将 GPU 以全虚拟化或半虚拟化的方式提供给客户虚拟机。GPUvm 使用开源的 GPU 驱动①在 Xen Hypervisor 中实现, 可支持 Fermi 和 Kepler 架构的 GPU。其架构如图 15 所示, GPU 访问聚合器仲裁对 GPU 的所有访问实现多虚拟机对 GPU 资源的共享。该机制实现了多虚拟机面向 GPU 内存资源的隔离使用; 对于计算资源, 其以非抢占式时间复用的方式实现。

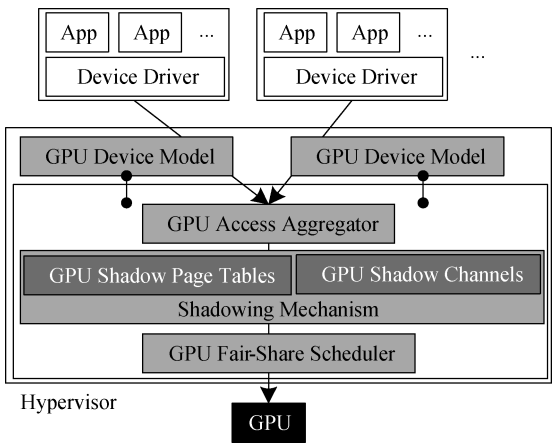


图 15 GPUvm 架构图[35]

Figure 15 Architecture of GPUvm[35]

GPUvm 提供 GPU 影子通道, 将对 GPU 的访问以虚拟机为单位隔离。GPU 通道的物理索引对所有虚拟机隐藏, 通过将物理索引映射到虚拟索引的方式, 完成虚拟通道的分配。为了确保一个虚拟机无法访问另一个虚拟机的内存区域, GPUvm 为每个 GPU 通道创建一个影子页表, 其不受客户操作系统的影响。所有 GPU 内存访问都由 GPU 影子页表处理; GPU 内存的虚拟地址是由影子页表转换得到, 而不由客户设备驱动程序设置。GPUvm 会验证影子页表的内容, 以保证 GPU 内存可在多个虚拟机间安全共享。通过使用 GPU 影子页表, GPUvm 保证了由 GPU 发起的 DMA 不会访问分配到虚拟机之外的内存区域。GPUvm 不能应对 GPU 内存中的页错误, 因此需要影子页表及时更新的更新。

GPUvm 是一份完整的 GPU 虚拟化方案, 但是其基于开源 GPU 驱动实现且无法实现计算资源空间复用的限制, 使其无法在公有云中大规模应用。

GPU 半虚拟化的工作还有 LoGV[36], VGRIS[37]等。LoGV 使用逆向工程得到英伟达的开源驱动, 在 KVM 上实现了 GPU 半虚拟化, 其也实现了支持多程序的多地址空间和内存保护。VGRIS 在 VMWare SVGA[38]上实现了半虚拟化。

基于以上的研究, 可以看到计算资源的空间复用技术和多地址空间的虚拟内存技术是单 GPU 虚拟化技术的发展方向, 二者的协同使用在理论上可提供云平台上安全可用的单 GPU 虚拟化环境。

3.2 “多虚一”的 GPU 虚拟化技术

在分布式集群中, 多个计算节点中有数量不等的 GPU, “多虚一”的 GPU 虚拟化提供给一个租户多个 GPU 使用, GPU 的位置对用户透明。相关工作如 rCUDA[39], PALMOS[40]和 VOCL[41]等。

rCUDA 是最早的分布式集群中共享 GPU 的虚拟化解决方案。rCUDA 允许集群中只在部分节点上部署 GPU, 当 GPU 通用计算的任务被提交至集群中时, rCUDA 框架将计算请求发送至 GPU 物理节点计算并将结果返回给客户端。rCUDA 实现了物理节点与 GPU 解耦, 形成逻辑上共享的 GPU 资源池, 从而增强集群配置的灵活性, 支持单个节点利用集群中安装的全部 GPU。CUDA 应用程序可以与集群中的任何 GPU 交互, 而不受其物理位置的影响, 这不仅节省了空间, 而且节省了能源、购置和维护费用。不足之处在于 rCUDA 框架并没有考虑安全控制并且需要对程序代码进行修改。图 18 为使用 rCUDA 的集群与传统集群的对比, 图 19 为 rCUDA 的结构。rCUDA 的实现中需要进程间的数据通信以及应用程序的调度, 该工作目前支持 InfiniBand 网络的加速。

PALMOS 是一种考虑进程安全的应用层实现的虚拟化层, 位于应用程序与操作系统之间, 将集群中的 GPU 提供给多进程或多用户使用。无需修改操作系统、OpenCL 运行时以及应用程序。其提供了细粒度的多 GPU 管理以及应用程序在集群中的调度。

① GPUvm 无法使用原生的英伟达设备驱动。

该工作已基于标准 OpenCL 实现, 具有可移植的特点并对应用程序透明。

如图 16(a)所示, 在标准的集群环境中, 应用程序将直接通过操作系统使用使用 GPU 资源, 若节点上没有 GPU 资源, 或 GPU 资源被占用, 则应用程序将无法启动或等待执行。如图 16(b)所示, PALMOS 通过对标准 OpenCL 的封装, 以及其运行时模块提供的资源管理与程序调度, 实现了 GPU 任务在集群中分发, 较好的应对单节点 GPU 负载压力过大的问题。PALMOS 是应用程序与操作系统之间独立的一层, 对客户端发送的 GPU 请求进行分析, 从集群的角度对 CPU 和 GPU 资源进行管理并对 GPU 任务进行调度, 以提高集群的吞吐量。

PALMOS 的设计如图 17 所示。Level0 为应用程序接口层, 负责将应用程序的 GPU 调用发送至 PALMOS 中。Level1 中的 PALMOS 运行时系统, 负责 GPU 资源管理与应用程序调度, 存储分配管理及安全管控。Level2 层负责 PALMOS 与操作系统和运行时库的连接。

Xiao 等人[41]提出了一种针对 OpenCL 应用的 GPU 虚拟化解决方案 VOCL。与 rCUDA 类似, VOCL 采用基于远程 GPU 的加速, 提供支持 OpenCL 的虚拟设备。VOCL 在客户端提供 OpenCL 封装器库, 在服务器端提供 VOCL 代理进程。代理进程接收来自封装库的输入, 并在远程 GPU 上执行它们。OpenCL 封装库和 VOCL 代理进程通过 MPI[42]进行通信。与其他传输方法相比, MPI 可以提供丰富的通信接口, 动态地建立通信通道。可以将 VOCL 理解为在集群中由 MPI 实现的 OpenCL 应用程序运行环境。

“多虚一”的 GPU 虚拟化技术使 GPU 卡与计算节点解耦合, 即 GPU 任务可以交由集群中远程的且数量有限的 GPU 卡完成。这种模式可以减轻集群的建造和运营成本。但同时增加了 GPU 任务在集群中的单节点上的同驻几率, 带来了新的安全漏洞。

3.3 硬件辅助虚拟化

3.3.1 NVIDIA MPS

GPU 多进程服务[43](MPS)是一种可选的、二进

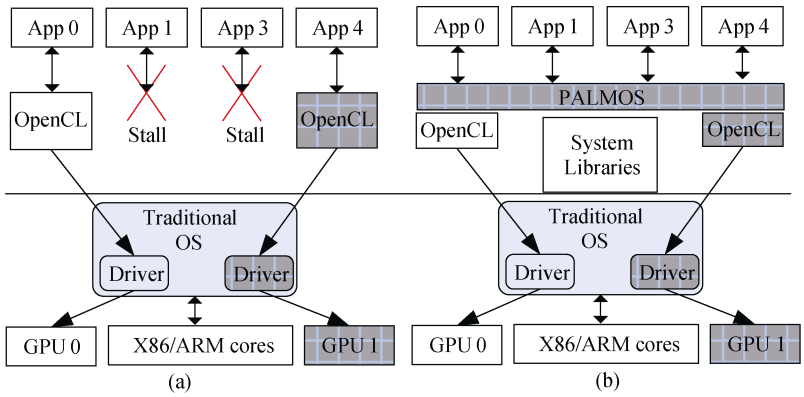


图 16 PALMOS 系统于传统系统的对比[40]

Figure 16 PALMOS vs. Traditional OS[40]

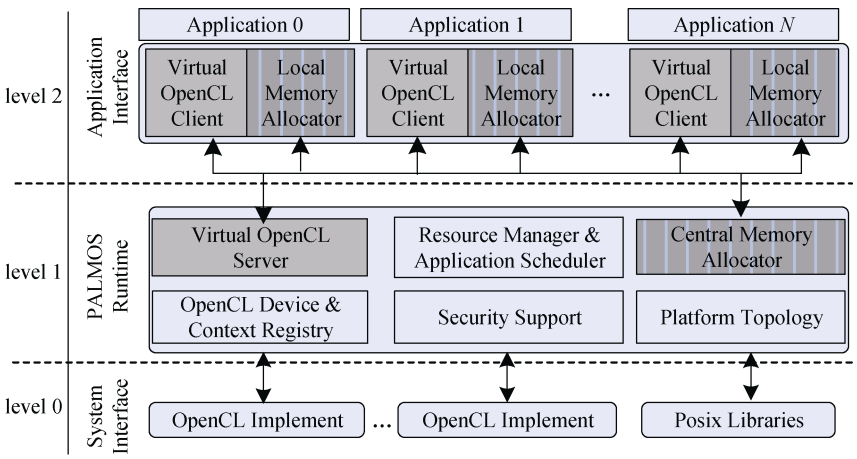


图 17 PALMOS 系统的设计[40]

Figure 17 Design of PALMOS[40]

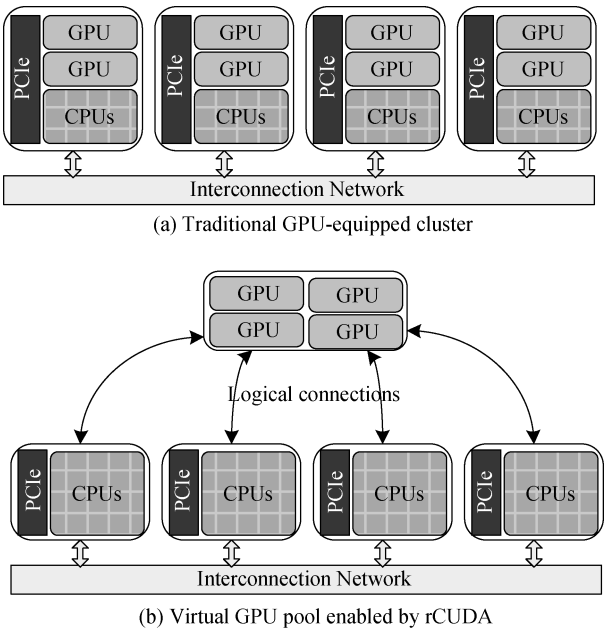


图 18　传统集群与 rCUDA 集群对比[39]

Figure 18　Traditional cluster vs. rCUDA cluster[39]

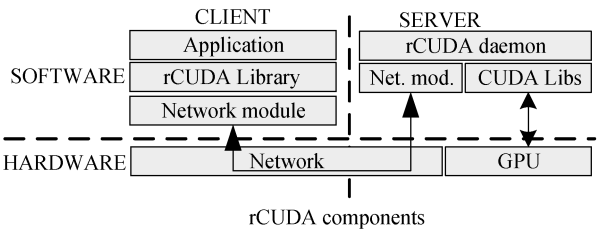


图 19　rCUDA 架构图[39]

Figure 19　Architecture of rCUDA[39]

制兼容实现的 CUDA 应用程序编程接口(API)。多进程服务被设计用来透明的支撑多个 CUDA 应用程序并行执行, 以充分利用 GPU 的 hyper-Q 的计算能力。英伟达最新的 Volta 架构[43]相比之前的架构对多进程服务的功能进行了升级, 相比于之前多进程服务实现中 CUDA 程序必须通过 CPU 端的服务向 GPU 端发送计算任务, 新的多进程服务的实现中 CUDA 程序可以直接将计算任务发送至 GPU 端, 同时新的多进程服务的实现支持多地址空间, 即每个 CUDA 应用程序拥有其自己的地址空间, 而不是多个 CUDA 应用程序分享 GPU 的统一地址空间。此外, 新的多进程服务的实现通过限制每个 CUDA 应用程序的资源实现服务质量控制。如图 20 中的对比所示。

需要注意的是, 多进程服务的客户端共享 GPU 的调度和错误报告, 即单个客户端的异常会被告知到所有的客户端(具体哪个客户端的异常并不告知), 同时单个客户端异常导致的 GPU 停机也会影响到所有的客户端。如果提供多进程服务的服务器同时包括 Volta 架构和之前架构的的 GPU, 多进程服务只能利用部分的 GPU(提供新的多进程服务则无法使用 Volta 之前架构的 GPU, 提供旧的多进程服务无法使用 Volta 架构 GPU)。Volta 架构的多进程服务通过限制每个客户端可使用的线程数量, 可以使客户端的任务被分配到一定数量的 SM 中。

3.3.2　NVIDIA GRID vGPU

本文之前讲述的 GPU 计算资源复用技术多构建在消费级 GPU 之上。为了满足市场上对 GPU 虚拟化的需求, 英伟达推出了 GRID 系列 GPU(其售价远高于消费级 GPU)。GRID 系列 GPU 通过硬件辅助虚拟化的方式支持虚拟 GPU (vGPU) [32]。虚拟机可以使用 vGPU 直接访问物理 GPU, 多个虚拟机可共享一个 GPU。其架构如图 21 所示, 基于 Hypervisor 中虚拟 GPU 管理器, GRID GPU 支持将多个 vGPU 直接分配给租户。vGPU 使用穿透(Pass through)技术实现, 因此客户虚拟机使用 vGPU 和物理 GPU 的方式相同: 客户虚拟机加载英伟达官方驱动后使用 vGPUs, 目

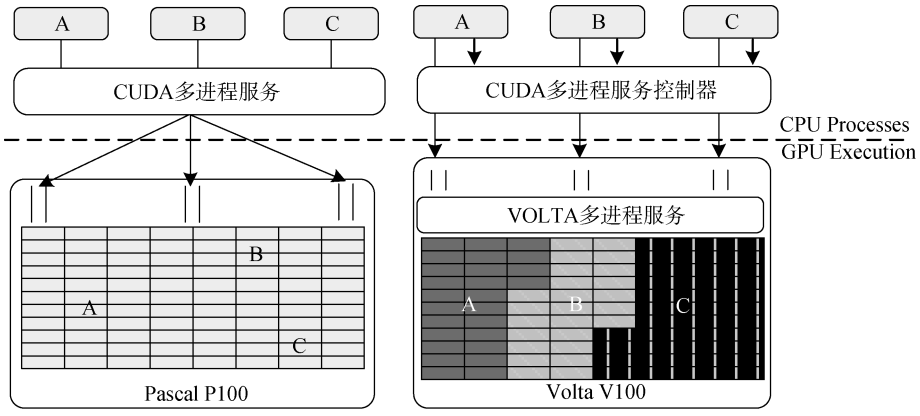


图 20　Volta 中的 MPS 与之前架构中 MPS 的对比[13, 43]

Figure 20　MPS of Volta vs. MPS of older GPU[13, 43]

前 vGPU 拥有固定数量的 GPU 内存和计算资源, vGPU 在其自身创建时从物理 GPU 内存中得到其独占的内存, 并在虚拟机销毁后释放。所有驻留在物理 GPU 上的 vGPU 共享 GPU 中计算单元。目前支持通用计算的 GRID GPU 有 M60, M10, M6 等, 其可以提供对 Xen 和 VMware vSphere 的支持。

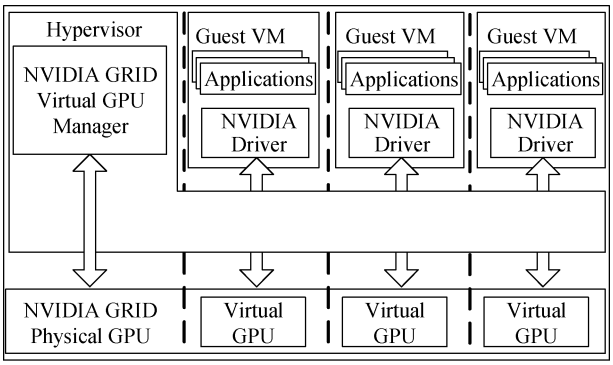


图 21 NVIDIA GRID 系列 GPU 中虚拟 GPU 实现[32]
Figure 21 vGPU of NVIDIA GRID GPU[32]

3.3.3 基于 Docker 的 GPU 虚拟化

容器将应用程序封装在相互隔离的虚拟环境中, 可以简化数据中心的部署。其是操作系统级的虚拟化技术, 通过操作系统内核机制来进行虚拟化实现。英伟达发布了 nvidia-docker[44], 其在 docker[45]上做了一层封装, 以支持容器中使用 GPU 实现业务, 即无需对 GPU 应用程序进行修改就可以在容器中使用 GPU, 同时可以利用 docker 中的部分隔离机制。目前 nvidia-docker 可实现跨环境的 GPU 支持, 方便云平台中 GPU 环境的部署与应用。但是目前该方法(最高版本为 2.0)不支持多进程服务(MPS), 容器引擎中(如 docker)GPU 的资源隔离是以单块 GPU 卡为粒度, 因此容器管理工具(如 Kubernetes)上的多个租户可以单块 GPU 为粒度隔离。例如: 可将租户 1 的镜像可被分配 0 号 GPU, 租户 2 被分配 1, 2 号 GPU;若两个租户同时使用一块 GPU, 则 GPU 内部无隔离, 两者的程序会依次顺序执行。

3.4 典型 GPU 虚拟化技术及其安全机制总结

如以上展示的内容, 学术界与产业界在近年来都在致力于开展 GPU 虚拟化相关工作。学术界多以消费级 GPU 为平台, 尝试在计算资源上开展时间复用、空间复用的工作, 同时提出 GPU 上内存的虚拟化方法。产业界多通过硬件增强的方式, 将学术界提出的方案进行产业化, 如英伟达的 GRID 系列 GPU。这其中我们可以看到, 为实现租户间 GPU 的资源分割、资源隔离与性能隔离的安全机制, 空间复用是对于计算资源进行分割与共享的较好选择, 使租户间无法产生片上资源的争用; 对于 GPU 上的全局内存, 需通过硬件增强的方式实现支持多地址空间的 GPU 虚拟内存, 同时实现虚拟内存对计算资源空间复用的支持, 对于无法支持多地址空间的 GPU, 可以选择基于开源驱动的方式实现内存段保护; 对于分布式集群中使用 GPU, 远程 GPU 逻辑池会增加同驻攻击的威胁。综上, 可以认为已有的典型 GPU 虚拟化方法并没有开展系统性的安全工作。

4 云平台中已有的 GPU 攻击方法分析

目前, 提供 GPU 通用计算服务的云平台, 多数选择使用英伟达的 GRID 系列 GPU, 基于穿透技术实现 GPU 物理资源在多虚拟实例间的分享。本文之后展示的 GPU 攻击方法都是在英伟达提供 GPU、SDK 和官方驱动上开展的。

本节对现有的 GPU 攻击方法进行分析, 包括隐秘信道攻击, 侧信道攻击和内存溢出攻击。在隐秘信道攻击中, 木马程序与间谍程序首先在 GPU 上实现同驻, 随后基于 GPU 物理资源的争用实现了间谍租户与木马租户间的信息传输; 在侧信道攻击中, 攻击者可以通过资源争用方式或非资源争用方式获取受害者的敏感信息; 内存溢出攻击中, 攻击者基于 CUDA 的漏洞获得受害者在内存中的敏感信息。

4.1 隐秘信道攻击

Naghibijouybari 等人[46]的工作首次实现了 GPU 上的隐秘信道攻击。该工作首先通过逆向工程推测英伟达 GPU 上 kernel 与 warp 两级的调度策略, 并以此实现木马程序与间谍程序在 GPU 上的同驻; 然后, 利用木马程序与间谍程序在 GPU 上的资源(cache, FU 和全局内存)争用建立起隐秘信道; 并对已建立的隐秘信道进行优化, 增加其带宽与鲁棒性。

隐秘信道攻击场景如图 22 所示, 多个虚拟实例的 GPU 程序同时在一台配备 GPU 的计算节点上运行, GPU 上并发执行的 kernel 中包含一个木马 kernel 和一个间谍 kernel。攻击假设木马与间谍可实现含有 GPU 的计算节点上的同驻(计算节点同驻属于云平台传统安全问题)。该攻击可以发生在当前的云环境中, 也可以发生在其他多任务在同一服务器上共享 GPU 的非云环境下。

GPU 中同驻关系的构建。英伟达 GPU 平台的 kernel 调度算法并没有公开, 因此, 必须通过逆向工程推测该算法。首先在 GPU 上发起两个不同的 kernel 并探测它们是否同驻在一个 SM 上, 每个 kernel 的线程读取它所使用的 SM 的标识号(SMID), 以此确定

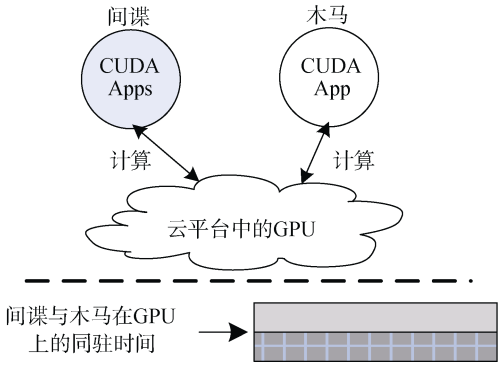


图 22　GPU 中隐秘信道攻击场景

Figure 22　Attack scenario of covert channel in GPU

线程块所使用的 SM, 同时使用计时函数来确定每个线程块的计算开始时间和结束时间。经多次不同数量和维度的线程块重复试验, 可逆向得出 GPU 程序的 kernel 以轮询算法(Round-Robin)分配给不同的 SM。如果有 GPU 中有 SM 处于空闲状态或 SM 内有剩余 warp 可执行运算, 则尝试放置第二个 kernel 的线程块, 若剩余资源不够第二个 kernel 的线程块启动, 则需排队等待资源, 这种资源分配方式为“剩余策略”。多个 GPU 程序除了争用 GPU 上的 SM 外, 也在 SM 内争用 warp 调度器。在大多数 GPU 中, 每个 SM 上有多个 warp 调度器。每个 warp 将被分配到一个 warp 调度器中, 单个 warp 调度器最多支持 4 个 warp。如果不同 kernel 的 warp 共用一个 warp 调度器, 则该 warp 调度器下的带宽也将被争用。经实验发现 warp 调度器也使用轮询算法对 warp 进行调度。基于 GPU 上的各层级调度算法, 间谍程序与木马程序可以通过设置线程空间大小和片上资源的使用量, 来实现 SM 级或 warp 级的同驻。具体而言: 每个 kernel 启动的线程块数量等于或超过设备上 SM 的数量(小于 GPU 支持的最大线程数量), 且每个线程块不会耗尽 SM 的资源, 那么它们可在 SM 中实现同驻。例如, 在 Tesla K40C 上(包含 15 个 SM 且每个 SM 上有 4 个 warp 调度器), 如果每个间谍和木马程序都使用 15 个线程块, 且每个线程块包括 4 个 warp(4*32 个线程), 即可在 GPU 上实现 SM 与 warp 调度器两个级别的同驻。

Cache 硬件细节的挖掘。由于常量内存体系在 L1 cache 和 L2 cache 上占用较小的空间, 容易在其上实现常量 cache 上的争用, 因此适合建立隐秘信道。为实现隐秘信道首先要了解常量 cache 的硬件细节, 使用固定步长模式(strided)从常量内存中读取不同尺寸的数组, 通过观测访问延迟的变化可确定 L1 常量 cache 和 L2 常量 cache 的容量、cache line 大小和 cache 组相联方式等。例如图 23(a)与(b)分别展示了英伟达 K40C 上 L1 cache 与 L2 cache 的延迟。GPU 中的内存与 cache 间使用组相联的映射方式。当 cache 中缓存了全部的数组时, 延迟是稳定的。若数组大小超出 cache 容量, 将在一个 cache 组中发生 cache 缺失, 进而导致延迟升高。在不停增大数组的过程中, cache 丢失会影响到剩余 cache 组, 进而得到 cache 组的数量等于图中阶梯的数量, 以此方法最终得到 L1 cache、L2 cache 的硬件细节如表 4 所示。

通过 L1 cache 建立隐秘信道。在间谍 kernel 与木马 kernel 同驻在一个 SM 上之后。木马通过制造 cache 争用发出 1 信号, 或不制造争用发出 0 信号。为了在一个 cache 组中制造争用, 木马开辟同 L1 cache 一样大的 2KB 的数组, 并以 512B(8 组*64B)的步长读取数据以确保被访问数据被载入同一个 cache 组中。间谍同样开辟 2KB 的数组, 并以相同的步长访问数组, 同时记录访问的延迟, 当获得高延迟时则为 1 信号, 低延迟为 0 信号。根据文献[46]结果显示, 在 Kepler 架构 K40C GPU 上, 当木马制造

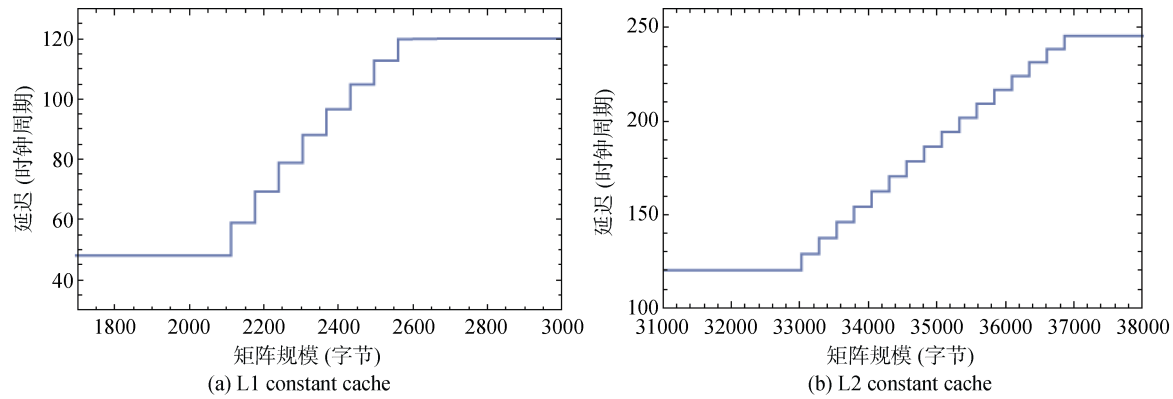


图 23　常量 cache 上的访问延迟[46]

Figure 23　Latency of constant cache[46]

表 4 常量 cache 的硬件细节

Table 4 Hardware details of constant cache

Cache	架构①	Cache line	Cache 组	联接方式	大小
L1 常量 cache	K, M	64B	8 组	4 路	2KB
L2 常量 cache	K, M, F	256B	16 组	8 路	32KB
L1 常量 cache	F	64B	16 组	4 路	4KB

cache 争用时, 间谍测量结果为 112 个时钟周期, 没有争用的情况为 49 个时钟周期。最终, 在一个 cache 组中制造争用可以获得 42Kbps 的带宽。

通过 L2 cache 建立隐秘信道。如果木马程序与间谍程序不能在同一个 SM 上同驻, 可以通过 L2 常量 cache 构建隐秘信道(L2 cache 被所有 SM 共享)。实现方式与 L1 cache 的相似, 木马和间谍使用 32KB 的数组大小, 并以 4096B 的步长读取数据以在一个 cache 组制造争用。L2 cache 得到 20Kbps 的带宽[46]。

基于 cache 争用建立隐秘信道的方法在 Fermi、Kepler 和 Maxwell 三种架构的 GPU 中都可生效。为了使间谍程序和木马程序在执行期间重叠且实现无误差通讯, 需要多次发送同一位的数据(对于 Kepler 架构, L1 上需要发送 20 次, L2 上需要发送 2 次[46])。原因在于, 一是没有木马与间谍同步的情况下, 必须足够多的重复次数确保间谍和木马间执行的重叠; 二是 GPU 上的计时函数对访存行为的计时会因不可避免的时钟漂移现象, 产生不稳定的计时信息。

功能单元(Functional unit, FU)上隐秘信道的建立。FU 的争用也可以建立隐秘信道, 当两个线程块向同一个功能单元发出指令时, 会得到比一个线程块单独发出指令更大的延迟。延迟为隐秘通道的带宽设置了上限, 单个操作出现争用的延迟是通信的最小周期。构建 FU 上的隐秘信道需要首先选择实现争用的硬件单元。FU 包括单精度计算单元(SP)、双精度计算单元(DPU), 以及特殊功能单元(SFU), 它们用以支持不同类型数据的不同计算请求。分次启动多个不同操作类型(sinf 与 sqrt 在 SFU 上执行, Add 与 Mul 在 SP、DPU 上执行)的 kernel, 同时增加 warp 的数量并测量因争用产生的延迟。计时结果显示加与乘操作的延迟不可见, 数量有限的 SFU 在使用 sinf 操作可在 warp 调度器与 SFU 上制造可测量的延迟, 故以此建立隐秘信道。每个 SM 中的 FU 数量取决于 GPU 的架构, 如表 5 所示。FU 上的争用也因架构不同而效果不同, 如 Maxwell 中每个 SM 被切分为 4 个象限, 每个象限有其独占的资源, warp 调度器负责一个象限的 FU 管理, 单个象限中的 FU 用尽后可观察到争用行为。而 Fermi 与 Kepler 架构的 warp 调度器没有独占的资源, 其上的多个 warp 调度器共享 FU, 并基于软件调度的方式实现资源共享, 由于其上包含数量更多的可用 FU, 延迟出现的时间相较于 Maxwell 架构更晚。

表 5 不同架构 SM 上硬件单元的个数[9-11]

Table 5 Number of physical units on SM of different architecture[9-11]

GPU	Warp Scheduler	Dispatch Unit	SP	DPU	SFU	LS/ST
Tesla C2050 (Fermi)	2	2	32	16	4	16
Tesla K40C(Kepler)	4	8	192	64	32	32
Quadro M4000(Maxwell)	4	8	128	0	32	32

与 cache 构建隐秘信道方法一致, 在实现 SM 上同驻后制造 SFU 争用发送 1 信号, 反之发送 0 信号。没有争用的情况下, 可在 Fermi 架构上观测 SFU 操作延迟为 41 个时钟周期[46]。争用会导致延迟增加到 48 个周期[46], 最终可在 Fermi 架构 C2050 GPU 上得到带宽为 21Kbps 的隐秘信道[46]。同样的攻击方法在 Kepler 架构和 Maxwell 架构上也成立, 分别在 K40C 和 M4000 GPU 上获得 24Kbps 和 28Kbps 的带宽[46]。

全局内存中建立隐秘信道。对于木马与间谍不能在 SM 上同驻的情况, 可使用全局内存上原子单元(atomic unit)的争用创建隐秘信道。具体实现方式分为 3 种: (1)木马程序和间谍程序的每个 GPU 线程在一个特殊的全局内存地址执行原子加操作; (2)两个程序的每个 GPU 线程在全局内存上以固定步长执行原子加操作, 且每个 warp 中的所有线程都执行合并访问; (3)两个程序的每个 GPU 线程在连续的全局内存中执行原子加操作, 且每个 warp 中的所有线程不执行合并访问。3 种方法的访存地址都由线程号决定。攻击实验结果显示, 方法(1)获得带宽最高。基于全局内存中的原子操作在 Kepler 和 Maxwell 架构的

① K, M, F 分别代表 Kepler、Maxwell 和 Fermi 架构, 数据出自 Tesla C2050, Tesla K40C, Quadro M4000 3 种 GPU。

GPU 上得到的带宽明显高于 Fermi 架构的 GPU, 这是因为其上的原子操作被 L2 cache 支持, 且添加了更多的原子单元, 因此提升了原子操作吞吐量。

隐秘信道的强化。在已有的隐秘信道的基础上, 基于 GPU 提供的大量硬件资源可进一步提高信道的带宽和鲁棒性。同时使用多种资源合作提升带宽是可行的, 例如, 同时传输 2 位数据, 分别使用 L1 常量 cache 和 SFU, 这种方式与在单资源带宽优化相正交, 可在单资源带信道强化的基础上叠加进行, 故不做深入研究。对于单物理资源上的信道(只在 cache, FU 或全局内存中, 不通过两种资源同时使用强化信道)强化, 可使用并行化通信的机会实现多个木马与多个间谍间的同时通信, 或使用同步机制以减少因时钟漂移或 kernel 连续启动带来的带宽损失。

Cache 隐秘信道带宽提升。间谍程序和木马程序间隐秘信道的同步机制可通过同时使用 3 个不同的 cache 组来完成, 使用一个 cache 组作为数据传输, 另外两个 cache 组以握手机制实现同步。为进一步提升隐秘信道的带宽, 可充分利用 GPU 上的硬件资源, 使用两个 cache 组实现同步机制, 同一 SM 上剩余的多个 cache 组同时用来发送数据。使用多个 SM 并行通讯可进一步提升带宽, 如果在多个 SM 上实现了木马和间谍的同驻, 每组同驻的实例都可以独立的利用 SM 上的资源进行通信。表 6 展示了多方法优化后的隐秘信道带宽。由此可见, 高质量高稳定性的隐秘信道带宽可以在 GPU 上实现。对于 L2 常量 cache 隐秘通道, 也可以制造多个 SM 的并行通信进行优化, 最终可观察到最高 8 倍的带宽提高[46]。

FU 信道的强化。和 L1 cache 隐秘信道相似, 理论上 K40C 上通过 15 个 SM 并行通信带宽可以提升 15 倍。且 warp 调度器的争用还可以进一步增加带宽, 由于 warp 调度器的争用是相互隔离的, 可使用多个 warp 调度器同时发送数据。相对于基础的 FU 隐秘信道, 基于 warp 调度器间和 SM 间的并行机制, 得到强化带宽的隐秘信道带宽表 7 所示。

表 6 强化后基于 L1 cache 的隐秘信道带宽[46]

Table 6 Improved L1 Channels[46]

GPU	L1	同步	同步加多位	同步, 多位加并行
Tesla C2050 (Fermi)	33Kbps	61Kbps	207Kbps	2.8Mbps
Tesla K40C(Kepler)	42Kbps	75 Kbps	285Kbps	4.25Mbps
Quadro M4000(Maxwell)	42Kbps	75Kbps	285Kbps	3.7Mbps

表 7 强化后基于 FU 的隐秘信道带宽[46]

Table 7 Improved FU Channels[46]

GPU	SFU	同步	同步加多 SM 并行
Tesla C2050 (Fermi)	21Kbps	28Kbps	380Kbps
Tesla K40C(Kepler)	24Kbps	84Kbps	1.2Mbps
Quadro M4000(Maxwell)	28Kbps	100Kbps	1.3Mbps

干扰控制。其他业务负载干扰隐秘信道的构建有两种方式: (1)同驻干扰, 其他业务负载阻止间谍和木马一起开启, 进而使它们分配到不同的 SM 上, 使得 SM 内部的高效率信道难以建立; (2)噪声干扰, 当木马与间谍同驻后, 会因其他业务负载使用构建隐秘信道的硬件资源, 使间谍收集到含有噪音的信息。由于 GPU 的高并发性, 这将极大的影响隐秘信道信息传输的准确性。为此, 可以根据 GPU 资源的限制, 利用剩余策略使间谍和木马程序获得 SM 级别或是整 GPU 级别的独占来控制噪声。例如, 设置间谍的每个线程块使用单个 SM 全部的 shared memory, 且木马的线程块块不使用共享内存, 此时木马和间谍程序可以同驻在 SM 上, 而其他使用 shared memory 的 GPU 程序因资源不足无法启动, 进而实现间谍和木马程序独占 GPU 资源进行通信。

4.2 侧信道攻击

4.2.1 基于资源使用度量的侧信道攻击

Naghibijouybari 等人[47]的后续工作实现了 GPU 上多类型的侧信道攻击, 分别是图形图像程序攻击图形图像程序、CUDA 程序攻击 CUDA 程序、图形图像程序攻击 CUDA 程序, 并给出了每种类型的侧信道攻击的几种案例。因本文只关心 GPU 作为通用计算平台的虚拟化技术及其安全问题, 故本节首先分析了 GPU 上的安全漏洞; 之后展示了可用于实现攻击的的三种侧信道; 最后对 CUDA 程序攻击 CUDA 程序的案例做深入分析, 具体而言为间谍利

用 CUDA 程序获取受害者由 CUDA 实现的神经网络模型。

GPU 上漏洞分析。上节的工作通过 GPU 上同驻的间谍和木马程序制造了在 cache、FU 和全局内存中原子单元上的资源争用, 并以此建立了隐秘信道, 但这种方法无法用于侧信道的建立, 原因在于 GPU 上活动线程数量大且 cache 容量较小, 使得在 GPU 上 cache 进行 prime-probe[48]或类似的侧信道攻击很难实现。同时, 已有的 CPU 侧信道攻击方法很难在 GPU 实现, 针对 CPU 的侧信道攻击方法包括: 基于计时分析、功率分析、分支预测分析, 或针对指令缓存等多种攻击方法, 它们都会因 GPU 的 SIMT 架构的限制无法开展。例如: 当 GPU 编程模型中线程块执行判断语句(if-else)时, 若两个分支都有线程满足条件, 那么两条分支都会执行, 使得无法观测到时间上的区别, 导致基于指令分支预测的侧信道无法在 GPU 上实现。

因此, 侧信道攻击不能通过细粒度的争用行为建立。而**通过跟踪 GPU 整体资源争用情况并对数据进行聚合分析是实现侧信道攻击的有效方法。**GPU 上实现间谍与受害者同驻后, 可通过以下方法构建侧信道搜集受害者的性能轨迹: (1)调用内存占用接口(API)获得 GPU 上剩余的可用资源数量; (2)调用接口获取 GPU 中硬件性能计数器的数据; (3)对受害者进程进行计时。以上 3 种方法都可被间谍循环调用, 在时间轴上构成对受害者的行为描述的性能轨迹。

GPU 内存资源分配的度量。当 GPU 被用于通用计算, 可通过 CUDA 或 OpenCL 程序使用厂商提供的 API 探测可用的物理 GPU 内存。反复查询这个 API 可跟踪可用内存空间使用量的轨迹。CUDA 应用程序可使用的 API 是“cudaMemGetInfo”, AMD 的 GPU 平台使用 OpenCL 编程, 其上也提供了类似的接口, 从而使此攻击可兼容 AMD 的 GPU。当 GPU 被用于图形图像渲染时, 可以通过调用 OpenGL 的内存分配接口, 获取 GPU 的内存资源分配轨迹。该方法可被应用于间谍使用图形图像程序对受害者的图形图像程序进行攻击的场景。

性能计数器的度量。间谍通过开启 CUDA 程序, 可利用英伟达的性能采集工具[49]获得 GPU 上性能计数器采集中的数据。表 8 展示了跟踪 GPU 行为的一些重要的事件和度量, 它们被分为 5 类: 内存、指令、SM、cache 和纹理(内存)。虽然 GPU 只允许用户观察与自己程序相关的计数器, 但间谍调用这些计数器返回的结果都受受害者 kernel 执行的影响, 例如, 受害者线程块使用 cache, 这可能与间谍产生争用, 在 cache 中替换掉间谍的数据, 之后间谍可通过与 cache 相关的计数器观测到 cache 缺失情况的发生。该方法被用于构建间谍的 CUDA 程序攻击受害者的 CUDA 程序的场景。

表 8 GPU 上的性能计数器[47]

Table 8 GPU performance counters[47]

类别	事件/度量
存储	设备内存读/写事件 全局/本地/共享内存加载/存储事件 L2 读写事件, 设备内存利用率
指令	控制流, 整数, 浮点数(单/双)指令 指令执行/发出, 指令发出/执行的 IPC 值 加载/存储指令 停机等待(数据请求、执行依赖等)
多 SM	单/双精度计算单元利用率 特殊计算单元利用率 纹理功能单元利用率, 控制流功能单元利用率
Cache	L2 命中率(纹理读/写) L2 吞吐量/事务(读/写, 纹理读/写)
Texture	统一 cache 命中率/吞吐量/利用率

时间度量。在间谍程序和受害者程序同时运行时, 也可以度量单个操作的时间来检测硬件争用行为。在间谍程序和受害者程序交织运行的情况下, 时间度量还可以获取受害者计算内核的执行时间。该方法与前两个方法协同使用, 可实现图形图像程序对 CUDA 程序的侧信道攻击。

针对神经网络模型的攻击如图 24 所示, 首先建立间谍 CUDA 程序与受害者 CUDA 程序的同驻, 然后间谍通过性能计数器获得受害者 CUDA 程序的性能轨迹, 最后通过数据聚合分析获得受害者以 CUDA 实现的神经网络模型的结构。具体而言: 受害者执行的程序是从 Rodinia 基准[50]中选择的用

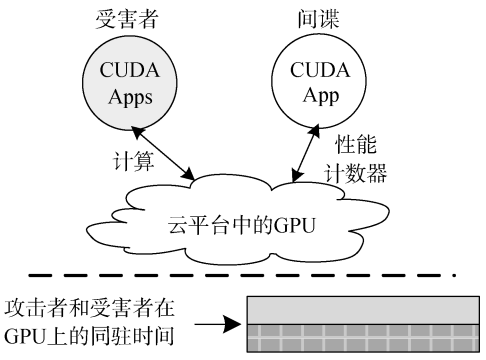


图 24 基于资源度量的侧信道攻击场景

Figure 24 Attack scenario of profile-based side channel

CUDA 实现的反向传播算法, 对于一个神经网络程序, 其网络结构是受害者的一个重要的商业秘密, 间谍通过 CUDA 程序获取受害者性能轨迹, 随后经分析实现对深度神经网络模型的提取。

实验首先根据**逆向工程的方法得到的关于硬件调度器的策略**, 即 GPU 上利用剩余策略给多个 kernel 分配硬件资源, 进而实现间谍 CUDA 程序与受害者 CUDA 程序在每个 SM 上同驻。间谍的程序中连续启动几百个 kernel, 确保可覆盖受害者 kernel 的执行过程。间谍 kernel 启动的数量随着受害者的执行时间的增加而增加。为了使用硬件性能计数器跟踪争用产生的特性, 间谍访问不同的 cache 组, 并在 FU 上执行不同类型的操作。当受害者在 GPU 上使用共享资源时, 根据其神经网络模型输入层大小的不同, 在缓存、内存和功能单元上的争用强度也会随着时间的推移而不同, 从而在间谍 CUDA 程序的性能计数器的测量中产生可测量的泄漏。由此间谍程序可由每个间谍 kernel 收集的性能计数器中的数值。

受害者程序为反向传播算法, 其输入层会输入不同尺寸的数据, 间谍同时开启超过 100 个间谍 kernel 以收集性能数据(在每次 kernel 执行结束时, 测量性能计数器读数)。攻击实验设置受害者 CUDA 应用程序的输入层大小在 64~65536 个神经元, 受害者在每个输入层大小下运行至少 10 个样本的反向传播计算。间谍利用全部间谍 kernel 采集到的数据**得到受害者 CUDA 程序的性能轨迹。**

攻击者选择基于全部时间序列构造特征并使用传统的机器学习方法构建分类器。具体而言, 构建一组统计特征, 包括最小值、最大值、斜率、平均值、标准偏差, 偏斜(Skew)和峭度(Kurtosis), 这些统计特征在受害者程序运行时通过侧通道收集。偏斜和峭度反映了性能轨迹分布的形状。偏斜表示性能轨迹的不对称程度, 峰度表示分布相对于正态分布的相对峰度或平坦度。表 9 展示了几种用于分类的主要特征, 并以此训练分类器(使用 10 倍交叉验证来识别数据集的最佳分类器)。

表 9 用于构建分类器的特征[47]

Table 9 Features for classification[47]

GPU 性能计数器	特征
设备内存写事件	偏斜, 标准差, 均值, 峭度
Fb_subp0/1_read_sector(发送到所有 DRAM 单元 0/1 子分区的读请求数)	偏斜, 峭度
独立 cache 吞吐量(B/min)	偏斜, 标准差
停滞等待	偏斜, 标准差
L2_subp0/1/2/3_read/write_misses(所有 L2 缓存单元的第 0/1/2/3 片上 L2 缓存累积读/写错误数量)	峭度

表 10 展示了通过侧通道攻击识别神经元数目的分类结果。基于 K 最近邻算法(K Nearest Neighbor with 3 neighbors, KNN3)可以高精度的识别出正确的神经元数目(准确率为 88.6%, f-measure 为 86.6%), 证明了 CUDA 应用间侧通道攻击的可能性。

表 10 神经网络模型检测的结果[47]

Table 10 Result of neural network detection[47]

	FM%	Prec%	Rec%
	μ(σ)	μ(σ)	μ(σ)
NB	80.0(18.5)	81.0(16.1)	80.0(21.6)
KNN3	86.6(6.6)	88.6(13.1)	86.3(7.8)
RF	85.5(9.2)	87.3(16.3)	85.0(5.3)

4.2.2 基于时间分析的侧信道攻击

Jiang 等人[51]的工作中完成了针对一种 AES 在 GPU 实现的攻击。AES 算法的 GPU 实现会对 S 盒[52]的多次访问, 由明文内容和加密密钥决定对 S 盒的访存地址, 而访存请求的地址决定了独立 cache line 请求的个数, 该工作发现 kernel 的执行时间与其执行中产生的独立 cache line 请求①数量成线性相关关系, 基于此种相关性关系攻击恢复了 AES-128 GPU 实现的 16 个字节的密钥信息。

攻击模型如图 25 所示, 加密任务在云端 GPU 服务器中完成, 客户端通过网络获得加密服务。客户端发送明文至加密服务器, 加密服务器通过其上的 GPU 实现对明文的加密, 之后将密文发回客户端。

攻击目标是 OpenSSL 0.97[53]中的 AES 算法 GPU 实现(基于 Kepler 架构)。具体而言, AES-128 使用 ECB 模式, 其 S 盒使用 T4 表的实现方法, 每个 GPU 线程负责对 16 字节的明文进行加密, 如图 26 所示。

① 独立 cache line 请求指多个 cache line 请求合并去重后的 cache line 请求。

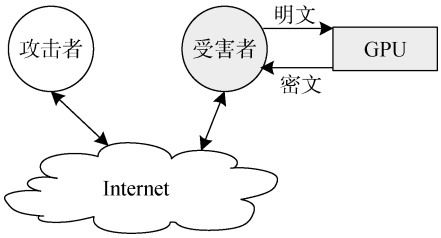


图 25　基于时间的侧信道攻击场景

Figure 25　Attack scenario of time-based side channel

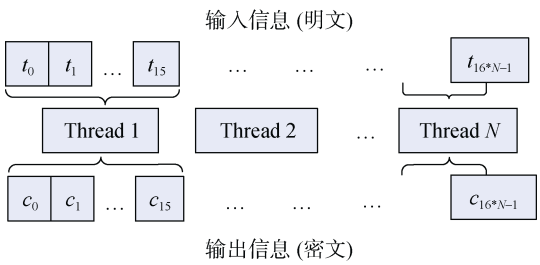


图 26　AES 算法在 GPU 中的实现[51]

Figure 26 AES implement in GPU[51]

密钥生成器会根据 16 位的密钥生成 160 字节的轮询密钥(AES-128 需要 10 轮的轮密钥加操作, 每轮需要 16 字节的密钥), 在前 9 轮中, 明文需要基于 T4 表进行字节替换, 之后进行行位移, 列混淆与轮密钥加操作, 第 10 轮操作中没有列混淆操作。本文的攻击以恢复第 10 轮的密钥为目的, 因为在获得第 10 轮的密钥后, 可以倒推出之前 9 轮的密钥进而获得全部密钥。第 10 轮如公式(1)所示。

$$\begin{aligned}
&c_0 = T4[t_3]_0 \oplus k_0, c_1 = T4[t_6]_1 \oplus k_1,\\
&c_2 = T4[t_9]_2 \oplus k_2, c_3 = T4[t_{12}]_3 \oplus k_3,\\
&c_4 = T4[t_7]_0 \oplus k_4, c_5 = T4[t_{10}]_1 \oplus k_5,\\
&c_6 = T4[t_{13}]_2 \oplus k_6, c_7 = T4[t_0]_3 \oplus k_7,\\
&c_8 = T4[t_{11}]_0 \oplus k_8, c_9 = T4[t_{14}]_1 \oplus k_9,\\
&c_{10} = T4[t_1]_2 \oplus k_{10}, c_{11} = T4[t_4]_3 \oplus k_{11},\\
&c_{12} = T4[t_{15}]_0 \oplus k_{12}, c_{13} = T4[t_2]_1 \oplus k_{13},\\
&c_{14} = T4[t_5]_2 \oplus k_{14}, c_{15} = T4[t_6]_3 \oplus k_{15}
\end{aligned} \tag{1}$$

其中 c_i 代表第 i 个密文字节, $T4$ 代表需要查找的 T 表, k_i 代表第 i 个密钥字节, t_i 代表第 9 轮的输出。产生一字节密文需要的操作包括一次表查询(返回 4 字节数据), 一次行位移和一次轮密钥加。这些操作在 GPU 上实现为读取、存储以及一些逻辑指令。CUDA 编译器在程序编译时会改变指令执行顺序。

GPU SIMT架构的漏洞。在GPU上的SIMT架构中, 当一个warp执行了一条内存读取指令, 32个物理线程会产生32个内存访问请求, 这些访问请求会被发送至合并访问单元。访存请求会首先转换为独立cache line请求, 之后与MSHR(Miss Status Holding Register, 存储丢失状态访存请求的寄存器)中的独立cache line合并。由于内存操作串行化与写回串行化, 一个warp产生的32个内存地址请求的时间会与独立cache line请求数量正相关。

基于 GPU 对内存的操作模式, 可发现执行时间和独立 cache line 请求数量间呈现正相关关系。图 27 (a)中, 以 3 个不同步长展示了独立 cache line 请求数量与计算时间间的关系, 三个步长分别是 32 字节、64 字节和 128 字节。结果显示, kernel 计算时间与独立 cache line 数量呈正相关关系。步长 64 字节与 128 字节的图像高度重合, 说明英伟达 K40 的实际 cache line 大小为 64 字节。为了测试 AMD 显卡是否也具有同样的特点, 使用 OpenCL 实现了同样的计算, 在步长为 64 字节的情况下, 得到图 27(b)。在英伟达的显卡上, 实验显示执行时间与独立 cache line 数量间的皮尔逊相关系数[54]为 0.96, 这高度相关性使基于执行时间获得独立 cache line 访问数量成为可能。在 AMD 平台中获得相关系数为 0.93。这表明主流独立 GPU 厂商的产品中都存在着 kernel 运行时间与独立 cache line 访问数量间高度相关的漏洞。

AES 实现的漏洞。攻击者的目标是基于已知的密文信息和测量得到的时间信息恢复 16 字节的加密密钥。攻击者测量得到时间信息, 进而推测出独立 cache line 请求数量。假设攻击者发送了 32 块(32 个 16 字节的明文)信息到加密服务器, 服务器端开启一个包含 32 个物理线程的 warp 对接受的信息进行加密。之后攻击者获得了 32 块加密信息以及该 warp 执行加密的时间, 密文和时间信息被保存为: $\{c^1_{0-15}, c^2_{0-15}, \cdots, c^2_{0-15}, T\}$, T 为对加密过程计时。

表查询的索引决定了哪个 cache line 将被访问, 因此, 加密时间由 160 次表查询的索引决定。为了了解独立 cache line 访问数量与执行时间见的关系, 基于已知密钥进行 100 万次 32 字节数据块加密实验, 得到图 28 中平均计算时间与独立 cache line 请求数量间的关系。不同数量的样本产生的独立 cache line 数量呈现方差较小的高斯分布, 可以认为大量的样本产生稳定的独立 cache line 请求。经计算得到, AES GPU 实现的加密时间与 cache line 请求数量间的皮尔逊相关系数为 0.0596。

在实际的攻击中, 因为不知道密钥信息, 所以无法计算出独立 cache 访问数量, 而且密钥的大小为

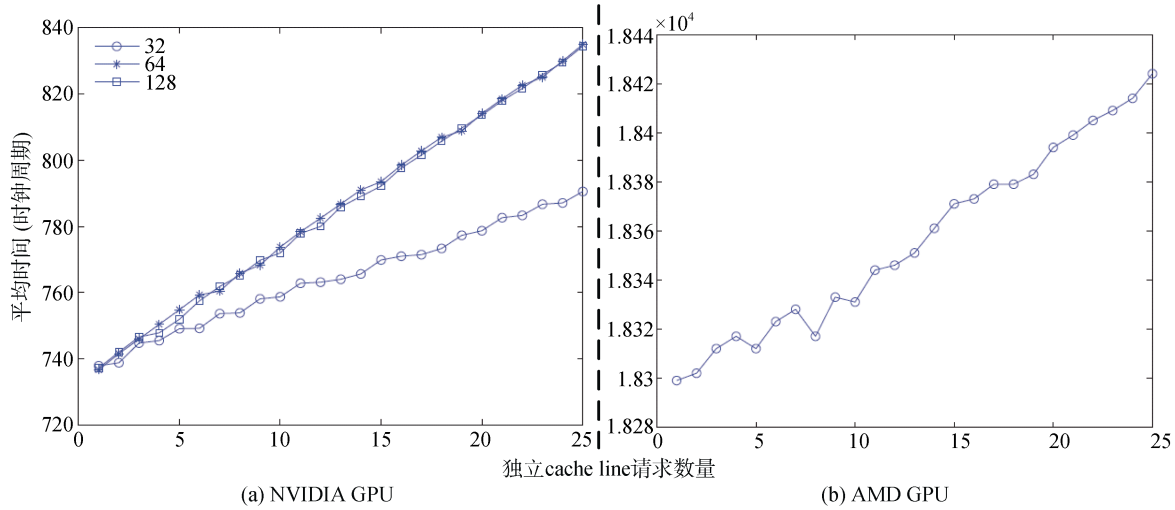


图 27　独立 cache line 请求数量与计算时间的关系[51]

Figure 27　Relationship between cache line requests and average computing time[51]

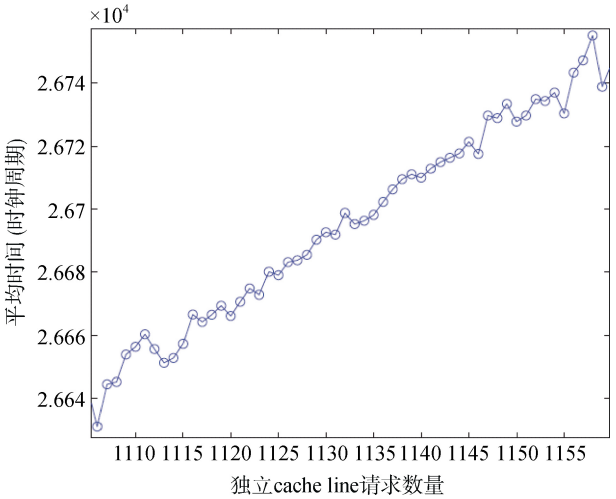


图 28　cache line 数量与计算时间关系(100 万样本)[51]

Figure 28　The average recorded time vs. the total number cache line requests (one million samples) [51]

128 位, 通过猜测整个密钥的内容进行验证也是不现实的(2^{128}=3.4*10^{38})。根据 AES 算法, 每轮加密操作中, 16 个字节的密钥按字节为单位分别对上轮输出的 16 字节的信息进行轮密钥加操作, 字节之间没有影响。因此在最后一轮 AES 加密中, T 表的查找索引可以通过一字节的密钥与对应的密文信息获得, 计算量减少为 16 次猜测 8 位数(0~256)中的正确值。根据公式 1, 可以将加密过程简写为: $c_j = T4[t_i] \oplus k_j$。

基于逆 S 盒, 如果已知最后一轮的轮询密钥 k_j, 我们可以算出最后一轮的输入 $t_i = T4^{-1}[c_j \oplus k_j]$, 在 GPU 的计算中, 32 个的信息块(单信息块 16 字节)由 32 个线程同时进行加密, 进而得到公式(2)。

$$\begin{aligned} t_i^1 &= T4^{-1}[c_j^1 \oplus k_j], \\ t_i^2 &= T4^{-1}[c_j^2 \oplus k_j], \\ &\vdots, \\ t_i^{32} &= T4^{-1}[c_j^{32} \oplus k_j] \end{aligned} \tag{2}$$

$T4$ 表访问 $t_i^1, t_i^2, \ldots, t_i^{32}$ 决定了独立 cache line 请求的数量。由于 $T4$ 表中每项为 4 字节, 且 cache line 的宽度为 64 字节, 所以每个 cache line 上存储了 16 个 $T4$ 表中的项。因此每个内存请求的 cache line 访问数量可以通过 $t_i^1, t_i^2, \ldots, t_i^{32}$ 右移 4 位决定, 得到 $<t_i^1>, <t_i^2>, \cdots, <t_i^{32}>$。最终的独立 cache line 访问数量是由$<t_i^1>, <t_i^2>, \cdots, <t_i^{32}>$中独立值的个数决定的。攻击对 k_i 进行猜测, 结果显示正确密钥值的皮尔逊相关系是是错误值的 36.9 倍。**可以认为最后一轮的 cache line 访问数量与 kernel 的执行时间成线性相关关系**。所以, 已知密文与加密时间, 基于独立 cache line 数量与加密时间的关系可获得 128 位密钥中正确的一个字节(一字节 8 位, 只需对比 256 个猜测值)。

该工作在干净的信道环境中, 得到 16 个正确的密钥字节的皮尔逊相关系数都远高于错误密钥字节的相关系数, 进而恢复了所有的 16 字节的密钥。对于有噪声的环境, 该工作将实验的加密样本数量增加以获取正确密钥值。AES 的侧信道攻击基于 GPU 的漏洞实现了攻击者对远程服务端密钥的恢复。这种攻击并不是发生在同驻虚拟实例之间, 当基于 GPU 实现的 AES 加密服务配置在提供 SAAS 服务的公有云端, 此种的攻击在理论上是可行的。

4.3　GPU 中的内存溢出攻击

CUDA 较早的版本中存在着内存溢出的问题,

为此 Pietro 等人[55]中对此做了验证。该工作针对使用 CUDA 4.2 驱动的 Fermi 架构的 C2050 和使用 CUDA 5.0 驱动的 Kepler 架构的 GT640 两块 GPU 进行内存溢出实验(CUDA Runtime 版本均为 4.2)。该工作发现, 若 GPU 上并发执行的两个 kernel, 利用 SM 上的寄存器、shared memory 和全局内存中的内存溢出机制, 恶意用户可获取受害者的信息。

寄存器溢出攻击利用英伟达 GPU 中寄存器溢出的机制来完成对受害者 GPU 任务在全局内存中数据的访问。原生的寄存器溢出机制允许进程申请比物理寄存器更多的寄存器, 如果一个变量没有物理寄存器空间存储, 那么它将会被写入到全局内存。通过使用 PTX[56]中间语言可以指定一个 kernel 在运行时所使用的寄存器数量。如果需要的寄存器数量超过了片上的寄存器数量, 编译器开始使用全局内存来存储变量。而片上的寄存器数量由 GPU 的型号决定。从攻击者的角度看, 寄存器溢出可以绕开 CUDA 运行时的访问原语获得全局内存访问权限。攻击者可以有效的通过利用寄存器溢出访问其他的 CUDA 程序的上下文信息; 或是访问受害者进程正在拥有的合法位置(受害者使用 cudaFree 指令之前)。

Shared memory 溢出攻击利用 GPU 任务切换时内存清理机制的漏洞获取受害者 GPU 任务在 shared memory 中的信息。具体做法为, 受害者 GPU 程序首先使用 shared memory, 在执行多次写操作过程中, 攻击者申请全部 shared memory, 进而获得受害者的信息。

全局内存中的溢出攻击利用 GPU 内存管理缺少内存归零操作的漏洞实现对受害者任务信息的读取。具体而言, 受害者结束工作后, 攻击者立即申请和受害者大小一致的全局内存, 最后以此实现攻击者对受害者信息的读取。该工作利用全局内存溢出机制的漏洞, 在 SSLShader 库中针对 AES 加密算法进行攻击, 实验结果显示, 此种方法可以恢复 AES 的加密密钥。

由此可见, 对于提供 GPU 服务的云平台运营商来说, 需要对其上运行的 SDK 进行测试, 防止内存溢出的漏洞发生, 因为 GPU 虚拟化需要运行在安全的 GPU 编程模型之上; 同时要对 GPU 任务过量使用寄存器、shared memory 等危险行为进行跟踪监测。

5 云平台中 GPU 虚拟化引入的安全威胁

5.1 已有的 GPU 上的安全威胁总结

如以上展示的多种攻击, 可见在云平台中 GPU 虚拟化的应用面临多种安全威胁。可以分为计算过程中产生的攻击和内存管理中产生的攻击。

计算过程中产生的攻击可以分为非资源争用型的攻击与资源争用型的攻击。非资源争用型的攻击包括: (1)通过对 GPU 硬件的执行时间进行度量, 利用 GPU 体系结构中访存与独立 cache line 命中数量之间的关系, 对 AES 的秘钥进行推测。在 GPU 上实现的诸多算法中, 存在大量的类似的访存行为, 如 OpenCV[57]库中 ADD、Remap 等算法的 GPU 实现, 若这些实现被用于处理敏感信息, 会产生类似的信息泄露风险。(2)利用恶意用户不断访问性能技术函数获得 GPU 资源利用量, 构建侧信道恢复受害者 kernel 中的有价值的信息。资源争用型攻击, 通过在 GPU 中构造多租户的多 kernel 的同驻, 利用 GPU 硬件资源可以构建间谍 kernel 与木马 kernel 间的隐秘信道。同时, 同驻的 kernel 间会因恶意用户的 kernel 的占用有限的共享资源, 使受害者的 kernel 无法获得 GPU 资源始终处于等待状态, 实现 DOS 攻击。

GPU 内存管理中产生的攻击是指同驻在 GPU 上的多个虚拟机或 GPU 程序间产生越界的内存访问, 在我们对 GPU 虚拟化技术的研究中, 我们发现多种方法未对内存的进行隔离, 这给内存数据篡改、内存溢出等攻击方式留下了空间。

传统云平台中的虚拟化安全问题可以分为: 虚拟化平台的安全问题; 虚拟实例的安全问题; 虚拟网络的安全问题。从已有的针对 GPU 的攻击, 我们可以看到 GPU 虚拟化在云平台的应用使已有的安全问题变的更加复杂, 问题(1)是在于 GPU 虚拟化的实现需要在已有的虚拟化平台中加入 GPU 管理模块, 实现 Hypervisor 层对 GPU 的管理, Hypervisor 层的漏洞会使 GPU 模块面临越权使用等问题; 问题(2)是 GPU 会给虚拟机的安全带来新漏洞, GPU 有其自身的计算与存储体系, 多个租户的计算负载在 GPU 上的并发执行可突破虚拟实例间的隔离, 建立隐秘信道或侧信道; 问题(3)是 GPU 上的恶意攻击会影响到原有平台的稳定性, 如 GPU 与 CPU 间的通讯可能占据大量的 PCIE 总线带宽及 DMA 的使用影响计算节点的网卡, 同时 GPU 遭遇 DOS 攻击后大量的计算带来的功耗开销会影响计算节点的正常运行。

问题(1)可以理解为 Hypervisor 层自身漏洞带来的威胁, 与传统安全问题类似(如 Hypervisor 层已有的漏洞会带来内存的越权读取等, 与 GPU 越权使用类似)。问题(2)与问题(3)需要研究 GPU 虚拟化在应用中带来的安全威胁的演进。

5.2 云平台引入 GPU 后安全威胁的演进

CPU 与 GPU 上的同驻威胁。虚拟机的同驻即云

服务提供商将不同租户的虚拟机放置到同一台物理服务器上。2009 年 Ristenpart 等[58]的工作中首次指出了云计算环境中虚拟机同驻安全的问题。2015 年, Varadarajan 等人[59]的工作中对谷歌、亚马逊和微软三家公有云的虚拟机放置算法进行安全评估, 结果显示攻击者完成同驻的可能性高达 90%。现有的同驻实现方法包括基于网路信息的虚拟机同驻方法、基于资源干扰的虚拟机同驻方法、基于隐秘信道的同驻方法。虚拟机同驻是云平台虚拟化环境中大多数攻击的前提, 使攻击者可以与受害者共用计算节点物理资源, 进而展开拒绝服务攻击、侧信道攻击、隐秘信道攻击和提权攻击等攻击。**云平台中引入 GPU 提供服务, 会将现有的同驻威胁问题变的更为复杂**。其原因在于: 一是由于 GPU 固定的 GPU 资源分配与任务调度方案, 且缺乏 GPU 上的同驻任务的安全检测手段, 使 GPU 上的同驻活动可绕开传统云平台中已有的同驻安全检测; 二是由于 GPU 的架构提供了更多的计算资源, 这方便攻击者更容易实现资源争用与性能监测; 三是在集群中部分节点部署 GPU 的环境中, GPU 任务更容易聚合在拥有 GPU 的节点上, 使同驻的实现变的更加简单。可见, 借由 GPU 上提供的物理资源与 GPU 引入后集群结构的变化, 云平台上现有的同驻威胁问题将变的更为复杂。值得注意的是, GPU 编程模型(OpenCL、CUDA 等)都是对 C 语言等的扩展, GPU 程序可和传统 CPU 程序以同样的方式使用 x86 平台的物理资源, 因此可在 GPU 上实现同驻, 然后使用传统的攻击方法。

GPU 引入后侧信道攻击的演进。CPU 中已有的侧信道攻击包括 Flush Reload[60]、Prime Probe、Flush Flush[61]等攻击, 其分别利用共享 LLC (Last Level Cache)争用, LLC 中 Cache 组争用、计时等方式实现。对比已有的 GPU 中的侧信道实现方法, 我们认为 **GPU 的引入会增加侧信道攻击的几率**。原因在于: 一是 GPU 上多层级的存储资源, 可以提供基于 cache、内存等资源争用发起侧信道攻击的平台; 二是 GPU 提供单精度、双精度、特殊计算、神经网络等计算单元, 相对于 CPU 上固定的且数量较少的计算核, 这些计算单元上的争用也会给侧信道攻击通过新的渠道; 三是由于 GPU 的体系结构特点和执行方式, 提供了新的基于时间或能耗进行侧信道攻击的渠道。

GPU 引入后隐秘信道攻击的演进。基于内存总线锁(memory bus lock) [59]等方法可在传统云平台中实现隐秘信道。结合之前展示的 GPU 上的隐秘信道攻击, 我们认为 **GPU 的引入会增加隐秘信道攻击的几率**。原因在于: 一是 GPU 上提供了新的硬件资源, 其被证实可以用于隐秘信道的构建; 二是 GPU 需要和 CPU 端进行数据传输, 不同的数据传输操作带来了不同的时间消耗, 这些特征可以被利用构建隐秘信道。

GPU 引入后 DOS 攻击的演进。如 Zhang 等人[62]工作中的 DOS 攻击为例, 攻击者绕过 Hypervisor 的管理, 使得受害者虚拟机无法正常使用内存资源, 破坏同驻虚拟机的可用性, 达到拒绝服务攻击目的。**GPU 的引入增加了 DOS 攻击的几率**。原因在于: 一是 GPU 本身可以作为 DOS 攻击的目标, 如之前的工作[14]中发现的循环机制不当, 可以导致 GPU 处于空转状态, 无法对外继续提供服务; 二是 GPU 的引入增加了 PCIe 总线上的数据传输负载, 使共用 GPU 的其他任务无法正常工作; 三是单节点上多块 GPU 的使用, 增加了功耗攻击[63]的可能性, 有报道称电源功率不足的情况会导致 GPU 无法启动[64], 进而导致节点无法提供有效的服务。

GPU 引入后内存溢出攻击的演进。云平台中一直受到内存溢出攻击的困扰[65], 原因在于程序编写的不规范或使用未经充分测试的开源项目等。GPU 上同样存在内存溢出的问题。**内存溢出攻击在 GPU 的引入后会变的更为复杂**, 因为 GPU 的内存溢出问题尚没有有效的检测和解决方案。

GPU 引入后提权攻击的演进。2016 年 Xiao 等[66]在的工作中利用行撞击(Row hammer)攻击, 通过修改内存中 SSH 服务器进程的代码来绕过密码验证, 在不知道目标密码的情况下登录一个同驻虚拟机的 OpenSSH 服务器。此外, Xen 等主流的 Hypervisor 会时常爆出可用于提权攻击的安全漏洞。**GPU 引入后, 会增加提权攻击的几率**。因为 GPU 虚拟化需要引入 GPU 管理组件, 与现有 Hypervisor 协作完成 GPU 服务的提供, 这些组件的引入增加了 Hypervisor 层被攻击的风险。

6 GPU 虚拟化的安全需求与安全技术

基于以上的研究可以看到云平台中 GPU 虚拟化的应用, 既面临着传统云平台中的安全问题, 也面临着与 GPU 引入后伴生的安全威胁演进。针对上述的 GPU 安全威胁及其虚拟化的需求(表 3 所示), 结合已有云计算安全的研究[67], 我们认为在 GPU 虚拟化的设计和实现中应满足以下的安全需求。

6.1 GPU 虚拟化的安全需求

GPU 是云平台中的新引入的硬件, 其基于 GPU 程序运行环境及 GPU 虚拟化管理模块向用户提供服务。为此, GPU 虚拟化的安全需求需要面对用户空间

层、GPU 服务层及 GPU 硬件设备层。

6.1.1 用户空间层的 GPU 虚拟化安全需求

对于用户空间层, 需要**静态分析租户的 GPU 程序**, 获取其控制流与访存行为, 在 GPU 程序运行中, 需要**实时监控** GPU 端与主机端的数据交互, 最终获取对 GPU 程序进行特征描述。

6.1.2 GPU 服务层的 GPU 虚拟化安全需求

对于 GPU 服务层, 需要做到:

有效的计算资源隔离, 使多租户间彼此的计算不受干扰, 确保租户被分配的 FU、Warp 调度器、寄存器、L1 cache 和 shared memory 等资源中不产生争用的情况, 使攻击者无法通过使用 GPU 中的有限资源的争用实现攻击。

实现有效的内存资源隔离, 攻击者可以尝试内存污染、内存溢出等方式, 获取同驻在 GPU 上其他虚拟实例的存储数据。因此 GPU 内存上也要实现有效的内存隔离, 需要支持多程序间相互隔离的多地址空间。同时要兼顾到全局内存中有限资源, 如 L2 cache、原子计算单元等的分配与管理。防止攻击者利用其构建攻击。

安全的 GPU 任务调度需要面向集群中的 GPU 任务调度与单 GPU 上的任务调度两个层次。对于集群中的 GPU 任务调度, 需要防止攻击者利用集群硬件相似性的关系实现同驻, 为了优化云平台的计算效率与功耗, 阿里云、亚马逊等云平台提供商现以在其平台上实现了业务敏感的任务分配, GPU 作为显著相同硬件需求会加剧 GPU 任务在计算节点上的同驻。对于单 GPU 上的任务调度, 要防止攻击者 GPU 调度机制实现同驻, 这需要通过使用安全的任务调度机制减少攻击者与受害者接触的可能性[68]。

GPU 资源使用情况的实时监控, 及时发现 cache、FU 等资源上租户任务间的资源干扰情况和整体 GPU 资源的使用情况, 基于此发现租户的异常行为和 GPU 的异常情况。

与 Hypervisor 层结合的 GPU 攻击阻断, 当攻击者利用 GPU 实现攻击后, 需要有效的阻断机制。这需要在 Hypervisor 层和 GPU 上的任务调度层两层次完成, 对于攻击已经产生较大危害的情况, 需要 Hypervisor 层介入, 通过调用 GPU 虚拟化管理模块实现 GPU 的任务中断与重启; 对于疑似攻击的行为, 通过 GPU 上的运行时模块将疑似攻击的任务迁移或挂起。同时也需要对 GPU 虚拟化模块与 Hypervisor 的接口进行防护, 防止对 Hypervisor 层的攻击。

隐藏 GPU 的伴生信息, GPU 的伴生信息分为两种, 一是攻击者通过计时等方式获得的信息, 二是 GPU 服务所提供的用于 GPU 程序优化的信息, 包括 GPU 资源的利用情况等。攻击者可以通过获取伴生信息实现攻击, 因此需要使用噪声加入、数据规整、数据脱敏等方法隐藏 GPU 的伴生信息。

6.1.3 GPU 设备层的安全需求

对于 GPU 设备层, 需要做到**能耗监控**, 防止攻击者基于能耗攻击, 使 GPU 产生的拒绝访问问题; 同时需要**对 GPU 虚拟化管理程序的响应**, 当 GPU 的资源被攻击耗尽时, 在获得重启等攻击阻断策略后能及时响应。

6.2 GPU 虚拟化的安全技术研究方向

针对 GPU 引入后带来的新的安全问题以及安全问题的演进, 为满足安全需求, 我们认为应该以如下的安全技术为研究方向展开工作。安全需求与安全技术的关系如图 29 中所示。

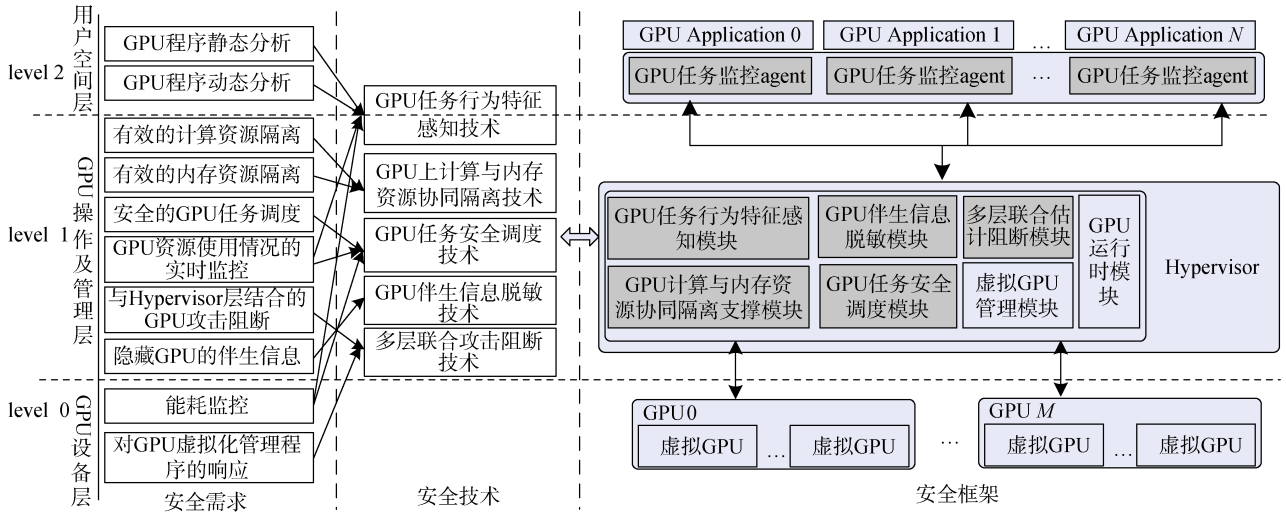


图 29 GPU 虚拟化安全需求、安全技术与安全框架

Figure 29 Security demand, security technology, and security architecture of GPU virtualization

GPU 上计算与内存资源协同隔离技术用以支撑虚拟实例间在 GPU 上的资源隔离。最有效的隔离机制为计算资源的空间复用技术与多地址内存空间的虚拟内存技术的协同使用。每个虚拟实例可以独占部分 SM, 确保虚拟实例间不会发生资源争用和地址空间干扰的问题, 英伟达 V100 GPU 因支持硬件辅助虚拟化, 故可实现协调隔离。对于不支持硬件辅助虚拟化的 GPU, 需要通过实现与虚拟 NDRange 相似的技术, 对 GPU 的片上资源进行分割。对于虚拟实例间的内存干扰问题, 可使用开源 GPU 驱动实现多地址空间; 若不能使用开源驱动, 则通过对虚拟实例的 GPU 任务进行行为跟踪与分析, 实时发现危险的内存访问操作, 并进行阻断。对于只能时间复用的 GPU, 通过 GPU 任务调度, 将确保安全的虚拟实例的 GPU 调度到一起执行, 减少攻击发生的可能性。

GPU 任务行为特征感知技术可以通过对 GPU 应用程序的运行时环境进行封装完成, 例如对 OpenCL 和 CUDA 的运行时进行 API 重定向, 在 GPU 应用使用到标准的运行时系统之前, 根据重定向的 API 调用获得其行为轨迹, 进行分析发现其危险行为, 例如, GPU 程序长时间占据片上大量共享内存, 或 GPU 程序长时间对性能指标函数进行访问, 都认为是疑似攻击的 GPU 程序。已有工作如 Burtscher 等人[69]对 GPU 任务的控制流与访存行为进行特征描述, 得到 GPU 程序的正常行为与异常行为, 该工作尝试基于行为的局部性提高 GPU 上的整体吞吐量。相似的方法可与 API 重定向方法结合应用于 GPU 任务的行为感知与攻击识别中。

GPU 任务安全调度技术。GPU 任务安全调度技术应分为单 GPU 上的任务安全调度与集群中的 GPU 任务安全调度技术。单 GPU 上的调度需要根据 GPU 业务负载的特点进行, 将安全的 GPU 任务可调度到相同的硬件资源上执行。若 GPU 可以使用空间复用技术实现资源分割, 需确保同一个 GPU 上的 GPU 任务分别位于不同的 SM 上。对于只能时间复用分割的 GPU, 需要减少有疑似安全威胁的 GPU 任务对其他 GPU 任务的影响。此外, 对于 GPU 任务在集群中的调度, 需要通过减少 GPU 任务间的接触面的方法尽可能减少同驻威胁发生的可能性。由于 GPU 任务的特点, 很难使用移动目标防御[70](Moving Target Defense, MTD)的方式实现计算任务动态迁移来减少同驻威胁, 所以 GPU 任务的调度要注意执行任务计算节点的初次分配, 以及 GPU 任务的运行时监控。

多层联合攻击阻断技术负责阻断已发现 GPU 攻击。阻断可在 Hyperisor 层与 CUDA 运行时(或 OpenCL)两层发生, 在攻击行为或疑似攻击行为未造成 GPU 服务整体暂停时, 可以运行时系统中对其进行阻断操作; 若攻击造成了 GPU 的宕机, 可以在 Hyperisor 层通过对虚拟 GPU 管理程序(类似 NVIDIA GRID Virtual GPU Manager)的调度实现攻击阻断, 解除恶意租户对 GPU 资源控制。

GPU 伴生信息脱敏技术负责管理伴生信息, 其根据租户的权限提供伴生信息, 确保虚拟实例只能获得其占用的 SM 上的资源利用数据。对于 GPU 程序优化中所必须的 GPU 整体资源利用情况, 需要对数据进行脱敏后提供。对于 SAAS 方式提供的 GPU 加速的应用(如 AES 攻击), 可以将其运算时间根据 SLA 的要求, 进行一定程度的规整后返回给用户, 以减少恶意用户对 GPU 资源利用情况的探知。

基于以上五种 GPU 虚拟化安全技术, 如图 29 所示我们给出 GPU 虚拟化安全框架作为参考。其中 GPU 任务监控代理模块负责获取程序的特征, 后基于提出的安全技术分别实现安全模块提供 GPU 程序的安全运行环境, 多个安全模块位于 hypervisor 层中以安全组件的形式存在, 并与 GPU 运行时模块和虚拟 GPU 管理模块交互。

7 总结

本文围绕——典型 GPU 虚拟化技术给云平台引入的潜在安全威胁和 GPU 虚拟化的安全需求及安全防护技术演进趋势——两个基本性问题, 系统性的论述 GPU 虚拟化及其安全技术。首先, 深入分析了“一虚多”、“多虚一”和硬件辅助等 GPU 虚拟化方法, 总结了云平台上隐秘信道、侧信道以及内存溢出等多种已有攻击, 并深入剖析了云平台中引入 GPU 后的安全问题的演进; 然后, 在总结了多层次的 GPU 虚拟化安全需求后, 提出了 GPU 计算与内存资源协同隔离、GPU 任务行为特征感知、GPU 任务安全调度、多层联合攻击阻断、GPU 伴生信息脱敏等五大安全技术研究方向; 最后, 给出了 GPU 虚拟化安全框架, 为云平台 GPU 虚拟化安全技术发展与应用提供有益的参考。

参考文献

[1] Xue M C, Ma J C, Li W T, et al. Scalable GPU Virtualization with Dynamic Sharing of Graphics Memory Space[J]. *IEEE Transactions on Parallel and Distributed Systems*, 2018, 29(8): 1823-1836.

[2] Mali GPU Architectures. https://developer.arm.com/architectures/media -architectures/gpu, 2019.

[3] K. Tian, Y. DONG, D. COWPERTHWAITE. A Full GPU Virtual-

ization Solution with Mediated Pass-Through[C]. *The 2014 USENIX Annual Technical Conference*, 2014: 121-132 .

[4] Ni H, Liu X. Multi-Core Optimization Technology of Unstructured Grid Based on Sunway TaihuLight[J]. *Computer Engineering*, 2019, 45(6): 45-51.
(倪鸿, 刘鑫. 基于神威 • 太湖之光的非结构网格众核优化技术[J]. *计算机工程*, 2019, 45(6): 45-51.)

[5] Amazon EC2 Instance Types. https://aws.amazon.com/ec2/instance-types/?nc1=h_ls, 2019.

[6] Instance families. Enterprise-level heterogeneous computing instance families. https://www.alibabacloud.com/help/, 2019.

[7] An Even Easier Introduction to CUDA. https://devblogs.nvidia.com/even-easier-introduction-cuda/, 2017.

[8] What's the Difference Between a CPU and a GPU?. https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/, 2019.

[9] Whitepaper NVIDIA Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[10] NVIDIA TESLA V100 GPU ARCHITECTURE. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017.

[11] Whitepaper Fermi. https://www.nvidia.com/content/PDF/Fermi _white _papers, 2009.

[12] Whitepaper Kepler GK110/210. https://www.nvidia.com/con-tent/dam/en-zz/Solutions/Data-Center/documents/NVIDIA-Kepler-GK110-GK210-Architecture-White- paper.pdf, 2014.

[13] Whitepaper NVIDIA GeForce GTX 1080. https://international.download.nvidia.cn/geforce-com/international/pdfs/GeForce_GTX_1080_Whitepaper_FINAL.pdf, 2016.

[14] Wu Z L, Zhang Y Q, Xu J L, et al. Research on Kmeans Algorithm Optimization Based on OpenCL[J]. *Journal of Frontiers of Computer Science & Technology*, 2014, 8(10): 1162-1176.
(吴再龙, 张云泉, 徐建良, 等. 基于 OpenCL 的 Kmeans 算法的优化研究[J]. *计算机科学与探索*, 2014, 8(10): 1162-1176.)

[15] Khronos OpenCL Working Group. The OpenCL specification, Version 1.2, 2012.

[16] CUDA C Programming Guide, https://docs.nvidia.com/cuda/cu-da-c-programming-guide/index.html, 2019.

[17] NVIDIA NVLINK, High-Speed GPU Interconnect. https://www.nvidia.com/en-us/design-visualization/nvlink-bridges/, 2019.

[18] Popek G J, Goldberg R P. Formal Requirements for Virtualizable Third Generation Architectures[J]. *Communications of the ACM*, 1974, 17(7): 412-421.

[19] Products by Category. https://www.vmware.com/products.html, 2019.

[20] GETTING STARTED. https://xenproject.org/developers/getting-started-devs/, 2019.

[21] Intel Virtualization Technology (Intel VT) . https://www.intel.com/content/www/us/en/virtualization/virtualization-technology/intel-virtualization-technology.html, 2019.

[22] AMD I/O Virtualization Technology (IOMMU) Specification. https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf, 2019.

[23] KVM-Features. https://www.linux-kvm.org/page/KVM_Features, 2019.

[24] Fu Y Z. Application of Network Virtualization Technology in Private Cloud Resource Pool[J]. *China Internet*, 2015(12): 53-58.
(付永振. 网络虚拟化技术在私有云资源池中的应用[J]. *互联网天地*, 2015(12): 53-58.)

[25] A. Rachata, V. Miller, J.Laudgraf, et al., Mask: Redesigning the gpu memory hierarchy to support multi-application concurrency. [J].*ACM SIGPLAN Notices*, 2018(53): 503-518.

[26] T. Bradley, Hyper-Q Example. https://developer.download.nvidia.cn/compute/DevZone/C/html_x64/6_Advanced/, 2013.

[27] Pai S, Thazhuthaveetil M J, Govindarajan R. Improving GPGPU Concurrency with Elastic Kernels[J]. *ACM SIGPLAN Notices*, 2013, 48(4): 407-418.

[28] J. A. Stratton, C. Rodrigues, et al., Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing[M]. Technical Report IMPACT-12-01, University of Illinois, at Urbana-Champaign, 2012.

[29] Eyerman S, Eeckhout L. System-Level Performance Metrics for Multiprogram Workloads[J]. *IEEE Micro*, 2008, 28(3): 42-53.

[30] Margiolas C, O'Boyle M F P. Portable and Transparent Software Managed Scheduling on Accelerators for Fair Resource Sharing[C]. *The 2016 International Symposium on Code Generation and Optimization*, 2016: 82-93.

[31] Yeh T T, Sabne A, Sakdhnagool P, et al. Pagoda: Fine-Grained GPU Resource Virtualization for Narrow Tasks[C]. *The 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017: 221-234.

[32] CUDA Toolkit Archive. https://developer.nvidia.com/cuda- toolkit-archive, 2019.

[33] Ausavarungnirun R, Landgraf J, Miller V, et al. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes[C]. *The 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017: 136-150.

[34] GRID VIRTUAL GPU User Guide. https://images.nvidia.com/content/grid/pdf/GRID-vGPU-User-Guide.pdf, 2016.

[35] Y. Suzuki, S. Kato, H. Yamada, et al. GPUvm: Why not virtualizing GPUs at the Hypervisor? [C]. *The 2014 USENIX Annual Technical Conference*, 2014: 109-120.

[36] Gottschlag M, Hillenbrand M, Kehne J, et al. LoGV: Low-Overhead GPGPU Virtualization[C]. *2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, 2013: 1721-1726.

[37] Qi Z W, Yao J G, Zhang C, et al. VGRIS: Virtualized GPU Resource Isolation and Scheduling in Cloud Gaming[J]. *ACM Transactions on Architecture and Code Optimization*, 2014, 11(2): 1-25.

[38] VMware vSphere 5.1 Documentation. https://pubs.vmware.com/vsphere-51/index.jsp?lang=en, 2019.

[39] Duato J, Peña A J, Silla F, et al. RCUDA: Reducing the Number of GPU-Based Accelerators in High Performance Clusters[C]. *2010 International Conference on High Performance Computing & Simulation*, 2010: 224-231.

[40] Margiolas C, O'Boyle M F P. PALMOS: A Transparent, Multi-Tasking Acceleration Layer for Parallel Heterogeneous Sys-

tems[C]. *The 29th ACM on International Conference on Supercomputing*, 2015: 307-318.

[41] Xiao S C, Balaji P, Zhu Q, et al. VOCL: An Optimized Environment for Transparent Virtualization of Graphics Processing Units[C]. *2012 Innovative Parallel Computing*, 2012: 1-12.

[42] MPI Documents. https://www.mpi-forum.org/docs/, 2019.

[43] MULTI-PROCESS SERVICE. https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf, 2019.

[44] NVIDIA Docker: GPU Server Application Deployment Made Easy. https://devblogs.nvidia.com/nvidia-docker-gpu-server-app-lication-deployment-made-easy/, 2016.

[45] Docker Platform. https://www.docker.com/products, 2019.

[46] Naghibijouybari H, Khasawneh K N, Abu-Ghazaleh N. Constructing and Characterizing Covert Channels on GPGPUs[C]. *The 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017: 354-366.

[47] Naghibijouybari H, Neupane A, Qian Z Y, et al. Rendered Insecure: GPU Side Channel Attacks are Practical[C]. *The 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018: 2139-2153.

[48] Liu F F, Yarom Y, Ge Q, et al. Last-Level Cache Side-Channel Attacks are Practical[C]. *2015 IEEE Symposium on Security and Privacy*, 2015: 605-622.

[49] nvidia profile manager. https://www.nvidia.com/en-us/geforce/forums/off-topic/25/259518/tool-nvidia-profile-manager/, 2019.

[50] Che S, Boyer M, Meng J Y, et al. Rodinia: A Benchmark Suite for Heterogeneous Computing[C]. *The 2009 IEEE International Symposium on Workload Characterization*, 2009: 44-54.

[51] Jiang Z H, Fei Y S, Kaeli D. A Complete Key Recovery Timing Attack on a GPU[C]. *The 2016 IEEE International Symposium on High Performance Computer Architecture*, 2016: 394-405.

[52] J. Bonneau and I. Mironov, Cache-collision timing attacks against AES[C]. *Cryptographic Hardware and Embedded Systems*, 2006: 201–215.

[53] OpenSSL 0.9.7. https://github.com/openssl/openssl/tree/OpenSSL-fips-0_9_7-stable, 2019.

[54] K. Pearson. Note on Regression and Inheritance in the Case of Two Parents[J]. *The Royal Society of London*, 1895, 58(347/348/349/350/351/352): 240-242.

[55] Pietro R D, Lombardi F, Villani A. CUDA Leaks[J]. *ACM Transactions on Embedded Computing Systems*, 2016, 15(1): 1-25.

[56] Inline PTX Assembly in CUDA, https://docs.nvidia.com/ cuda/inline-ptx-assembly/index.html, 2019.

[57] OpenCV 4.1.1, https://opencv.org/, 2019.

[58] Ristenpart T, Tromer E, Shacham H, et al. Hey, You, Get off of my Cloud: Exploring Information Leakage in Third-Party Compute Clouds[C]. *The 16th ACM conference on Computer and communications security*, 2009: 199-212.

[59] V. Varadarajan, Y. Zhang, T. Ristenpart, et al., A placement vulnerability study in multi-tenant public clouds[C]. *The 24th USENIX Security Symposium*, 2015: 913-928.

[60] Y. Yarom, K. Falkner, FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack[C]. *The 23rd USENIX Security Symposium,* 2014: 719-732.

[61] Gruss D, Maurice C, Wagner K, et al. Flush+Flush: A Fast and Stealthy Cache Attack[C]. *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016: 279-299.

[62] Zhang T W, Zhang Y Q, Lee R B. DoS Attacks on your Memory in Cloud[C]. *The 2017 ACM on Asia Conference on Computer and Communications Security*, 2017: 253-265.

[63] Xu Z, Wang H N, Xu Z C, et al. Power Attack: An Increasing Threat to Data Centers[C]. *The 2014 Network and Distributed System Security Symposium*, 2014(14).

[64] System halted when calling ‘model.fit’ #751. https://github.com/fastai/fastai/issues/751, 2019.

[65] Wei Z, Pierre G, Chi C H. CloudTPS: Scalable Transactions for Web Applications in the Cloud[J]. *IEEE Transactions on Services Computing*, 2012, 5(4): 525-539.

[66] Y. Xiao, X. Zhang. One Bit Flips, One Cloud Flops: Cross- VM Row Hammer Attacks and Privilege Escalation[C]. *Usenix Conference on Security Symposium*, 2016: 19-35.

[67] Feng D G, Zhang M, Zhang Y, et al. Study on Cloud Computing Security[J]. *Journal of Software*, 2011, 22(1): 71-83.
(冯登国, 张敏, 张妍, 等. 云计算安全研究[J]. *软件学报*, 2011, 22(1): 71-83.)

[68] Miao F B, Wang L M, Wu Z L. A VM Placement Based Approach to Proactively Mitigate Co-Resident Attacks in Cloud[C]. *2018 IEEE Symposium on Computers and Communications*, 2018: 285-291.

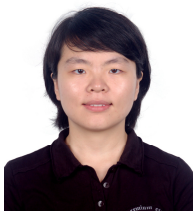
[69] Burtscher M, Nasre R, Pingali K. A Quantitative Study of Irregular Programs on GPUs[C]. *2012 IEEE International Symposium on Workload Characterization*, 2012: 141-151.

[70] W. Peng, F. Li, C. Huang, et al. A moving-target defense strategy for cloud-based services with heterogeneous and dynamic attack surfaces[C]. *2014 IEEE International Conference on Communications*, 2014: 804-809.

吴再龙 于 2014 年在中国海洋大学大学计算机软件与理论专业获得硕士学位。现任中国科学院信息工程研究所第五研究室助理研究员, 并于中国科学院大学网络空间安全专业攻读博士学位。研究领域为 GPU 虚拟化技术与安全、大数据安全、云安全。Email: wuzailong@iie.ac.cn

王利明 于 2007 年中国科学院软件所计算机科学与技术专业获得博士学位。现任中国科学院信息工程研究所第五研究室正高级工程师, 博士生导师。研究领域为云计算与网络安全、天基信息安全、移动通信系统安全、大数据安全分析、关键基础设施安全。Email: wangliming@iie.ac.cn

徐震 于 2005 年在中国科学院软件所计算机科学与技术专业获得博士学位。现任中国科学院信息工程研究所第五研究室主任, 正高级工程师, 博士生导师。研究领域为云计算安全、可信计算、移动互联网安全、系统安全。Email: xuzhen@iie.ac.cn



杨婧 于 2019 年在中国科学院信息工程研究所信息安全专业获得博士学位。现任中国科学院信息工程研究所第五研究室助理研究员。研究领域为安全数据智能分析、网络与系统安全、分布式计算与虚拟化技术。Email: yangjing@iie.ac.cn

李宏佳 于 2011 年在北京邮电大学电路与系统专业获得博士学位。现任中国科学院信息工程研究所第五研究室副研究员。研究领域为 LTE 与 5G 移动通信系统安全、异构蜂窝网络组网与优化。研究兴趣包括: 安全虚拟化、HetNet 与 MEC 等面向服务安全方法、优化理论与方法。Email: lihongjia@iie.ac.cn