中图分类号：TP311　　论文编号：1028716 15-S079
学科分类号：081200

硕士学位论文

基于多 GPGPU 并行计算的虚拟化技术研究

研究生姓名　　张玉洁
学科、专业　　计算机科学与技术
研 究 方 向　　高性能计算
指 导 教 师　　袁家斌　教授

南京航空航天大学

研究生院 计算机科学与技术学院

二 O 一五年一月

Nanjing University of Aeronautics and Astronautics
The Graduate School
College of Computer Science and Technology

The Research of Virtualization of Parallel Computing Based on Multi-GPGPU

A Thesis in

Master of Computer Science and Technology

by
Zhang Yujie

Advised by

Prof. Yuan Jiabin

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Engineering

January, 2015

承诺书

本人声明所呈交的硕士学位论文是本人在导师指导下进行的研究工作及取得的研究成果。除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南京航空航天大学或其他教育机构的学位或证书而使用过的材料。

本人授权南京航空航天大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密的学位论文在解密后适用本承诺书）

作者签名：_____

日　　期：_____

摘 要

跟 CPU 相比，GPU 在计算能力、能耗上具有显著的优势，被广泛应用于高性能计算领域。虚拟化是云计算的主要支撑技术之一，屏蔽硬件基础设施使多台虚拟机透明地共享集群中的 GPU 设备，从而降低配置成本，提高资源利用率。目前，GPU 通用计算的虚拟化技术尚处于研究阶段，虚拟化环境下资源共享方案普遍缺乏对 GPU 的有效支持。

本文以通用计算框架 CUDA 为研究对象，设计一种基于多 GPGPU 并行计算的虚拟化的方案，实现一种可动态调度、支持多任务并发的 GPU 虚拟化解决方案。具体工作包括以下几个方面：

第一，采用动态库拦截的方法将 GPU 引入虚拟机，设计了基于多 GPU 计算资源特征的动态分配与管理的架构，该架构分为虚拟化用户层、虚拟化资源管理层和虚拟化资源服务层，解决了 GPU 通用计算在虚拟化环境下的适应问题，实现 GPU 资源在多个计算节点间的共享。

第二，针对大规模计算任务场景，提出了在虚拟化环境下多 GPU 并行计算的实现方案，使用多线程或流处理的方式实现多 GPU 并行计算，分析了 GPU 多层次存储结构、传输、通信等方面内容，通过实验分别对数据松耦合交互模式（如蒙特卡罗方法）和紧耦合交互模式（如 QFT 算法）实现多 GPU 并行计算。

第三，提出了一种基于动态负载量多负载状态的 GPU 负载均衡算法 DMLS-GPU（Dynamic and Multi-Load Status algorithm for GPU），通过将负载与 GPU 设备的硬件能力和任务本身的特性相结合，解决了虚拟化方案中动态评估 GPU 设备计算能力的问题。实验分析表明，在虚拟化环境下可实现多个 CUDA 程序并发地使用一块或者多块 GPU 设备，并验证了本文的虚拟化方案具有良好的可扩展性和高效性。

本文工作针对 GPU 通用计算虚拟化过程中面临的挑战和制约，研究虚拟化环境下的多任务 GPU 资源共享和多 GPU 并行计算，以进一步拓展其应用空间。

关键词：虚拟化，GPU 通用计算，CUDA，并行计算，资源共享

ABSTRACT

GPU has significant advantages compared to CPU on computer power and energy consumption to be widely used in high performance computing. Virtualization technology is one of the main technologies in cloud computing, which can make multiple virtual machines share the GPU devices in the cluster transparently by shielding the hardware infrastructure. Therefore, it can reduce the resources cost and improve the resource utilization. At present, virtualization based General purpose GPU technology is still in the research stage. Resource sharing solutions are generally lacked of effective support for GPU in the virtual environment.

The general computing framework CUDA is taken as the research object in this paper. We design a solution of virtualization of parallel computing based on multi-GPGPU. In this way, the solution can schedule the tasks dynamically and process the multi tasks concurrently. The specific work includes the following aspects:

First, we introduce the GPU into the virtual machine by intercepting the library dynamically and design the dynamic allocation and management architecture based on multi GPU computing resources. The architecture includes virtual user layer, virtual resource management layer and virtual computing resource service layer. The general calculation adaptation problem in virtual environment is resolved and it can share the GPU resources among multiple computing nodes.

Second, the solution of multi-GPU parallel computing under virtualization is proposed for the large-scale computing tasks. The multi-threads and flow mode are supported to implement the multi GPU cooperative computing. The multi-level storage structure for GPU, transmission, communication and other aspects are analyzed to accelerate the program. We implement the multi GPU cooperative computing for the loosely coupled interaction pattern of data (such as Monte Carlo method) and tight coupled interaction model (such as QFT algorithm).

Third, the Dynamic and Multi-Load Status algorithm for GPU(DMLS-GPU)has been proposed for estimating GPU computing capacity，which combines the load value with the GPU equipment hardware capacity and the task characteristic. And it resolves the dynamic evaluation of GPU computing capacity in the virtualization solution. The experiment shows that multiple CUDA programs can be performed concurrently on one or more GPU devices under virtualization. And we verify the good extension ability and the high efficiency of the virtualization solution in this paper.

This paper is aimed at the challenges and constraints in GPU general purpose computing under virtualization. Multi-tasks share the GPU resources and multi-GPU parallel computing is studied

under virtualization in order to further expanding their application space.

Key words: Virtualization, General-Purpose computing on GPU, CUDA, Parallel computing, Resource Sharing

目 录

图表清单

注释表

i	GPU 上第 i 个任务	N	当前 GPU 上的任务数
$Scale_i$	GPU 上第 i 个任务的规模大小	LGg	单块 GPU 设备综合负载评价值
CN_i	任务每秒计算的浮点次数（Flops）	PNg	GPU 的计算核心的数量
PFg	处理器内核频率（GHz）	GMg	GPU 的全局内存（G）
BWp	PCI-E 的总线带宽（GBps）	NW	每个节点的所有网络接口带宽
T_1	数据通过 PCI-E 的接口拷贝的时间(s)	T_2	数据通过网络传输的时间（s）
α_1	任务计算能力强度的影响因子	α_2	全局存储器使用率的影响因子
α_3	显存带宽使用率的影响因子	α_4	网络通信带宽使用率的影响因子
U_{QFT}	么正操作符	e	自然对数的底数
a_{ij}	矩阵 A 第 i 行第 j 列的元素	CI	一致性指标

缩略词

缩略词	英文全称
GPU	Graphic Processing Unit
GPGPU	General-purpose computing on GPU
CUDA	Compute Unified Device Architecture
VM	Virtual Machine
VMM	Virtual Machine Monitor
CKX	CKX Concurrent Kernel Execution
RPC	Remote Procedure Call
SPMD	Single Program Multiple Data
HLSL	High Level Shader Language
VDA	Virtual Desktop Agent
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
Pthread	POSIX threads
MPI	Message Passing Interface
DMA	Direct Memory Access
UVA	Unified Virtual Addressing
VMCI	Virtual Machine Communication Interface
QFT	Quantum FourierTransformation
SSD	Solid State Disk
NIC	Network Interface Card
DFT	Discrete Fourier Transform

第一章　绪论

虚拟化技术对物理资源进行逻辑抽象和统一表示，可复用硬件平台、提高物理资源的利用率、透明使用硬件平台、降低资源管理的复杂度[1]，无论在商业界还是学术界都是研究重点。GPU 的高带宽、低成本、高速率的运算被越来越多使用于高性能计算中，以 CUDA（Compute Unified Device Architecture）[2]为代表的并行计算架构将 GPU 用于通用计算，但没有针对多 GPU 的 API 标准。由于 GPU 设备的特殊性，对 GPU 的虚拟化研究尚处于起步阶段，且在多节点多任务的环境下，如何有效使用 GPU 资源成为一大难题。基于多 GPGPU 并行计算的虚拟化技术的研究对基于 CPU+GPU 异构计算模型的云计算、网格计算具有重大意义[3]。

1.1 研究背景及意义

虚拟化（Virtualization）是一种实现对计算机资源进行抽象模拟的技术，在计算机操作系统、编程语言等方面获得广泛的应用，在计算机技术的发展过程中占据重要的位置[4]。从 90 年代开始，随着计算机研究的逐渐深入（PC 的普及和 Java 虚拟机的问世）以及市场需求的变化（VMWare Workstation 等多款虚拟机的推出），虚拟化在服务器、网络以及数据存储等方面的优势逐渐得到重视。特别是随着近年来云计算技术的提出，虚拟化作为其中重要的一环，又成为学术界和企业界广泛关心的热门话题。

虚拟化技术能够将原先在真实物理环境下运行的计算机系统移植到虚拟的环境中运行，并可以在软硬件资源的不同层次间创建一个虚拟化层，并成为解除硬件与软件两层间耦合关系的中间层。虚拟化对现有的硬件资源（如 CPU、内存、I/O 设备等）抽象模拟，形成一部分或一整套的虚拟硬件资源。这些虚拟硬件资源统称为虚拟机。从软件层的角度来讲，虚拟机与真实的机器没有区别，即对于用户来说虚拟机的实现与运行是透明的。虚拟化技术按照资源类型的不同分为进程级虚拟化和系统级虚拟化。本文研究的是系统级虚拟化，通过对软硬件的整合或划分，实现将硬件资源合并或者分割为一个或多个环境。虚拟机监视器（Virtual Machine Monitor，VMM）[5]处于虚拟机与物理硬件之间的中间层，运行在特权模式，主要用于隔离及管理上层运行的虚拟机。

在 2004 年虚拟化技术被华尔街日报评为全球技术创新奖软件类的第二名，在国内，虚拟化计算技术的研究被列为了国家重大研究计划[6]。系统级虚拟化主要是从逻辑上对资源进行重新划分，由于计算机系统中的 I/O 资源有限，为了能够满足多个虚拟机的需求，必须通过虚拟化方式来共享有限的外设资源。图 1.1 为系统级虚拟化结构图，显示了 GPU 虚拟化所处的地位。至今，CPU 虚拟化方案已经日渐成熟，一个 CPU 可以模拟成多个 CPU 并行，允许一个物理节

点上同时运行着多个不同的操作系统，并且应用程序之间相互独立，互不影响。但是，针对 GPU 虚拟化的方案还处于起步阶段，现有的 GPU 虚拟化大多针对图形处理，在通用计算方面涉及的甚少。GPU 的浮点计算能力大大超过 CPU，并行计算能力不断提升，因而现在很多高性能计算集群都采用 GPU。目前虽然虚拟化越来越普及，但是完善的适用于 GPU 通用计算的虚拟化方案并不存在，这方面的空白严重地影响了虚拟化在高性能计算集群上的大规模部署和应用。

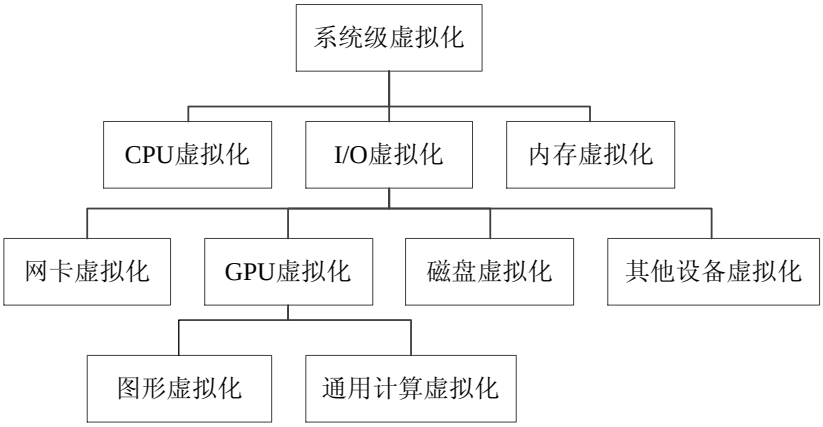

图 1.1 系统级虚拟化结构图

自虚拟化技术诞生以来，一个重大挑战就是对 I/O 资源的虚拟化，如何在虚拟机中共享 I/O 设备并获得良好的性能是 I/O 虚拟化的重要问题。I/O 虚拟化主要是对 I/O 设备的虚拟抽象，现有的 I/O 虚拟化的解决方案有设备全虚拟化模型、驱动分离模型以及直接分配模型。设备全虚拟化模型采用软件模拟的方式，通过软件模拟实现与物理设备一致的接口，驱动程序在客户操作系统中无需改动即可直接驱动虚拟设备。驱动分离模型最典型的是 Xen 前后端驱动模型，客户操作系统中的驱动程序称为前端（Front-End），由 VMM 提供的简化驱动程序称为后端（Back-End），系统通过前后端模拟 I/O 设备。从客户操作系统中拦截对 I/O 设备的访问，通过特殊的通信机制转发到后端的驱动程序，后端驱动处理完请求后再发送给前端[7]。直接分配模型（也称透传模型），是指将物理设备直接分配给某个虚拟机，由虚拟机直接访问 I/O 设备（不需要通过 VMM）。GPU 是 I/O 设备中比较特殊的一种设备，它主要用于图形处理以及通用计算两部分。除少数 2D 显示标准外，显卡硬件电气接口没有统一的设计规范和国际标准，几大生产厂商之间的接口并不兼容。此外，出于商业目的，所有厂商设计的细节不对外公开，因而在操作系统层开发以适应不同的硬件驱动程序堆栈是不现实的。基于以上原因，对 GPU 虚拟化时存在一定的困难，学术界将虚拟化的思路放到了软件栈，利用驱动分离模型的方式实现 I/O 虚拟化。

随着科学研究日益精细化，科学计算对大规模并行计算提出了迫切需求。图形处理器（GPU，Graphic Processing Unit）的出现使计算机的运算能力不断提升，吞吐率不断提高。2006 年，

NVIDIA 公司统一架构 CUDA 的诞生，GPU 由单一的图形处理向通用计算领域发展，出现了通用计算图形处理器（GPGPU，General Purposecomputing on Graphics Processing Unit）的迅猛发展，大大简化了 GPU 的编程环境，使得更多的应用可以运用 GPU 强大的计算能力。

无论从计算能力还是存储器带宽上，GPU 相对 CPU 都有明显优势，在功耗和成本上的代价也较小。在 GPU 中将大量的晶体管用作 ALU 计算单元，而 CPU 为优化串行代码将晶体管作为复杂的控制单元和缓存，GPU 注重快速低延迟地实现某个操作。二十一世纪以来，大规模集群系统中很多都采用 CPU+GPU 异构计算模式，在 2014 年 11 月的全球超级计算机 500 强排名中，位于中国国防科技大学的天河二号位居榜首；美国能源部在橡树岭国家实验室的一台 Cray XK7 系统的超级计算机泰坦，排名第二。天河二号和泰坦均采用了基于“CPU+GPU”异构的混合计算模型。由此，我们可以看出 CPU+GPU 异构的计算模型具有向大数据计算领域发展的趋势。

本文认为，结合虚拟化技术和 GPU 通用计算技术，可以有效利用 GPU 资源，提升计算效率。在虚拟化环境下进行高性能计算，通过将 GPU 融入到虚拟化计算基础架构中，利用系统虚拟化技术开展高性能计算的应用。在多 GPU 并行计算方面，目前尚未有标准的 API，在虚拟化环境下使用多 GPU 设备对应用程序加速的研究尚处于空白阶段。同时，设计面向多任务的 GPGPU 并行计算，提高 GPU 设备利用率，为虚拟化技术和 GPU 在高性能计算集群乃至云平台的应用提供有力支持。

zkq 20151222

1.2 国内外研究现状

在虚拟化方面，实现 CPU 虚拟化比 GPU 虚拟化要容易。因为 CPU 利用内置的分时机制，可以很容易地中止当前进程并切换上下文到另一个进程。与此相反，GPU 虚拟化比较困难。因为 GPU 在一个时间节点内只能运行单个任务，并且不能切换到其他进程上。此外，因商业考虑，GPU 制造商不提供驱动源代码，因此 GPU 不能由其他系统方案控制，包括虚拟化管理程序。

近年来，NVIDIA 公司在 GPU 并行计算方面注重与云服务器的嫁接，在美国已经有多家云服务商提供了 GPU 并行的服务。2009 年 10 月，NVIDIA 与 Mental images 联合推出一款基于云计算的高端服务器——RealityServer。2010 年 11 月，NVIDIA 与 Amazon 联合推出亚马逊的集群 GPU 实例[8]。2012 年 5 月，NVIDIA 公司推出使用 GPU 加速云计算的技术。2013 年，NVIDIA 给出了在云计算领域最新的产品服务器平台——NVIDIA GRID，实现 GPU 虚拟化，用云端渲染再次改写历史。

针对 GPU 运算在底层的封闭性的问题，学术界在较高软件栈层次上设计出实现 GPU 虚拟化的方案。在图形显示方面，几乎所有使用 GPU 的应用程序都使用了 OpenGL[9]和 Direct3D[10]这两种图形处理接口。其中 OpenGL 是唯一能够支持跨平台图形 API。为了实现在虚拟机中使

用硬件加速渲染能力，VMGL[11]是首个针对 OpenGL 在虚拟化平台下实现的接口。通过在虚拟机中部署伪库来取代 OpenGL 库，通过 GL 网络传输重定向到远端服务器或者虚拟机管理器中，虚拟机管理器借助 GPU 驱动以及 OpenGL 库，可以访问到实际的 GPU，从而完成 OpenGL 调用。由此可见，VMGL 不用修改源代码也不需要二进制改写即可实现虚拟化环境下的图形处理，保证了完整透明的服务。

GPU 的计算能力不断增强，信息界从 2003 年开始，将 GPU 的研究目光延伸至通用计算领域，利用异构的计算资源进行大规模的并行计算，将 GPU 用于图形渲染以外的领域的计算称为 GPGPU[12]。2007 年，NVIDIA 公司第一个推出可以进行通用计算的编程框架——CUDA。2009 年以后涌现出一批基于 CUDA 的虚拟机应用方案。最早提出在 GPU 通用计算方面的虚拟化解决方案是在 2009 年，中国湖南大学 vCUDA[13]以及佐治亚理工学院的 GViM [14]。随后在国际上涌现出多个相关的衍生项目，如 gVirtus[15]、rCUDA[16]，DS-CUDA[17]及 GridCUDA[18]等。所有这些框架通常都是基于客户端/服务器的分布式体系结构，即虚拟机作为前端拦截 CUDA API 调用，特权域或者物理机作为后端执行程序。不同的虚拟化架构有不同的特征。

vCUDA[13]（virtual CUDA）采用客户/服务器模式，包含三大部分：vCUDA 库、虚拟 GPU 以及 vCUDA 服务器。在应用层通过 vCUDA 库拦截 CUDA 的调用，并在虚拟机中建立维护 GPU 状态的逻辑映像——vGPU（virtual GPU），通过 RPC（Remote Procedure Call，远程过程调用）系统将请求发送到 vCUDA 服务器端。最初 vCUDA 在客户端及服务端的通信采用 XML-RPC 的通信方式，性能损失高达 1600%。主要性能损失在于传统的 RPC 在虚拟化环境中的效率极低，所以后来 vCUDA 开发了 VMRPC，性能开销仅有原来的 21%。vCUDA 是一个比较通用的 GPU 虚拟化方案，它不依赖于特定的虚拟化平台，但是并不对外直接公开。

GViM[14]（GPU-accelerated Virtual Machines）针对特定的虚拟化平台 Xen，它的一个重要的特性是其执行 Kernel 的性能与虚拟机直接访问加速器的速度相仿。通常，在虚拟机中编写 CUDA 程序，当使用 malloc()函数时，数据存放在用户虚拟地址空间。使用 GPU 设备时要经过两次拷贝，首先将数据拷贝到特权域的内存，其次将其传输到 GPU 的显存上。GViM 的研究重点实现了数据传输方面的优化，通过 Xen 中的特定通信方式类似 XenLoop[19]实现了零拷贝（zero copy），即通过 bypass 技术直接访问到 GPU，效率大大提升。但是，GViM 要求在客户端及服务端都插入一个定制的虚拟化模块，对用户不是完全透明。此外，GViM 仅支持有限的 CUDA API，支持 CUDA1.1 版本，因而缺乏通用性。

gVirtus[15]（GPU Virtualization Service）是 2010 年意大利开发的项目，思路上借鉴了 vCUDA 和 GViM，同样采用客户端/服务端的架构。gVirtuS 支持 CUDA3.2，并且独立于管理器，因而保证平台无关性，适用于 KVM、Xen、VMware 等虚拟化平台，尤其对 KVM 虚拟机进行了优化。为加速虚拟机和特权域之间的通信，gVirtuS 加载了 VMSocket 模块。

rCUDA[16]（remote CUDA）是西班牙的研究小组在 2010 年开发的项目，此后随着 CUDA 版本的更新而更新。最新的 rCUDA 能够支持 CUDA 6.0，并且支持用于 Ethernet 和 InfiniBand 的特殊通信机制。它设计的初衷是远程实现 CUDA 的框架，使得不拥有 GPU 的节点也可以使用 GPU 设备。rCUDA 实现了 GPU 的复用，通过在服务端开不同的进程来执行每个远程调用，每个进程创建一个新的 GPU 上下文。rCUDA 在服务端与客户端采用的是 Socket 通信，比传统的 RPC 通信效率提升很多。

GridCuda[18]是中国台湾于 2011 年开发的项目，旨在用于网格计算中支持 CUDA 程序开发。用户编写的 CUDA 程序，将被 GridCuda 拦截并重定向到远端的 GPU 上执行，这一点同 rCUDA 类似。针对跨网域资料传递所衍生的时间成本问题，GridCuda 添加压缩演算法来最小化资料传输时间对效能的影响。此外，GridCuda 支持多线程编程，从而实现 CUDA 程序可以使用分布的多个 GPU 并行执行，但是支持的版本过低。

DS-CUDA [17]（Distributed-Shared CUDA）是 2012 年日本开发的项目，DS-CUDA 支持 CUDA 4.1，并有针对支持 InfiniBand 的特殊的通信库。DS-CUDA 也实现了访问远程 GPU 的功能，这个与 rCUDA 类似。不同的是，rCUDA 主要用于降低集群里 GPU 的数目，从而减少集群构建以及维护的开销，而 DS-CUDA 主要目的是以一种简单可靠的方式尽可能地利用集群中的 GPU。此外，DS-CUDA 含有容错机制，通过使用多个 GPU 执行冗余计算。

随着这些虚拟化框架方案的提出，一些学者针对这些虚拟化方案进行一些科学研究。文献[20]提出了一种能够使得应用程序在虚拟机中透明地共享一个或者多个 GPU 的框架。通过扩展开源的 GPU 虚拟化方案可以有效地共享 GPU，并且提出了 Kernel Consolidation（核聚合）的概念。此方案的构想源于新版的 CUDA 在计算能力 2.0 以上的设备上支持 CKX（Concurrent Kernel Execution，并发内核执行）的特性。当将要执行的任务的数目大于可用的 GPU 的数量时，使用此方案可以动态整合多个内核到单内核，并且合并时不需要在 GPU 内核或用户应用程序任何的源代码级别变化。在该文中，使用了八个 Kernel 实例证明核聚合方法的有效性，但是这并不是一个通用的核聚合解决方案。在实际应用中，可能会有成千上万个核，并非所有的核都有核聚合的价值。此外，核聚合有一定的开销，其并行效率有待进一步研究，并且在多 GPU 的管理中也未涉及，所以可行性不是很高。

目前高性能计算机中的协处理器如 GPU 一般采用 SPMD（Single Program Multiple Data，单程序多数据）模型，要求微处理器和协处理器之间平衡计算资源，确保整个系统的利用率。但是在使用 GPU 作为协处理器的高性能计算系统中，不对称 GPU 数量的分布会导致整体系统资源利用不足情况的发生。针对这种问题，文献[21]在 vCUDA、gVirtuS 和 GViM 三个项目的基础上，提出了一个 GPU 资源虚拟化方法以实现 GPU 资源的均衡调度。从 SPMD 的特点出发，该文认为在 CPU+GPU 异构计算模式的体系架构下，在物理层面满足 CPU 与 GPU 的一一匹配

并不现实。它在设计的 GPU 虚拟化方案中，通过对同一物理 GPU 资源进行多路复用，达到在逻辑上对 SPMD 执行平台进行构建的目的。

文献[22]结合 GPU 程序计算的特点，提出了一种结合了集中式调度与分层式调度的 GPU 集群调度管理的设计方案。该设计方案目的是在多 GPU 组成的 GPU 集群上并行工作，充分利用 GPU 的资源，并且保证系统具有良好的扩展性，保证在 GPU 数量增长的情况下，系统总体计算能力也呈线性增长。实验通过某种哈希算法进行验证，但只进行了 GPU 集群的计算能力与众高性能 CPU 的比较，并没有涉及到随着集群 GPU 数量的增长其系统计算能力也随之增加的测试，所以此方案的可行性有待研究。

文献[23]融合云计算中的 MapReduce 编程技术与 CPU+GPU 异构混合编程模式，充分利用 GPU 强劲的计算能力以及 CPU 的逻辑处理能力，结合云计算虚拟化技术，使用分布式文件系统 GFS 和 HDFS 对异构数据信息进行存储，实现云系统对资源的高效利用，从整体上提高云系统的海量信息处理能力。

综上，由于 GPU 优异的性能功耗比以及通用计算和虚拟化技术在各个领域的广泛使用，使得 GPU 通用计算虚拟化成为了近几年的一个研究热点。但是上述虚拟化方案将重点放在如何在虚拟化环境下使用 GPU 设备，并没有针对 GPU 自身的结构特点设计出适合在 GPU 虚拟化环境下的面向多任务的调度算法，且对单个计算任务全部由单个 GPU 来完成。为更好地支持多任务在虚拟化环境下高性能计算的并行处理，本文构建了一种基于多 GPU 计算资源特征的动态分配与管理的架构，实现一种在虚拟环境下面向多任务的动态调度 GPU 资源的虚拟化方案，解决通用计算在虚拟化环境下的适应问题。同时，在虚拟化环境下，针对大规模的计算程序，设计一种单任务多 GPU 并行计算的加速方法，充分利用 GPU 的计算能力。

1.3 主要研究内容

为充分利用虚拟化技术和 GPU 计算能力，拓宽 GPU 虚拟化的适用范围，本文从现有的 GPU 虚拟化发展过程中的新需求出发，以通用计算框架 CUDA 为研究对象，设计一种基于多 GPGPU 并行计算的虚拟化方案。在虚拟化环境下设计多 GPU 并行计算的方法，实现将大规模程序划分到多块 GPU 设备上并行计算。同时结合此虚拟化方案，提出 GPU 虚拟资源池的概念，针对此虚拟化方案中动态评估 GPU 设备计算能力的问题，提出了 GPU 负载均衡的算法，使得在虚拟化环境下实现多任务共享 GPU 资源，并通过实验分析系统的有效性和高效性。

本文的贡献主要体现在：

（1）设计了一种在虚拟环境下面向多任务的动态调度 GPU 资源的虚拟化方案。以 CUDA 为研究对象，设计了基于多 GPU 计算资源特征的动态分配与管理的架构，解决通用计算在虚拟化环境下的适应问题，GPU 虚拟化使 GPU 可以在多计算节点间共享，该架构分为虚拟化用户

层，虚拟化资源管理层和虚拟化资源服务层。

（2）提出了虚拟化环境下多 GPU 并行计算的实现方案。在虚拟化环境下，使用多线程或流处理的方式实现多 GPU 并行计算。为加速 CUDA 程序的运行，分析并验证 GPU 多层次存储结构并在异构平台中优化传输带宽。由于虚拟化环境下会存在很大的开销，本文探讨了在虚拟化环境下多 GPU 并行计算的通信策略，在 Xen、VMware 系列的虚拟化平台下对现有的虚拟机域间通信方式进行对比，得出在特定虚拟化环境下虚拟机域间最高效的通信方式。并分析了在多 GPU 并行计算时，多 GPU 的通信策略问题。

（3）提出虚拟资源池的概念，研究基于 GPU 的虚拟资源池映射，分析虚拟资源的映射关系。针对动态评估 GPU 设备的计算能力问题，提出 DMLS-GPU 算法，将负载与 GPU 设备的硬件能力和任务本身的特性相结合，计算出的综合负载值用于衡量 GPU 设备的负载情况。通过实验分析，在虚拟化环境下可实现多个 CUDA 程序并发地使用一块或者多块 GPU 设备，并验证了本文的虚拟化方案具有良好的可扩展性和高效性。

1.4 论文组织结构

本文主要围绕基于多 GPGPU 并行计算的虚拟化展开研究，本文共有六章，具体的内容安排如下：

第一章为绪论，介绍了本文的研究背景及意义、国内外的研究现状、本文的主要研究内容与贡献。

第二章介绍了 GPU 计算和 GPU 虚拟化相关的研究。阐述了 GPU 的发展历程，由主要用于图形的 GPU 技术发展到 GPU 通用计算技术。针对虚拟化技术方面，主要探讨了系统级虚拟化。重点分析了 GPU 通用计算和虚拟化交叉的相关成果，探讨了目前实现 GPU 虚拟化的方法，并对存在的问题加以提炼和分析。

第三章详细阐述了虚拟化环境下多任务 GPGPU 并行计算的总体框架。采用经典的客户/服务器架构，构建在由大量服务器组成的计算机集群上，在这些物理节点上，有的物理节点拥有 GPU 设备，有的没有 GPU 设备，有的安装虚拟机，有的没有安装虚拟机。该架构分为虚拟化用户层，虚拟化资源管理层和虚拟化资源服务层。针对本文提出的系统框架，说明其主要的工作流程，并且对于 CUDA 库函数在虚拟化环境下的使用流程进行了详细说明。

第四章针对目前 CUDA 中没有针对多 GPU 的 API 以及现有的 GPU 虚拟化方案只针对单 GPU 的问题，提出在虚拟化环境下使用多 GPU 并行计算加速大规模程序的方案。为提升性能，对多 GPU 并行计算的关键问题如多层次存储技术和数据传输等技术展开研究，并用实验验证了该方案的可行性和高效性。

第五章提出 GPU 虚拟资源池的概念，详细分析了在虚拟机中虚拟 GPU 与物理 GPU 的映射

关系。在集群管理技术和负载均衡方法的基础上，提出适用于本文虚拟化框架的 GPU 负载均衡算法。通过层次分析法，确定影响因子的系数。实验结果表明，在虚拟化环境下，可以实现多个 CUDA 程序并发地共享一块或者多块 GPU 设备。对比本文提出的算法与随机的算法，验证了本算法可以有效提高硬件利用效率和集群的吞吐率。

第六章对本文的研究工作进行总结，并展望下一步需要研究的内容。

第二章　GPU 计算和 GPU 虚拟化相关研究

2.1 GPU 计算技术

GPU（Graphics Processing Unit，图形处理器）最初是为图形处理的需要而产生的，它在图形处理方面的造诣不言而喻，1999 年 NVIDIA 公司在发布 GeForce 256 时首次提出 GPU 的概念。摩尔定律预言了集成电路的发展，CPU 芯片的进步符合其阐述的趋势，并且十分准确。然而 GPU 的发展速度却远远超过了摩尔定律“18 个月性能翻倍”的预测，不仅对计算机的图形图像处理能力有了极大的提升，而且 GPU 的飞速发展还带动了图形处理、计算机仿真和虚拟现实等相关领域的发展。

2.1.1 基于图形的 GPU 技术

在 20 世纪八九十年代，计算机图形主要使用 CPU 在工作站上进行处理。随着图形处理器的功能不断发展和完善，使得图形处理从 CPU 向 GPU 转移。最早的 GPGPU 开发直接使用图形学 API 编程，它的图形渲染流水线依赖于图形学 API 和着色语言，如景深效果、透明/半透明处理、遮挡剔除等。这一时期称为可编程架构时期，编程人员需要深入地研究 GPU 下的图形绘制语言与流编程语言，才能利用 GPU 强大的计算能力。这些汇编语言或者高级着色语言包括：OpenGL[24]、Cg[25]、HLSL[26]、BrookGPU[27]等。

2000年，微软发布DirectX 8.0，首次引入了像素渲染的概念，提出了Vertex Shader（像素着色器）和Pixel Shader（顶点着色器）。Vertex Shader主要负责每个顶点的信息的计算，Pixel Shader主要负责片源颜色等的计算。2002年，ATI推出了第一个符合DirectX 9.0规范的加速器，进一步增加Vertex Shader和Pixel Shader的功能，灵活地实现循环和长浮点数的运算，使原来的汇编级语言发展成为C语言风格的高级语言HLSL[28]。

学术界进一步抽象了着色器语言，将与图形学相关的数据表示和操作屏蔽或者隐藏。2004年，斯坦福大学的Brook就是该领域的典型作品，该项目使得GPU在通用计算的应用越来越广。Brook设计了一个实时编译器，采用类似C语言的流处理语言Brook C，并且将有关图形学API的实现细节隐藏，甚至不需要了解图形图像在处理时是如何运作的，大大简化了开发的难度。AMD采用Brook的改进版本Brook+，将其用于GPGPU通用计算产品Stream中。Brook+在效率上相对于Brook有所提高，主要因为Brook+的编译器工作方式不同于Brook[29]。

基于图形的 GPU 技术使用了通用的图形学 API（如 OpenGL），使得它的可移植性和通用性较高。经典 GPGPU 可以运行于存在图形加速功能的图形设备上或者支持图形学 API 的系统上，为后来基于 GPGPU 开发语言的设计提供了方向性的指导。至今，仍然有一些通用计算的

算法由通用的图形学 API 实现，使用经典的 GPGPU 编写程序可以方便地使用这些模块。为了使用经典的 GPGPU 开发应用程序，对程序开发人员来说，必须先学习图形学并对图形 API 有较为深入的了解。此外，传统的图形学 API 最初并不是为通用计算设计的，它的重点是图形学的实现，若将传统的图形学 API 应用于通用计算领域会存在某些方面的缺陷，很难为特定的硬件架构提供统一的编程接口。由于以上缺陷，这个时期的 GPU 大多部署在科学实验环境中，实际环境下用的比较少。

2.1.2 GPU 通用计算技术

随着微软首次提出统一渲染架构（Unified Shader）以来，GPU 的可编程性也随之增强，研究 GPU 在非图形计算的任务越来越多，GPU 可以利用自身的图形处理单元完成对应用程序的加速。

2006 年，NVIDIA 公司推出统一架构 G80 架构，并在同年发布了其首版通用并行计算架构 CUDA。CUDA 架构的诞生简化了应用程序在 GPU 上的编程，使得可以利用 GPU 强大的计算能力实现更多的应用。CUDA 编程模型完全抛开了传统的图形编程接口，采用类 C 语言作为基本编程语言。为支持 CUDA，GPU 的硬件架构的关键特性做出了显著的改变，一是采用了统一处理架构，与前期专属的计算单元不同，统一架构采用片上设计并配置了标量流处理器，其功能是执行统一计算；二是引入共享存储器，支持同一块内的线程相互通信，提高计算效率，这些改进使得 GPU 更加适合通用计算的需求。自 CUDA 推出后，GPU 已广泛应用于高性能计算和科学计算领域，其中包括分子动力学、生物医药、金融分析、线性代数、石油勘探以及地震探测等领域[30]，在效率上比传统的 CPU 运算获得不小的加速比[31]。

作为另外一家 GPU 生产厂商 AMD 公司，在 CUDA 平台推出后第二年推出其统一架构产品，发布了首个版本的 Stream[32]开发平台来支撑统一架构理念下的硬件设备，实现通用计算。AMD 的设备架构兼顾线程级并行与指令级并行，基本运算单元是流程处理器。为了实现 GPU 强大的并行处理能力，采用流编程与流计算模型。最初，AMD 采用 Brook+开发语言，提供一种简洁的流编程模型和流计算机制。但是，这种模型虽然与直接利用图形学 API 的方式相比简化了开发过程，但是编译效率仍然很低，并且很难使得 AMD 的硬件物尽其用，在努力了一段时间后，AMD 转向另一种新的工业标准和规范上，即 OpenCL。

Khronos 研究组在 2008 年同 Intel、AMD、NVIDIA 等业内制造商一起合作，投入到 OpenCL（Open Computing Language，开放的计算语言）[33]异构平台下的并行程序开发，旨在建立一个真正支持异构计算网络的应用程序接口。与 CUDA 架构不同，OpenCL 不仅支持 CPU-GPU 架构，同时还支持 FPGA、DSP 等多种异构计算核心，相对于 CUDA 更加通用、开放。OpenCL 吸收并借鉴了 CUDA 架构的特点，计算架构分为四个模型，分别是平台模型、存储器模型、执

行模型和编程模型，前三个与 CUDA 架构非常类似。OpenCL 既提供了对各种 CPU、GPU 和其他硬件的支持，也提供了一套对底层硬件结构的模型抽象，使得模型对底层硬件的支持成为了可能。正是 OpenCL 的易用性（即融合性），许多第三方纷纷开发或收购 OpenCL 的开发平台。其中最具影响力的两款应用开发平台是 Google 公司的 PeakStream[34]与 Intel 公司的 RapidMind[35]。

微软公司的 Windows 操作系统在计算机领域占有大量的份额，它并未支持 OpenCL 的标准，而是自行开发和推广一种用于 GPU 通用计算的应用程序接口——DirectCompute，集成在 Microsoft DirectX （微软创建的多媒体编程接口）内。DirectCompute 编程模型基于 Direct3D 着色器架构，采用高级渲染语言 HLSL（High Level Shader Language）进行编程。在微软发布的 Windows 7 操作系统中设计了 DirectCompute 应用程序接口支持异构通用计算，目前，DirectCompute 在人工智能、物理加速视频转码等方面均得到很好的应用。

2.2 GPU 虚拟化技术

2.2.1 虚拟化技术概述

虚拟化技术由于其拥有隔离性（Isolation）、透明性（Transparency）和可迁移性（Migration）等突出的优点[36]，正迅速改变着 IT 的面貌，并从根本上改变着人们的计算方式。虚拟化技术不但能够将一台服务器上的某个应用迁移到别的服务器上，还能实现将不同平台上的应用整合到一台服务器上，因而能够降低硬件采购以及运行使用成本，提高服务器的利用率。目前具有代表性的虚拟化产品有 VMware[37]系列（如 Workstation、ESXi 等）、Xen[38]、Virtual PC[39]、Hyper-V[40]、KVM[41]等。由于采用的技术不同，系统虚拟化包括以下三种虚拟化技术：

（1）完全虚拟化（Full Virtualization）。通过在硬件和客户操作系统之间捕获和处理那些对虚拟化敏感的特权指令，无需修改操作系统内核就能运行，但是在性能方面不是特别优异。在完全虚拟化下，VMM 向虚拟机虚拟出和真实硬件完全相同的硬件环境，为每个虚拟机提供完整的硬件支持服务，包括虚拟设备、虚拟内存管理等。这种虚拟化方式自由度较高，采用这种虚拟化平台的有 KVM、VMware、VirtualBox 等。

（2）半虚拟化（Para-virtualization）方式，与完全虚拟化有一些类似，都利用 VMM 实现对底层硬件的共享访问。在半虚拟化下，VMM 需要操作系统的支持，完成对敏感特权指令的虚拟化，在执行特权指令时，由 VMM 完成特权指令，而非特权指令直接执行。在这种情况下，Guest OS 是知道自己运行在虚拟机中的，与全虚拟化相比，架构更精简，在速度上也有一定的优势，但是需要对 Guest OS 进行修改，所以在用户体验方面比较麻烦。常见的虚拟机平台有 Denali，Xen 等。

（3）硬件虚拟化（Hardware-assisted Virtualization），又称硬件辅助虚拟化，就是说 VMM

需要硬件的协助才能完成对硬件资源的虚拟，消除了特权指令对主机的影响。基于 Intel 公司的 VT-x 技术和 AMD 公司的 AMD-V 技术[42]，引入新的指令和处理器运行模式，使得 Guest OS 和 VMM 工作在不同的特权级中。通过引入硬件技术，使得虚拟化技术更加接近主机的速度，今后可能成为虚拟化的主流趋势[43]，常见的平台有 Xen。

相应的，I/O 设备的虚拟化在完全虚拟化、半虚拟化和硬件虚拟化中都不尽相同，其核心在于 I/O 设备原生驱动（Legacy Driver）的存放位置以及 VMM 对 I/O 设备的处理方式。

（1）完全虚拟化方式中，VMM 直接运行在物理硬件中，I/O 设备的驱动不需要作修改。VMM 对 I/O 指令进行解析并映射到实际物理设备，然后直接控制硬件完成，结构如图 2.1 所示。I/O 完全虚拟化的方式缺点在于 VMM 需要直接控制 I/O 设备，其设计会变得非常复杂。

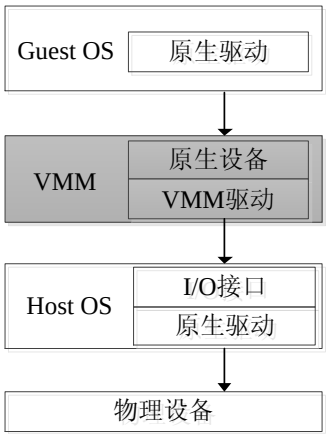

图 2.1 全虚拟化下的 I/O 设备虚拟化

（2）半虚拟化方式中，需要修改 Guest OS 内核，将原生设备驱动从 Guest OS 中移出，放在经由 VMM 授权的设备虚拟机中。在 Guest OS 内部，系统为每个虚拟 I/O 设备安装一个特殊的驱动程序 Paravirt Driver（半虚拟化驱动程序，也称副虚拟驱动），由该驱动程序负责 I/O 请求的传递，如图 2.2 所示。在这种方式下，I/O 的虚拟化性能较好，但是需要修改系统驱动，从而实现前后端驱动的对应。

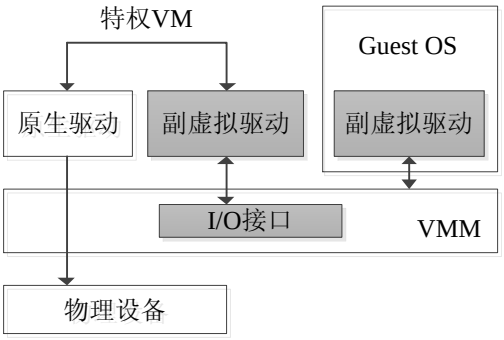

图 2.2 半虚拟化下的 I/O 设备虚拟化

（3）硬件支持的 I/O 虚拟化方式中，虚拟机内 Guest OS 借助于虚拟设备来完成虚拟 I/O

设备访问，而这些虚拟 I/O 设备访问与实际硬件的交互则通过特权虚拟机的设备驱动完成，如图 2.3。这种方式最大的优势在于虚拟机系统可以由 VMM 的请求转发，实现对硬件设备的直接访问[44]。

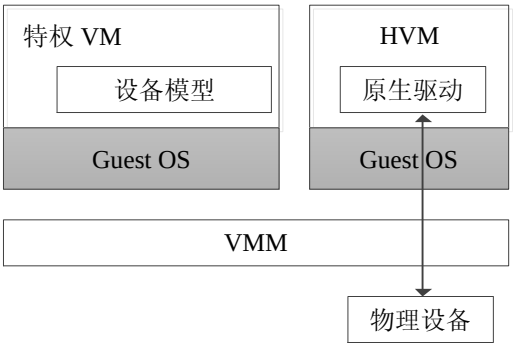

图 2.3 硬件虚拟化下的 I/O 设备虚拟化

2.2.2 GPU 虚拟化技术

在第一章阐述了虚拟化 GPU 困难的原因，至今尚未有完善的 GPU 虚拟化方案。目前，针对 GPU 虚拟化的方案主要分为四类：硬件虚拟化、设备仿真虚拟化、设备独占虚拟化以及 API 重定向虚拟化。

1）硬件虚拟化

2013 年，NVIDIA 公司推出 GRID GPU 为主的平台突破了 GPU 硬件虚拟化在技术上的瓶颈，实现了 GPU 的硬件虚拟化，充分发挥了 GPU 的优势。NVIDIA 公司的 Kepler 系列 GPU 在硬件架构上，将硬件虚拟化扩展到 GPU 上，从而使得云计算架构优势在图形处理上也可以得到很好发挥。此外 Citrix 公司 XenDesktop 技术使得 GPU 虚拟化更进一步创新，其成果就是真正以硬件支持虚拟化的专属 GPU 产品──NVIDIA 的 GRID K1 和 K2，以及 AMD 的 FirePro S10000、S9000 和 S7000。NVIDIA GRID vGPU 技术能够实现每台虚拟机的图形命令直接传递到 GPU，不用 VMM 来译码，提供顶级的共享虚拟化图形性能。这一技术可以实现 GPU 硬件的时间分片（Time-sliced），为虚拟化解决方案带来 NVIDIA 硬件加速图形的所有优势，使得多名用户共享一块 GPU 设备时使虚拟桌面达到堪比本地 PC 的性能，vGPU Manager 最多允许八名用户共享一块物理 GPU。

2）设备仿真虚拟化

设备仿真（device emulation）方法通过使用 CPU 来模拟 GPU，为虚拟机提供一个伪设备，模拟出真实的 GPU 设备的所有接口语义。这种方式在功能上是可行的，但是在效率上相差却很大。这是由于目前的 GPU 设备内部结构含有上百亿个电气单元，内部结构极为复杂，在大量的计算密集型和密集内存访问的计算应用上比 CPU 拥有更大的优势。如果纯粹靠 CPU 和虚拟化

平台来仿真 GPU，对于一般的文书处理需求，设备仿真方法可以应付；但是对于更高阶的 3D 图形、动画设计等，设备仿真方法就难以负荷。

在图形方面的设备仿真，VMware 公司在它的产品如 EXSi 等实现了 VMware SVGA II[45] 的仿真设备，能够支持 2D 显示，同时还支持 Direct3D 接口。Xen 和 KVM 等支持全虚拟化技术的虚拟机的访问模型基于 QEMU[46]，可以模拟 GPU 中的具有 2D 功能的显卡——Cirrus CLGD 5446 PCI VGA card。

针对传统图形流水线，GPU 的模拟器有 Victor Moya 等人于 2006 年 7 月发布的 Atila[47] 和 J. W. Sheaffer 等人于 2004 年发布的 Qsilver。Atlia 是一个比较完备的针对传统的 GPU 流水线的性能模拟器，它的设计是以 OpenGL 标准指定的 Rasterization 流水线为蓝图，实现图形程序能够在该模拟器上运行。Qsilver 的流水线结构与 Atlia 模拟器的流水线结构相似，即由一系列 cache 和队列将不同的功能单元相连接。J. W. Sheaffer 利用 Qsilver 在文献[48]中验证了 Qsilver 在性能模拟上的正确性，验证了细节丰富的图像成像过程。

另一方面，GPU 在通用计算领域的设备仿真，比较广泛使用的是在 2009 年 2 月由 Sylvain Collange 发布的 Barra，以及在 2009 年 6 月由 UBC 大学发布的通用计算模拟器 GPGPU-Sim[49]。Barra 是对 G80 体系结构进行建模的 GPGPU 模拟器，它基于 UNISIM[50]系统，使用 CUDA 内核函数经过 NVCC 编译器编译后生成的二进制 cubin 文件作为输入，能够对内核函数在执行时的一些因素进行分析，以此可以对 GPU 进行通用计算时的性能进行分析和评估。Barra 的结构划分为两个部分，一个是对 GPU 硬件结构进行建模的部分，另一个是 Barra 的驱动库函数。由于 Barra 的源文件含有大量的宏定义，以及 Barra 不含有针对所模拟的 GPU 硬件结构的配置文件，所以使用难度比较高。同基于 UNISIM 系统的 Barra 相比，GPGPU-smi 最初版本是基于 Simplescalar 实现的，它与实际的 GPU 硬件结构相对应，更适合对统一架构 GPGPU 进行模拟。GPGPU-smi 实现了并行功能模拟模式（Parallel Functional Simulation Mode），即需要并行部分的线程在解码栈进行功能模拟。它可以实现对线程块内的同步进行模拟，因为执行模式与 GPU 一样，使用线程块和 warp 的方式进行调度。GPGPU-smi 功能比较强大，配置选项全面，兼容性好，绝大部分 CUDA 程序无需改动即可在 GPGPU-smi 上直接编译运行。

设备仿真方案中，实际的设备状态位于由 VMM 管理的 CPU 或内存中，所以虚拟机的高级特性如虚拟机多路复用、实时迁移等功能可以延伸到 GPU 相关的领域。设备仿真方式下的 GPU 虚拟化虽然支持虚拟机特性，但是其性能却大幅度下降，与实际 GPU 擅长的并行计算相比，执行时间是 GPU 的数百倍甚至是上千倍，所以实用性不高。

3）设备独占虚拟化

设备独占方式，即PCI pass-through。PCI pass through需要硬件支持IOMMU，允许客户操作系统绕过VMM直接控制物理设备（Intel VT-d或AMD IOMMU），主要用于网卡，U盘等PCI设

备的直接访问。由于GPU的特殊性，比如集成显卡需固定内存作为显存，使其不能直接使用PCI pass-through。设备独占方式的GPU虚拟化是让用户使用的每一台VM，都能配置一颗专用的GPU，而不与其它用户或者VM共享，以便获得最大的加速处理效能。此方式确保了GPU的独立性和完整性，在性能上接近本地环境。在运作架构上，GPU pass-through的配置相对Shared GPU要容易的多，只需考虑VM与底层的硬件GPU，但是在VMM层与VM上的桌面代理程序（Virtual Desktop Agent，VDA）仍有一些限制。在虚拟机上的客户操作系统只需安装NVIDIA提供的GPU驱动程序，即可透传VMM层到物理GPU设备上去执行图形加速。目前支持GPU pass-through的VMM平台，有VMware ESXi和Citrix XenServer，VM桌面管理软件支持该模式的，有VMware View 5.2和Citrix XenDesktop 5.6 FP1。支持此架构的NVIDIA的GPU有Quadro 2000到6000系列、K2000到K6000系列，以及GRID K1/K2。

设备独占解决方案不需要模拟设备进行请求转换，客户机可以根据最新硬件，使用客户操作系统直接使用原生的显卡驱动程序获得对 GPU 的使用，所以访问速度高。但是，此方式是以牺牲设备的共享能力为代价的，绕过了 VMM 层，因而缺乏 VMM 跟踪和硬件的维护状态。在此方案中，不能实现虚拟机的高级特性，如实时迁移、快照等功能，因而这种 GPU 虚拟化的方案与虚拟化的本质不符，不仅增加了成本，而且在性能上较本地也有一定程度的下降。另外，设备直通方式中的设备只能被某一个虚拟机占用，因而难以充分利用 GPU 设备。

4）API 重定向虚拟化

API重定向虚拟化，顾名思义，就是指对应用程序重定向，具体来讲，就是在应用层拦截与GPU相关的API，通过重定向或者模拟的方式将应用程序发送到真实的机器，利用真实的硬件如GPU完成相应的功能，最后再将计算结果返回给应用程序。API重定向虚拟化被广泛应用在传统的图形学API方法中。VirtualGL（virutal machine library）[51]，具体的实施策略是在远程截取GLX库的协议指令流，将OpenGL中的指令拦截、解析，重定向到远程图形加速卡，在远端完成图形绘制，通过GLX协议传送给本地服务器，最后提交给某个特定场景窗口。VirtualGL的优势在于其非入侵性，即现有的OpenGL程序无需任何改动即可享有远端硬件加速。VMGL是首个在虚拟化环境下针对OpenGL渲染加速的解决方案，其目的是为虚拟机环境提供硬件加速功能。VMGL包含Guest端和Host端，Guest端在虚拟机中部署了一个虚拟OpenGL库，也就是伪库，该伪库替代原生的OpenGL库，伪库的API与OpenGL库相同，用来拦截OpenGL指令；同时，通过网络传输将伪库截获的OpenGL指令发送给宿主机中的Host端。宿主机中的Host端拥有原生的OpenGL库，直接驱动真实的GPU执行本地OpenGL调用，最后将结果返回给调用者。整个过程对用户来说是透明的，无需为虚拟化平台做任何的改动。

API 重定向虚拟化在 API 层对设备参数进行拦截、封装，在一定范围内可以掌握 API 设备的状态，所以能够支持虚拟化诸如实时迁移等高级特性。但是，API 重定向虚拟化存在着不足

之处：首先，并非所有的 API 都可以使用 API 重定向虚拟化的方案，因为很多 API 在设计之初并未考虑虚拟化和远程过程调用，加上这些功能可能会破坏原有的语义，同时虚拟显卡驱动的设计需要对所支持的所有 GPU 的 API 进行仿真。其次，API 重定向虚拟化需要了解目标 API 的实现细节，对于标准透明、公开的 OpenGL，实现起来比较容易，而对于封闭的 CUDA 架构则非常困难。最后，API 重定向虚拟化方案需要面对不同的 API 标准，如：DirectX，CUDA，OpenGL，OpenCL 等。

下面将上述几种 GPU 的虚拟化解决方案总结如表 2.1 所示：

表 2.1 GPU 虚拟化方案

虚拟化方案	图形渲染	GPU 通用计算	高级性能的支持
硬件虚拟化	NVIDIA GRID 显卡	无	支持图形渲染
设备仿真	QEMU、Atlia 、Qsilver Xen VGA pass-through	Barra GPGPU-Sim	支持图形渲染
设备独占	VMware Virtual GPU VMware VMDirectPath	Xen VGA pass-through VMware VMDirectPath	不支持
API 重定向虚拟化	VMGL VMware Virtual GPU(3D)	vCUDA、GViM、 gVirtuS rCUDA、 DS-CUDA、GridCuda	部分支持

综上可以看出，无论在商界还是学术界都对 GPU 虚拟化有一定的研究并取得了一些成果，但仍存在一定的问题：

（1）从表 2.1 可以看出，目前 GPU 虚拟化方案中很多都是针对图形渲染，在通用计算方面并没有完整的解决方案。这主要因为 GPU 在图形渲染方面的研究很成熟，而通用计算方面提出时间较短，所以经验不足，并且通用计算框架的细节未公开。

（2）虚拟化环境与本地环境相比有不小的性能损失，主要原因是 I/O 设备在设计之初并未考虑在虚拟化环境下的使用。特别在内存密集型数据中，涉及大量数据和运算，在虚拟化和数据传输时占用大量资源，性能损失更为严重。

（3）目前 GPU 虚拟化方案都只针对单块 GPU 设备的使用，要么只在本地虚拟化平台，要么只在远端虚拟化平台，应用场景比较单一。当有多个任务共享 GPU 资源时，并未有完善的解决方案。

2.3 本章小结

本章首先介绍了 GPU 的发展历程，由主要用于图形的 GPU 技术发展到 GPU 通用计算技术，在 GPU 通用计算领域发展中作出重大贡献的主要是 NVIDIA 提出的 CUDA 架构和苹果公司联

合 AMD 以及其他厂商提出的 OpenCL。针对系统虚拟化，主要有完全虚拟化、半虚拟化、硬件虚拟化这三种方式，I/O 虚拟化是虚拟化方式中比较困难的部分，GPU 作为特殊的 I/O 设备，主要探讨 GPU 虚拟化技术，分为硬件虚拟化、设备仿真、设备独占、API 重定向虚拟化，比较各自的优缺点，总结得出 API 重定向虚拟化是目前针对 GPU 通用计算虚拟化的有效方案，为后续章节奠定了一定的理论基础。

第三章　虚拟化环境下多任务 GPGPU 并行计算总体框架设计

3.1 虚拟化方案总体设计

3.1.1 虚拟化方案设计原则及目标

虚拟化技术对物理资源进行逻辑抽象和统一表示，被广泛运用于云计算中，其优势在于使硬件平台透明化，复用底层硬件资源。但是虚拟化技术主要运用于 CPU 虚拟化，对 GPU 虚拟化缺少相应的支持。本文虚拟化方案沿袭了虚拟化的特性，以及利用 GPU 强劲的计算能力，基于以下原则设计了虚拟化环境下多任务 GPGPU 并行计算框架。

高效性。GPU 通用计算的主要目的在于处理计算密集型任务，但是由于在虚拟机中使用客户操作系统需通过 VMM 与硬件交互，增加了数据通信层，不可避免地会带来额外的开销。如果虚拟化技术的开销大于 GPU 加速的能力或者完全抵消，则 GPU 虚拟化就是没有意义的。所以需要在虚拟化的开销与 GPU 高性能计算所取得的加速比之间找到一个阈值，使得 GPU 加速能力大于虚拟化带来的开销。本文虚拟化框架应尽量减少这种开销，在效率上具有可用性。

弹性收缩资源。虚拟化环境下需要灵活地使用计算资源，这些计算资源并不是一成不变的，可能随着硬件故障而撤销某些资源，也有可能在集群中增加某些资源以增加计算能力。所以在本文的虚拟化框架中应具备良好的资源收缩能力。无论增加还是撤销计算资源，数据的完整性不能遭到破坏，任务分配不能受到影响，并且能够充分利用虚拟集群里的所有计算资源。

透明性。本文的透明性包含两个方面，一方面是用户编写 CUDA 程序时能够保证完全的二进制兼容，无需修改操作系统内核并且无需改变 CUDA 源代码，保证兼容 CUDA 程序。另一方面，使用虚拟化技术，为上层软件（客户操作系统、应用程序等）屏蔽资源的异构性。综上，使用本文的虚拟化框架，用户无需考虑底层硬件环境，也无需改变 CUDA 源代码，即可在虚拟化环境下无缝运行。

通用性。通用性是指本文的虚拟化框架不限定于特定的 VMM 体系，可以部署在任意的虚拟化平台上，如 Xen，Vmware，KVM 等。这种通用性强调不同 VMM 框架之间的通信，对于拥有不同异构资源的云环境十分重要。

根据以上原则，本文虚拟化框架采用以下策略保证设计的目标。

保证高效性。目前并没有完善的虚拟机域间通信方式，通过在特定平台下使用不同的通信方式，提升效率。

保证弹性收缩资源。在系统中设置一个配置文件，在这个配置文件中，只需知道 GPU 设备的 IP 地址以及设备号，即可以增加或删除 GPU 设备。

保证透明性。一方面，在虚拟化用户层使用伪库拦截和重定向 CUDA 函数，将拦截的 CUDA 函数通过 Socket 通信发送到相应的 GPU 设备上运行，最后将结果回传给用户，整个过程对用户是透明的，使用如同与本地一样。另一方面，利用虚拟化资源管理层，屏蔽了计算资源层自身的差异和复杂度，对于不同的客户操作系统、应用软件等，其运行不受任何限制。

保证通用性。在本文的 GPU 虚拟化方案中采用 Socket 接口通信。Socket 是比较底层的通信接口，一些其他的通信方式都是基于 Socket 的，如 RPC 等，但效率没有 Socket 高。充分考虑不同 VMM 的共同点，将所有功能建立在 VMM 体系间公共接口上。

一般情况下，为了增加高性能计算集群性能，往往在集群中添加大量的 GPU 设备。但这样有一些缺点：1）不仅增加 GPU 设备配置成本，而且增加了维护、管理和空间等开销；2）GPU 是一种耗电的设备，因而增加能源成本；3）因为很少有需要极度并行的程序，GPU 在集群中的利用率比较低。针对这些问题，在高性能计算集群中加入虚拟化技术可有效改善上述问题。通过 GPU 虚拟化技术，集群中所有虚拟机可以透明地共享安装在部分节点中的 GPU 设备。GPU 虚拟化可通过启用较少的 GPU 而达到较高的性能，从而降低配置成本以及能源消耗，同时提高资源利用率。

目前主流的虚拟化环境下的资源共享方案缺少对 GPU 计算资源的支持，本文的系统架构采用动态库拦截的方法将 GPU 引入虚拟机，并且可以实现在本地无 GPU 资源的情况下仍可进行 GPU 并行程序的开发和运行。虚拟化资源管理层接收多个虚拟机提交的任务，并根据虚拟化环境中物理的 GPU 设备的负载进行合理调度。系统设计的主要目标如下：1）灵活的调度管理功能：对任务进行合理划分并分配到虚拟集群中 GPU 设备资源上；2）良好的透明性：程序员无需修改 CUDA 程序，即可在虚拟化环境下执行；3）平台无关性：本文虚拟化框架不限定于特定的 VMM 体系；4）良好的扩展性：能够动态地加入 GPU 设备，GPU 计算资源可能会随着平台的不断扩建而增多，也可能因硬件故障而将部分节点撤销。

3.1.2 虚拟化方案总体框架的设计

本文的虚拟化框架采用经典的客户/服务器架构，构建在由大量服务器组成的计算机集群上，这些物理节点按照功能分为服务节点和管理节点。为了充分利用集群中的节点，也可将高性能的计算机同时作为服务节点和管理节点。在这些物理节点中，有的拥有 GPU 设备，有的没有 GPU 设备，有的安装虚拟机，有的没有安装虚拟机。在装有虚拟机的物理节点上，物理硬件资源上层运行 VMM 来提供硬件映像。在虚拟机中的操作系统称为 Guest OS，它不能直接使用 GPU 设备，在本地环境下的操作系统称为 Host OS，它安装有原生的 GPU 显卡驱动和 CUDA 库，能够直接访问到物理 GPU。为更好地支持多任务在虚拟化环境下的高性能计算并行处理，本文构建了一种基于多 GPU 计算资源特征的动态分配与管理架构，实现一种基于多 GPGPU

并行计算的虚拟化方案，解决通用计算在虚拟化环境下的适应问题。根据用户所提交任务类型和规模，以及平台中 GPU 的使用情况，虚拟化资源管理层对 GPU 资源采用灵活的管理机制，实现资源请求、资源监控、资源分配与调度等工作，充分地利用集群的计算性能，大幅度提高系统的易用性，降低使用成本。

本文虚拟化框架包括三层：虚拟化用户层、虚拟化资源管理层以及虚拟化资源服务层（如图 3.1）。

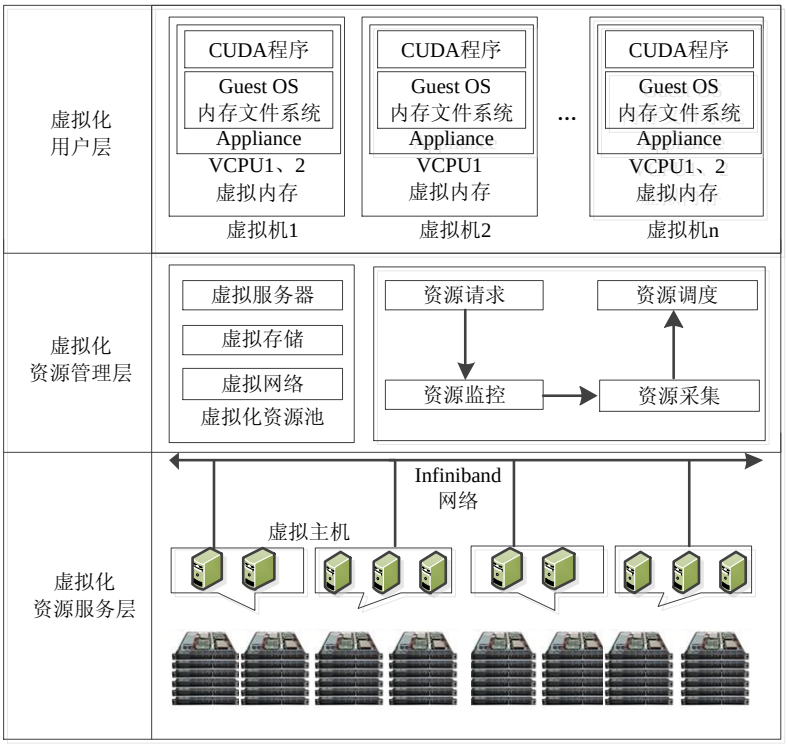

图 3.1 多 GPU 虚拟化架构

3.2 虚拟化方案的构建

3.2.1 虚拟化用户层

在虚拟化环境下使用 CUDA API 时，并不能直接在 GPU 设备上运行，系统采用动态库拦截的方法，可实现大部分的 CUDA 程序在不做修改的情况下仍能正确地运行在虚拟化环境下。使用动态库拦截的方式实现虚拟化方案中的通用性，因为拦截在系统层之上，与底层硬件无关。其主要方法是在用户层上提供代替原 CUDA API 的封装库，也称伪库（fake library），以及维护 CUDA 相关状态的虚拟 GPU（vGPU）。原生的 CUDA 有两个动态库，分别为 libcudart.so 和 libcuda.so。libcudart.so 对应的是 CUDA 运行时 API，libcuda.so 对应的是 CUDA 驱动 API。NVIDIA 官方推荐的是运行时 API，并且运行时 API 使用广泛，故本文虚拟化方案针对 CUDA

运行时 API 进行虚拟化。

通过使用动态库拦截的方法，即在虚拟机端建立一个与原生库同名的伪库 libcudart.so，此伪库与原生库的接口相同，当运行 CUDA API 时，伪库会将其拦截，通过 Socket 通信转到服务端使用物理 GPU。但是，伪库的实现细节与原生库不同，这主要因为原生库并不开源，无法得知其具体实现，另一方面，此伪库主要用于拦截重定向，达到虚拟化的目的。

虚拟化用户层主要实现：1）CUDA API 动态库拦截；2）局部性检查是否有函数依赖关系；3）将参数连同函数标识符一起打包、编码；4）对服务端返回的结果进行解码，并返回给应用程序。此外，由于用户层中存在多个虚拟机用户，每个虚拟机用户就相当于一个任务，每个任务里又有许多个子任务，故需对这些任务统一管理，从而充分利用虚拟化环境中所有 GPU 资源。所以当用户层中虚拟机客户端在运行 CUDA 任务之前，首先到虚拟化资源管理层中的管理节点请求 GPU 设备资源，每次独立地调用都需到虚拟化资源管理层请求资源，实现对任务的实时调度以及系统资源的均衡。

NVIDIA 官方提供的 CUDA API 都有接口定义，包含函数名称、函数参数、数据类型、返回值等。通过 API 拦截，即使不了解 CUDA 内部实现细节，也可将所获得的函数及参数等重定向到物理环境下重建。CUDA 状态包含 CUDA 程序运行时的硬件状态和软件状态，硬件状态包含显存、指令以及各级存储器的信息，软件状态包含用户库状态和驱动 API 所维护的 GPU 上下文状态，CUDA 上下文与系统中一个指定的设备（系统中可能存在多个设备）相关联。

在进行 CUDA API 虚拟化时，根据不同 API 的功能和实现特点，采用不同的封装和虚拟化方法，将其分为远端执行型和本地替代型。远端执行型是指在虚拟机中不能执行的 CUDA API，通过伪库将函数名和函数参数传递给 Host OS，由 Host OS 驱动物理的 GPU 执行，并将结果返回给虚拟机。这种方法虽然采用了远端执行的方式，并且转换了地址空间，但整个过程对于用户来说是透明的，如同在本地操作一样。目前绝大部分 CUDA API 都采用这种方式。本地替代型是指一些 CUDA API 并非都需要在远端执行，这样可以减少传输的开销。为了实现本地替代型的 CUDA API，在虚拟机端设置一个 vGPU。vGPU 与 vCPU（虚拟 CPU）概念不同，vGPU 主要用来维护虚拟机中 CUDA 状态与物理环境中 GPU 的一致性，保证与真实设备的同步。故可以利用 vGPU 使得本地替代远端执行。但这类的 API 并不多，主要集中在 GPU 设备管理的 API 中，如 cudaGetDeviceCount，cudaGetDeviceProperies 等。还有一部分 API 是 CUDA 的高级特性，需要较强的软硬件耦合度，无法模拟这种耦合特性，只能选择用本地函数替代。这类 CUDA API 如零拷贝所需的 cudaMallocHostMapped，由于虚拟机中的虚拟 GPU 无法知道内存的物理地址，只有真实的 GPU 才知道，故选择普通的 malloc 函数取代。

3.2.2 虚拟化资源管理层

虚拟化资源管理层作为统一虚拟化平台中的核心模块，主要负责整个资源池中资源的统一管理和调度，以及整个系统的运行监控。该层通过各种虚拟化技术将服务器资源、存储资源、网络资源整合起来，构建一个组织完善的资源池，用于动态管理和分配各项设备和资源，实现对资源的集中化、智能化调度和管理。虚拟化资源管理层接收用户层的 GPU 资源请求，同时对 GPU 虚拟集群中的 GPU 计算资源进行统一管理。调度的基本原则是尽量使发出资源请求的虚拟机调用本地物理机上的 GPU 设备，如果不能够满足需求，则重定向到远程物理机的 GPU 设备。当局部 GPU 资源压力过大时，由 5.3 节的 DMLS-GPU 算法，在全局调整负载，将任务发送到负载较轻的 GPU 资源上。

为了充分利用虚拟集群中 GPU 强大的计算能力，虚拟化资源管理层实现了在更高的逻辑层次上划分、隔离和调度。主要功能有：1）接收任务：用户层中拥有大量的虚拟机，每个虚拟机可以看成是一个任务，每个虚拟机里面又会运行多个 CUDA 程序，将这些任务发送到虚拟化资源管理层；2）实时接收负载信息：虚拟化资源服务层定时收集各个节点内的 GPU 的负载信息，并发送给虚拟化资源管理层；3）动态调度：根据接收到的任务以及负载信息，分配合适的 GPU 资源给多个任务，并且当用户任务结束时，虚拟化资源管理层回收资源；4）负载平衡：任务种类不同，对 GPU 资源有不同的需求，这样会导致 GPU 的局部压力过大，通过 DMLS-GPU 算法均衡 GPU 的分配；5）差错转移：当 GPU 设备发生故障时，将任务转移到另外可用的 GPU 资源上，重新运行 CUDA 任务。

3.2.3 虚拟化资源服务层

虚拟化资源服务层拥有若干个计算节点，在这些节点上部署服务端组件，服务端组件面向物理 GPU，是对某些 CUDA API 进行计算处理的部分。为了减少网络通信的开销，本架构的各个计算节点之间使用 Infiniband[52]交换设备连接。服务端组件拥有完整的 CUDA 库和驱动，能够直接访问硬件，因此直接使用物理 GPU 来进行运算。

虚拟化资源服务层的每个节点上都部署服务端组件，服务端组件面向真实 GPU，其主要功能有：1）对本节点内的 GPU 信息进行收集，并定时传输给虚拟化资源管理层；2）接收虚拟机发来的数据报，解析出 API 调用和变量定义；3）审核 API 调用和变量定义；4）使用本节点内的 CUDA 库和 GPU 驱动计算审核过的 API 调用；5）将计算结果编码，并返回给虚拟机。

服务端组件首先需要将支持 CUDA 的 GPU 设备信息发送给虚拟化资源管理层。每个服务端组件统一管理本地的硬件资源，按照虚拟机的要求以及当前资源分配策略提供虚拟的通用计算资源。服务端组件可以为多个虚拟机程序服务，这些程序可能是来自多个不同的虚拟机，也可能是同一虚拟机中的多个 CUDA 程序。在应对虚拟机程序请求时，在服务端创建服务线程，

每个服务线程对应一个应用程序。服务线程与虚拟机程序相对应，所以它开始于第一个虚拟机程序请求，结束于最后一个虚拟机程序完成。当虚拟机请求到达时，必须分配合适数量的服务线程，也可与其他虚拟机程序的服务线程合并。分配服务线程的数量主要取决于可用资源的使用情况、虚拟机程序的特点、负载均衡等，而合并线程要考虑 GPU 资源的使用情况，如全局内存、寄存器数量是否超出硬件极限等。

服务端组件管理本节点内的 GPU 资源，并根据虚拟化资源管理层的调用策略，将所需 GPU 设备传给虚拟机。为了标识虚拟化集群中的 GPU 设备，采用“IP 地址+设备号”标注每个 GPU。服务线程接收来自 CUDA 程序数据报中的 API 和参数，对其进行处理以符合 CUDA API 的接口定义。服务端组件还需创建计算线程，一个计算线程对应一个 GPU 设备，服务线程将调整后的 API 参数列表传送给计算线程。在任务切换机制上，GPU 与 CPU 有很大不同，在 CPU 中，可以使用中断处理机制从一个任务切换到另一个任务，而 GPU 核数目很多，无法保存所有核的状态，重建 CUDA 上下文开销很大，所以在计算线程中采用先来先服务的原则处理来自虚拟化资源管理层的任务。计算线程获取数据报中的 API 和变量信息，调用本地 CUDA 库和物理 GPU 进行运算，将结果返回给服务线程（如图 3.2）。最后服务线程将计算结果作为远程调用的返回值。

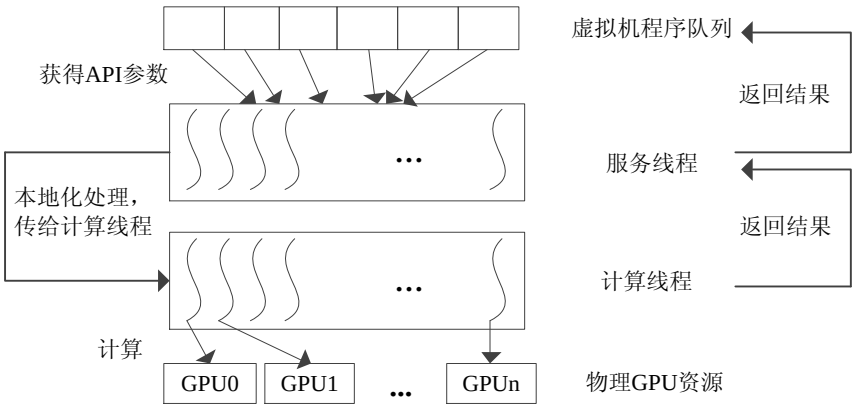

图 3.2 CUDA 服务端工作示意图

3.3 主要工作流程

3.3.1 系统架构的流程

本文工作流程参照云计算平台中常用的 MapReduce 架构[53]，该系统采用主从（Master/Slave）模式构建（如图 3.3）。系统含有两类节点：一个资源管理节点和多个计算节点。其中资源管理节点（Master 节点）负责接收客户端发送的任务以及计算节点中 GPU 设备的管理，计算节点（Slave 节点）利用 GPU 设备完成计算任务。由于系统中 GPU 设备的计算能力有所不同，需根

据 GPU 能力的情况分发给多个任务执行，以实现资源最大化的使用。

虚拟化环境下多任务 GPGPU 并行计算主要包含如下步骤：

用户程序初次到达系统时，首先更新本地节点的 GPU 负载评价值，选择合适的 GPU 计算资源，执行任务调度。一段时间后，随着一些任务的完成以及各个节点的 GPU 设备负载的变化，需要重新调整任务分配，以将 GPU 资源分配得更合理和均衡。

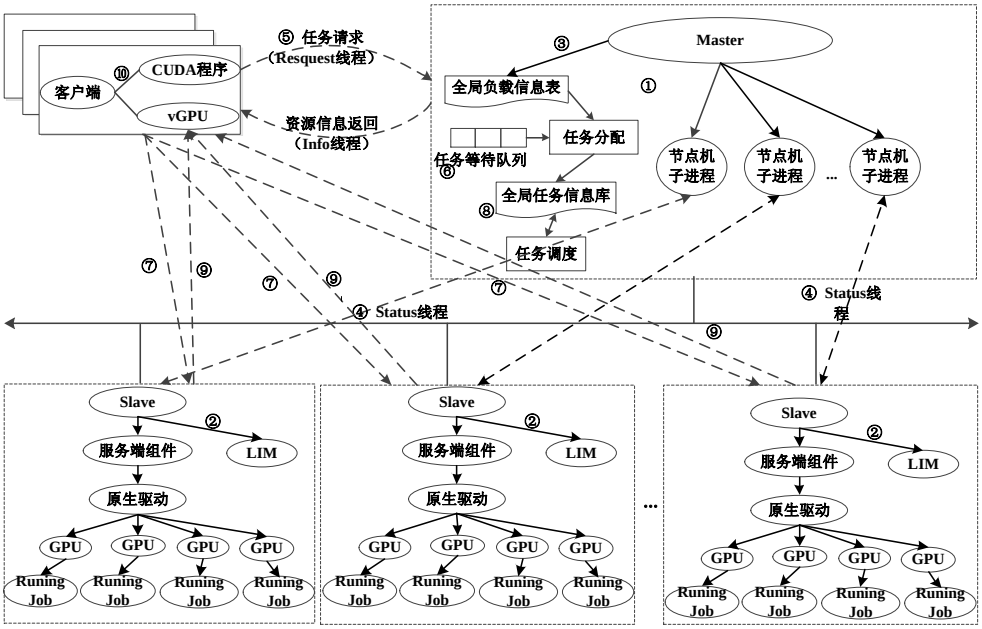

图 3.3 工作集群的逻辑结构

（1）启动 Master 节点，监听虚拟机客户端的资源请求及各个 Slave 节点的资源注册请求；

（2）启动 Slave 节点，在指定的时间间隔内，每个 Slave 节点运行 LIM（负载信息管理器）进程，负责收集本节点内的 GPU 设备的负载信息，向 Master 节点发送注册请求；

（3）Master 节点接收 Slave 节点的注册请求（Status 线程），建立一张维护当前集群内所有 GPU 设备的状态表（全局负载信息表），完成后返回成功；

（4）Slave 节点接收到注册成功的信息，立刻监听指定端口；

（5）启动虚拟机客户端模块，当虚拟机中出现对 GPU 设备的调用，则向 Master 节点发送资源请求（Request 线程）；

（6）Master 节点接收到虚拟机客户端的资源请求，将此任务存入任务等待队列，等待分配目标 GPU 设备。Master 节点检查全局负载信息表中已获取的 GPU 设备的状态，根据 DMLS-GPU 算法，向虚拟机客户端模块分配最匹配的 GPU 资源集合（Info 线程）；

（7）虚拟机客户端接收到分配的 GPU 设备资源，并且同该 GPU 设备资源的集合建立数据传输连接，虚拟机客户端将拦截的调用封装，并通过 Socket 通信发送至选择出的 GPU 资源；

（8）将分配结果更新至系统全局任务信息表，等待调度模块调度此任务执行；

（9）执行调用，并将结果返回直至执行结束；

（10）虚拟机客户端接收到程序的返回结果。

目标节点选择依据来自于其他各节点的全局信息，从而使调度结果更为准确。此外，各节点上多个模块分别监测和收集相应的数据和信息，并同步目标节点的初步选择结果，大部分调度过程由系统中的各节点并行完成，具有很好的实时性和准确性，大幅度提高了算法的效率。

3.3.2 库函数的执行流程

在虚拟化用户层提到 CUDA API 库函数分为远端执行型和本地替代型，本小节主要说明常用的 CUDA API 中远端执行型的执行流程，其余的远端执行型流程以此类推。

1）设备内存分配 cudaMalloc (void **devPtr, size_t size)

cudaMalloc 函数用于在 GPU 设备内存中分配一块地址空间，这个函数调用的行为类似于 C 语言中的 malloc()函数，参数有 devPtr 和 size。参数 devPtr 是一个指针，指向用于保存新分配内存地址的变量，size 表示分配内存的大小，单位是 bytes 表示。除了分配内存的指针不是作为函数的返回值外，这个函数和 malloc()函数是相同的，并且返回类型为 void*。

在虚拟机中运用 CUDA 程序编程时，设备内存分配的流程如图 3.4 所示。当 cudaMalloc()函数被调用时，虚拟机中的客户端会将 size 参数传递给远端的服务端。在服务端将会调用 NVIDIA CUDA 驱动 API 中的 cuMemAlloc()函数去配置大小为参数 size 的设备内存，并将这个内存空间的开始地址回传给客户端。最终，客户端会将此位置写入 devPtr 参数中，以便为了之后存取远端设备内存的内容使用。

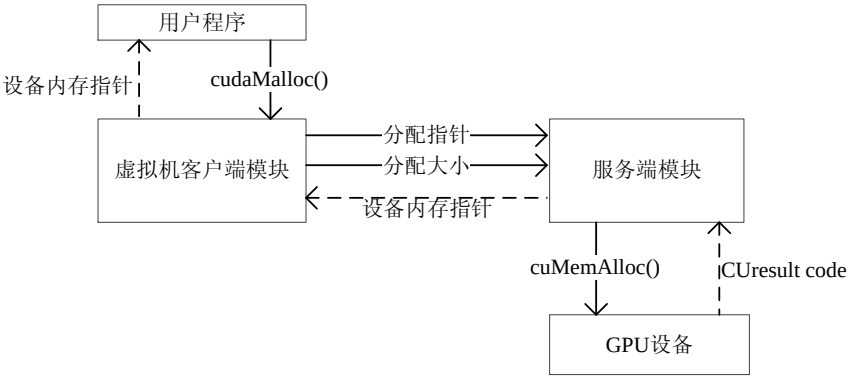

图 3.4 cudaMalloc 处理流程

2）设备拷贝函数 cudaMemcpy(void *dst, const void *src,size_t count, cudaMemcpyKind kind)

cudaMemcpy 函数用于数据的拷贝。参数中的 dst、src 分别代表欲拷贝数据的目的地址和源地址，count 表示欲从源地址指向的存储区域中将 count 个字节复制到目的地址指向的存储器区域，kind 代表数据在哪两者直接进行拷贝，分为四种类型，分别为 cudaMemcpyHostToHost（H->H）、cudaMemcpyHostToDevice（H->D）、cudaMemcpyDeviceToHost（D->H）和

cudaMemcpyDeviceToDevice（D->D）。当 cudaMemcpy 执行时，客户端会根据使用的 kind 参数，使用不同的伪库运行时 API 去执行主机与远端 GPU 设备的数据拷贝，如图 3.5 所示。

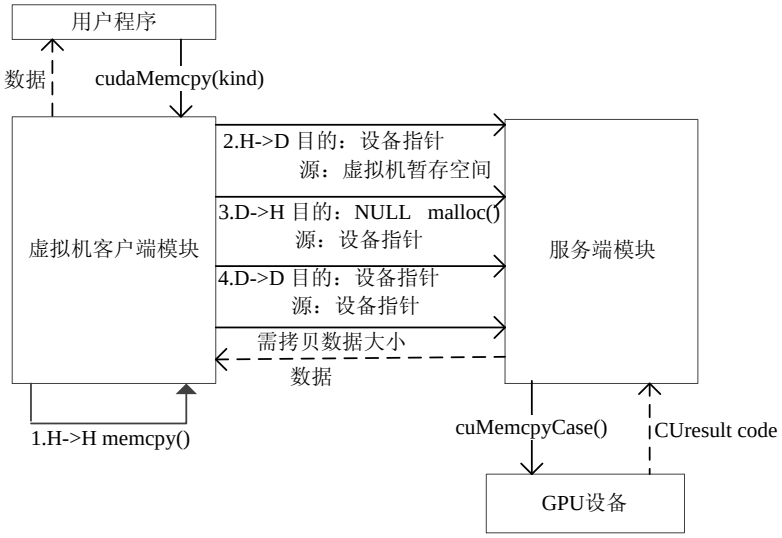

图 3.5 cudaMemcpy 处理流程

在此处，重点说明 cudaMemcpyHostToDevice、cudaMemcpyDeviceToHost 这两个常用的拷贝函数的执行流程。

cudaMemcpyHostToDevice 类型是指从主机端拷贝数据到设备端，在虚拟机客户端会动态地配置出一个暂存空间存储来自源端（Host）的数据，并将相关参数以数据包的方式发送到服务端。此数据包除了暂存空间的资料还包含参数 count 以及 dst。当服务端接收到数据包时，解包，并使用 NVIDIA 驱动 API 中的 cuMemcpyHtoD()，将 host 端的数据拷贝到参数 dst 指定的 device 地址。数据将从 device 的起始地址 dst 拷贝到 dst+count-1 的地址，并将执行结果回传 CUDA Error types code 给 client 端。

cudaMemcpyDeviceToHost 类型是指从设备端拷贝数据到主机端。客户端会将参数 src、count 传到服务端，服务端接收到这些参数后，会调用 C 语言中动态配置空间的 malloc()函数，配置大小为 count 的空间。调用 NVIDIA 驱动 API 中的 cuMemcpyDtoH()，将 GPU 设备来源端的起始地址拷贝到先前分配出的内存暂存空间上，最后回传至客户端。当客户端接收到拷贝数据的内容再将其写回主机端内存，数据将从 host 的起始地址 dst 拷贝至 dst+count-1。

3）kernel 函数的执行

所有 kernel 函数使用<<<dimGrid，dimBlock>>>或者 cudaLaunch 函数在设备上执行，GPU 所使用的线程数为 dimGrid *dimBlock，利用 cudaSetupArgument()来执行。当执行到 cudaLaunch(const char* entry)时，entry 表示 GPU 设备中调用 kernel 函数的名称，服务端会调用 NVIDIA 驱动 API 中的 cuLaunchGridAsync()执行 GPU 设备中指定的 kernel 函数。当 kernel 函

数执行完毕，服务端会将执行结果回传 CUDA Error types code 给 client 端，如图 3.6 所示。

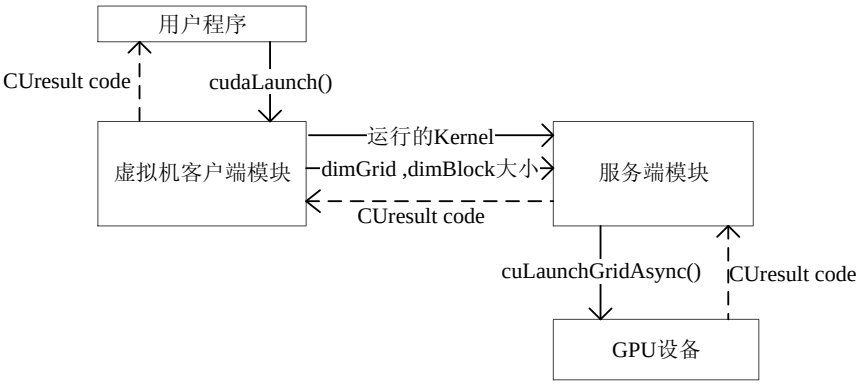

图 3.6 kernel 函数处理流程

4）设备内存释放函数 cudaFree (void *devPtr)

cudaFree 函数的功能是在 GPU 设备上释放所用的空间，参数 devPtr 表示欲释放空间的起始位置，执行流程如图 3.7 所示。当调用 cudaFree()函数时，客户端将参数 devPtr 传送至服务端。接收到此参数的服务端会调用 NVIDIA 驱动 API 中的 cuMemFree()，释放参数所指定的 GPU 内存地址，并将结果回传 CUDA Error types code 给 client 端。

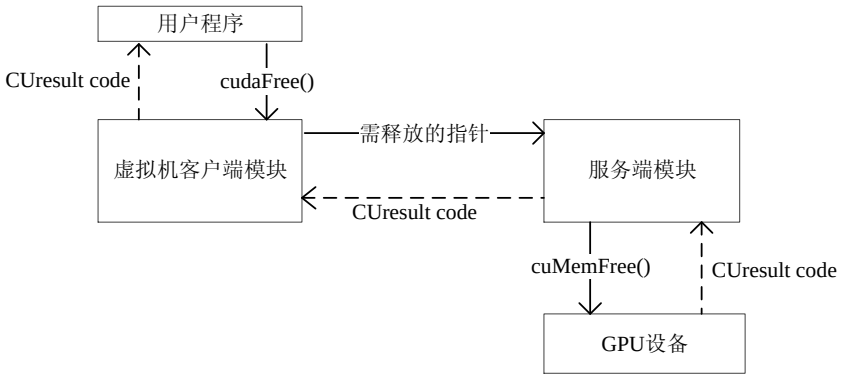

图 3.7 cudaFree 函数处理流程

3.4 本章小结

本章首先说明 GPU 虚拟化方案的设计原则及目标，根据这些原则和目标，设计虚拟化环境下多任务 GPGPU 并行计算总体框架，使得在虚拟化环境中仍能使用 GPU 进行 CUDA 运算，并且根据虚拟化的特性，使得 GPU 可以在多个计算节点间共享。根据 GPU 虚拟化框架，构建了虚拟化用户层、虚拟化资源管理层、虚拟资源服务层，并对每层的功能进行了详细的说明。接着针对提出的系统框架，说明其主要的工作流程，并且对于 CUDA 库函数在虚拟化环境下使用流程进行详细说明。

第四章 虚拟化环境下多 GPU 并行计算

随着对计算能力的要求不断增加以及硬件成本的不断降低，在一台计算机内安装有多块 GPU 设备以提高性能成为广泛采用的方案。目前 CUDA 中没有针对多 GPU 的 API，且现有的 GPU 虚拟化解决方案大多针对单块 GPU 设备，在多 GPU 环境下，应用程序使用多 GPU 进行加速的研究较少。为了能够更加充分地利用多块 GPU 设备智能地扩展应用性能，加快工作流程，本章根据 CUDA 规范，将多线程技术引入到 GPU，实现了在虚拟化环境下多线程控制多 GPU 的并行计算；对于在主机端只有单个主线程的情况，实现了在虚拟化环境下基于流处理的多 GPU 并行计算，使得单个主线程也可以访问到所有 GPU 设备，并且通过流处理的方式，使得传输和计算并行执行。在虚拟化多 GPU 环境下，为了优化程序的设计，充分利用 GPU 强大的计算能力，本章分析了 GPU 多层次存储结构，并优化存储器间的传输带宽。

由于虚拟化本身的特点，在虚拟化环境下进行应用程序的开发不可避免地会带来性能的损耗，特别是不能直接虚拟化的 GPU 设备。传统的 RPC 系统如 CORBA[54]、XMLRPC[55]、ICE[56] 主要用于网络环境或者分布式环境，将这些 RPC 通信方式移植到虚拟化这种受限的环境时并不适用，在系统吞吐率、延迟等方面有很大的影响，一般而言在实际中基本不会使用上述方法。针对这些问题，本章在基于 Xen 和 VMware 虚拟化平台下根据 CUDA 应用的延迟和吞吐率情况找出最优的域间通信方式。此外，当使用多 GPU 并行处理大规模紧耦合的数据时，探讨了多 GPU 之间通信的问题。

4.1 多 GPU 并行计算技术研究

4.1.1 基于 OpenMP 和 Pthread 的多 GPU 并行计算

在 CUDA 4.0 以前，多个 GPU 设备需要多个主机端线程来进行管理。在每个 CPU 线程中，必须在执行任何 CUDA 运行时 API 函数之前调用一次 cudaSetDevice()函数，以此与一个 GPU 设备关联，并且以后也不能与其他 GPU 设备关联。CPU 线程的数量可以比设备数量多，但一个时刻一块 GPU 设备上只有一个 CPU 线程的上下文。一般认为，为达到较好的性能，最好使 CPU 线程数量等于设备数量，以使每个线程与设备一一对应。虽然多个 CPU 线程会增加 CPU 的开销，但是相对于多个 GPU 带来的强大计算能力而言，这样是值得的。

将通常用于多核 CPU 编程的 OpenMP[57] （Open Multi-Processing）和 Pthread[58] （POSIX threads）等多线程技术引入到 GPU 编程中，实现多 GPU 的并行计算。OpenMP[57]是基于共享内存编程模型的应用程序接口，采用 fork/join 并行模式，在源代码中加入制导性语句（Compile Directive）pragma 来指明程序的并发性。当主线程执行到并行区时，会派生出其他线程，这些

线程与主线程共同完成并行区的任务，必要时可加入通信和同步函数等。当忽略这些制导性语句时，程序自动转成串行程序。OpenMP 移植性好、可扩展性高，已成为共享存储系统中的并行编程标准，适合控制多 GPU 进行运算。

Pthread 是 POSIX 线程的简称，它与 OpenMP 类似，但更加注重底层实现。Pthread 是在 Linux 下开发的，定义了各种平台下均可使用的 C 语言函数、类型和常量，因而易于移植。Pthread 根据系统库创建活动线程，并发执行各自线程内的任务。对每个创建的线程适当控制，如通过加锁操作，可以保证各个线程之间不会交叉，每个线程只运行自己的那一部分。根据 Pthread 的特性，其同样适合控制多 GPU 进行运算。

4.1.2 基于流处理的多 GPU 并行计算

基于流处理的方式也可实现多 GPU 并行计算。流（Stream）是对 CUDA 并发进行管理的一种方式，默认情况下，CUDA 在一个 GPU 内部创建一个执行流，若不指定编号，默认为 0。所有的数据传输与 Kernel 调用都在这个流中排队等候，并按队列顺序依次执行。通过显示地创建与使用多个执行流，能够在单位时间内进行更多的工作，使应用程序执行得更快。

1）隐式同步使用多 GPU

在单个 CUDA 程序中使用多个 GPU 最简单的方式是为每个设备的每个上下文隐式地创建一个流，通过 cudaSetDevice()更改当前设备来使用多个 GPU。CUDA 在创建上下文时通过 cudaMalloc()引起上下文状态的改变，从而引入上下文环境的创建。GPU 设备驱动通过在设备驱动程序中为应用程序提供多个上下文环境，可以使单个 CUDA 应用程序使用多个设备。在每个设备上对任务进行排队，进而通过增加系统中 GPU 数量来提升应用程序性能。

2）显式同步使用多 GPU

在 CUDA 4.0 及以后版本中，可以使用单线程方案轮番作用于多个设备。通过显示同步使用流，使用单 CPU 线程就能进行控制。这种方式减少了 CPU 的开销，但在分配工作时需要额外处理不同 GPU 间的切换，这会消耗一定的时间，而且流之间相对无序，无法保证行为的正确性。通过在上下文环境间切换，设备驱动可以如同一个小的操作系统一样，使多个 GPU 计算程序多任务化。执行完所有 Kernel 函数后，使用 cudaStreamSynchronize()函数在指定流上执行同步，等待指定流执行完成之后再继续执行后面的代码。

4.1.3 基于 MPI 的多 GPU 并行计算

MPI[59]（Message Passing Interface）属于消息传递模型，主要用于多节点的运算，在集群式高性能计算机、服务器集群中有着广泛的应用。在多节点上使用 MPI 进行粗粒度的通信，每个节点内部使用 CUDA 进行运算，实现多节点共同运作。在这种混合编程模型中，每个节点都启动一个 MPI 进程，使用消息传递在不同节点之间传输数据，实现粗粒度的并行。在每个节点内

部，根据程序内部特性，设置 CUDA 线程个数。这种方式通信成本较高，适合任务划分明确、通信较少的粗粒度并行。通常可以将多个编程模式混合使用，提高运行速度。

4.2 GPU 多层次存储技术与优化设计

4.2.1 GPU 存储层次分析

为了优化应用程序的性能，CUDA 使用了多层次存储器技术，不同类型的存储器面向不同应用中的数据结构，对其合理利用可以极大地改善程序的运行效率，减少数据传输时间。在 CUDA 模型中使用了主/从模式，CPU 是主机端，GPU 是协处理器，作为设备端。被 CPU 访问的内存称为主机端内存，可分为可换页内存和页锁定内存。可换页内存是一种通过操作系统分配的存储器空间；页锁定内存也叫不可分页内存，操作系统不会对这块内存分页并与磁盘交换，确保内存时钟驻留在物理内存中，允许 GPU 上的 DMA（Direct Memory Access，直接内存访问）控制器请求主机内存而不需 CPU 主机处理器的参与。

GPU 设备内存拥有复杂的存储层次，可以分为三层：每个线程内部都拥有自己的一块局部存储器（Local memory）和寄存器（Registers）；在每一个 Block 中，存在着一块共享存储器（Shared Memory），提供该 Block 内的所有线程共用；能够被一个应用程序中所有线程访问的全局存储器（Global Memory）。另外，板载显存上还存在着可被所有线程访问的只读存储器：常量存储器（Constant Memory）和纹理存储器（Texture Memory）。图 4.1 为 GPU 的多层次存储器空间。

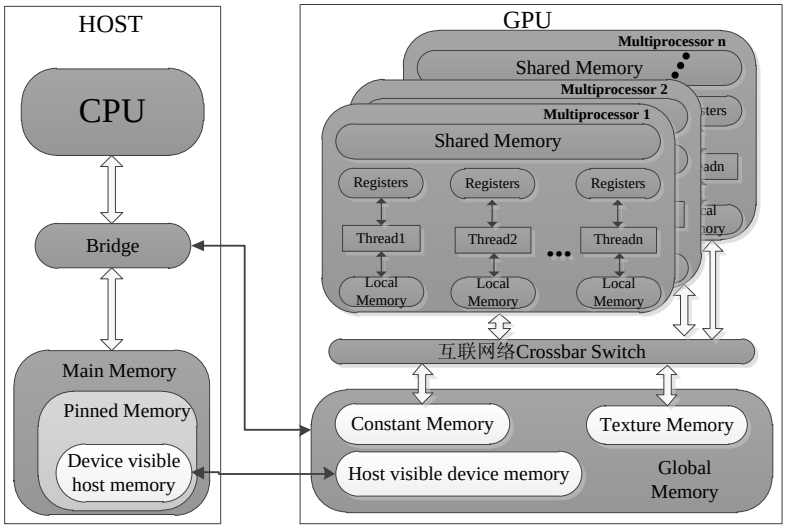

图 4.1 GPU 多层存储器空间

GPU 片上含有大量的寄存器资源，根据硬件不同，每个流多处理器（SM）可供所有线程使用的寄存器空间大小也不同，它为大规模线程的并发提供了零开销切换。每个线程的每个变量都会占用一个寄存器，当线程中的寄存器不够用时，会自动溢出到片外局部存储器中。在线程执行过程中由于增加了大量的全局访存事物，会使机器速度大幅度减慢，降低应用性能。共

享存储器是一种可受用户限制的一级缓存，属于片上存储器，速度仅次于寄存器，支持在一个 Block 中临时计算结果的重用或共享。当线程束中的所有线程同时访问相同地址的存储体时，共享内存会触发一个广播机制到线程束的每个线程中，对数据重复利用或者线程之间有共享数据时性能会得到很大的提升。常量内存是为多个线程只读类型的广播操作而优化的内存，通过广播机制，可以快速地将数据分配到线程束的多个线程中。纹理存储器通常用来做局部优化，即它希望数据提供给连续的线程进行操作。全局存储器是一个 CPU 和 GPU 都可以对其进行写操作的存储，任何设备都可以通过 PCI-E 总线对其进行访问。全局存储器属于片外存储器，对其访问需要消耗大量的时间，对设备存储器的一次访问需要花费数百个时钟周期，存储速度非常慢。表 4.1 详细列出多层次存储器的相关属性。

表 4.1 存储层次和相关属性

存储空间	位置	缓存	访问	延迟	作用域
寄存器	片上	否	读/写	1 个周期	一个线程
共享存储器	片上	N/A	读/写	1~32 个周期	一个块内的所有线程
局部存储器	片外	否	读/写	400～600 个周期	一个线程
固定存储器	片上 cache	是	只读	400～600 个周期	全局+主机
纹理存储器	片上 cache	是	只读	400～600 个周期	全局+主机
全局存储器	片外	否	读/写	400～600 个周期	全局+主机

由于拥有多层次存储技术和存在多种访存模式，用户能够充分有效地利用片上存储器，挖掘计算 Kernel 之间的时间和空间局部性，实现各级存储层次上负载均衡。

4.2.2 异构平台中传输带宽优化设计

1）数据传输模式的优化

尽管 GPU 拥有很强的计算能力，但有限的传输带宽在性能优化方面仍然是一大瓶颈。通常情况下，在可分页内存中数据从主机端拷贝到设备端，拷贝速度将受限于 PCI-E 传输速度和系统前端总线速度相对较低的一方。在某些系统中，这些总线在带宽上存在着巨大的差异。因此当在 GPU 和主机间复制数据时，这种差异会使页锁定主机内存的性能比标准可分页内存的性能要高大约 2 倍。即使 PCI-E 的速度与前端总线的速度相等，由于可分页内存需要更多一次由 CPU 参与的复制操作，因此会带来额外的开销。合理分配使用页锁定内存能够提高传输带宽，但是由于页锁定内存不能交换到磁盘上，在应用程序中使用时都需要分配物理内存，故与可换页内存相比，系统内存将会更快地耗尽，所以使用完毕后，应立即释放。对于使用多 GPU 的应用程序，如果支持统一虚拟寻址（UVA，Unified Virtual Addressing），指定所有页锁定内存为可共享的是一个非常好的方式。

一般情况下 CPU 只能访问主机内存，GPU 只能访问设备内存。零拷贝内存也是页锁定内存，它被映射到 GPU 地址空间，在内核程序中可以直接对其读取或写入。采用零拷贝内存和写

结合策略可以有效提高带宽的利用率。对于两个输入缓冲区，运行时可将内存分配为“合并式写入（Write-Combined）”内存，虽然这种方式不能改变应用程序本身的性能，但却可以提升 GPU 读取内存时的性能。对于计算密集型程序，计算密度很大，采用全局内存合并方法隐藏 PCI-E 延迟。采用零拷贝内存，将内核传输操作分解为更小的块，通过流水线的方式执行，减少整体时间。

2）隐藏数据传输的延时

对某一数据集的操作，一般是将数据从主机传输到设备、对数据集进行计算，然后将结果传输回主机。这种方式是完全串行的，将导致主机和 GPU 在一段时间内都是闲置的，浪费了计算能力和传输能力。流是 GPU 上虚拟的工作队列，使用流计算模式或者异步传输方式在一定程度上隐藏了 CPU 与 GPU 之间的传输延时。在 Kernel 函数执行期间，设备端能够把控制权返回给主机线程，这样主机端和设备端能够并行执行任务。使用双缓冲技术，CPU 和 GPU 在这两个中转缓冲区之间来回交互，尽管 GPU 正在将结果传输回主机并且请求一个新的工作包，但另一个缓冲区仍然能被计算引擎用来处理下一个数据块。

在 CUDA 中，异步并发操作可以使用流处理来管理，异步操作使得处于同一流内的 Kernel 计算和数据传输是顺序执行的，但在不同的流之间，它们的执行是没有顺序的，因而可以使 GPU 中的执行单元和存储器中的控制单元同时工作，进而提高资源利用率。通过流处理操作可以实现多个 Kernel 函数与数据传输的异步并发执行，其中 Kernel 函数和数据传输分别在不同的流中。

4.3 虚拟化环境下多 GPU 并行计算通信策略

4.3.1 虚拟机域间通信策略

本文的 GPU 虚拟化方案是将请求发送到客户操作系统的前端，通过 TCP/IP 协议栈，调用 Socket 接口，前端与后端守护进程进行通信，由后端的驱动程序完成请求的处理，这种虚拟化架构和网络传输会带来很大的性能开销。为减少这种开销，本小节在不同的虚拟化平台中设计或应用不同的通信机制。

GPU 虚拟化方案中通信策略主要是在虚拟机域间通信，在目前跨 VMM 通信方面还不存在有效的软件方案，只能依靠使用高速的传输设备如 Infiniband 这类特殊的硬件机制。Infiniband 技术不是用于一般网络连接的，设计的主要目的是针对服务器端连接问题的，使用了特殊的通信协议，支持多并发链接的“转换线缆”技术，其中 Mellanox 公司的 InfiniBand FDR 速度高达 56Gb/s。针对在远端无法使用虚拟化平台的特性，通信方式采用 Socket。Socket 方式比 RPC 的方式更加底层，效率较 RPC 方式高。

目前主流虚拟化平台并不存在高效且通用的通信方式，为了使本文 GPU 虚拟化方案有效使用，通过对 Xen 和 Vmware 主流的虚拟化平台中设计的特殊通信机制进行全方位的对比，在不

同平台下得出在不同的虚拟化环境下针对 CUDA 程序应用最佳通信方式。

1）基于 Xen 的虚拟机通信方式

Xen 平台下域间通信效率低的原因主要有 TCP/IP 操作、复杂的通信路径和页面切换。针对这些因素，研究人员提出域间通信优化的解决方案，具体有 XenSocket[60] 、XWAY[61]和 XenLoop[19]。

XenSocket[60]利用标准的 Socket API 接口，在 Linux 内核创建一种使用共享内存来传输数据的新的 Socket，使得各个虚拟机能够共享同一物理硬件平台。为防止内存读写重读，XenSocket 加入了互斥锁机制，用来控制共享缓冲池的读写。XenSocket 的共享内存分为两部分，一部分是由 32 个 4KB 缓冲区内存页所组成的共享循环缓冲池，总计 128KB，主要负责数据的通信；另一部分是由 4KB 内存页组成的控制信息页，主要用来存储通信双方共享状态的控制信息。XenSocket 的体系结构如图 4 .2 所示。XenSocket 在通信双方的共享缓冲池只有一个，所以只能提供数据的单向传输，这种方式适合于非对称的广播通信。此外 XenSocket 对用户不透明，需要修改程序源代码才能使用，另外它还不支持动态迁移。

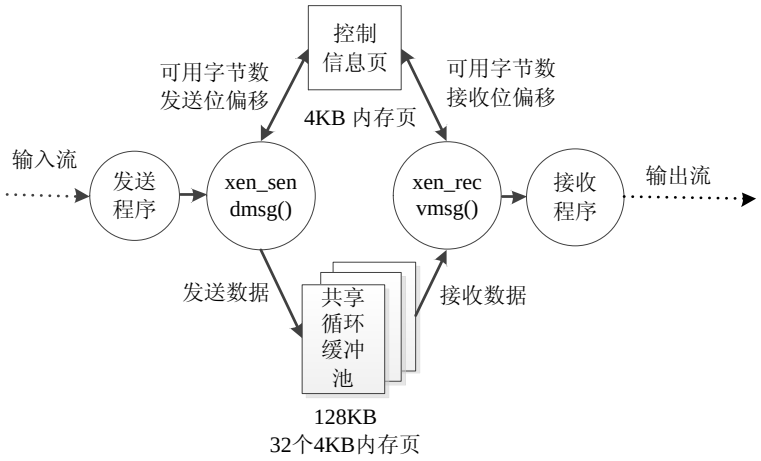

图 4 .2 XenSocket 的体系结构

XWAY[61] 是 Xen 平台上又一重要的域间通信方式，它绕过 TCP/IP 通信的协议栈，同时也避免了页面切换的开销。相对于 XenSocket，实现了对用户应用程序透明和全双工通信，在同一物理机上的不同虚拟机之间建立一个可直达的通信通道，从而实现虚拟机间高速通信。在内核中，XWAY 能够动态判定通信双方是否在同一物理机器上，如果在就用 XWAY 通信，否则就使用 TCP 协议通信。XWAY 同样使用了共享内存传输数据，并且相对 XenSocket，能够兼容二进制核外应用程序。XWAY 的体系结构如图 4.3 所示。XWAY 系统分为四个组成部分：XWAY 虚拟网络接口和 XWAY 设备驱动，以及为了兼容标准 Socket API，额外增加了 XWAY 转换器和 XWAY 协议层。XWAY 转换器在传输层工作，负责链路的选择，即如果目标虚拟机在同一节点上，则使用 XWAY 通道，使用 bind()函数与 XWAY Socket 绑定，否则将请求传给 TCP 层。

XWAY 协议层将传输层和网络层功能相结合，通过 XWAY 设备驱动程序传输实际数据。XWAY 设备驱动不仅模拟了网卡驱动本身的功能，而且同 XenSocket 一样，使用共享内存技术，加速数据传输。

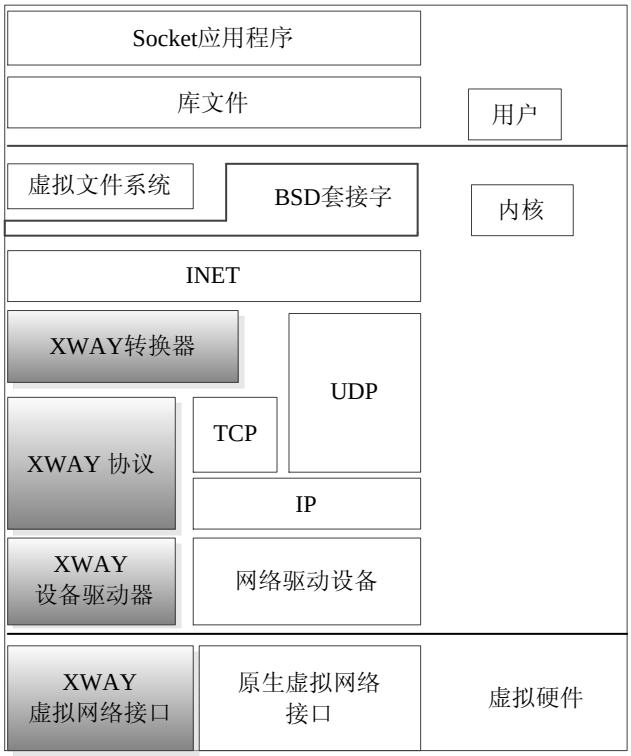

图 4.3 XWAY 的体系结构

XenLoop[19]相对于 XenSocket 和 XWAY，透明性更加良好，无需修改程序源代码也无需修改内核程序的协议栈，即可实现无缝运行。此外，XenLoop 可以实现同一物理机器中虚拟机之间直接通信而不需要第三方如特权域的参与。图 4.4 为 XenLoop 的体系架构图。XenLoop 在网络层和驱动层之间实现了 XenLoop 层，包含工作在需要通信的虚拟机（DomU）的 XenLoop 软件桥和工作在特权域（Dom0）的特权域软件桥。XenLoop 软件桥对某些端口进行监听，拦截网络层的数据报，通过查看 IP 地址检查两个通信域是否在同一物理机器上，若是则通过共享内存与目的虚拟机通信，否则采用 Socket 方式通信。

XenLoop 有一最大的特色是支持实时虚拟机迁移等虚拟化平台的高级特性，任务在不受影响的情况下转移到另一台物理机器上运行。此外，通过 Xen 特有组件 XenStore 实现虚拟机的动态发现，检索其中的 MAC 地址和 ID 号以及 XenLoop 标志位判定是否可以使用 XenLoop 通信，检索 MAC 地址和 ID 号判定是否在同一物理机器上，XenLoop 标志位是判定通信双方是否都配置了 XenLoop，若同时满足，则可用 XenLoop 通信。

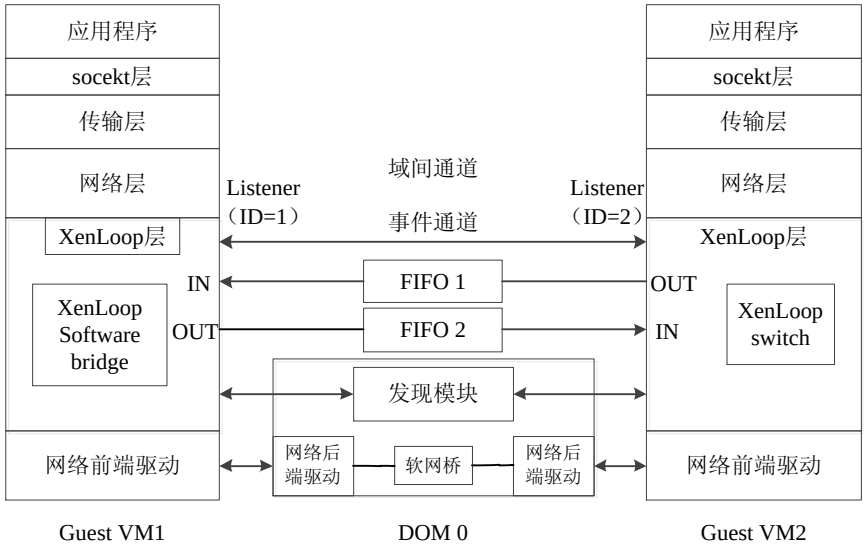

图 4.4 XenLoop 的体系架构

2）基于 VMware 的虚拟机通信方式

VMware 在通信方面也提供了类似的解决方案 VMCI[62]（Virtual Machine Communication Interface）。在 VMware 系列虚拟机中，VMCI 提供虚拟机平台下的高效域间通信通道，使得虚拟机与虚拟机之间、虚拟机与物理机实现高速的通信。VMCI 属于 VMware 的商业产品，其内部细节并未公开。它实现了两类接口，一类是数据报 API，一类是共享内存 API。图 4.5 是 VMCI 的结构图，图中 VMware workstation 中虚拟机上的应用程序利用 VMCI 进行通信。

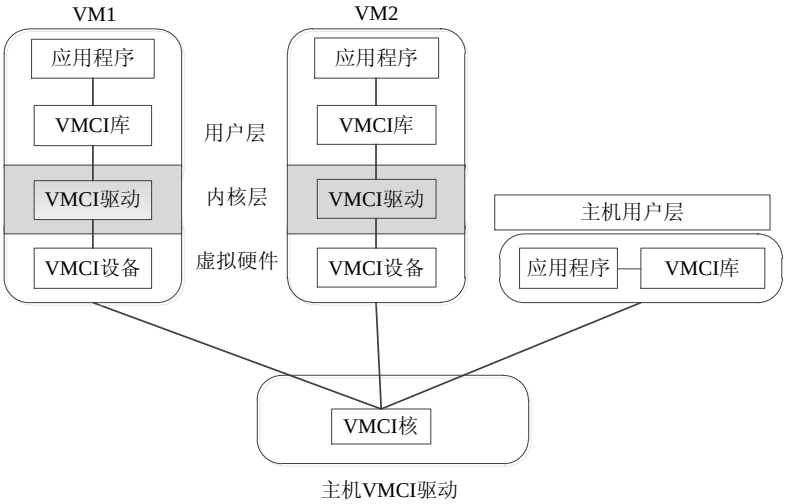

图 4.5 VMCI 的体系结构

4.3.2 多 GPU 通信策略

当使用多个 GPU 对某一数据集进行运算时，如果这一数据集具有高耦合度，即在运行过程

中需要别的 GPU 上的数据，这时 GPU 与 GPU 设备之间就需要通信。如何高效地进行数据通信，是虚拟化环境下进行多 GPU 并行计算不可越过的难题。传统方式中，如果 GPU 与 GPU 之间需要通信，一般通过 CPU 中转，这样就会带来“路程”的开销。NVIDIA 发布了 GPU Direct 技术，使得多 GPU 之间通信更为方便，本小节针对 GPU 数据交换、传输机制进行分析。

1）单节点内多 GPU 通信

在 CUDA 早期版本中，主机端内存与 GPU 设备存储器被看做是独立的存储模块，每块 GPU 设备也被看做是独立的存储模块。当 GPU 之间需要通信时，首先通过 cudaMemcpy()函数将数据从设备端内存拷贝到主机端内存，再由主机端内存传输到目标 GPU 中的设备内存，传输过程如图 4.6 所示。从本身速度角度来说，主机内存可能代表了传输的瓶颈，特别对于比较老的处理器，仅仅通过主机内存在 GPU 间执行一些传输就能很快用尽全部主机内存带宽，由于它将和 GPU 传输一起竞争主机内存带宽，将会严重限制 CPU 的性能。

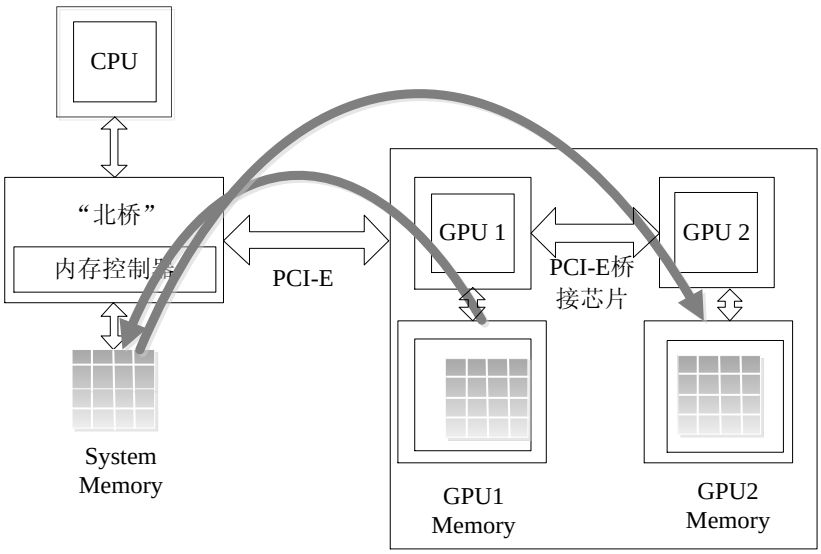

图 4.6 GPU 通过主机端拷贝

在 CUDA 4.0 及以后版本中，将多块 GPU 设备当做工作网络中的节点，在通信上各个节点之间是对等的。为了让两个 GPU 互相使用对方的内存，每个 GPU 必须显式地映射对方的内存，通过点对点的方式直接传输。将主机端内存与设备端的资源看做统一的存储器池，数据保存在主机内存空间之外，在 GPU 设备间通过 PCI-E 总线直接传输，不再需要主机端中转。这种方式不仅可以避免 CPU 与 GPU 的数据交换，而且能够高效利用 GPU 片上高速通信带宽。传输过程如图 4.7 所示。

2）多节点多 GPU 通信

在多个节点中，传统的 GPU 设备与网络的通讯模式采用总线控制芯片并经过 CPU 寻址的方式，GPU 存储器与网络设备之间没有直接的传输通道，需要通过源端和目的端上 CPU 内存

以及网络信道进行数据传输，所以通信复杂度高。针对这样的问题，NVIDIA 提出 GPUDirect RDMA 技术，在开普勒架构显卡以及 CUDA 5.0 以上版本中，允许使用如 SSD（Solid State Disk，固态硬盘）、NIC（Network Interface Card，网络适配器）、Infiniband 等第三方设备，直接访问系统内多个 GPU 上的存储，绕开 CPU 的主机内存，降低 MPI 从 GPU 内存接收/发送消息的延迟。但是这种方式对硬件要求很高，并且要在相同的系统中，所以目前在使用中有很大的局限性。

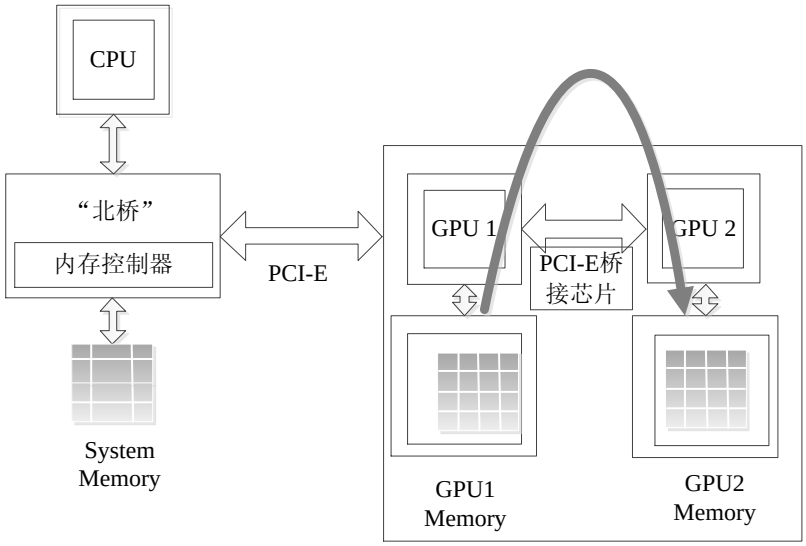

图 4.7 GPU 间点对点通信

4.4 虚拟化环境下多 GPU 并行计算的实现

4.4.1 实现方案

本文在虚拟化环境下多 GPU 并行计算的实现方案如图 4.8 所示，分为客户端和服务端。客户端即为虚拟机端，服务端即为可以访问到物理 GPU 设备的特权域或物理机。实现方案的框架分为横向和纵向，横向分为应用层和系统层，纵向分为物理机和虚拟机。客户端位于虚拟机应用层，服务端位于物理机应用层，系统层中存在有若干物理 GPU。

在客户端编写 CUDA 程序，采用动态库拦截的方式转移到服务端，服务端调用物理 GPU 进行运算，最后将结果返回给虚拟机，详细的执行流程在第三章已叙述，这里不再赘述。依据现有的多 GPU 并行计算的方法，将其引入到虚拟化环境中，使得 CUDA 程序在虚拟化环境下仍然可以调用多块 GPU 设备进行运算。如图 4.8，在虚拟机中配置了 vCPU（虚拟 CPU），虚拟内存和 vGPU（虚拟 GPU），vCPU 和虚拟内存都与物理 CPU 和内存相对应，通过物理 CPU 和内存映射，开辟虚拟的 CPU 和内存，而 vGPU 只是在虚拟机中配置所需真实 GPU 的逻辑映像，在虚拟机中并不能直接使用。当一个 CUDA 程序需要多块 GPU 设备运行时，可以使用单线程

控制多个 GPU 设备，也可使用多线程控制多个 GPU 设备。其中，使用多线程控制多 GPU 的方案需在虚拟机端开设多个线程分别映射到物理 CPU，使用物理 CPU 线程控制 GPU，一个线程控制一块 GPU 设备；当使用单个线程时，通过轮转控制 GPU 设备。在虚拟化环境下进行多 GPU 并行运算时，涉及到数据分解、数据计算和数据合并这三个步骤。

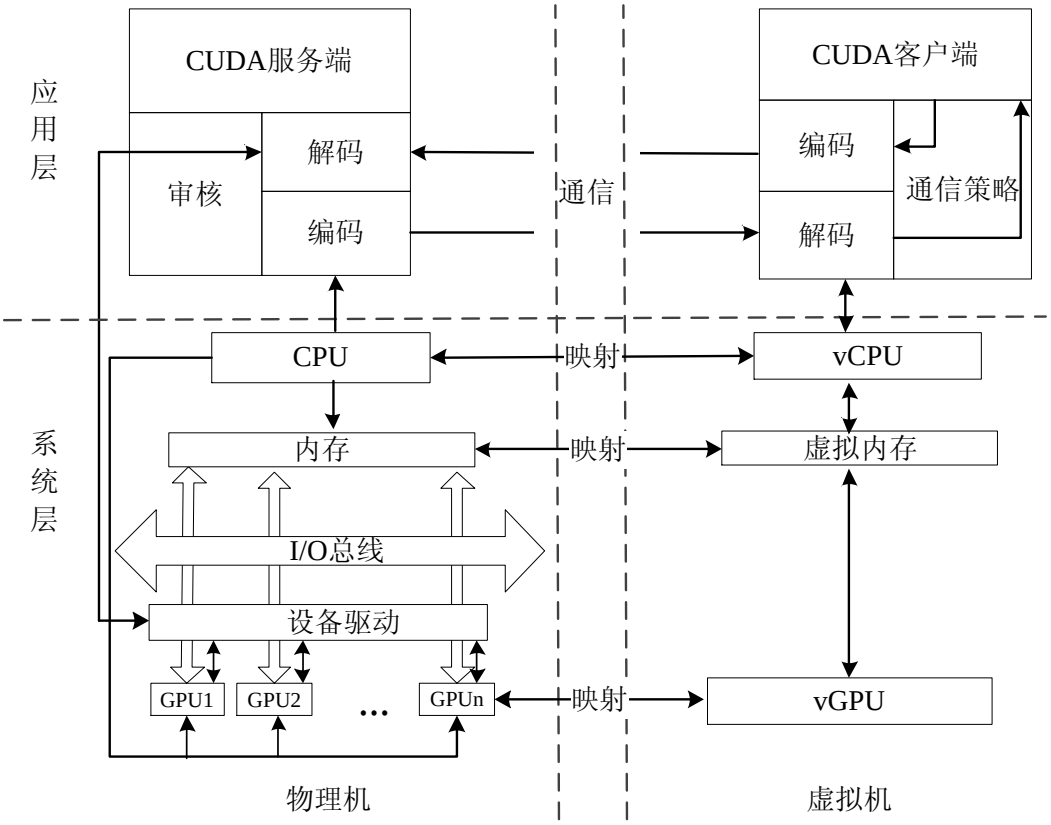

图 4.8 虚拟化环境下多 GPU 系统计算实现方案

数据分解是将程序（一般是比较大规模的）分解为多个小规模的程序，根据每块 GPU 设备的计算能力，对任务进行合理划分，交由多块 GPU 设备并行执行。对于计算能力相同的设备，对数据平均分配，交由各块 GPU 设备是最有效的方式。使用一个或多个 CPU 线程控制多块 GPU 设备，将划分好的数据传给多 GPU。假定 GPU 数目为 *N*，数据规模为 *M*，则每块 GPU 设备上分到 *M/N*。在传输时，通过设置主机端指针和设备端指针，主机端指针根据数据划分规模指向不同的起始位置，并通过 cudaMemcpy()函数从主机端线程的位置拷贝 *M/N* 的数据到每块目的 GPU 设备上。数据计算是在多块 GPU 设备上同时运行的过程，在运行过程中各块 GPU 设备可能是互不相干的，这种称为松耦合模式，各自独立计算，最后将结果回传给主机端。当然也有可能运行过程中需要别的 GPU 设备上的数据，这种称为紧耦合模式，在计算过程中需要通信，计算完成后对数据进行合并。数据合并是指所有运算在各自独立的 GPU 设备上运行完之后，结果回传给主机端并将所有数据合并，得到最终的结果。

本小节对上述所提方案及一些策略进行实验，实现在虚拟化环境下多 GPU 并行计算。实验

环境如表 4.2 所示。服务器配备 Intel(R) Xeon(R) CPU E5410（2.33GHZ），32G 内存和 1TB 硬盘，并包含 4 块 Tesla C2070 GPU 设备。

表 4.2 实验环境

实验环境	参数
操作系统	虚拟机 Ubuntu 12.04，物理机 CentOS 6.5
虚拟化平台	Xen 4.0.1，ESXi 5.0
GPU 平台	NVIDIA Tesla C2070*4
CPU 平台	Intel(R) Xeon(R) CPU E5410（2.33GHZ）
GPU 编译器	nvcc
CPU 编译器	gcc 4.4.7，g++ 4.4.7
CUDA 版本	CUDA 5.5
测试数据格式	输入单精度浮点数——输出单精度浮点数

4.4.2 数据传输优化

为了比较不同的数据传输模式，在本文的实验平台上，使用 SDK 中 bandwidthTest 测试不同数据大小的传输速度。平台中的主板都兼容 PCIe2.0 x16 标准，理想带宽可以接近 8.0GB/s。实验结果如图 4.9 所示。

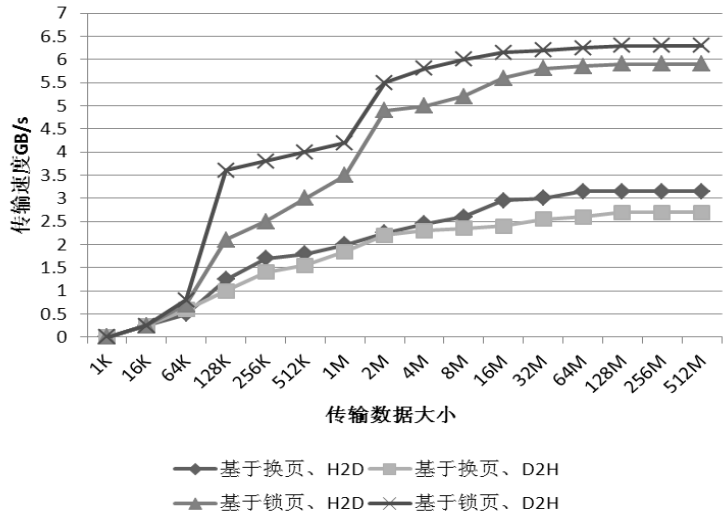

图 4.9 CPU-GPU 传输带宽测试

其中，H2D 表示从主机端到设备端；D2H 表示从设备端到主机端。可以看出，使用页锁定内存和可分页内存有显著区别，页锁定内存写入速度快 2 倍，读取速度快 2.48 倍。随着数据规模的增加，页锁定内存的传输带宽也在增长，最终达到 6.3GB/s。页锁定内存中主机端到设备端稳定带宽达到 3.3 GB/s，反向的设备端到主机端稳定带宽仅为 2.7GB/s。在主机端可以发现

一个特殊的现象，PCI-E 提供两个方向（来自设备以及传输到设备的）速度相同的全双工通信，但是设备端到主机端的带宽比主机端到设备端的带宽高 10%。这种差异是依赖于硬件的。要想获得接近峰值的带宽，传输数据的大小需要是 2MB 左右。事实上，只有传输数据大小为 16MB 或以上时才能获得绝对峰值带宽。所以，在传输数据时，可以考虑将多个传输组合起来以增加总线的整体使用带宽。

4.4.3 域间通信方式的选择

虚拟机系统的总体性能受虚拟机域间通信的速率和质量的影响，本小节通过对比分析域间通信方式，从通信方向、透明性、对标准协议的支持等方面全方位地分析，并得出在虚拟化环境下运用不同策略对 CUDA 应用程序性能的影响情况，尽量使用高吞吐量低延迟的域间通信模式。

基于 Xen 平台的高效通信方式在上文中已提及，主要有 XenSocket，XWAY 和 XenLoop。总结对比如表 4.3 所示。

表 4.3 Xen 域间通信方式

通信方式	通信方向	透明性	标准协议支持	动态迁移	共享内存安全性	延迟
XenSocket	半双工	不透明	不支持	不支持	未考虑	较长
XWAY	全双工	半透明	只支持 TCP	不支持	未考虑	较长
XenLoop	全双工	透明	TCP 和 UDP	支持	考虑	很短

XenSocket 和 XWAY 都支持 Xen 平台下的高速域间通信，但都存在一定的不足。XenSocket 功能比较简单，只能实现单向通信，并且不支持 TCP 和 UDP 协议，在使用时需要修改用户源程序。在数据量较少的情况下（小于 16KB）单向吞吐量较高，是 XenLoop 的两倍，但是其固定缓冲区大小无法满足大数据量传输的要求。XWAY 是一种面向 TCP 连接的通过拦截 Socket 底层调用来提供透明虚拟机域间传输的通信机制，较 XenSocket 有二进制兼容的优点，且实现全双工通信，但仍需修改操作系统内核，对用户不完全透明。这两种方式都未考虑到共享内存的安全性。XenLoop 相对前两种方式，实现了良好的透明性和支持动态迁移，提供了高性能的虚拟机域间网络回环通道，不仅支持 TCP 协议，而且对 UDP 协议也支持。XenLoop 实现了全双工通信，对共享存储区的访问实现了安全机制和同步机制，支持虚拟机的动态迁移，保证迁移前后数据发送的正确性。综上考虑，XenLoop 是 Xen 虚拟化平台下较为完善的虚拟机域间通信解决方案，且对用户使用透明，故本文虚拟化方案中不需要针对 Xen 平台设计通信程序，仅使用 Socket 方式实现，开启 XenLoop 加速域间通信。

由于 VMware 不开源，在 VMware 平台上虚拟化通信解决方案并不多，只有其商业产品 VMCI。它是一个在应用层工作的解决方案，在 VMware 平台上提供快速、高效的域间通信通

道。VMCI 的接口定义及编程流程类似于 Socket，相对于 XenLoop，不仅支持虚拟机操作系统为 Linux 的环境，而且支持 Windows 环境。本文在 VMware 虚拟化系列平台下均采用 VMCI 进行域间加速，在虚拟机中部署客户端组件，在 Workstation 中使用宿主物理机部署服务端组件，访问物理 GPU 设备。

4.4.4 虚拟化性能分析

在本小节，使用测试程序分析本文的虚拟化方案性能损耗情况。所使用的测试程序为 NVIDIA 官方 SDK 中能够反映不同的 CUDA 应用程序特征的基准程序，如 I/O 的访问，计算的负载以及数据的大小。表 4.4 显示了虚拟化环境与非虚拟化环境的比较。在 Host 环境下，每个 CUDA 程序运行在实际的物理环境上，能够直接使用物理 GPU 进行运算。此时，CUDA 程序的性能是 CUDA 框架下最好的效率，并将此运行时间作为标准，对比在虚拟化环境下的性能。统计特征有显存和内存之间需要传递的数据通量、任务类型、实际物理环境下执行时间（单位/s），虚拟化环境下的执行时间（单位/s），以及虚拟化环境相对物理环境的比值。

表 4.4 SDK 虚拟化环境与非虚拟化环境的比较

Benchmark name	Data transfers	Category	Host	Virtualization	Norm.
alignedTypes(AT)	413.26MB	memory	3.905	18.916	4.844
bandwidth Test(BT)	32.00MB	memory	0.867	8.693	10.026
blackScholes(BS)	76.29MB	memory	2.585	11.635	4.501
binomialOptions(BO)	0MB	calculation	4.726	5.230	1.106
clock(CLK)	2.50KB	calculation	0.278	0.312	1.122
matrixMul(MM)	79.00KB	calculation	0.489	0.538	1.100

实验结果表明，对于不同的应用程序，虚拟化环境相对于本地环境有一定的性能损耗。性能开销主要有以下几个部分：VMM 的开销（环境切换和虚拟机调度），伪库的开销，Socket 传输的开销等。从表 4.4 看出，对于数据量传输特别大的 I/O 密集型 AT 和 BS，执行时间超出 Host 一个数量级。相反，当数据传输量较小时，如计算密集型的 BO、CLK 和 MM，性能接近 Host。这说明， GPU 虚拟化中数据传输为主要的性能瓶颈。

4.4.5 数据松耦合交互模式下的多 GPU 并行计算

蒙特卡罗（MC）方法使用概率的方式求解近似解，目前被广泛应用于期权定价方案中[63]。由于采样次数多而导致计算开销大，故而可以采用并行的方法来加速蒙特卡罗方法计算，提高采样次数来减少误差。通过随机采样次数（为 N）得到 N 个可能预期价格，再对这 N 个预期价格的平均值作为某个时刻期权价格的预估值，得出最终结果。可以看出，整个过程中每次采样

以及处理过程都是相互独立的，适合大规模的并行化处理，故而可使用 GPU 高性能计算对其进行加速。同时，由于其独立性，可以使用多 GPU 并行计算的方法进一步提升性能，本小节主要验证在虚拟化环境下多 GPU 并行计算的加速效果。

整个过程如图 4.10 所示，将期权数量分割到多 GPU 上，每个 GPU 设备上进行随机数生成，采样计算以及采样结果处理这三个步骤。随机数的生成采用 CUDA 中的 CURAND 库，生成伪随机序列。在多 GPU 并行计算过程中，每个 GPU 设备的处理过程相同，不同之处在于生成的随机数不同。将计算划分到 *M* 个 GPU 设备上后，需要保证划分后的随机数要与原来未被划分的 *N* 次采样有相同的随机特性，即随机数质量和随机数分布类型一致，而对内容和顺序没有要求。

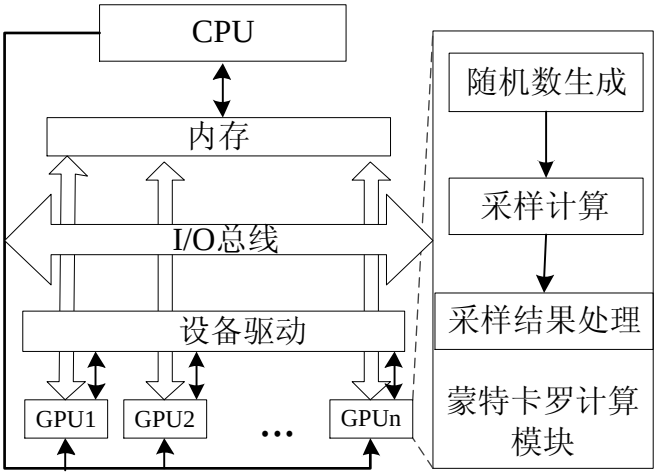

图 4.10 多块 GPU 设备处理蒙特卡罗过程

下面验证蒙特卡罗方法在多 GPU 并行计算的效率。随机采样次数越多，误差就越少。在本实验中，路径数量采用 524288 个，期权数目为 1024 个。分别使用一块 GPU 设备和 4 块 GPU 设备进行运算，实验结果分别如图 4.11 和图 4.12 所示。图 4.11 为使用单 GPU 进行运算得到的效果图，不考虑 GPU 生成伪随机数的时间，采用一个期权数量对应一个 block 的方法，则共计使用 1024 个 block。为了使 GPU 资源利用率达到最大化，通过使用 NVIDIA 提供的 CUDA_Occupancy_calculator.xls[64]设置不同的线程数目对 GPU 资源利用率进行计算，经过计算得到当 GPU 端的线程数目为 256 时 GPU 利用率最大（图 4.11 中②），蒙特卡罗方法运算时间为 595.503ms（图 4.11 中①）。当使用 Pthread 多线程的方式进行计算时，即一个线程控制一块 GPU 设备，在主机端开设 4 个线程，如图 4.12 中④所示（由于受界面篇幅限制未能全部显示），建立了 4 个 GPU 上下文，每个上下文可以同时进行运算，互不干扰。采用与单 GPU 同样数目的期权和路径数目，将 1024 个期权数量平均分到 4 块 GPU 设备上，每块 GPU 设备将有 256 个期权数目，因而每块 GPU 设备的规模降低四分之一。0 号 GPU 设备的蒙特卡罗方法计算时

间为 156.673ms（图 4.12 中①），其余几块设备的计算时间都与 0 号设备接近，且各设备同时运行。故加速比达到 3.801，接近于 GPU 的数目。最后将 GPU 设备同步，并将计算结果传回给主机端。由于需要计算每个期权的路径值最后的总和，因此使用了共享存储器，数组的大小与每个 block 中的线程数相同，即 threadsPerBlock，这样线程块中的每个线程都能将它计算的临时结果保存到这个数组中，然后启动并行计算的最后规约，如图 4.12 中③所示。

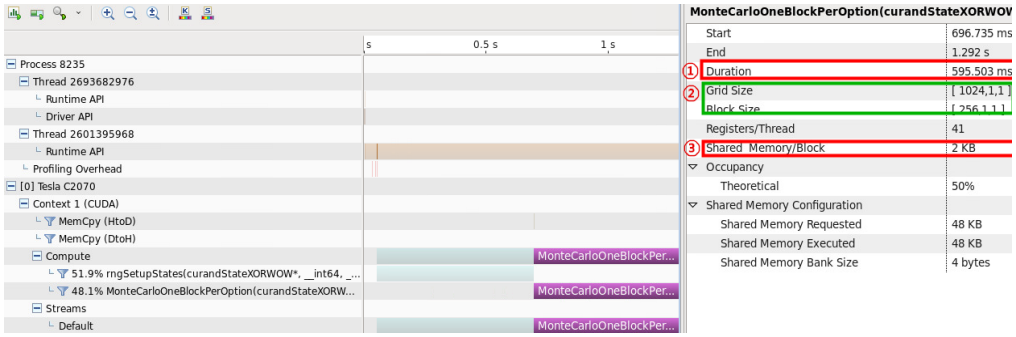

图 4.11 使用单线程单 GPU 设备对蒙特卡罗函数的计算

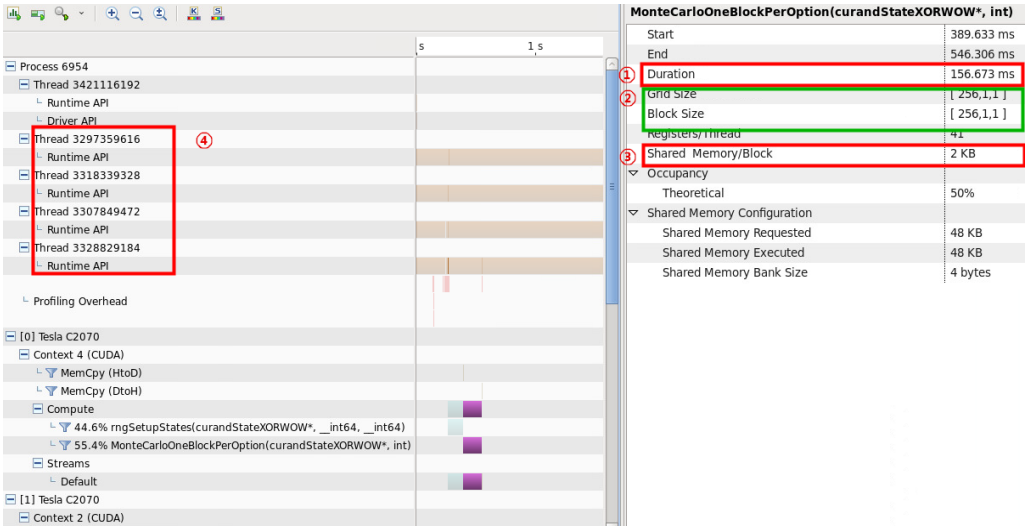

图 4.12 使用多线程方式 4 块 GPU 设备对蒙特卡罗函数的计算

同样，使用流处理的方式使得一个 CPU 线程控制多块 GPU 设备同时运算（图 4.14 中④），使用一块 GPU 设备和 4 块 GPU 设备的运行结果分别如图 4.13 和图 4.14 所示。当使用单块 GPU 设备时，蒙特卡罗的计算使用了 595.474 毫秒（图 4.13 中①）。使用 4 块 GPU 设备时，通过流处理的方式建立了 4 个 GPU 上下文，并且在每块 GPU 设备上分配一个乱序的流（图 4.14 中⑤），在 0 号设备上蒙特卡罗的计算函数使用了 156.953 毫秒（图 4.14 中①），其余设备的运行时间与 0 号设备接近。故加速比为 3.794，接近 GPU 数目。在使用流处理的方式时，最后需调用 cudaStreamSynchronize()函数执行同步，等待所有流执行完成之后再继续执行后面的代码。使用流处理的方式流之间相对无序。由于流之间复杂的并发性，使用流处理来开发算法可能会很难

以调试，CUDA 提供了一些同步的函数来便于调试和计时。

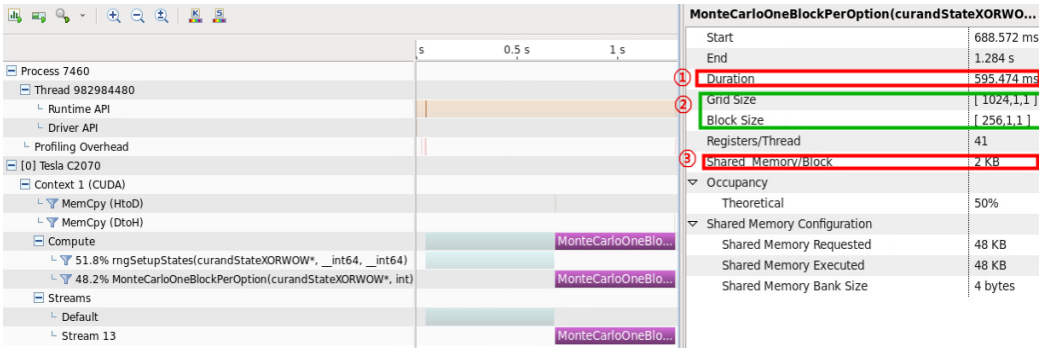

图 4.13 使用流处理的方式单 GPU 设备对蒙特卡罗函数的计算

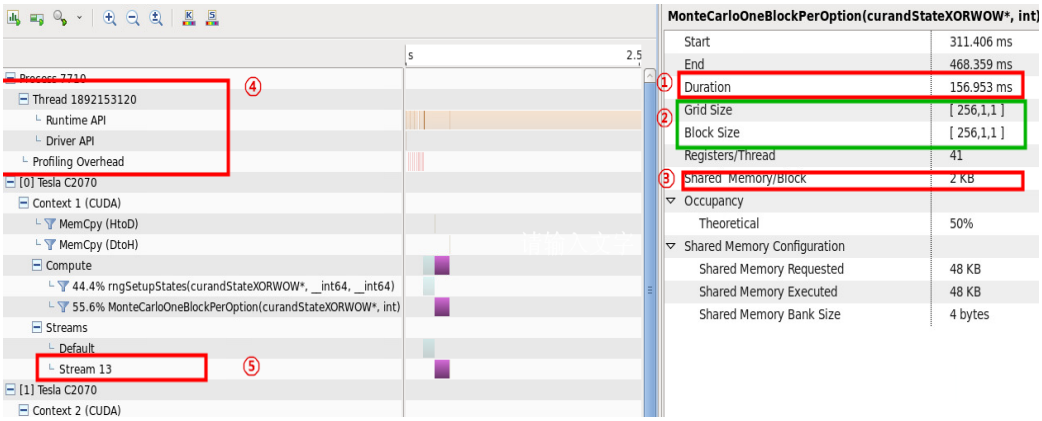

图 4.14 使用流处理的方式 4 块 GPU 设备对蒙特卡罗函数的计算

4.4.6 数据紧耦合交互模式下的多 GPU 并行计算

一些大规模的应用程序可能会受到计算机的存储容量和计算能力的限制，需要将程序分解成多个模块，进而分散在多块 GPU 设备上运算。在多 GPU 并行计算的过程中，这些模块可能需要 GPU 设备之间交换一些数据来完成原来的功能。本小节以 QFT(Quantum FourierTransformation，量子傅里叶)算法为例，说明紧耦合交互模式下的多 GPU 并行计算。

量子离散傅里叶变换定义如式 4.1[65]:

$$QFT: U_{QFT} \mid x >= \frac{1}{\sqrt{2^n}} \sum_{t=0}^{2^n-1} e^{2\pi itx/2^n} \mid t > \qquad (4.1)$$

其中，n 为量子态 x > 的量子比特位数；U_{QFT} 为一个么正操作符，QFT 为一个 2^n 维的么正变换，作用主要是将一个单态转为一个叠加态。量子计算仿真需要在目前线性的物理空间上提供指数级规模的计算空间和存储空间。因为在量子系统中每个量子态是 2^n 个基态的叠加，一次量子变换相当于 2^n 次经典计算。在进行高量子比特位的仿真时，所需的数据计算量之大，普通的 CPU 已经无法满足，而 GPU 强大的并行处理能力刚好满足这一需求。利用 GPU 强劲的计算

能力，将其移植到 GPU 上。对于计算一个 20 位的量子比特，其概率幅为 2^{20} 个，实部和虚部由两个浮点数来表示，每个浮点数占 4 Byte，故需要 2^{20}*2*4 Byte 的空间，而对于一般的显卡而言，存储空间有限，这时可以划分到多块设备上进行运算。

下面说明 QFT 算法在多 GPU 并行计算的效率。对于量子比特为 23 位的 QFT 算法，分别使用一块 GPU 设备和两块 GPU 设备进行运算，实验结果如图 4.15 和图 4.16 所示。

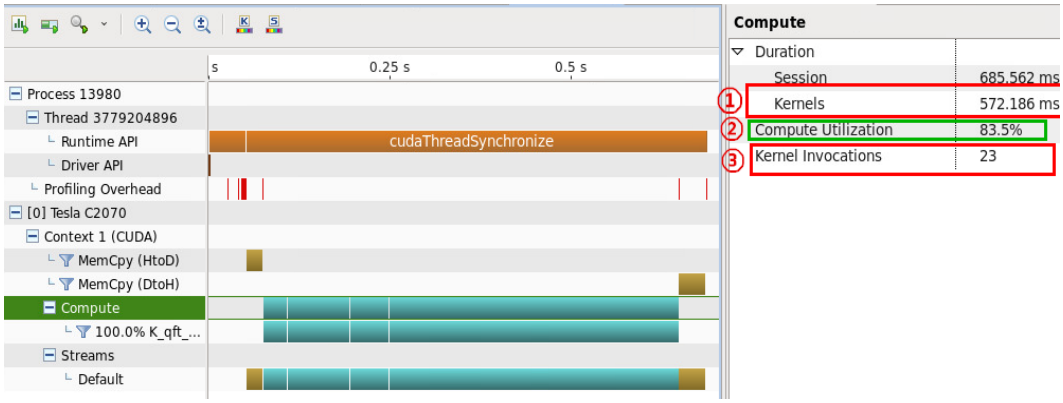

图 4.15 使用单线程单 GPU 设备 QFT 算法的计算

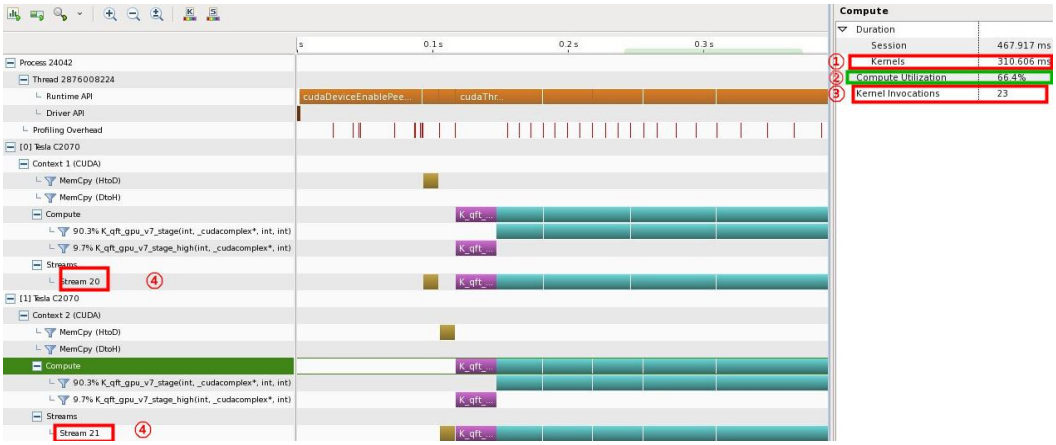

图 4.16 使用流处理的方式 2 块 GPU 设备对 QFT 的计算

图 4.15 为使用单 GPU 进行运算得到的效果图，设置不同的线程数目对 GPU 进行计算，通过计算得到当 GPU 端的线程数目为 512 时 GPU 利用率最大，QFT 计算一共使用了 572.186ms（图 4.15 中①），23 位的量子比特，需要调用 23 次 Kernel 函数（图 4.15 中③）。使用 2 块 GPU 设备时，通过流处理的方式建立了 2 个 GPU 上下文，并且在每块 GPU 设备上分配一个乱序的流（图 4.16 中的④）。由于在 1 号设备上运算时，需要到 0 号设备上读取对应状态的概率幅，每计算一次都要读取一次 0 号设备上的概率幅，共计需要读取 2^{22} 次，每次读取 8 Byte（实部和虚部）。由于读取次数太多，传统的通过 CPU 中转的方式已经不适合，在此使用了点对点的通信方式，即设备 1 从设备 0 上直接读取数据，这样不仅减少了“路程”的开销，而且可以利用

GPU 片上高速的通信带宽。在 1 号设备上，Kernel 运算的时间为 310.606ms（图 4.16 中②），这个时间包含运算的时间以及运算过程中需要从 0 号设备上读取数据的时间。在计算完成后，将结果传回主机端。

图 4.17 所示为不同量子位数的 QFT 算法分别由单块 GPU（Single Card）和两块 GPU（G_G）分别执行的计算时间，两块 GPU 设备之间的传输方式采用点对点的通信方式。随着量子比特位数的增加，单块 GPU 所计算时间的增长速度相对两块 GPU 设备所计算的时间快，并且由于量子计算的特殊性，时间呈指数级增加的态势。图 4.18 描述的为其加速比。

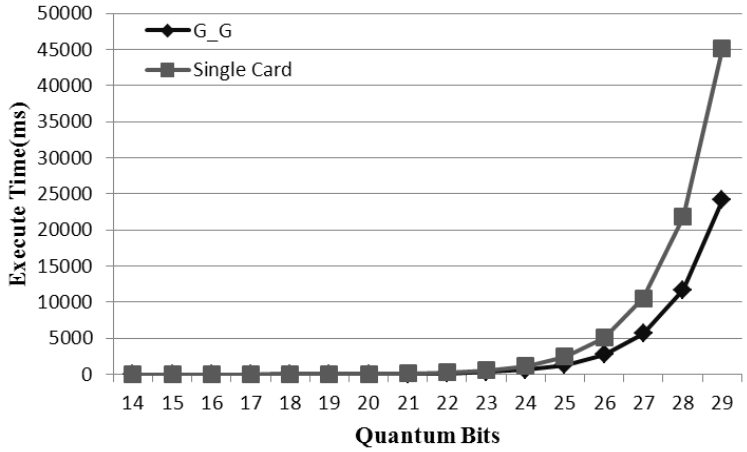

图 4.17 QFT 的多 GPU 和单 GPU 随量子比特位数变化的计算时间

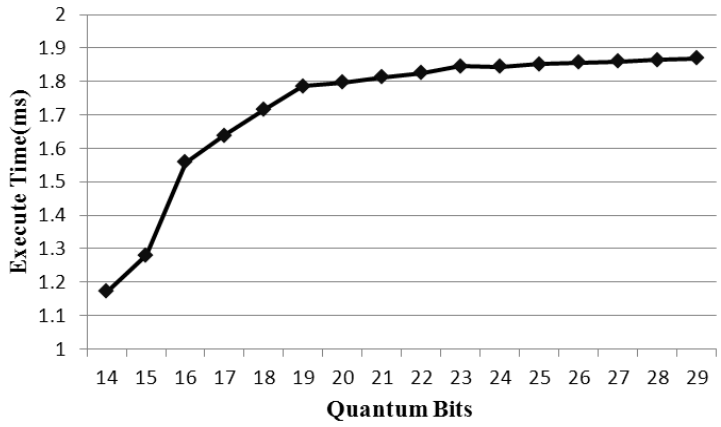

图 4.18 QFT 的多 GPU 与单 GPU 的加速比

由图 4.18 可以看出，虽然使用多块 GPU 设备并行计算可以提高并行计算效率，但是当量子比特位数较少时，加速比不是很高。这主要因为，在单块 GPU 设备上计算时，由于需要的计算量比在多块 GPU 运算上的计算量要多，因而 GPU 计算的利用率比分成多块 GPU 的利用率高（如计算 23 位量子比特时，图 4.15 中②计算利用率为 83.5%，当划分到两块 GPU 设备上时，1 号 GPU 设备的利用率为 66.4%，如图 4.16 中②所示）。量子比特位数较少时，QFT 的计算时间相对于整体的时间（数据传输+数据计算）的比例较低，因此加速比也较低；随着量子比特

位数的增加，分到每块 GPU 上的计算量也会增多，QFT 的计算时间相对于整体时间的比例逐渐提高。当量子比特位数达到 22 位时，加速比逐渐趋于稳定状态。一方面由于当量子比特位数达到一定的数值，能够充分利用 GPU，量子比特位数继续增长时，并未有明显增长；另一方面，数据量达到一定数值时，能够充分利用 GPU 设备间的带宽。多块 GPU 相对单块 GPU 的加速比增加，说明了计算时间相对于数据传输的时间而言，计算时间占据主导地位。

4.5 本章小结

本章首先对多 GPU 并行计算技术进行研究，使用多线程的方式用一个 CPU 线程控制一块 GPU 设备，或者使用流处理的方式用单个 CPU 线程控制多块 GPU 设备。为提升程序的性能，探讨了 GPU 多层次存储技术与优化技术。由于在虚拟化环境下会存在很大的通信开销，本章探讨了在虚拟化环境下多 GPU 并行计算的通信策略，对现有的虚拟机域间通信的优化方案从体系结构和底层实现方面进行了详细介绍和对比，选取 GPU 虚拟化环境下的最优通信策略。对于多 GPU 环境下运算数据耦合性很大的程序，研究多 GPU 之间的通信策略。最后给出虚拟化环境下多 GPU 并行计算的实现方案，并用实验验证了该实现方案的有效性和可行性。

第五章　虚拟化环境下多任务 GPU 资源共享

5.1 研究基础

5.1.1 集群管理系统

集群是将若干台分散的同构或者异构计算机通过网络相互连接而形成的计算机集合，在可视化集成开发环境以及并行程序设计的支持下，统一管理集群中资源，并行处理，统一调度，实现并行且高效的系统，达到超级计算机和并行机的计算效果[66]。从上面的叙述可以看出，仅仅由网络将分散的计算机相互连接起来，并不能称之为集群，还需对这些计算机进行统一管理，这样的管理系统称之为集群管理系统。

集群管理系统对各个处于高度自治的计算资源统一进行管理，旨在强调任务、资源的全面分布。集群管理系统与分布式计算、并行计算、网格计算和云计算密切相关，集群的很多技术为这些计算提供了解决方案。集群管理系统应具备以下功能：1）资源管理：监控系统资源使用情况并根据需要分配合适的系统资源，此处资源是个很广泛的概念，包括各种设备、程序和数据等；2）监控：监控集群内每个节点的状态；3）作业管理和调度器：一般将用户发来的作业排成一个队列，根据资源可用性、资源需求、作业类型等来执行作业调度，一方面要决定作业的调度次序，即确定作业在队列中的排列，另一方面，对作业所带的资源需求，根据一定的策略，把合乎要求的资源分配给作业。通过作业调度和资源管理可实现资源共享，有效地支持不同位置的用户能够共享系统中的信息和软硬件资源。目前在普通集群中已经有很多成熟的产品，如 LSF、PBS、SGE、LoadLeveler 等。

集群管理系统在目前的科学计算及商业应用领域都有着广泛的应用。2009 年后，GPU 通用计算受到越来越多的关注，新建的高性能计算机集群上都采用 GPU[67]，但针对 GPU 集群管理的软件大多属于商业产品，并不对外公开具体细节。

5.1.2 负载均衡关键技术

集群计算系统大多都是多用户分时共享的系统，系统主要目的是将独立的计算机通过网络互连，形成一个对用户透明的系统，实现系统范围内的资源共享。资源的有效共享是通过作业调度技术实现的，需对集群提供负载均衡和调度的服务，只有均衡的负载才能最大限度地使用资源，从而提高系统整体效率。从系统角度来看，资源使用率越高，作业的总体周转时间越少，资源共享的效果就越好；从用户角度看，作业响应时间越少，系统效能就越好。

负载均衡是一种计算能力共享的方法，在并行计算系统内，根据不同服务器的运行状况和

性能监测，找到一种任务对计算节点的映射关系，将来访任务以高效、透明、经济的方式分配到多个计算节点间，达到服务器间负载均衡的效果，从而提高集群吞吐量和系统利用率。负载均衡包含两方面含义，第一，将大量任务并发发送到多台计算节点分别处理，不同的服务器响应不同的任务；第二，复杂的任务被分成若干个小任务，分配到多台设备上并行处理，当每个节点处理完毕后，将结果汇总返回给用户。根据策略的不同，一般将负载均衡分为静态负载均衡和动态负载均衡[68]。

静态负载均衡算法利用集群的统计信息以概率的方式向计算节点分配任务，在任务分配之前，已经有相应的分配方案，这个方案可能考虑到当时负载状况，一旦设定就不考虑以后负载的变化，确定了任务具体在哪个节点运行，直至任务结束都不能更改。这种算法简单并且开销小，不必监测节点负载状态，也无需考虑将任务从过载节点上迁移引起的开销。但是这种方法缺乏灵活性，未考虑到任务之间的差异性，只能根据经验进行大致评估，适用于应用相对固定的专用系统。常见的静态负载均衡算法有轮转算法，随机算法，遗传算法和模拟退火等。

动态负载均衡[69]算法跟静态策略不同，在系统运行过程中，不断计算网络中每个计算节点的负载，交换系统的状态信息，动态地将任务加载到每个计算节点。调度贯穿整个系统的执行过程，根据不断变化的负载值，使程序的执行时间和调度时间最小。动态负载均衡策略相对静态策略具有动态、可伸缩的优点。但是这种算法会造成很多额外的开销，如信息收集和分析，并且信息会存在一定的误差，不能实时反应节点状况。如果调度得当，这种开销相对于整个系统的改善还是可以接受的，在实际应用中往往比静态负载均衡更加有效。

在虚拟化环境下，虚拟集群中存在大量虚拟资源，数量庞大且动态变化。由于物理服务器处理能力有差异以及任务的不确定性，导致虚拟集群中资源的负载失衡。虚拟化环境下负载均衡弹性资源管理有助于集群的垂直扩展和提升并行化处理能力，根据负载动态分配资源给多个任务，充分融合资源管理和负载均衡，提高系统整体能力。

5.2 基于 GPU 虚拟资源池的映射

5.2.1 GPU 虚拟资源池

目前，集群里的 GPU 计算资源复杂多样，这些 GPU 处理能力各有不同，并且每个计算节点里的 GPU 数目不定。对于这种资源差异性的考虑和资源不同处理能力的考量还比较缺乏，如何将任务合理地调度到这些差异化资源，并且保证资源的负载均衡，提高系统资源的利用率，是有待解决的问题。虚拟化技术可以将资源集中，形成一个动态分配的资源池，能够对资源动态伸缩，并具备可靠性和复原能力。为了满足虚拟机客户端对资源的不同需求，达到软硬件解耦和资源共享的目的，本文在虚拟化服务层维护多个 GPU 计算资源，将多个物理的 GPU 计算资源整合成虚拟的计算资源池，根据虚拟机客户端需求动态分配不同的计算资源，并实现逻辑

硬件资源的“隔离，划分，整合”。

隔离性体现在提供给虚拟机客户端的 GPU 计算资源之间不能相互影响，即使是同一个物理 GPU。及时更新服务端的硬件状态，并同步到虚拟机客户端的虚拟 GPU 上，避免不同的虚拟机之间相互干扰，宏观上来看每个虚拟机都有各自的 GPU 资源，在微观上，按顺序为每个虚拟机提供完整的 GPU 资源映像。划分体现在提供给虚拟机客户端的 GPU 是 GPU 资源池中的一个实际物理硬件的子集。根据用户的需求，可能会需要不同数目的 GPU，将物理资源空间划分，给虚拟机客户端提供相应的 GPU 资源。在实际物理机器的服务端，可为不同的客户端分配服务线程，能够实现多个虚拟机客户端共用一块显卡，也可实现使用多块显卡。整合性体现在借鉴传统的硬盘池和内存池技术，将多个物理 GPU 资源映射为单一的虚拟 GPU。

将 GPU 资源合理分配给虚拟机客户端，物理 GPU 资源处于资源提供者的角色，而虚拟机客户端处于资源消费者的角色，虚拟资源池则是维系两端角色的中间件，消耗集群中的 GPU 资源，提供给虚拟机客户端。GPU 虚拟资源池横跨整个虚拟集群，当 GPU 资源加入到虚拟集群中，会被虚拟资源池监测并统一调度分配。当从集群中移出，则资源不能被资源池和其子对象使用。GPU 虚拟资源池是以资源使用-回收的工作模式运作，而非分配-销毁的模式运作方式。GPU 虚拟化资源池不仅实现根据用户需求的弹性扩充，并且具备统一监控调度和分配的能力。

5.2.2 虚拟资源映射

虚拟化的一个重要的功能就是设备复用，VMM 模拟底层的硬件，为客户操作系统提供访问虚拟设备的接口。当用户使用虚拟机时，会造成这样一个假象，似乎都独占使用底层的设备。而实际上，虚拟机管理器将单一的、独占设备分解成多个可复用的设备，向客户虚拟机提供虚拟的设备映像。由于 CUDA 等通用计算框架在驱动层是不透明的，所以 GPU 虚拟化方式跟一般的 I/O 设备是不同的。

GPU 设备的 Tesla 方案，可支持 CUDA 三种不同的计算模式：Defalult 计算模式（多个主机端线程分享一块 GPU 设备），Exclusive 计算模式（任何时间都只允许一个主机端线程使用设备）和 Prohibited 计算模式（不允许任何线程使用 GPU）。

虚拟化资源管理层根据虚拟机请求以及集群中资源的特性，有不同的映射关系，建立物理资源和虚拟资源之间的映射关系。通过映射，虚拟资源与物理资源相关联。物理-虚拟资源映射关系分为三种：

1）One-to-One（1:1）映射关系

直接分配设备模型就属于 1:1 映射关系，是指将硬件平台中的一个 GPU 设备映射为单一的虚拟 GPU，并分配给一个虚拟机，而其他虚拟机不能对该设备进行访问，如图 5.1 所示。1:1 映射关系给客户操作系统能够访问真实物理设备的假象，就像 VMM 不存在一样。从本质上说，

直接分配模型是一个不参与虚拟化的真实物理设备，客户操作系统对 GPU 的设备访问直接传送给物理 GPU 设备。客户操作系统无法“感知”到底层 VMM 存在，无法共同来实现对外部设备的高效访问。

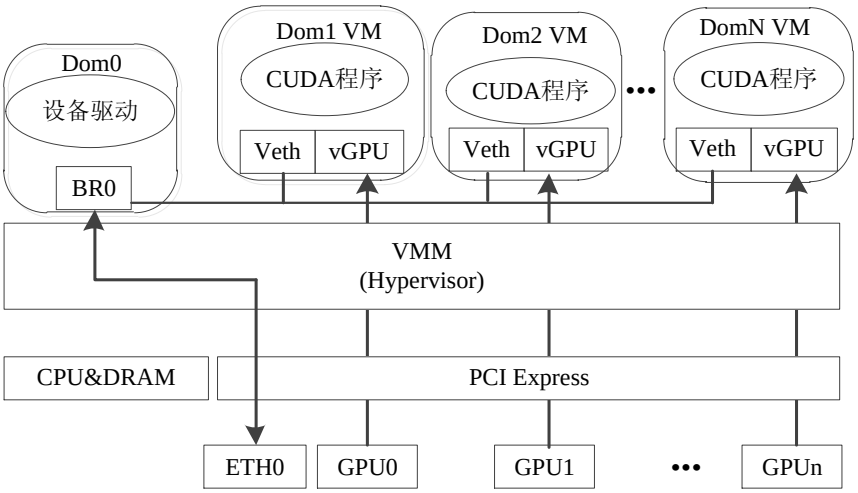

图 5.1 1:1 映射关系

2）One-to-Many（1:n）映射关系

在 1:n 映射关系中，是指将硬件平台中的一个 GPU 设备映射为多个虚拟 GPU，并分配给多个虚拟机，如图 5.2 所示。这里的虚拟 GPU 与通常意义上的虚拟 CPU 并不是同一概念，此处虚拟 GPU 主要用来维护 CUDA 相关软硬件状态。根据 CUDA 的 Defalult 计算模式，多个虚拟机可以分享一块 GPU 设备。

在这种模式中，用户使用 GPU 资源时，分别在软件层次和硬件层次竞争资源。软件层次是指 CUDA 的虚拟地址空间。在 CUDA 4.0 以后版本中支持统一地址空间，能够支持上万个的并发线程。如果每个内核启动的线程块的数量足够小，将导致内核并发执行，目前的 Fermi（费米）架构最多可实现 16 个 Kernel 同时执行。在硬件层次，由于共享存储器和寄存器资源属于比较稀缺资源，并且与 Kernel 的生命周期相同，所以当不同虚拟机同时运行 Kernel 时才会产生竞争。当使用全局存储器时，在一个 CUDA 程序的使用周期内，包含从 cudaMalloc 到 cudaFree 的过程中，全局内存都没有释放。在这种模型中，主机操作系统为每个虚拟机提供一个服务线程，服务端组件还需创建计算线程，每个 GPU 设备对应一个计算线程，计算线程直接控制物理 GPU。当多个虚拟机请求同时在一块 GPU 设备上时，由 CUDA 运行时库来处理不同线程之间的冲突。

但是，在实际使用中，并非所有 GPU 设备和通用计算架构都支持 CUDA 的 Defalult 计算模式，因而这种虚拟化方案缺乏通用性。特别是对并发性要求很高的 CUDA 应用，强制使用 Exclusive 计算模式时，这种映射方式就更加不能使用了。下面将引出 n:n 的映射模式。

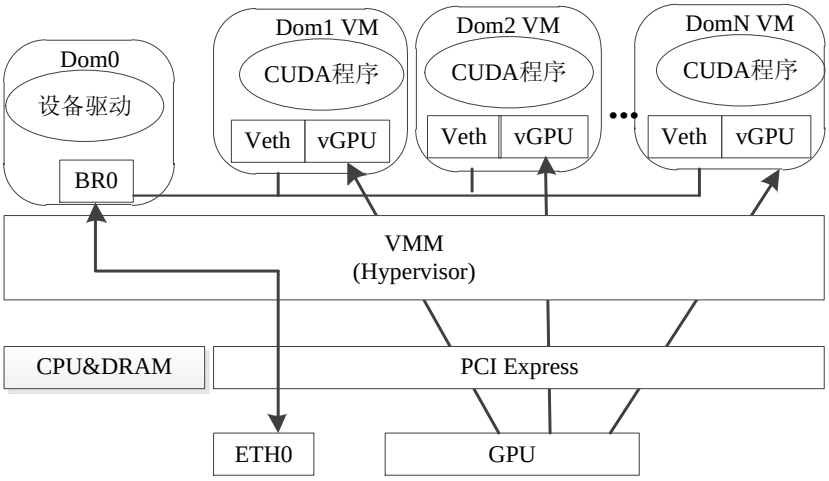

图 5.2 1:n 映射关系

3）Many-to-Many（n:n）映射关系

在 n:n 映射模型中，是指在硬件平台中存在多个 GPU 设备，这些设备可能在同一节点，也有可能在不同的节点中，将多个 GPU 设备映射为多个虚拟 GPU，并分配给多个虚拟机。在这个模型中，包含了 1:n，n:1 的关系，即在设备允许和不用 Exclusive 计算模式的 CUDA 程序时，可以使用 1:n 的关系，当虚拟机客户端的 CUDA 程序请求使用多个 GPU 设备运作时，即蜕化成 n:1 的关系，这个主要在第四章多 GPU 并行计算中重点阐述。当使用 n:n 映射关系时，应对多个虚拟机请求，如何分配多个设备，成为主要问题。下面将具体阐述如何充分调度虚拟集群中多块 GPU 设备，计算其处理能力，并将计算出的资源加入到 GPU 虚拟资源池中，进行统一地调度与分配，从而实现混合资源的负载均衡分布，保证计算性能，提高资源易用性，并降低使用成本。

5.3 一种基于动态负载量多负载状态的 GPU 负载均衡算法 DMLS-GPU

为充分利用虚拟化环境下多 GPU 计算资源，解决资源分配不均的问题，综合考虑 GPU 设备的处理能力及负载状况等因素指导任务的分配，本文提出了 DMLS-GPU 算法，利用负载权值从 GPU 虚拟资源池选择分配 GPU 设备的集合，保证性能高的 GPU 设备分配到较多的任务负载，使得所有 GPU 设备各尽其能。

5.3.1 基于 GPU 的负载评价算法

本小节介绍了影响 GPU 性能的关键指标，并探讨了 GPU 性能评估理论。GPU 的运算效率与 GPU 本身的计算核心数量和处理器内核频率的乘积即为 GPU 的浮点性能大致成正比，且计算效率受 GPU 全局内存的大小影响也很大，如果任务的规模大于 GPU 可用全局内存的大小，则该计算在 GPU 上不能一次完成，会导致额外的通信开销，因此数据通过 PCI-E 传输的开销也要考虑在内。此外，在集群环境中，网络传输的开销是不可忽视的，GPU 设备的负载与网络接

口带宽 *NW* 成反比。故负载信息中主要包含处理计算核心的数量、处理器内核频率、全局内存、PCI-E 总线带宽及网络接口带宽，同时将任务规模纳入负载评价的考量范围，并根据这些因素对 GPU 资源进行调度。

本文为服务端的所有 GPU 设置一个综合负载评价值 LDg，如式 5.1:

$$LD_g = \sum_{i=1}^{N}(\alpha_1 * \frac{CN_i}{PN_g * PF_g} + \alpha_2 * \frac{Scale_i}{GM_g} + \alpha_3 * \frac{Scale_i / T_1}{BW_p} + \alpha_4 * \frac{Scale_i / T_2}{NW}) \quad (5.1)$$

参数定义如表 5.1 所示。

表 5.1 调度模型主要参数

参数名称	意义
i	GPU 上第 i 个任务
N	当前 GPU 上的任务数
$Scale_i$	GPU 上第 i 个任务的规模大小
LDg	单块 GPU 设备综合负载评价值
CN_i	任务每秒计算的浮点次数（Flops）
PNg	GPU 的计算核心的数量
PFg	处理器内核频率（GHz）
GMg	GPU 的全局内存（G）
BWp	PCI-E 的总线带宽（GBps）
NW	每个节点的所有网络接口带宽（GBps）
T_1	数据通过 PCI-E 的接口拷贝的时间（s）
T_2	数据通过网络传输的时间（s）
α_1	任务计算能力强度的影响因子
α_2	全局存储器使用率的影响因子
α_3	PCI-E 带宽使用率的影响因子
α_4	网络通信带宽使用率的影响因子

其中，任务规模 $Scale_i$ 由 CUDA 应用程序提供在接口参数中。 Master 节点维护着一张全局负载信息表，每次进行任务分配时，都要根据负载表的信息来决定 GPU 设备的选取。α_1、α_2、α_3、α_4 为影响 GPU 性能关键指标的权重系数，它们的值根据基于 AHP 多属性决策方法确定，见 5.3.2 节。*PNg*、*PFg*、*GMg*、*BWp*、*NW* 等均由 Slave 节点中的 LIM 所提供。CN_i、T_1、T_2 根据不同的任务类型得到不同的值。采集 GPU 设备的负载信息根据周期 *T* 定期更新，因而综合负载计算也是周期性的。当周期 *T* 很短时，虽然可以更为精确地反映 GPU 设备的状态，但是频繁的地执行负载计算及传输会带来额外的负载。在本文中，收集 GPU 设备的负载时间间隔设为 1 秒，在系统开销及使用效果上可以达到较为合理的平衡。

Master 节点依据 CUDA 任务的调度原则返回 GPU 信息中 *LDg* 值最小的集合，*LDg* 值越小，说明当前 GPU 上的负载越小。此外，调度的另一原则是 CUDA 任务在本地节点上优先处理，这是由于虚拟化方案的通信机制决定的，以减少数据传输带来的性能开销。Slave 节点提交 GPU 设备信息时，不仅要提交本 GPU 所在服务器的 IP 地址和设备号，还需提供 GPU 的参数信息。在 Master 节点计算每个 GPU 负载值时，还需乘以一个权值系数，该权值系数表明任务在本地节点优先处理的程度，并加入表项中排序，得出负载较小的 GPU 设备集合，根据 CUDA 任务的需求，分配合适的 GPU 数目。

5.3.2 多属性决策的负载量确定

由于 GPU 性能的好坏是由多方面因素决定的，有些因素对总体性能影响可能大些，而有的可能小些。因此，在进行综合评判时，必须给出各个因素在总评价中的权重系数（重要程度）。目前，很多方案都采用经验赋值，后续通过实验不断调整值，这种方法有很大的盲目性。层次分析法（AHP）[70]是由美国的匹斯堡大学 Saaty 教授提出的一种将定性与定量分析方法相结合的多属性决策分析方法，根据前文介绍，引入多评价指标来确定 GPU 设备的负载量，而后根据负载量并设定阈值确定负载状态，适用于对决策结果难以直接准确计量的问题。为了计算各项成本评估指标的权重，本文根据少量定性信息直接构建判断矩阵并采用层次单排序计算方法计算各项评估指标的权重，主要计算步骤如下[71]：

第一步. 建立层次结构模型

将问题分解为元素的各个组成部分，把这些元素按属性不同分成若干个组，以形成不同的层次。按照目标层、准则层、方案层的形式进行分层排列，处于最上面的层次通常只有一个元素，一般是分析问题的预定目标或理想结果。如图 5.3 所示：

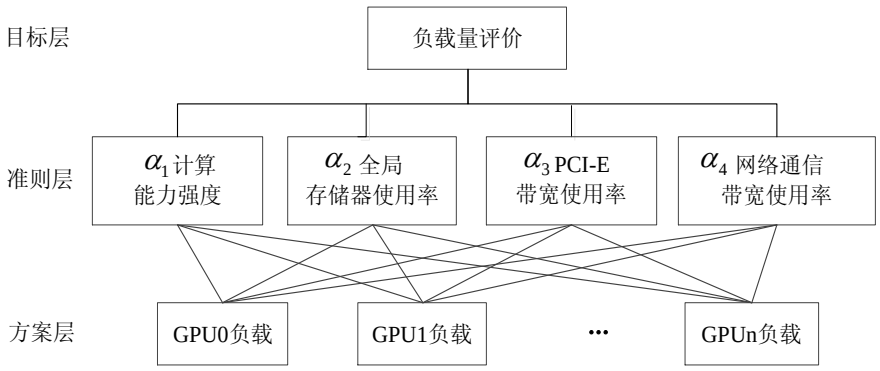

图 5.3 负载量评估指标层次结构

第二步. 构造判断矩阵

层次分析法的一个重要特点就是用两两重要性程度之比的形式表示出两个方案的相应重要性程度等级。如对某一准则，对其下的 n 个方案进行两两对比，并按其重要性程度评定等级。

记 a_{ij} 为第 i 和第 j 方案的重要性之比，a_{ij} 是要素 a_i 对 a_j 的相对重要性，表 5.2 列出 Saaty 给出的 9 个重要性等级及其赋值。

表 5.2 比例标度表

a_i 比 a_j	极端重要	强烈重要	明显重要	稍微重要	同样重要
量化值	9	7	5	3	1

在 AHP 中，为了确定 α_1、α_2、α_3、α_4 成本评估指标的权重，通常采用如表 5.2 所示的比例标度法。对于负载量 LDg，其计算能力强度比全局存储器利用率稍微重要一些，所以在相应的位置赋值为 2；而 GPU 计算能力强度与 PCI-E 带宽使用率相比则为一般重要，所以赋值为 3，比网络通信带宽使用率重要得多，所以赋值为 5；以此类推，得到准则层的判断矩阵如式 5.2 所示：

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}=\begin{bmatrix} 1 & 2 & 3 & 5 \\ 1/2 & 1 & 2 & 4 \\ 1/3 & 1/2 & 1 & 2 \\ 1/5 & 1/4 & 1/2 & 1 \end{bmatrix} \tag{5.2}$$

其中，$a_{ii}=1$，$a_{ij}=1/a_{ji}$，且 $a_{ij}>0$ (i, j = 1, 2, 3,4)。

第三步. 评估指标权重的计算

Saaty 通过求取最大特征值对应的特征向量而得到权值，为了从判断矩阵中提炼出有用的信息，达到对事物的规律性认识，为决策科学提供科学依据，就需要计算判断矩阵的权重向量。

根据步骤 2，计算判断矩阵 A 的特征向量，令式 5.3 等于 0：

$$\begin{bmatrix} 1\text{-}\lambda & 2 & 3 & 5 \\ 1/2 & 1\text{-}\lambda & 2 & 4 \\ 1/3 & 1/2 & 1\text{-}\lambda & 2 \\ 1/5 & 1/4 & 1/2 & 1\text{-}\lambda \end{bmatrix}=0 \tag{5.3}$$

计算得到 $\lambda_{\max}$=4.0211，λ_{max} 是判断矩阵的最大特征根。令式 5.4 等于 0

$$\begin{bmatrix} 1\text{-}\lambda & 2 & 3 & 5 \\ 1/2 & 1\text{-}\lambda & 2 & 4 \\ 1/3 & 1/2 & 1\text{-}\lambda & 2 \\ 1/5 & 1/4 & 1/2 & 1\text{-}\lambda \end{bmatrix}\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix}=0 \tag{5.4}$$

最终可计算得到评估指标权重向量 $\alpha=(\alpha_1,\alpha_2,\alpha_3,\alpha_4)^T=(0.509,0.288,0.204,0.080)^T$.

第四步. 一致性检验

在实际过程中，当判定矩阵的阶数 n>2 时，对两两重要性程度之比呈现出非等比的特性，可能会出现 *A* 比 *B* 重要，*B* 比 *C* 重要，而 *C* 又比 *A* 重要的情况，所以对判定矩阵的一致性检验非常有必要。判断矩阵偏离一致性条件应有一个度，为此，必须对判断矩阵是否可接受进行鉴别，这就是一致性检验的内涵。为了检验判断矩阵的一致性，则需要计算矩阵的一致性指标

CI、平均随机一致性指标 *RI* 以及随机一致性比例 *CR*。

计算一致性指标 *CI*，如式 5.5。*CI* 表示特征值 λ_{max} 相对于 λ_{max}=n (完全一致性时的最大特征值)时的偏离程度的描述。

$$CI=(\lambda_{max}-n)/(n-1)=0.007 \tag{5.5}$$

随机一致性指标 *RI* 表示平均一致性指标，通过多次计算各阶随机判断矩阵的特征根，然后取特征根的平均值作为平均一致性指标。它的值可以根据矩阵的阶数从表 5.3 中查得。

表 5.3 矩阵的平均一致性指标

阶数	1	2	3	4	5	6	7	8	9
指标 RI	0	0	0.52	0.89	1.12	1.26	1.36	1.41	1.46

判断矩阵的一致性指标 *CI* 与平均随机一致性指标 *RI* 的比值称为随机一致性比例 *CR*，记为式 5.6：

$$CR=CI/RI=0.008 \tag{5.6}$$

如果一致性指标 $CR\le 0.1$ 时，则可认定判断矩阵的一致性是可接受的；如果 $CR>0.1$，则需要重新调整判断矩阵，直到判断矩阵的一致性可以接受。至此，我们获得了层次单排序，如表 5.4 所示，由式(5.6)可计算出 $CR=0.008<0.1$，因此，所求结果满足一致性检验，是可以接受的。

表 5.4 准则层指标权重层次单排序表

因子	计算能力	显存资源	显存带宽	网络带宽	权重 W
计算能力	1	2	3	5	0.509
显存资源	1/2	1	2	4	0.288
显存带宽	1/3	1/2	1	2	0.204
网络带宽	1/5	1/4	1/2	1	0.080

5.4 实验及性能分析

5.4.1 实验环境

实验的系统体系结构为：虚拟化用户层（4 个虚拟机构成）、虚拟化资源服务层（5 台服务器构成）以及虚拟化资源管理层（位于其中一台服务器）。其中虚拟化资源服务层为四台服务器，每台服务器都配备 Intel(R) Xeon(R) CPU E5410（2.33GHZ）处理器，32G 内存和 1TB 硬盘，并包含 4 块 Tesla C2070 GPU 设备，服务器间的连接使用高速的 Infiniband 交换网络。在服务器端部署服务端组件，并且在指定的时间间隔内，向 Master 节点发送 GPU 设备的负载注册请求。虚拟化用户层为 4 台虚拟机，虚拟机通过 Vmware Workstation 开启，具体配置为两个 vCPU、1G 的内存以及 20G 的外存。虚拟化资源管理层位于其中一台服务器，配置与服务器相同。

5.4.2 可扩展性测试

本文的架构要求能够自动发现计算设备，动态加入 GPU 计算资源，GPU 计算资源可能会随着平台的不断扩建而增多，也可能因硬件损坏而将部分节点撤销。可扩展性一般指问题的规模随着系统规模的增长仍能保持合理的性能。假定，在集群中 GPU 设备的数目为 N，每块 GPU 设备上的工作负载为 W，W 是 N 的函数，即 $W=W(N)$。设系统的加速比函数 $S=(N,\ W)$，当 GPU 设备数目由 N 增长到 N'，则系统在 GPU 设备数目由 N 变为 N' 时，加速比函数由 S 变为 S'，则可扩展性为 $Scale(N,N')=(S'/S-1)/(N'/N-1)$，其中 $N'\neq N$。$Scale(N,N')$ 的含义为当系统中 GPU 数目增加时，加速比的增加倍数 $(S'/S-1)$ 与 GPU 设备的增加倍数 $(N'/N-1)$ 之间的比值。当 GPU 数目由 N 变为 N' 时，$Scale(N,N')$ 值越大，说明加速比增长的倍数就越大，则系统性能就越好。当 $N'>N>0$ 时，如果 $Scale(N,N')=1$，则表明此系统的加速比会随着 GPU 数目的增加而线性增加的，为线性可扩展的。证明如下：

$$Scale(N,N')=1 \Rightarrow (S'/S-1)/(N'/N-1)=1 \Rightarrow (S'/S-1)=(N'/N-1)$$
$$\Rightarrow S'/S=N'/N \Rightarrow S'=(N'/N)*S$$

在实验中，为了验证此架构的可扩展性，本文使用 CUDA SDK 里的 SimpleMultiGPU 进行验证。通过改变数据量，观察任务执行的变化。单 GPU 设备的存储和计算能力是有限的，通过不断的向计算数据集中增加重复数据以增加计算量，数量的大小与 GPU 的数量呈正比关系。观察整个计算任务在单位时间内执行的指令数，以此评估平台的可扩展性。结果如图 5.4 所示，随着数据量的线性增加和所使用的 GPU 数量的增加，计算任务中单位时间内执行的计算指令数呈现线性增长的趋势，说明该平台有良好的可扩展性。

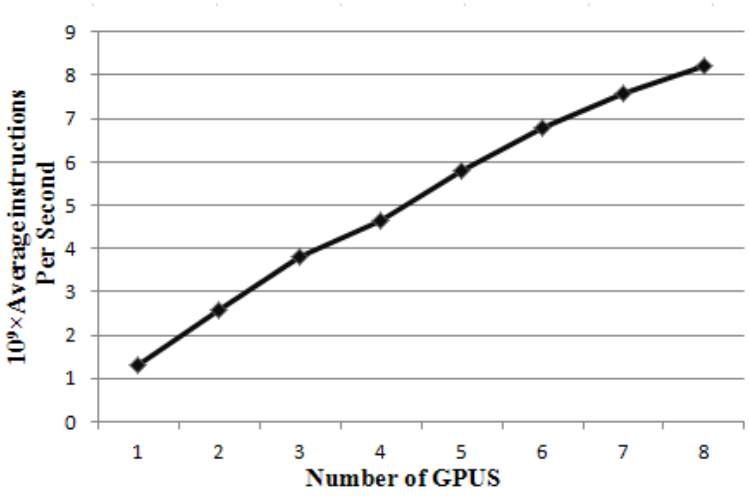

图 5.4 可扩展性验证

5.4.3 多个 CUDA 程序并发

在 1:n 映射关系中，一个物理 GPU 被映射到多个虚拟机，每个虚拟机都可以执行一个或多

个 CUDA 程序。多机并发是指同时执行两个以上的 CUDA 程序的计算能力，在一个 GPU 上下文并发执行多个 Kernel。Femi（费米）架构有 16 个 SM（流多处理器），所以最多可同时执行 16 个 Kernel。当然这些 Kernel 的并发执行有限制，比如全局内存的大小，block 的数量，thread 的数量等等。在本文的虚拟化环境下从三个方面验证 GPU 共享的性能。

1）相同的 CUDA 程序同时运行

图 5.5 表示同时运行同一个应用程序的两个实例，选用的实例为 4.4.4 节中 CUDA SDK 中的基准程序，分别使用独占模式（Exclusive Mode）和共享模式（Sharing Mode），独占模式是指一次只能运行一个 CUDA 程序，在一个 CUDA 程序运行时，另外一个 CUDA 程序只能等待，属于串行的运行方式，共享模式是指利用 Femi 架构以及 GPU 虚拟化的特性，多个 CUDA 程序在同一 GPU 设备上并发执行。在此选取 NVIDIA 官方 SDK 中的具有能够反映不同的 CUDA 应用程序特征的基准程序，如计算的负载以及数据的大小。

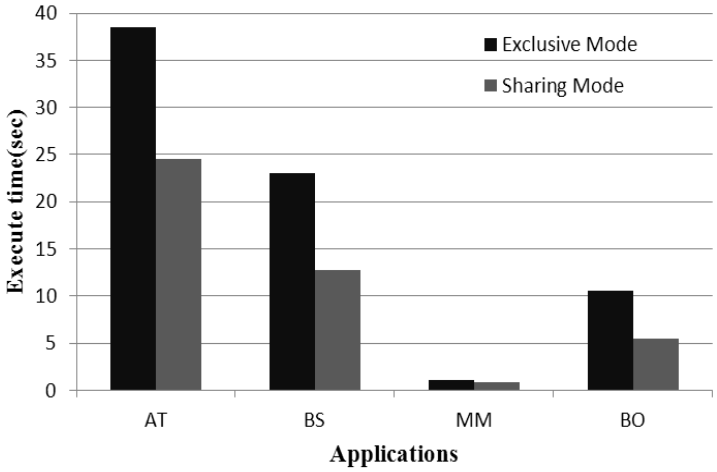

图 5.5 同一应用程序两个实例不同模式的对比

从图中可以看出，共享模式可以显著减少程序运行时间。这主要原因是独占模式是以串行的方式运行，而共享模式是在流多处理器中以并发的方式执行多个 CUDA 应用程序，提升 GPU 的利用率。并且，对于不同的应用程序，独占模式与共享模式之间的差距也不一致，这主要因为 CUDA 程序类型不同，当多个 CUDA 程序并发执行时，对全局存储器的要求等会出现不一致的情况，对于 AT 类型这种数据传输量很大的程序，它的差距也较大，而对于 MM 这种计算密集型的 CUDA 程序，差距较小。

2）与其他程序同时运行

在这个实验中，测试两个不同的 CUDA 应用程序同时运行，同样使用独占模式和共享模式。图 5.6 为使用 BO 与其他样例同时运行，查看他们各自运行时间。选择 BO 作为参考并发程序，是因为 BO 执行时消耗较少的 GPU 资源，能够和其他程序同时执行。Single 模式为任务在虚拟化环境下单独执行所需时间，Concurrent 模式为任务同 BO 并发执行所需时间。

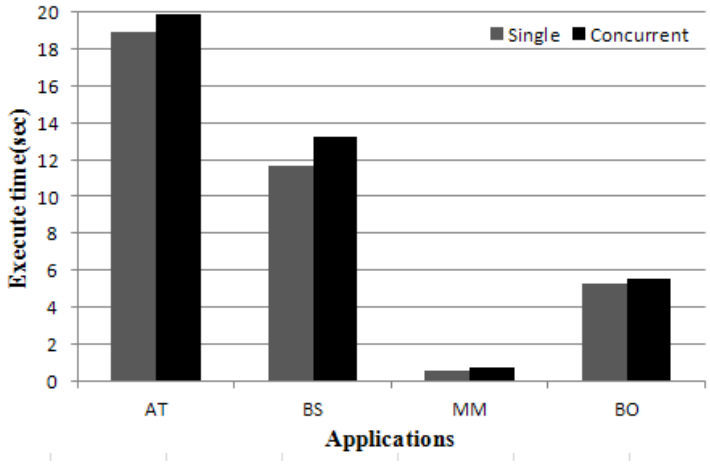

图 5.6 不同应用程序并发

从图中可以看出，在虚拟化环境下多个不同的应用程序使用共享模式同样可以使用一个物理 GPU，且时间要比独占模式少。具体原因与同一应用程序的两个实例类似，在这就不具体阐述。各个应用程序之间时间差相差很大主要原因是任务类型不同，对 GPU 资源的消耗也不一致。

3）共享的可扩展性

上文提到，Femi 架构最多支持 16 个 Kernel 函数并发执行，在本次实验中，增加应用程序的数目，比较使用独占模式和共享模式。图 5.7 显示了程序运行结果。在大于 4 个应用程序的情况下，将上例中 4 个应用程序复制一份，即使用 8 个应用程序时使用 4 个 SDK，每个 SDK 使用两个实例。由于最多支持 16 个，所以应用程序数量扩展为 16 个。

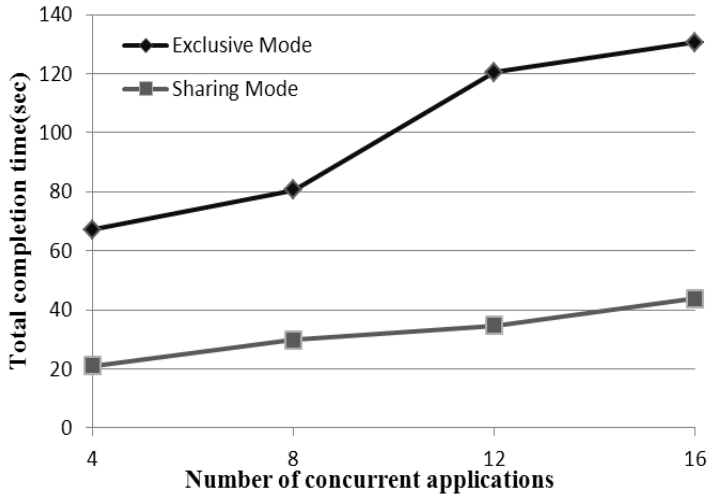

图 5.7 应用程序数目的增长总体执行时间

从图 5.7 中可以看出，使用共享模式随着 CUDA 应用程序数量的增加，执行时间呈现出线性增长的趋势；而使用独占模式时，应用程序的执行时间突然上涨。主要原因是，在独占模式中，每个应用程序都要等待前面的程序执行完才可执行下一个，每个应用程序的执行时间不定，

所以不能呈现出线性增长的趋势。

5.4.4 多机环境下的性能

针对本文的调度算法，本文从两个方面验证其可行性及高效性。一是使用同种类型的任务，在各虚拟机上随机生成一定的任务，计算其平均周转时间；二是使用多种不同类型的任务，计算随着时间的变化，查看 GPU 设备的实际利用率。

针对第一种方法，使用 DFT（离散傅里叶变换）程序进行测试，在一定的时间内，虚拟机随机生成一定数量的 DFT 任务，此处 DFT 的输入点数的规模为 30000 点，图中显示的其平均周转时间。图 5.8 为使用本文提出的 DMLS-GPU 算法和使用 Random 算法的任务调度的比较，Random 算法是指通过取随机数的方式，随机地从系统中调用 GPU 设备。

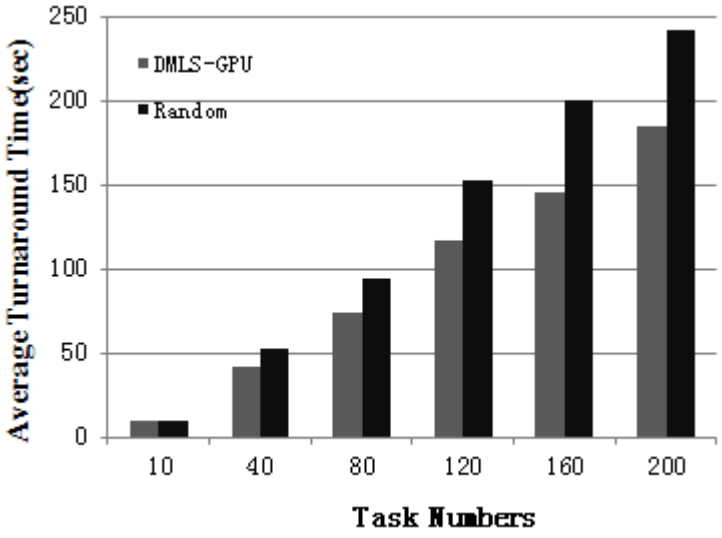

图 5.8 GPU 虚拟化随机生成任务的完成情况

从图 5.8 可以看出，随着提交任务数量的增多，使用 DMLS-GPU 算法的周转时间明显小于使用 Random 算法的周转时间。这是由于，在实际环境中，Master 节点使用 Random 算法时，并未对其 GPU 的真实状况进行分析，随机选择 GPU 设备，这将会导致多个任务堆积在同一个 GPU 设备上，在运行时，需要等待前面的任务完成，才能运行下一个任务。此外，在实际情况下，每块 GPU 设备的运算能力也不一致。这种算法会导致有些 GPU 设备很繁忙，而有些 GPU 设备却很空闲的情况。本文的 DMLS-GPU 算法，不仅充分考虑每块 GPU 设备的计算能力，而且考虑任务的数目及任务在运算过程中的实际状况，如可以通过 nvidia-smi 查看 GPU 实际利用率，GPU 实际用了多少全局存储器，动态测试任务计算、传输所消耗的时间等，将这些参数带入到 DMLS-GPU 算法中。采用本文的 DMLS-GPU 算法对 GPU 负载进行评价，不仅能动态实时计算 GPU 的实际负载值，在均衡 GPU 负载的前提下，更能降低任务的周转时间。

针对第二种方法，测试 GPU 设备的利用率，使用多种不同类型的任务，计算随着时间的变

化，查看 GPU 设备的利用率。在选择任务时，分别使用单 GPU 任务和多 GPU 任务，每种又可分为“弱任务”和“强任务”。“弱任务”是指占用 GPU 的资源较少，而“强任务”则占用较多的 GPU 资源，这里 GPU 资源主要是指流多处理器和存储资源。实验所用任务类型及参数如表 5.5 所示。

表 5.5 任务类型及参数

计算任务	所需 GPU 数量	存储量 (MB)	占用流多处理器数目	PCI-E 传输时间 (ms)	主机间的传输时间(ms)	执行时间 (ms)
vectorAdd_weak	1	150	4	516.7	1234.8	3348.4
vectorAdd_strong	1	3120	14	8789.4	24011.5	11729.3
2GPU_weak	2	114	7	2341.3	912.8	23735.3
2GPU_strong	2	1200	14	2735.7	9589.9	20875.2
4GPU_weak	4	82	7	2479.5	656.7	22659.7
4GPU_strong	4	1110	14	4747.7	9380.1	17112.9

在本实验中使用 4 台虚拟机，在每台虚拟机上都产生一个任务队列，队列中随机产生上述六种类型任务。将任务队列同时执行，查看在任务运行过程中 GPU 设备的利用率，结果如表 5.6 所示。在此处，一个节点中包含两台虚拟机，一共两个计算节点。独占模式是指事先指定能够用到的 GPU 设备的数目，在任务运行中不能占用别的 GPU 资源，在本文实验条件下，只能启动两个任务队列。而资源共享模式中，采用虚拟化的方式，可以访问到多块 GPU。FCFS 是经典的先来先服务算法，DMLS-GPU 是本文提出的算法，可以启动全部虚拟机，即四个任务队列。

表 5.6 任务执行过程中各块 GPU 的资源利用率

	独占资源	FCFS	DMLS-GPU
虚拟机数目	2	4	4
Card0	82.15%	56.40%	82.57%
Card1	69.70%	68.23%	83.95%
Card2	32.85%	53.06%	95.02%
Card3	32.85%	44.81%	81.03%
Card4	82.15%	55.62%	82.57%
Card5	69.70%	67.01%	80.08%
Card6	32.85%	53.93%	91.76%
Card7	32.85%	45.93%	83.91%
平均利用率	54.39%	55.62%	85.11%

从 GPU 设备的平均利用率来看，独占资源的方式和 FCFS 算法的利用率相当，而本文提出的 DMLS-GPU 显然要高于前两种。在独占模式中，Card0 和 Card4 利用率很高，在任务队列中

几乎都用到这两块设备，而 Card2 、Card3 、Card6 和 Card7 仅当任务需要 4 块 GPU 设备时才会利用，所以 GPU 利用率较低。独占资源与 FCFS 算法的本质其实是一致的，都是希望能够最大化地使用资源，其区别为独占资源相对于整个队列来讲，而 FCFS 算法相对于某一个任务。本文提出的 DMLS-GPU 算法，其利用率均达到 81%以上，主要原因是利用虚拟化技术及负载均衡算法，隐藏了数据准备以及传输的时间，将各块 GPU 设备均衡利用，从而提高了利用的效率。

5.5 本章小结

本章在 GPU 虚拟化环境下，提出了 GPU 虚拟资源池的概念，研究基于 GPU 的虚拟资源池映射，分析虚拟资源的映射关系。借鉴在普通集群环境下集群管理技术和负载均衡方法，提出基于动态负载量多负载状态的 GPU 负载均衡算法 DMLS-GPU，将计算出的综合负载值用于衡量 GPU 设备的负载情况。实验结果表明，在虚拟化环境下，可以实现多个 CUDA 程序并发地共享一块或者多块 GPU 设备。通过与 Random 算法相比，验证了本算法在多任务共享 GPU 时的可行性和有效性，提高了硬件资源的利用效率和集群系统的吞吐率。

第六章　总结与展望

虚拟化技术是一种实现对计算机资源进行抽象模拟的技术，其优势在于复用硬件平台、透明使用硬件平台并降低资源管理的复杂度。与传统的集群并行架构相比，GPU 的并行不仅体现了节能、性价比高、体积小等优势，而且计算加速比更优。将上述两者相结合，GPU 虚拟化会使用较少的 GPU 而达到较高的性能，因而降低配置成本及能源消耗，同时提高资源利用率。但由于 GPU 设备的特殊性，对 GPU 虚拟化的研究处于起步阶段。为充分利用虚拟化技术和 GPU 计算能力，拓宽 GPU 虚拟化的适用范围，本文从现有的 GPU 虚拟化发展过程中的新需求出发，以通用计算框架 CUDA 为研究对象，设计一种在虚拟化环境下多任务的 GPGPU 并行计算的虚拟化方案。

6.1 论文研究工作总结

目前学术界主流的 GPU 虚拟化解决方案有 gVirtuS、vCUDA、GViM、rCUDA 等，借鉴这些虚拟化技术，本文设计了一种在虚拟化环境下多任务的 GPGPU 并行计算的方案。在虚拟化环境下设计多 GPU 并行计算的方法，实现将大规模程序划分到多块 GPU 设备上并行计算。同时结合此虚拟化方案，提出 GPU 虚拟资源池的概念，针对此虚拟化方案中动态评估如何 GPU 设备计算能力的问题，提出了 GPU 负载均衡的算法，实现多任务在虚拟化环境下共享 GPU 资源，并通过实验分析系统的有效性和高效性。

下面对本文的研究工作进行总结：

（1）介绍了虚拟化及 GPU 通用计算的概念，说明了 GPU 虚拟化的意义和前景，对比分析了目前在 GPU 虚拟化方面国内外的研究现状，阐述了 GPU 虚拟化方法，对现有的 GPU 虚拟化在图形渲染和通用计算中的方案进行了总结。

（2）从集群中管理任务的方案出发，设计出在基于多 GPGPU 并行计算的虚拟化方案。以 CUDA 为研究对象，设计了基于多 GPU 计算资源特征的动态分配与管理的架构，解决通用计算在虚拟化环境下的适应问题。GPU 虚拟化使 GPU 可以在多计算节点间共享，该架构分为虚拟化用户层、虚拟化资源管理层和虚拟化资源服务层。

（3）提出了在虚拟化环境下多 GPU 并行计算的实现方案。在虚拟化环境下，对于大规模的应用程序，使用多种方式实现多 GPU 并行计算，并在实验中分别对数据松耦合交互模式（如蒙特卡罗方法）和紧耦合交互模式（如 QFT 算法）实现多 GPU 并行计算。

（4）为减少 CUDA 程序中通信的开销，讨论了虚拟化环境下多 GPU 并行计算的通信策略。对主流的虚拟化平台下的虚拟机域间通信方式进行对比，得出在特定虚拟化环境下针对 CUDA

运算虚拟机域间最高效的通信方式，减少虚拟化带来的性能损耗，并分析了在多 GPU 并行计算时，多 GPU 的通信策略问题。

（5） 提出 GPU 虚拟资源池的概念，研究基于 GPU 的虚拟资源池映射，分析虚拟资源的映射关系，分为一对一、一对多，多对一、多对多几种方式。一对一是大多数 GPU 虚拟化研究的方案；一对多的关系在实验多机并发进行验证；多对一在本文第四章重点阐述；多对多，即研究实现多个任务划分到多块 GPU 设备上的问题。

（6）针对上面多对多方式存在的问题，防止出现 GPU 使用不均衡的情况，提出 DMLS-GPU 算法，将计算出的综合负载值用于衡量 GPU 设备的负载情况。实验结果表明，在虚拟化环境下，可以实现多个 CUDA 程序并发地共享一块或者多块 GPU 设备。通过与 Random 算法比较，验证了本算法在多任务共享 GPU 设备时的可行性和有效性，提高了硬件资源的利用效率和集群系统的吞吐率。

6.2 进一步的工作展望

本文对 GPU 虚拟化进行详细研究，并提出一种在虚拟环境下多任务的 GPGPU 并行计算的虚拟化方案。但在虚拟化环境下对 GPU 通用计算的研究仍处于早期阶段，在性能和可靠性方面仍有很多问题可以研究。

下面是对未来工作的进一步展望：

（1）虚拟化给系统可靠性、系统管理、透明使用资源等带来好处，但不可避免地也带来很大的开销，特别是对 I/O 密集型程序。目前虚拟机域间通信方式只是针对某一平台，并没有通用的解决方案，可以针对 CUDA 程序本身设计适合 CUDA 程序的通信解决方案。

（2）设计针对多 GPU 的 API，目前都是使用 CPU 控制 GPU 的方式使多 GPU 进行并行计算。因此可根据 CUDA 程序的特性设计统一接口将单 GPU 中程序透明地移植到到多 GPU 中。

（3）GPU 虚拟化方案支持虚拟机的高级特性，如虚拟机迁移、克隆等。目前在虚拟机迁移后，CUDA 程序并不能如预期直接使用，因此下一步可以从虚拟化的高级特性研究 GPU 虚拟化。

参考文献

[1] Overby E. Process Virtualization Theory and the Impact of Information Technology [J]. Organization Science archive, 2008, 19(2): 277-291.

[2] CUDA [EB/OL]. http://www.nvidia.cn/object/cuda-cn.html.

[3] 石林. GPU 通用计算虚拟化方法研究 [博士学位论文]. 长沙: 湖南大学, 2012.

[4] 陈滢, 王庆波, 金淳, 等. 虚拟化与云计算 [M]. 北京: 电子工业出版社, 2009.

[5] Rosenblum M, Garfinkel T. Virtual Machine Monitors Current Technology and Future Tends [J]. IEEE Computer Society, 2005, 38(5): 39-47.

[6] 金海等. 计算系统虚拟化--原理与应用 [M]. 北京:清华大学出版社, 2008.

[7] Waldspurger C, Rosenblum M. I/O virtualization [J]. Communications of the ACM, 2012, 55(1): 66–73.

[8] Announcing Cluster GPU Instances for Amazon EC2 [EB/OL]. http://aws.amazon.com/ec2/.

[9] Köhn A, Drexl J, Ritter F, et al. GPU Accelerated Image Registration in Two and Three Dimensions [J]. Bildverarbeitung für die Medizin, 2006, 19(21): 261-265.

[10] Wolfgang F. Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks [M]. Wordware Publishing, 2002: 113-115.

[11] Lagar-Cavilla H A,Tolia N, Satyanarayanan M, et al. VMM-independent Graphics Acceleration[C]. In: Proc of Virtual execution environments. New York, 2007: 33-43.

[12] 张舒,褚艳利,赵开勇等.GPU 高性能运算之 CUDA[M].北京: 中国水利水电出版社,2010.

[13] Shi L, Chen H, Sun J. vCUDA: GPU Accelerated High Performance Computing in Virtual Machines[C]. In: Proc of International Parallel & Distributed Processing Symposium. Rome, 2009: 1-11.

[14] Gupta V, Gavrilovska A, Schwan K, et al. GViM: GPU-accelerated virtual machines[C]. In: Proc of ACM Workshop on System-level Virtualization for High Performance Computing. New York, 2009: 17-24.

[15] Giunta G, Montella R, Agrillo G, et al. A GPGPU Transparent Virtualization Component for High Performance Computing Clouds[C]. In: Proc of EuroPar conference on Parallel Processing. Berlin, 2010: 379-391.

[16] Duato J, Pena A, Silla F, et al. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters[C]. In: Proc of International Conference on High Performance Computing

and Simulation. Caen, 2010: 224-231.

[17] Oikawa M. DS-CUDA: a middleware to use many GPUs in the cloud environment [C]. In: Proc of High Performance Computing, Networking, Storage and Analysis. Salt Lake City. 2012:1207 - 1214.

[18] Liang T Y. GridCuda: a grid-enabled CUDA programming toolkit[C]. In: Proc of Advanced Information Networking and Applications. WAINA, 2011: 141 - 146.

[19] Wang J, Wright K, Gopalan K. XenLoop: A Transparent High Performance Inter-VM Network Loopback[C]. In: Proc of International Symposium of High Performance Distributed Computing. Boston, 2008, 109-118.

[20] Ravi V T, Becchi M, Agrawal G, et al.Supporting GPU sharing in cloud environments with a transparent runtime consolidation framework. In: Proc of international symposium on High performance distributed computing. New York, 2011: 217-228.

[21] Li T, Narayana V K ,El-Araby E,et al. GPU Resource Sharing and Virtualization on High Performance Computing Systems[C]. In: Proc of International Conference on Parallel Processing. Taipei, 2011: 733-742.

[22] 李文亮. GPU 集群调度管理系统关键技术的研究[硕士学位论文]. 武汉: 华中科技大学, 2011.

[23] Stuart J and Owens J. Multi-GPU MapReduce on GPU clusters [C]. In: Proc of 2011 IEEE Intl. Parallel & Distributed Processing Symposium. Washington, 2011: 1068–1079.

[24] Randi J R. OpenGL(R) Shading Language [M], Redwood City, CA: Addison Wesley LongmanPublishing Co, 2004.

[25] Mark W R, Glanville S, Akeley K, Kilgard M J. Cg: A System for Programming GraphicsHardware in A C-Iike Language [J]. ACM Transaction on Graphics, 2003, 22(3): 896-907.

[26] Peeper C, Mitchell JL. Introduction to the DirectX 9 High-Level Shader Language [EB/OL]. http://msdn.microsoft.com/librap//default.asp?url=/library/en-us/dnhlsl/html/shaderx2 introductionto.asp.

[27] Buck I, Foley T, Horn D, Sugerman J, et al. Brook for GPUs: Stream Computing on Graphics Hardware [C]. In: Proc of the ACM Transaction on Graphics. New York, 2004:777-786.

[28] Nolan Goodnight, Cliff Woolley, David Luebke, et al. A Multigrid Solverfor Boundary value Problems Using Programmable GraPhics Hardware[C]. In: Proc. of GraPhics Hardware. SanDiego, 2003:102-111.

[29] John D. Owens1, David Luebke, et al. A Survey of General-Purpose Computation on Graphics Hardware [J]. COMPUTER GRAPHICS forum, 2007, 26(1): 80–113.

[30] Fatica M. Accelerating Linpack with CUDA on Heterogenous Clusters [C]. In: Proc of 2nd Workshop on General Purpose Processing on Graphics Processing Units. New York, 2009:46-51.

[31] Stone S, Haldar J P, Tsao S C, Hwu W, et al. Accelerating Advanced MRI Reconstructions on GPUs [J]. Journal of Parallel and Distributed Computing, 2008, 68(10): 261-272.

[32] Jang B, Do S, Pien H, Kaeli D. Architecture-aware optimization targeting multithreaded stream computing[C]. In: Proc of 2nd Workshop on General Purpose Processing on Graphics Processing Units. New York, 2009: 62-70.

[33] Barak A, Ben-Nun T, Levy E. A package for OpenCL based heterogeneous computing on clusters with many GPU devices[C]. In: Proc of Cluster Computing Workshops and Posters 2010, Heraklion, 2010: 1 – 7.

[34] Papakipos M. The PeakStream Platform, High-Productivity Software Development for Multi-Core Processors [EB/OL]. http://download.microsoft.com/download/d/f/6/df6accd5-4bf2-4984-8285-f4f23b7b1f37/winhec2007_peakstream.doc.

[35] McCool M D, Wadleigh K, Henderson B, Lin H Y. Performance Evaluation of GPUs Using the RapidMind Development Platform [C]. In: Proc of the 2006 ACM/IEEE Conference on Supercomputing (SC). New York, 2006:11-17.

[36] Ublig R, Neiger G, et al. Intel Virtualization Technology [J]. IEEE Computer Magazine, 2005, 38(5):48-56.

[37] VMware[EB/OL].http://www.vmware.com/products/Workstation/.2013-12-14.

[38] Barham P,Dragovic B,Fraser K,et al.Xen and the Art of Virtualization [C].In:Proc of 19th ACM Symposium on OperatingSystems Principles.Bolton Landing,2003:164-177.

[39] Virtual PC [EB/OL].http://www.microsoft.com/windows/virtual-pc/.2013-12-16.

[40] Hyper-V [EB/OL].http://technet.microsoft.com/en-us/library/cc816638(v=ws.10).aspx.2013-12-16.

[41] Kivity A, Kamay Y, Laor D, et al. KVM: The Linux Virtual Machine Monitor [C]. In: Proc of Linux Symposium.Ottawa,2007: 225-230.

[42] Darren A, Jeff J, Sridhar M, et al. Intel(R) Virtualization Technology for Directed I/O[J]. Intel Technology Journal, 2006, 10(3):179-192.

[43] Wei J, Jackson J R, Wiegert J A. Towards scalable and high performance I/O virtualization - A case study[C]. In: Proc of 3rd International Conference on High Performance Computing and

Communications, HPCC 2007. Houston, 2007: 586-598.

[44] Maruyama T, Yamada T. Sharing IO devices using hardware virtualization method for component-based industrial controllers.in Emerging Technologies and Factory Automation[C]. In: Proc of IEEE International Conference on, Hamburg, Germany: Institute of Electrical and Electronics Engineers Inc., 2008:705-708.

[45] VMware SVGA Device Developer Kit [EB/OL].VMware-svga.sourceforge.net.2013-12-15.

[46] Bellard F.QEMU, a fast and portable dynamic translator[C].In: Proc of the annual conference on USENIX Annual Technical Conference.Berkeley, 2005: 41-41.

[47] Moya V, Gonzalez C, J. Roca, and A. Fernandez. ATTILA: a cycle-level execution-driven simulator for modern GPU architectures[C]. 2006:231-241.

[48] Sheaffer J W, Luebke D, and Skadron K. A Flexible Simulation Framework for Graphics Architectures[C]. In: Proc of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware. 2004:85-94.

[49] Bakhoda A, Yuan G, Fung W, et al.Analyzing CUDA Workloads Using a Detailed GPU Simulator[C]. In: Proc of IEEE Analysis of Systems and Software. Boston, 2009:163-174.

[50] August D, et al. UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development [J]. IEEE Computer Architecture Letters, 2007, 6(2): 1556-6056.

[51] VirtualGL [EB/OL].http://www.virtualgl.org/.2013-12-16.

[52] Shui-xia H., Guo-sun Z, Yi-ming T. Scalability Analysis of Heterogeneous Computing Based on Computation Task and Architecture to Match [J]. Institute of Electronics, 2010, 38(11):2585-2589.

[53] Dean J and Ghemawat S. Mapreduce: simplified data processing on large clusters [C]. In: Proc of the 6th conference on Symposium on Opearting Systems Design & Implementation. New York, 2004:137-150.

[54] Vinoski S.CORBA: Integrating diverse applications within distributed heterogeneous environments [J]. IEEE Communications, 1997, 35(2):46-55.

[55] XMLRPC [EB/OL].http://xmlrpc.scripting.com/default.html.2013-12-18.

[56] Henning M. A new approach to object-oriented middleware [J].IEEE Internet Computing, 2004, 8:66-75.

[57] Hongyi M, Diersen R S, W. Liqiang, et al. Symbolic Analysis of Concurrency Errors in OpenMP Programs [C]. International Conference on Parallel Processing. Lyon, 2013: 510-516.

[58] Wei Z, Gulsah A, Xinmin T, et.al. Parallel protein secondary structure prediction schemes using Pthread and OpenMP over hyper-threading technology [J]. The Journal of Supercomputing, 2007, 41(1): 1-16.

[59] Gabriel E, Fagg G E, Bosilca G, et al. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation [J]. Recent Advances in Parallel Virtual Machine and Message Passing Interface, 2004, 32(41): 97-104.

[60] Zhang X, McIntosh S, Rohatgi P, et al.Xensocket: A high-throughput interdomain transport for virtual machines[C].In: Proc of International Middleware Conference. Berlin, 2007:184-203.

[61] Kim K,Kim C,Jung S,et al.Inter-domain Socket Communications Supporting High Performance and Full Binary Compatibility on Xen[C].In:Proc of International Conferenceof Virtual execution environments.Seattle,2008:11-20.

[62] VMCI [EB/OL].http://pubs.vmware.com/vmci-sdk/.2013-12-18.

[63] 李亚妮.蒙特卡罗方法及其在期权定价中的应用 [硕士学位论文].西安：陕西师范大学, 2007.

[64] CUDA_Occupancy_calculator.xls[EB/OL].http://developer.download.nvidia.com/compute/cuda/4_0/sdk/docs/CUDA_Occupancy_calculator.xls.2013-12-20.

[65] 夏培肃. 量子计算[J] . 计算机研究与发展, 2001, 38(10): 1153-1171.

[66] 都志辉. 高性能计算并行编程技术--MPI 并行程序设计[M].北京：清华大学出版社，2001年.

[67] 方旭东. 面向大规模科学计算的 CPU-GPU 异构并行技术研究 [硕士学位论文].湖 南 : 国 防科技大学, 2009.

[68] Saeed S, Seyed A. M., Mohammad K. Akbari. A predictive and probabilistic load-balancing algorithm for cluster-based web servers [J]. 2011, 11(1): 970–981.

[69] Peiyou Z, Ji G, Ron C, et al. The research on dynamic load balancing algorithm for heterogeneous systems[C]. In: Proc of Institute of Electrical and Electronics Engineers. Dalian, 2006: 4420-4424.

[70] Saaty T L, Vargas L G. The seven pillars of the analytic hierarchy process [J]. Models, Methods, Concepts & Applications of the Analytic Hierarchy Process, 2001, 34: 27-46.

[71] 李银钊，倪天权，薛羽. 基于层次分析法的并行干扰资源调度模型 [J]. 舰船电子对抗, 2013, 36(03): 88-91.

致 谢

又到夜里了，夜深人静时，不受外界干扰，思维更加活跃。终于写到致谢部分了，不禁回想起当时跟实验室的一群人爬紫金山的情景。许久的不运动，在半山腰上就爬不动了，实验室的同胞们鼓励我说这爬山就像是你的大论文，你现在正写到第三章，峰顶就是你的致谢部分，没走一步就是一个跨越，一点点的跨越最后就是成功。人生何尝不是如此，就在一点一滴中一步步向前跨越。写完这篇论文，我的学生生涯也接近尾声了，在人生最美好最灿烂的时光，我很荣幸在南京航空航天大学度过，每跨越一步都离不开导师、实验室的同胞以及亲友们的帮忙，在此深深地向你们说一声：谢谢你们，辛苦了！

首先要感谢我的导师袁家斌教授！在本文的研究工作中袁老师谆谆教导仍在耳边回荡，在承担二十个学生的指导下仍对我悉心地指导，从论文的开题、课题的研究到最后论文的撰写无不凝聚着老师的心血。在学术上，袁老师渊博的知识、国际化的视野、严谨的态度、前沿而精髓的学术造诣，都让我永志不忘，对我以后的人生和生活都会有很大的影响；在生活中，经常跟我们讨论如何做人，如何做事，一个人的成功需要格局和境界，格局是一个人受先天的影响，家庭环境、自我天赋等等，而境界就是一个人情商、逆商的体现，需要有开阔的胸襟、越挫越勇的精神。研究生阶段为跟着这样的导师十分幸运，同时也收获不少。

感谢在研一研二的时候王箭老师周末组织的研讨会，在研讨会中大家互相交流，从讨论中思维得以发散，王箭老师提出了很多宝贵的建议。感谢在研三阶段由袁老师牵头，刘虎老师、许娟老师、翟向平老师一起参与的研讨会，在研讨会中，老师们宽广的视野和严谨的学术态度不断鞭策着我在学业中前进。感谢学校网络中心的老师们，特别是张兰兰老师，在生活中对我们无微不至的照料。

感谢实验室已毕业和正在读博的师兄师姐们，以及仍在科研前线奋斗的师弟师妹们。他们充满活力和朝气，为实验室带来了欢声笑语，大家相互学习、互相关心，一起度过了美好的时光！感谢吕相文博士，为我的论文提出了宝贵的意见，解决了很多技术难点。感谢实验室全体同学，尤其是段博佳、张唯唯、赵兴方、张珮、王雪、丁伟杰，大家一起创造了良好的学术氛围，帮我一起修改论文。感谢我的家人、男朋友和室友们，感谢你们无微不至的关心和鼓励，有你们的陪伴让我研究生生活充满幸福和欢乐！

最后，感谢各位评审专家对本文的评阅。在跨年之际，祝大家身体健康，新春愉快，阖家幸福！

在学期间的研究成果及发表的学术论文

攻读硕士学位期间发表（录用）论文情况

1. Yujie Zhang, Jiabin Yuan, Xiangwen Lu and Xingfang Zhao. Multi-GPU Parallel Computing and Task Scheduling under Virtualization [J]. International Journal of Hybrid Information Technology. （EI 期刊，第一作者，已录用）
2. 张玉洁，吕相文，张云洲. GPU 虚拟化环境下的数据通信策略研究 [J]. 计算机技术与发展.（中文核心期刊，第一作者，已录用）
3. 张唯唯，张玉洁. 基于 GPU 的并行报文分类方法 [J].计算机与现代化. (中文核心期刊，第二作者，已录用）
4. 吕相文，袁家斌，张玉洁. 云计算环境下多 GPU 资源调度机制研究 [J].小型微型计算机系统. (中文核心期刊，第三作者，已录用）
5. 肖骁，龚正，张玉洁.机场噪声感知节点泛网格化布局仿真研究[J]. 计算机技术与发展. (中文核心期刊，第三作者，已录用）

攻读硕士学位期间获得软件著作权情况

1. 机场噪声值实时监测显示系统. 2013SR145202.
2. 中小型企业生产管理系统. 2014SR158372.

攻读硕士学位期间参加科研项目情况

1. 解放军理工大学某国家重大科研项目
2. 国家自然科学基金“面向机场感知的噪声监测及其环境影响评估”