中图分类号：TP302 论文编号：1028716 14-S054
学科分类号：081200

硕士学位论文

虚拟化环境下的 GPU 通用计算关键技术研究

研究生姓名 张云洲
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 一四年一月

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

Research on Key Technologies for General-Purpose computing on GPU in the Virtualization Environment

A Thesis in

Master of Computer Science and Technology

by

Zhang Yunzhou

Advised by

Prof. Yuan Jiabin

Submitted in Partial Fulfillment

of the Requirements

for the Degree of

Master of Engineering

January, 2014

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

随着计算机硬件功能和软件开发环境的不断成熟，GPU 应用逐渐从图形计算向通用计算领域发展，使用基于 GPU 的高性能计算平台进行海量数据计算科学研究的例子不胜枚举。虚拟化技术是当今计算机领域的一个研究热点，在虚拟化和 GPU 通用计算的结合处，学术界的研究处于起步阶段，本文对虚拟化环境下的 GPU 通用计算关键技术进行研究。

本文对虚拟化技术和 GPU 通用计算进行了详细的分析，对现有的 GPU 虚拟化解决方案进行总结，在 GPU 虚拟化方案和 GPU 资源调度算法的基础上，立足于降低任务的周转时间，对现有的 GPU 调度算法进行改进。通过设置综合负载评价值的方法实现负载均衡，将 GPU 特征、任务规模和计算复杂度纳入负载评价考量，实现更细粒度更准确的负载评价。针对大规模计算程序，本文在虚拟化环境下设计了基于 OpenMP 的多 GPU 协同计算方法，根据不同任务类型和任务规模进行不同的任务划分，采用经典的矩阵运算和复杂度较高的离散傅里叶变换等具有代表性的科学计算实例进行实验验证。实验结果表明，随着计算规模的增加，多 GPU 的加速比可以达到接近 GPU 的个数。

为了降低由于虚拟化本身给 GPU 通用计算带来的性能开销，本文对现有的虚拟机域间通信优化解决方案进行总结，在特定虚拟化平台下采用对 CUDA 性能影响最低的通信方式。同时，在 GPU 内部数据交互的方式上，设计了两种数据传输方式对 GPU 协同计算进行测试，通过对理论计算和实际测量进行对比，验证了预测结果的正确性并得出影响多 GPU 协同运算效率的主要因素。

关键词：虚拟化，GPU 通用计算，资源调度，周转时间，OpenMP，协同计算，CUDA

ABSTRACT

With the maturation of computer hardware and software development environment, GPU computing applications gradually evolved from graphic computing to general-purpose computing field, and the scientific research examples of huge amounts of data computing carried on the platforms of GPU-based high performance computing are numerous. While virtualization is a hot topic in today's computer industry, the academic study in the junction of virtualization and GPU general computing is still in its infancy. This paper focuses on the study of the key technologies using the GPU for general purpose computing in virtualization environment.

In this paper, it makes a detailed analysis on virtualization technology and GPU computing and a summary on the existing GPU virtualization solutions. On the basis of GPU virtualization solution and GPU resource scheduling algorithm, it brings up a way to improve the existing GPU scheduling algorithm, which aims at reducing the turnaround time. By setting the value of a comprehensive load evaluation it realizes the load balancing. Take the GPU features, the size and complexity of the task into the consideration of load evaluation to achieve a more fine-grained and more accurate load evaluation. For large-scale computing programs, this paper designs a multi-GPU collaborative computing technology based on OpenMP in a virtual environment, which divides different tasks according to the different types and scales of tasks, and makes experimental verification of representative scientific computing examples such as classical matrix calculation and discrete Fourier transformation of high complexity. Experimental results show that with the increase of the calculation scale, it can achieve a speed-up ratio close to the number of GPU.

In order to reduce the GPU general computational overhead of performance due to the virtualization itself, the paper makes a summary of the existing optimization solutions of inter-domain virtual machine communication, and then get the optimal method of communication which has the least affection in CUDA in a specific virtualization platform. Meanwhile, in the exchange mode of GPU data it designed a method of using two data transmission ways to test the GPU synergistic calculation, and at last get the main factors which affect multiple GPU collaborative computing efficiency via the comparision of theoretical calculating and actual measuring.

Key words: Virtualization, General-Purpose computing on GPU, Resource scheduling, Turnaround time, OpenMP, Collaborative computing, CUDA

目 录

图表清单

注释表

N	当前 GPU 上的任务数	$Scale_i$	GPU 上第 i 个任务的规模
$complx_i$	第 i 个任务的计算复杂度	P	GPU 内核数
R	GPU 时钟频率	G	GPU 的全局内存
α_i	处理能力对第 i 个任务的影响因子	β_i	全局内存对第 i 个任务的影响因子
A_{ij}	矩阵 A 第 i 行第 j 列的元素	e	自然对数的底数

缩略词

缩略词	英文全称
GPU	Graphic Processing Unit
GPGPU	General-purpose computing on GPU
VM	Virtual Machine
VMM	Virtual Machine Monitor
SLI	Scalable Link Interface
CUDA	Compute Unified Device Architecture
CKX	Concurrent Kernel eXecution
SPMD	Single Program Multiple Data
SP	Stream Processors
MRT	Multiple Render Targets
OpenCL	Open Computing Language
MPI	Message Passing Interface
OpenMP	Open Multi-Processing
CUBLAS	CUDA Basic Linear Algebra Subroutines
DFT	Discrete Fourier Transform
VMCI	Virtual Machine Communication Interface
UVA	Unified Virtual Addressing

第一章　绪论

目前，无论是学术界还是工业界，虚拟化技术的研究和应用都是热点。虚拟化技术的优势在于复用硬件平台、拓展硬件平台并且可使硬件平台透明化，但是目前国内外工业界虚拟化的解决方案却普遍缺乏对 GPU 的支持，学术界对 GPU 虚拟化的研究也处于起步阶段。在通用计算领域，以 CUDA[1]为代表的并行计算架构将 GPU 引入到科学计算中，但它们都没有针对多 GPU 环境下的 API。虚拟化环境下面向通用计算的 GPU 协同计算的研究对基于 CPU+GPU 异构计算模型的网格计算、云计算具有重大的理论和现实意义[2]。

1.1 课题研究背景及意义

虚拟化本身是个广义的术语，涉及的范围非常广，它随着计算机技术的不断发展而出现，对计算机技术的发展产生了很重要的影响。上世纪五十年代虚拟化的概念首先被提出，六十年代 IBM 在大型机上实现了虚拟化的商业应用。九十年代以来，随着计算机科学研究的逐渐深入和市场需求的变化，虚拟化技术所带来成本的节约、安全性的增强等优势逐渐得到重视，在计算数据保护、服务器整合、网络安全、高性能计算和可信计算等领域得到了大量的应用。随着计算机技术的不断发展，从早期的操作系统的虚拟内存发展到后来的 Java 虚拟机，再到目前服务器虚拟化技术的蓬勃发展，虚拟化已经在计算机领域的各个层面得到了蓬勃的发展。近年来，随着服务器虚拟化技术的普及，数据中心（Data Center，DC）的部署和管理方式出现了全新的方法，这使得数据中心管理员拥有了便捷高效的管理体验。此外，虚拟化技术的成功部署，不仅可以提高 DC 硬件资源的利用率，更重要的是满足了人们对低功耗的需求。伴随着诸多优点，虚拟化技术迅速在工业界和学术界成为了研究的重点。

在虚拟化技术中，被虚拟的实体是各种各样的 IT 资源，按照这些资源的类型不同可以将其主要分为基础设施虚拟化、软件虚拟化以及系统虚拟化。由于本文的研究领域属于系统虚拟化，文中所述的虚拟化均特指系统虚拟化。系统虚拟化的核心理念是通过虚拟机监视器（Virtual Machine Monitor，VMM）在一台宿主物理机上虚拟出若干台虚拟机（Virtual Machine，VM），使得各个虚拟机之间可以互不干扰地运行在自己的隔离区中。其中，虚拟机是具有完整硬件功能、在一个隔离环境中运行的逻辑计算机系统，一般由客户操作系统（Guest OS）和在其上运行应用程序组成。系统级虚拟化抽象的对象是计算机完整的硬件，包括 CPU、外存和内存等物理资源。在系统级虚拟化中，多个客户操作系统可以相互隔离地同时运行在同一个物理机上，共享着同一物理机中的资源（CPU、内存、外存等）。不同类型的系统虚拟化，对虚拟机运行环境的设计和实现也不全相同。但在系统级虚拟化中，虚拟机监视器都需要为在其上运行的虚拟

机提供一套能独立运行的软硬件环境。

系统级虚拟化的本质主要是从逻辑上对资源进行重新划分，如果在各个层面按照被重构硬件的不同可以将虚拟化研究内容进一步细分，本文所述的面向通用计算 GPU 虚拟化在整个虚拟化的层次中所处的位置可在如图 1.1 体现。不管在学术界还是工业界，对于 CPU 虚拟化的研究由来已久且日益成熟，学术成果也已被广泛应用，但针对 GPU 虚拟化的解决方案还不成熟，且现有的对 GPU 虚拟化的研究大多在图形计算领域，在通用计算领域则是很少被涉及。随着 GPU 的并行计算能力不断提升，其浮点计算能力远远超过 CPU，在高性能计算平台上应用 GPU 对应用程序进行加速可以在计算能力方面取得显著的提升。目前虚拟化技术越来越普及，但是并不存在完善的 GPU 通用计算虚拟化方案。GPU 通用计算虚拟化技术方面的空白严重影响了虚拟化技术在高性能计算平台上的大规模部署和应用。

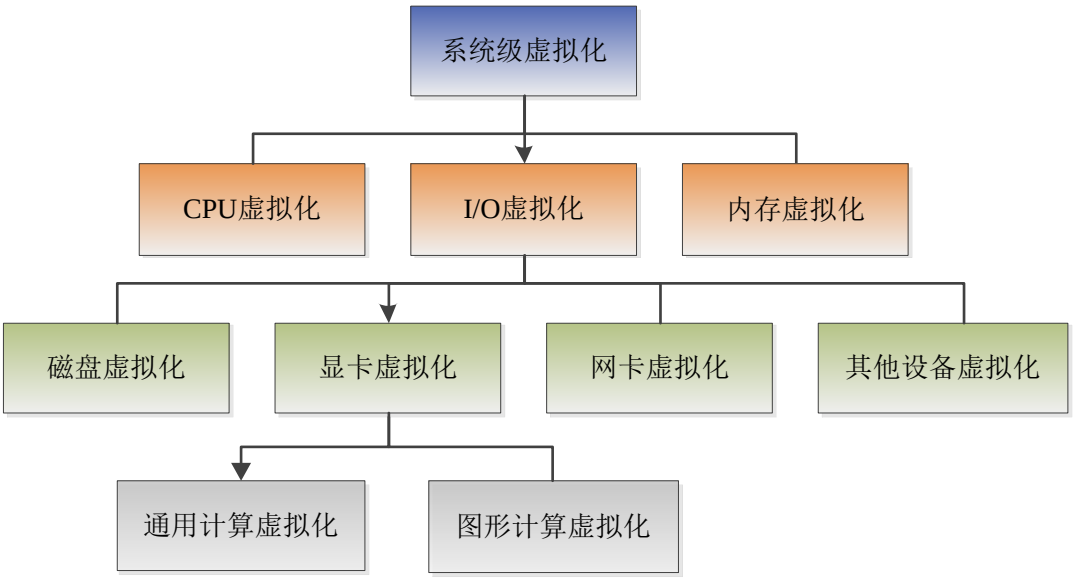

图 1.1 系统级虚拟化结构图

经过业界多年的研究和应用，系统级虚拟化在分布计算、网格计算、企业信息化以及信息安全等领域有着越来越广泛的应用。IBM 从事虚拟化的历史已经超过 40 年，世界上第一台采用了虚拟化技术的计算机是 20 世纪 60 年代中期由 IBM 公司在 Thomas J.Watson 实验室设计和实现的 IBM 7044(M44)[3]。随后 HP、Dell 等服务器解决方案提供商纷纷推出了自己主导的虚拟化技术，微软也在 2008 年推出了 Windows Server 2008 和 Hyper-V。虚拟化技术是对硬件资源在系统层进行抽象并统一表示，在数据中心的管理方面有着不可替代的作用，对网格计算、云计算提供着最主要的理论和技术支持[3]。在虚拟化技术发展的过程中，一个重大挑战是对 I/O 资源的虚拟化，这是由于 I/O 外设的多样性和很大一部分 I/O 外设的闭源性所决定的，使得在虚拟机中的运行的应用程序在很多场景下都无法充分地利用 I/O 资源，降低了由于虚拟化的部署而带来较低的资源利用率。以 GPU 为例，它的功能主要分为图形计算和通用计算两个部分，除了少数 2D 标准外，显卡硬件的接口在国际范围内并不统一，几个主要生产厂商（如 AMD、

NVIDIA 等）之间的电气接口也互不兼容，即使是同一厂商的不同产品型号、不同批次产品之间，其硬件特性也存在着很多不同的地方。除此之外，从商业的角度考虑出发，几大显卡制造商并不对外界开放显卡的设计细节，驱动源码也是闭源的，这就使得在操作系统层开发不同的驱动程序来统一显卡之间的硬件标准并不现实。目前主流的虚拟机监视器（如 ESXi、Xen、KVM 等）都只是在较小的范围内解决图形计算问题，但是对 GPU 的 3D 加速和通用计算支持却基本上不涉及。

针对 GPU 在底层虚拟化遇到的复杂性、封闭性问题，学术界将虚拟化的思路放到了更高层次的软件栈。例如，GPU 的 3D 加速功能是通过对传统的图形 API（OpenGL）的调用进行拦截，将其中相关的参数和命令进行解析并封装，将其通过 socket 通信或者其他虚拟机域间通信方式（如 VMsocket、XenLoop 等）从虚拟机传递至 VMM，然后在 VMM 上进行与 3D 加速功能相关的运算。将虚拟化从两个层面进行升级，在硬件层上升到软件栈层、驱动层上升到应用层，虽然解决了 GPU 虚拟化原本在底层遇到的难题，但针对图形 API 的虚拟化并不能完全解决通用计算的问题。2007 年由英伟达（NVIDIA）公司推出的 CUDA（Compute Unified Device Architecture，统一计算设备架构）是当前基于 GPU 的主流通用计算框架，其作为一个新的基础架构将 GPU 用来解决科学、工业以及商业方面的复杂计算问题。CUDA 作为一个 C 语言的最小扩展集，程序开发者可以轻松入手，不再需要依赖传统的图形 API（如 OpenGL 等）。

基于 GPU 的通用计算（General-Purpose computation on GPU，GPGPU）是现代并行计算领域的一个分支。基于 CPU 的并行计算技术已经相对比较成熟，学术界和工业的研究重点已经从实现层向应用层转移。然而基于 GPU 的并行计算情况却与 CPU 大不相同，原因有两个方面：其一，现实中的应用和针对这些应用的解决方案不止一种，不同的问题对计算机软硬件的环境和计算机性能的要求有着很大的差异，即使是同一问题，它们的规模、环境的差异对计算机性能的要求也有着很大的不同，传统的超级计算机很难满足中小规模计算的低成本、低功耗等的实际需求；此外，基于 CPU+GPU 的异构计算模式为传统的拥有大规模数据的串行算法向并行方向的转移提供了机遇。随着信息界对 GPU 通用计算的热情的逐渐高涨，利用 GPU 在图形计算以外的时间来完成通用计算，不仅大幅度提升了计算性能，更充分地满足了人们对性能功耗比的需求。二十一世纪以来，采用 CPU+GPU 异构计算模式的大规模集群系统大量出现，在 2013 年 11 月全球超级计算机 500 强的排名中，中国国防科技大学的天河二号位列第一名，第二名是一台安装在美国能源部（United States Department of Energy）的橡树岭国家实验室 Cray XK7 系统的超级计算机——泰坦。天河二号和泰坦均采用了 CPU+GPU 的异构混合计算模式。

在多 GPU 协同计算领域，随着近几年研究的深入，工业界针对多 GPU 渲染的技术愈发成熟。SLI（Scalable Link Interface，可升级连接接口）是 NVIDIA 公司的多 GPU 并行渲染的技术，它通过一种特殊的接口连接方式，在一块支持双 PCI Express X16 的主板上，同时使用两块同型

号的 PCI-E 显卡。随着 Fermi 架构的不断创新，基于 SLI 的扩展技术得到了前所未有的发展，双 GPU 通过 SLI 技术在理论上可将显卡的图形处理能力提升一倍。而在实际环境下的应用中，除了少数测试数据之外，在实际游戏中图形渲染性能只能提高 80%左右，在有些环境下性能甚至没有提升。SLI 在 GPU 中内置了一个智能的通信协议，用一个高速数字接口来加速 GPU 之间的数据交流；有一系列软件提供负载均衡、高级渲染和合成，来确保最新款的游戏达到最大兼容和最高性能。CrossFire 是 AMD 针对多 GPU 设计的并行渲染技术，也称交火技术。它是通过主卡和从卡来实现，其中主卡负责对数据进行分配，将需要处理的部分数据传输到从卡上并行处理，然后再把从卡计算完成的信息返回到主卡上一同输出，其中连接显示输出信号的显卡为主卡。由于高端显卡通信带宽的要求较高，故需要交火连接器，而低端显卡只需要主板的 PCI-E X16 通道就能够满足带宽需求。

在通用计算领域，成熟的多 GPU 协同计算的标准尚未形成，学术界将研究重点放在 CPU 和 GPU 协同计算上面，对多 GPU 的协同计算方法仍处研究的起步阶段，在虚拟化环境下更是空白。本文认为，通过在虚拟化环境下利用 GPU 对通用计算进行加速，可以为 GPU 在云计算平台上的应用提供强有力的技术支持。同时，通过在虚拟化环境下采用多 GPU 协同计算方法，对于在单节点多 GPU 环境下的任务具有很好的加速效果。

1.2 国内外研究现状

为了解决 GPU 运算体系在底层虚拟化时遇到的封闭性问题，学术界在较高层次上设计了 GPU 虚拟化的方案，主要围绕着 GPU 的两种完全不相同的接口：3D 图形计算接口和通用计算接口。VMGL[4]是首个对 OpenGL 进行虚拟化的方案，它通过设计了一个伪库（fake library）来取代 OpenGL 的原生库并部署在客户操作系统中，设计的伪库和 OpenGL 原生库有着相同的接口，但伪库都被实现为对宿主操作系统或者远程服务器的远程调用，这样所有的本地调用都被重定向至远程服务器，远程服务器拥有 OpenGL 库，并能够实际驱动物理 GPU，它负责完成 OpenGL 调用。VMGL 在 Xen 下实现，但是它是一个比较通用的方案，并不依赖于特定的虚拟化平台和 GPU。在图形处理方面，VMware 主持开发的一个特定的 GPU 虚拟化架构 VMware's Virtual GPU[5]，这个方案类似于设备仿真的方法，但是它包含了 API 远程处理的特点。

上述两个 GPU 虚拟化解决方案主要针对 GPU 的图形计算方面。随着 GPU 在通用计算领域的发展，它在通用计算领域的功能逐渐被发掘出来，形成了特定的服务于通用计算的 GPU。2007 年，NVIDIA 推出了当时第一个基于 GPU 的通用计算框架——CUDA，2010 年以后出现了一批探讨 CUDA 在虚拟机内应用的方案，例如 vCUDA、GViM、rCUDA、gVirtuS 等。

vCUDA[6]（virtual CUDA）通过在应用层对 CUDA 的调用进行拦截并重定向，同时在虚拟机中建立 GPU 的逻辑映像——vGPU（virtual GPU），展示了 CUDA 在虚拟机内应用的雏形。

vCUDA 采用了 XML-RPC 的方式来处理服务端和客户端之间的通信，这使得开发难度大大降低，同时将由于虚拟化带来性能损失降低至 21%。另外，vCUDA 不依赖于特定虚拟化平台，是一个相对通用的方案。

GViM[7]（GPU-accelerated Virtual Machines）则依赖于特定的 Xen 虚拟化平台，它通过建立与 Xen 命名规则相匹配的前后端，总共有四个部分：位于客户端的拦截库和前端驱动，位于宿主端的库封装器和后端驱动。其中，客户端中的拦截库通过对 CUDA 接口中的参数进行收集并发送至前端驱动，前端驱动再将其传递到后端。后端则利用库封装器在本地执行 CUDA 调用并将结果返回给客户端。GViM 将其研究重点放在对数据传输的优化并实现零拷贝（zero copy）等，但是该方案依赖于特定虚拟化平台，需要在客户端和服务端同时插入一个定制的模块——前端驱动和后端驱动，因而它对用户则不是完全透明的，同时对 CUDA API 的支持有限。

rCUDA[8]（remote CUDA）是在 2010 年出现的一个项目，至今已发布到 4.0.1 版本，有 Linux 和 Windows 共 12 个版本。它是远程执行 CUDA 的框架，设计思路与 vCUDA 类似，同样采取了 C/S（client/server）模式，客户端采用库封装器来对 CUDA API 进行拦截并获得其参数，在服务端运行本地 CUDA API 并将执行完成的结果返回给客户端。与 vCUDA 不同的是，在 rCUDA 中，服务端与客户端之间的通信直接采用的是 socket 而不是 RPC（Remote Procedure Call）方式，在效率上比 vCUDA 提升不少。rCUDA 主要针对的是驱动 API 进行虚拟化，而 vCUDA 则针对的是运行时 API 进行虚拟化。

gVirtuS[9]（GPU Virtualization Service）也是 2010 年出现的项目，它在思路上借鉴了 GViM 和 vCUDA 两个项目，并在它们的基础上独立地实现了 GPU 在虚拟机中的整体架构。gVirtuS 由于不需要在客户端和服务端定制任何驱动，gVirtuS 相对于 GViM 实现了完成的透明化。同时 gVirtuS 还保证了平台无关性，适用于 Xen、VMware、KVM 等主流的虚拟化平台，同时改善了数据通道的性能。从用户体验和数据传输的性能方面来看，gVirtuS 是目前比较完善且面向 CUDA 的 GPU 虚拟化解决方案，本文在第三章将以 gVirtuS 为基础，设计出用户体验更好、性能更高的通用计算解决方案。

随着 GPU 通用计算虚拟化方案的发展，学术界中涌现了一些利用这种虚拟化框架的方案。文献[10]同时引用了 GViM、rCUDA、vCUDA、gVirtuS，并在此基础上实现了核聚合（Kernel Consolidation），同时它对 CKX（Concurrent Kernel eXecution，内核并行执行）的特性也有着很好的支持，允许在计算资源有限的条件下，多个 CUDA 内核函数可以同时在一个 GPU 上执行。CheCUDA[11] 提出了如何实现 CUDA 状态的保存与恢复，在原有检查点方案的基础上实现了 CUDA 的检查点方案，它直接借鉴 GPU 通用计算虚拟化方案中对 CUDA 状态的管理。

文献[12]介绍了在基于 CPU+GPU 异构计算模式的集群中怎样有效地对资源进行调度以完成 SPMD（Single Program Multiple Data，单程序多数据）任务，它通过提出的 GPU 虚拟化解

决方案来实现 GPU 资源的全局调度。该文献在 vCUDA、GViM 和 gVirtuS 三个项目的基础上，从 SPMD 的特点出发，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 完成一一匹配并不现实。它最后在设计的 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。

综上，基于远程 API 的虚拟化解决方案把研究重点或者放在对域间通信和数据传输的优化上，或者依赖于特定的虚拟化平台，并没有针对 GPU 自身的结构特点设计出适合在 GPU 虚拟化环境下的单 GPU 多任务的调度算法，且针对单个计算任务全部由单个 GPU 来完成。本文从提高 GPU 资源的利用率和降低任务的周转时间的需求出发，在 gVirtuS 的基础上引入 GPU 注册中心，注册中心采用集中、灵活的机制对 GPU 资源进行统一管理，对 GPU 资源进行统一调度，通过对多类型多任务计算时间的反馈信息来设置综合负载评价值以实现负载均衡。同时，在虚拟化环境下，通过在主机端设置多个线程控制每块 GPU，根据不同的任务类型和规模设计不同的任务划分方法。针对大规模的计算程序，设计一种单任务多 GPU 协同计算的加速方法，充分利用了 GPU 的计算能力。

1.3 主要研究内容

本课题从 GPU 通用计算虚拟化技术和 CPU+GPU 异构模型发展过程中出现的新需求出发，对虚拟化环境下的 GPU 通用计算技术展开研究，以 NVIDIA 的通用计算框架（CUDA）作为研究对象，对现有的虚拟化环境下 GPU 资源的调度算法进行改进；在虚拟化环境下设计基于 OpenMP 的多 GPU 协同计算方法，对大规模的计算程序进行任务划分，将其分到多块 GPU 进行协同计算，通过实验和实例分析，对虚拟化方案以及多 GPU 协同计算方法的有效性高效性进行验证。

本文的主要贡献如下：

（1）对虚拟化环境下的 GPU 调度算法进行改进。从实现 GPU 的负载均衡及降低任务的周转时间出发，本文通过设置负载评价值对每个节点中每个 GPU 进行负载评价。负载评价值不仅考虑 GPU 的内部结构特征，同时将任务规模、计算复杂度以及时间复杂度纳入考量，使得对 GPU 负载的评价更接近于实际的情况。由于不同的任务类型对 GPU 的计算能力不同，通过对 GPU 计算能力和全局内存设置影响因子，使得负载评价更加客观。对于第一次提交的任务，通过反馈调节算法对影响因子进行调节，得出周转时间最短的最佳影响因子配置比。

（2）设计了一种在虚拟化环境下基于 OpenMP 的多 GPU 协同计算方法。采用科学计算中常用的矩阵相乘和相加的复合运算以及运算量较大的离散傅里叶变换，根据不用的任务规模进行不同的任务划分，通过主机端开多个线程控制多个 GPU 同时进行计算，并将结果汇总到主机端。实验结果表明，随着计算任务规模的增大，多 GPU 相对于单 GPU 的加速比可以达到接近

GPU 的个数。

（3）通过对虚拟机域间通信方式从多个层面进行对比，得出本文 GPU 虚拟化环境下最高效的通信方式。为了最大限度地降低由于虚拟化本身带来的性能影响，本文通过在 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。最后通过在服务端设计两种不同的 GPU 内部数据传输方式，从实验结果中验证了影响多 GPU 协同计算的主要因素。

1.4 论文组织结构

第一章为绪论，介绍了与本文相关的研究背景及研究意义、国内外研究现状、主要研究内容，并概括了本文的主要贡献。

第二章为相关理论基础。介绍了目前主要的 GPU 通用计算方式和 GPU 通用计算虚拟化方面的研究成果，然后介绍了 OpenMP 的应用领域，最后概括了目前 GPU 通用计算虚拟化的研究现状，并对存在的问题加以提炼和分析。

第三章对提出的虚拟化环境下面向多任务的 GPU 通用计算解决方案进行了详细阐述。面向多任务的 GPU 通用计算虚拟化方案解决了当前的 GPU 虚拟化方案对计算任务进行粗粒度调度的问题。引入 GPU 注册中心将 GPU 计算资源在逻辑层次上进行统一管理、统一调度。通过设计的反馈算法对影响因子进行调节，通过最佳的配置比例达到了降低任务周转时间的目的，在理论和实验中对提出的方案进行了分析和验证。

第四章通过在 Xen 和 VMware 两种虚拟化平台下部署 GPU 虚拟化服务，通过对虚拟机域间的不同通信方式在支持性和性能方面进行对比，得出在特定虚拟化环境下虚拟机域间最高效的通信方式。最后设计两种不同的 GPU 内部数据传输方式，从实验结果中验证了影响多 GPU 协同计算的主要因素。

第五章对本文的研究工作进行总结，并对后续可以继续深入研究的工作进行进一步的展望。

第二章　相关理论基础

2.1 基于 GPU 的通用计算

GPU（Graphics Processing Unit，图形处理器）是现代计算机的一种重要的设备之一，它最初的设计是用于图形处理和 3D 渲染。GPU 是显卡的核心，相当于 CPU 之于计算机的作用，GPU 的硬件配置决定着显卡的性能。GPU 因其拥有众多的的处理单元、高速度的带宽、高效的并行性以及超长的图形流水线，这就使得了其在大量重的计算密集型和密集内存访问的计算应用上比 CPU 拥有了更大的优势。GPU 的概念是由 NVIDIA 在 1999 年 8 月发布 GeForce 256 芯片时提出的，之后，GPU 的发展方向出现了重大变革，不仅专注于图形处理领域。经过 NVIDIA、AMD 等 GPU 的主要生产商的努力，通过对硬件和软件进行一系列的升级和改进，使 GPU 的可编程性越来越高，随之也出现了多种多样的 GPU 通用计算接口。

随着 GPU 计算能力的不断增长，GPU 发展方向变革的时机也成熟了。将 GPU 用于图形渲染以外领域的计算称为 GPGPU[13]。信息界对 GPGPU 进行深入研究从 2003 年开始，GPU 俨然从由固定功能单元组成的并行处理器进化成了以通用计算为主的架构，这就直接奠定了基于 GPU 的通用计算的发展基础。到了 DirectX 9 Shader Model 3.0 时代，新的渲染模型在流控制、指令槽方面得到了明显的增强，同时也大大地提升了 GPU 的可编程性，基于 GPU 的通用计算也正在此时得到了快速的发展。2006 年 8 月，ATI（后被 AMD 收购）联手 Stanford 大学，在 Folding@Home[14]项目（蛋白质折叠过程）中对 ATI Radeon X1900 显卡提供支持，在其将显卡加入该项目后，科研的进展速度得到了成倍的提升，同时提升了 GPU 通用计算的普及程度并拓广了其应用领域。随后 NVIDIA 发布了 GeForce 8800GTX，这是显卡领域的首个 DirectX 10 的 GPU，实现了其在 GPGPU 领域的重大进步，在 2007 年提出 CUDA 的概念后，使其在通用计算领域后来居上。第一个支持 DirectX 10 显卡的诞生是一个分界点：统一着色器架构直接取代了在这之前只能处理像素的处理单元，这大大地降低了开发人员的开发难度，并可以方便地对统一着色器进行控制[15]。DirectX 10 时代的 GPU 主要以 NVIDIA G80 和 AMD R600 为代表，它们拥有着数百个运算器（即内核），能够将之前在 CPU 上运行的算法进行并行化达到数十倍甚至数百倍的加速比，这就使得它所提供的计算能力超过了之前所有 GPU。在图形计算领域之外，GPGPU 在财务计算、科研教育以及工业领域都有着非常广泛的应用，在学术界关于它的学术成果也层出不穷。许多实验数据和科研成果已经表明：将 GPU 用于解决在 CPU 中运行的大规模的计算程序可以达到很高的加速比。

2.1.1 经典 GPGPU 技术

最初 GPU 只能处理图形计算任务，它的图形渲染流水线依赖于图形 API 和着色语言，被作为不可编程的硬件处理单元集成到图形处理器中。1998 年以来，GPU 的功能迅速增强，平均每年都会有新一代的 GPU 产生。第一代基于 PC 的图形处理器出现在 1998 年后期，主要以 NVIDIA 的 TNT2，ATI 的 Rage 和 3dfx 的 VOODOO3 为代表，这些 GPU 主要负责图形的光栅化处理。从 1999 年后期开始，第二代图形处理器以 NVIDIA 的 GeForce256、ATI 的 RV200 和 S3 的 Savage3D 为代表，它们可以处理顶点的光照计算和矩阵变换，因此也被称为真正意义上的 GPU。但更深入的变革来源于第三代 GPU，以 NVIDIA 的 GeForce 3 和 GeForce 4Ti 为代表，它们首次引入了可编程性的概念，并将图形硬件的流水线作为流处理器（Stream Processors，SP）来解释，就在此时，基于 GPU 的通用计算正式出现。第四代 GPU 以 ATI 的 R300 为代表，之于第三代 GPU，它们的像素以及顶点可编程性不仅得到了大大的扩展，但其最重要的特点还是它支持了浮点格式的纹理，可以做任意数组，这为通用计算的发展起到有巨大的作用。以 NVIDIA 的 GeForce 6800 为代表第五代 GPU，功能相比较前 4 代在灵活性、丰富程度上更胜一筹，顶点程序对动态分支操作有着很好的支持；像素着色器开始支持分支操作，包括循环和子函数调用，TMU 支持 64 位浮点纹理的过滤和混合，光栅操作处理器对 MRT（Multiple Render Targets，多目标渲染）也同样提供支持等。第六代的 GPU 主要以 NVIDIA 的 GeForce 7800 为代表，第七代主要以 NVIDIA 的 GeForce 8800 为代表，第八代主要有 AMD 的 RV670、RV770 以及 NVIDIA 的 GeForce 200 系列等。与此同时，还出现了一些高级的着色器语言如 Cg（C for Graphics）、HLSL（High Level Shader Language）和 GLSL（OpenGL Shading Language）等，提供了更方便的使用着色器的接口[16]。

随着统一渲染架构的出现，GPU 的可编程性也随之加强，利用 GPU 来完成非图形计算的任务的研究越来越多。最初，经典 GPU 通用计算技术都使用了传统的图形 API，如 OpenGL 等。作为实现手段，这使得它具有高度的可移植性和通用性，对操作系统和 GPU 都没有限制性的要求。经典 GPU 通用计算可以运行在任何存在图形加速功能的图形设备上或者任何支持图形 API 的系统上。至今，仍然有大量的 GPU 通用计算的算法由通用的图形 API 实现，使用经典的 GPU 通用计算编写程序可以方便地使用这些模块。传统的图形 API 本来并不是为了设计通用计算的程序而出现的，它的重点针对图形学的实现，而对于通用计算领域的线程的分配和控制能力较弱，很难为特定的硬件架构提供细致的编程接口。

高级着色器语言提供了给程序开发者对绘图管线更多的控制，但却不需要使用汇编语言或硬件规格语言。使用高级着色器语言来实现 GPU 通用计算虽然在一定程度上降低了开发难度，但并没有在根本上解决编程思想的问题，其使用的编程概念（如纹理、着色器等）都是图形计算中的专业术语，在传统的通用计算编程环境中不能找到对应的对象，对图形计算不了解的程

序开发者需要经历很长的学习时间才能熟练使用。鉴于此，学术界在更高层次对高级着色器语言进行抽象，将纹理表示为流（stream），即数据流，同时将着色器表示成内核（kernel），即对数据流操作的集合。Brook[17]就是该领域的典型作品，Brook 是由斯坦福大学的 Ian Buck 等人开发的流编程语言，它对 ANSI C 进行有限的扩展，同时包含了一个 brcc 编译器。程序开发人员可以使用类 C 语言的 brook C 语言编写程序代码，Brook 通过 brcc 编译器可以将编写的代码编译成 Cg 代码，这就大大地降低了 GPGPU 的开发难度。AMD 公司通过采用了 Brook 的改进版本——Brook+，作为其 GPGPU 产品 Stream 的高级开发语言，Brook+在概念上仍使用上述的流与内核。虽然 Brook 等降低了 GPU 通用计算编程的难度，但它们在底层仍然使用传统的图形学 API，编译效率较低，并缺乏高效的数据通信机制。经典的 GPGPU 计算方法作为最先出现的基于 GPU 的通用计算概念，为后来基于 GPGPU 开发语言的设计提供了方向性的制导。经典的 GPGPU 使用了通用的 API（比如 OpenGL），这使得它的可移植性和通用性较高，对软件的运行环境以及 GPU 的硬件都没有特定的要求。对于程序开发人员来说，为了使用经典的 GPGPU 方法来开发应用程序，程序开发人员必须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 API 并不是为了通用计算而设计的，将传统的图形 API 应用于通用计算领域存在某些方面的缺陷，例如它对于线程的控制能力较弱，在功能上和应用的场景中限制很多，可移植性不高等，这就使得程序开发人员使用图形 API 来实现基于 GPU 的通用算法难度大增。正是由于开发难度大、学习周期长，这一时期的 GPGPU 大多应用都部署在实验室场景中，在实际条件下的使用很少。

2.1.2 独立 GPGPU 技术

自从微软首次提出统一渲染架构以来，基于 GPU 的通用计算的发展进入了新的阶段。统一渲染架构为用户提供了一个通用处理器阵列，对于大规模的并行运算有着较好的支持。所有处理器对程序开发者都是透明的，有统一的软硬件接口，不用考虑硬件构造、功能等的差异，把精力放在算法设计与开发上。各大 GPU 生产厂商针对统一渲染架构发布了新的产品，同时推出了各自独立的 GPGPU 架构，引发了 GPU 通用计算的变革。

目前主流基于 GPU 的通用计算框架主要包括 NVIDIA 的 CUDA 架构、Khronos Groupde OpenCL（Open Computing Language，开放的计算语言）[18]以及微软的 DirectCompute（DirectX 的 GPGPU 解决方案）。其中，CUDA 是目前应用最广泛的一种，CUDA 是 NVIDIA 的私有标准，OpenCL 和 DirectCompute 为开放标准。与 NVIDIA 不同，另一个主要的 GPU 生产厂商 AMD 已经完全放弃了 Brook+，转而支持 OpenCL 标准。从市场角度看，目前 CUDA 是最流行的 GPU 通用计算架构，同时它也是目前较为完整的 GPU 通用计算解决方案。

2008 年，苹果公司和 AMD、IBM、因特尔、NVIDIA 等业界知名制造商一起合作，希望

建立一个真正支持异构平台的应用程序接口，这样的异构平台包含着各种各样的硬件资源，如CPU、GPU、内存、FPGA 等。2008 年 6 月，由各个软件和处理器制造商代表所组成的 Khronos[17] 计算工作组宣布成立，它主要负责在苹果公司计划书的基础上继续对该异构平台架构的功能进行改进。这个跨平台的计算框架就是 OpenCL，它包含了一套用于控制计算平台的 API 和一个用于编写内核的语言 OpenCL C，其结构类似于传统的图形学 API OpenGL。由于 CUDA 架构在工业界和学术界的广泛应用，OpenCL 在很多方面都吸收和借鉴了 CUDA 架构的特点，同时相对于 CUDA 更加的通用、开放。在通用性方面，OpenCL 必须保证不同硬件平台之间的通用性，这使得其很多时候不能利用硬件平台的自身结构特性，从而在执行效率上存在着诸多不足。基于上述缺点， OpenCL 在其第二个版本 OpenCL 1.1 中，增加了大量提高并行计算功能性、灵活性和执行效率的新特性。

作为软件领域的领军者，微软并没有支持 OpenCL 的标准，而是自己提出了一套用于 GPU 通用计算的应用程序接口——DirectCompute，并集成在 DirectX 内部。支持 DirectX 10 的 GPU 可以利用 DirectCompute 的一个子集进行通用计算，支持 DirectX 11 的 GPU 则可以使用完整的 DirectCompute 功能[17]。从 DirectX 11 开始，DirectX 增加了一种计算着色器（Compute Shader），专门为 GPU 通用计算设计。DirectCompute 正是对这种计算着色器调用和控制的接口，同时支持与其他 DirectX 特性之间的交互。DirectCompute 目前仅限于 Windows 上应用，但是由于微软在操作系统上的统治地位，DirectCompute 被所有主流的 GPU 生产商支持。

2007 年 6 月，NVIDIA 发布了其首版通用并行计算架构 CUDA，向用户提供着利用 NVIDIA GPU 产品进行通用计算开发的全套工具，它不只提供一种编程语言，同时提供了 NVIDIA 对于 GPU 通用计算完整的解决方案：不仅提供支持通用计算并行架构的硬件——GPU，同时提供了为实现计算所需的硬件驱动程序、API、程序库、编译器以及调试器等。NVIDIA 提供了一种较为简便的方式编写应用于 CUDA 架构上的 GPGPU 代码，这是一种基于 C 的高级语言——CUDA C。CUDA C 是含有 NVIDIA 限制和扩展的类 C 语言，它对现有的大多数 C 语言的语法和指令都提供支持，同时加入一定的语言扩展使程序能在 GPU 上顺利地进行多线程计算。GPU 的硬件架构为了支持 CUDA 的关键特性做出了显著的改变，一是采用统一处理架构，可以更加充分有效的利用所有的计算资源，二是引入片内共享存储器，对随机写入和线程间的通信提供了支持，提高了计算效率，这些改进都使 CUDA 更加适应了 GPU 通用计算的需求。CUDA 被推出以后，在流体力学模拟[19]、导航识别、军事模拟、生物医药工程、数理统计分析、天文计算等领域得到了广泛的应用，在计算效率上相对于传统的 CPU 获得了很高的加速比。目前，CUDA 已经成为了当下主流的通用计算解决方案，并成为了事实上的工业标准。

2.2 GPU 虚拟化

2.2.1 虚拟化技术

虚拟机管理器直接对底层硬件进行着控制和管理，并对物理资源进行划分、复用、整合，向上层提供统一、高效、灵活的基于底层硬件的完整的抽象资源。VMM 在虚拟机平台上的作用类似于传统平台下的操作系统，而虚拟机上操作系统的作用类似于传统意义上的应用程序。虚拟机监视器提供了快速部署、资源复用、热迁移、快照等高级功能，广泛应用在信息安全、服务器整合、云计算等领域。

目前具有代表性的 VMM 产品有 Xen[20]、VMware[21]系列（如 Workstation、ESXi 等）、Hyper-V[22]、Virtual PC[23]、KVM [24]等。VMM 可以按其结构不同分为经典 VMM（classic VMM）、宿主型 VMM（host VMM）和混合型 VMM（hybrid VMM）。经典 VMM 结构如图 2.1(a)所示，此类 VMM 直接运行在裸机之上，对硬件资源拥有着完整的控制权，为在其上层运行的虚拟机提供了抽象的硬件资源。虚拟机之间完全隔离，它们对硬件资源的访问都是向 VMM 提出资源请求来实现，并且相互之间自主、独立地运行各自的操作系统和应用程序。由于此类 VMM 直接在裸机上运行，所以 VMM 的内核必须对处理器拥有最高的控制权（如基于 x86 结构下的 ring0），相应地，在虚拟机上运行的操作系统则对处理器拥有较低的控制权。早期的 Xen 为经典 VMM 的代表产品，如图 2.1(b)所示，寄宿型 VMM 与经典 VMM 不同，它并不是直接运行于裸机之上，而是与普通的应用程序类似，运行于操作系统之上。此类 VMM 并没有拥有对硬件的直接控制权，而是通过现有的操作系统来申请硬件资源的使用。寄宿型 VMM 效率上比经典的 VMM 高不少，但是宿主型 VMM 的安装和实现比较简单，同时可以通过宿主操作系统提供一些额外的定制化服务，对用户提供了更好的透明度，从而拥有较大的市场份额。VMware Workstation 为经典的寄宿型 VMM 产品。而混合型的 VMM 综合前两类 VMM 的优点，在拥有了寄宿型 VMM 特性的同时也避免了过高的性能损失。混合型 VMM 处理每一个特权指令都采用纯软件解释执行的方式，而对非特权指令的处理则通过直接执行的方式，KVM 则可以归于这一类型的 VMM。

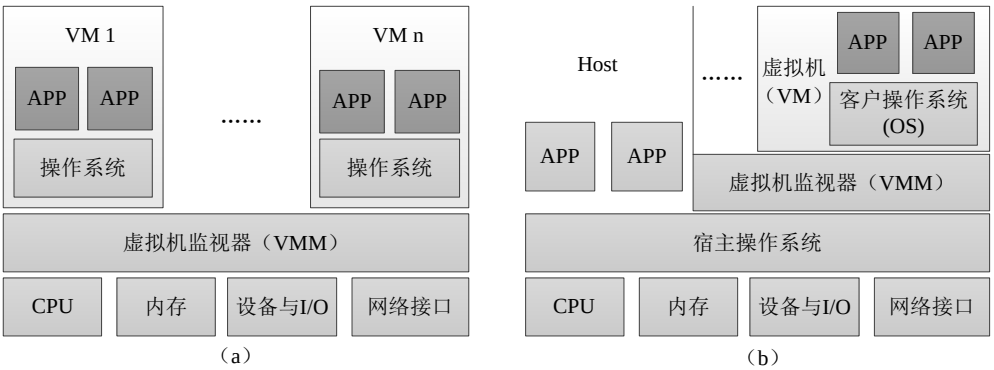

图 2.1 虚拟机监视器结构

2.2.2 GPU 虚拟化

如今 I/O 外设复杂多样，对 I/O 的虚拟化一直都是 VMM 面临的主要瓶颈。GPU 是属于 I/O 设备中较为特殊的设备，首先生产 GPU 的厂家不止一个，同一生产厂家也有多种系列，它们之间的规格大多都不相同，更新速度较快，没有通用的行业标准。其次几个主要的 GPU 生厂商将各自的产品都视为商业机密，并不对外开放任何驱动源码和电气设计的细节。至今，几乎不存在关于 GPU 的统一标准，仅仅在 2D 显示方面有明确的规范，例如 VGA、SVG 等。针对 CPU 的虚拟化是一种硬件方案，支持虚拟化技术的 CPU 通过对硬件的改造和指令集的扩充，使得 VMM 的性能得到进一步的提高，由于 CPU 虚拟化技术研究的比较早，技术也相对较为成熟。而 GPU 无法直接在硬件电气层次进行虚拟，与此同时 GPU 往往涉及大规模的并行计算，其计算效率也无法用 CPU 来模拟。基于上述原因，GPU 至今没有出现完善、通用的虚拟化解决方案。在目前主流的 VMM 中，在虚拟机中仅仅包括一个支持基本 2D 显示的虚拟显卡，对于 3D 渲染和通用计算的支持却基本不涉及。

目前，学术界对于 GPU 虚拟化的解决方案主要分为三类：设备仿真、设备独占以及 API 虚拟化。

1) 设备仿真

设备仿真（device emulation）通过模拟出完整的硬件环境为系统提供一个伪设备。通过设备仿真的方法来实现 GPU 虚拟化，即通过模拟出 GPU 的所有接口语义，为系统在使用 GPU 时提供一个仿真的 GPU。由于目前的 GPU 种类繁多，并且内部机构极为复杂，一般都会包含上百亿个电气单元，其运算速度要远远超过传统的 CPU，如果通过 CPU 来模拟 GPU，虽然在功能上可行，但在效率上相差却非常大，不具备作为 VMM 虚拟设备的条件。NVIDIA 为 CUDA 的早期版本在软件层提供了模拟器用于软件调试的功能，但效率很低，且经过模拟器调试的程序在部署到物理 GPU 时依然会出现一些问题，通过模拟的硬件与真实硬件在效率上和功能上依然存在着不小的差距。例如 Xen 中 HVM Domain（全虚拟化虚拟机）的 I/O 访问模型基于 QEMU[25]，对于 GPU 设备的使用都是通过 QEMU 仿真的显卡——Cirrus CLGD 5446 PCI VGA card，通过仿真的设备仅仅具有 GPU 基本的 2D 显示功能。VMware 在其产品中（Workstation、ESXi 等）实现了一个仿真设备 VMware SVGA II[26]，对于 3D 有一定的支持。

在通用计算方面对 GPU 的设备仿真中，较有影响的两个方案为 Barra、GPGPU-Sim。Barra[27] 是一个 GPU 功能模拟器，它是首个针对 G80 体系架构而设计的，在 2009 年由 Sylvain Collange 等人发布，它是基于 UNISIM 框架的模拟器，针对 CUDA 程序开发过程中的模拟执行而设计的，使用 NVIDIA 的 CUBIN 文件作为输入，能够对执行过程中的状态和行为进行监视。Barra 的运行效率是通过 CPU 模拟 GPU 方式（即 CUDA device_emu 模式）效率的数倍。2009 年 6 月，英属哥伦比亚大学的 Tor M Aamodt 等人发布了一个针对 GPU 通用计算的模拟器

——GPGPU-Sim[28]，GPGPU-Sim 是一个针对统一架构 GPGPU 的体系结构模拟器，可以对各种 GPU 和 CPU 架构进行模拟，既包含功能模拟，又包含性能模拟，是一个全面、兼容的模拟器。Barra 是一个针对 CUDA 的模拟器，而 GPGPU-Sim 是一个全面兼容的模拟器，功能更加强大。

在设备仿真中，实际仿真的设备状态位于 CPU 或内存中，而 VMM 则可以直接管理这一部分，这就使得在设备仿真方式下的 GPU 虚拟化方案可以无缝地继承 VMM 所有的高级特性，如热迁移、快照等。但是使用设备仿真方式在效率上却有严重的下降，CPU 不具备 GPU 超强的大规模并行运算的能力，执行效率太低，在实际环境中并不具有实用价值。

2) 设备独占

设备独占即 PCI pass-through，是指允许客户操作系统直接控制物理设备，可以通过 PCI pass-through 给客户操作系统分配的物理设备有 HBA、USB controller、NIC、声卡等，这会让客户操作系统能够直接地、完全地控制物理设备。设备独占方式的 GPU 虚拟化方案将 GPU 分配给某个特定的虚拟机，只有该虚拟机可以使用该 GPU，拥有完全的、最高的权限。此方式保证了客户操作系统对 GPU 的完全控制，同时也保留了 GPU 的完整性和独立性，使得性能接近于本地环境下的性能。同时，这种方式也保证了其对通用计算领域的支持，但却与虚拟化的本质（资源共享、动态分配）不相符合，不仅增加了计算成本，相对于本地环境下的性能还有一定程度的下降。虚拟化平台对于设备独占方式的部署在早期就已存在，PCI pass-through 技术利用 Intel VT-d[29]技术可以使虚拟机与 I/O 设备直接访问。然而，GPU 并不是简单的 PCI 设备，它需要支持许多遗留下来的 x86 功能，如传统的 IO 端口，文本模式、VGA BIOS、VESA 视频模式、传统的内存范围、正确的和预期的操作等。Xen 4.0.0 版本增加了 VGA pass-through[30]技术，支持了虚拟机在设备独占方式使用 GPU，满足了如上述所述的技术细节，使用 VGA pass-through 方式，虚拟机对 GPU 的使用与在本地使用 GPU 基本相同，但只能被单一虚拟机独占，不能提供类似于 CPU 的分时复用的功能，失去了虚拟化的本质特点，很少在实际中应用。VMware ESXi 中包含了一个类似于 PCI pass-through 的方案——VMDirectPath I/O[31]，它通过设置同样允许虚拟机直接与 PCI 设备交互，包括显卡。

设备独占解决方案在本质上是客户操作系统直接使用原生的显卡驱动来对 GPU 获得使用，绕过了 VMM 参与，缺少 VMM 跟踪和硬件设备的维护状态，此方式下的解决方案不对虚拟机的实时迁移、快照等高级特性提供支持。VMware 在 Workstation 中明确指出，开启了 VGA pass-through 或 VMDirectPath I/O，其对应的虚拟机将会失去虚拟机提供的高级特性。

3) API 虚拟化

API 虚拟化即对应用程序接口的重定向，它通过对与 GPU 相关的 API 进行拦截，使用重定向或者模拟的方式，将计算任务发送至物理机，在物理机中利用相应硬件的 GPU 完成相应功能，

最后将计算的结果通过指定的通信方式返回给客户端应用程序。在 API 层对 GPU 进行虚拟化的方法被学术界广泛应用于传统的图形 API 方案中，如经典的 OpenGL 等。VirtualGL[32]通过使用 API 虚拟化方式对 GLX 库进行虚拟化，对 OpenGL 中的指令进行拦截、解析，并将其重定向至远程服务器，从而通过在远程服务器上进行图形计算来达到渲染的目的。Chromium[33]通过将特定模块插入到 OpenGL 指令流中来对 OpenGL 的执行流程进行干预，实现图形呈现和渲染的大规模分布式处理。虚拟化技术出现之后，VMGL 是第一个针对 OpenGL 应用于虚拟化平台的项目，它使用 API 虚拟化方式对 OpenGL 进行虚拟化，通过在客户操作系统中部署一个伪库来取代 OpenGL 的原生库，伪库具有和 OpenGL 库相同的 API，但伪库实现为对宿主操作系统或者远程服务器的调用，宿主操作系统或者远程服务器都拥有着原生的 OpenGL 库，能够直接驱动物理 GPU 以负责执行本地 OpenGL 调用，并将执行结果进行封装传递给调用者，整个过程对于用户和系统来说是完全透明的，使用 OpenGL 的程序无需对虚拟机监视器作任何的改动。VMGL 是在 Xen 虚拟化平台下实现的，浙江大学何家俊等人实现了 KVM 中图形应用程序的 OPENGL 加速[34]，他们使用的方法只是将 VMGL 中使用的方法移植到基于 KVM 的虚拟机监视器中。VMware 也提供了自己的 GPU 虚拟化方案，它采用 VMware 传统的命令方式前端-后端，其中前端部署在客户操作系统，由 VMware SVGA II 和伪库构成，后端部署于宿主操作系统，由原生显卡驱动和管理线程构成。对于普通的 2D 图形处理，通过设备仿真的 VMware SVGA II 处理，对于 3D 图形的处理则使用对应的 OpenGL 或者 Dircet3D 的 API 虚拟化方式处理。前端与后端的通信采用了 DMA 和管道模式，VMware 系列虚拟化平台在 Guest OS 与 Host OS 之间设置了共享内存区域虚拟显存，明显地减少了数据的复制操作，大大降低了由于虚拟化所带来的性能损失。

API 虚拟化方式通过对 API 层次上的设备状态和参数进行拦截、封装，在一定程度上可以掌握针对特定的 API 设备内部状态，由此以支持在虚拟化平台下的高级特性。VMGL 实现的是针对 OpenGL 的 3D 加速在虚拟机中的挂起/恢复功能和分时复用等一部分高级特性。VMware 则实现的是 2D 的图形处理和基于 OpenGL 和 Direct3D 的 3D 应用的快照、复用、挂起/恢复以及实时迁移等虚拟化平台的高级特性。当然，API 虚拟化的解决方案同样存在着一些不足：首先，并非对于所有的 API 都可以采取 API 虚拟化的解决方案，因为部分 API 设计之初并未充分考虑远程过程调用和虚拟化，在对其虚拟化时有可能破坏原有的语义，如上述所列举的 Chromium 项目目前并不能完全支持 OpenGL 2.0。其次，API 虚拟化需要了解 API 的内部实现细节，对开源的 API 实现起来比较容易，存在着详细的代码注释和文档，但是对于闭源的 API 则是困难重重。再次，API 虚拟化最大的缺点是在对 API 版本的支持上，一般而言，对于某个版本的 API 进行虚拟化并不代表同时支持它的所有不同版本，如果两个版本之间有着较大规模的更改，时常为了支持最新版本，需要将原来的虚拟化方案进行小规模的更改，工作量较大。

最后，API 虚拟化引入了通信开销，这对那些实时性要求很高的应用并不是太适合。当然，在设计良好的数据通路下其性能受到的影响基本可以忽略不计，然而在某些体系中，通用性和高效性往往不能同时实现。

下面将上述三种 GPU 的虚拟化解决方案总结如下表所示：

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
设备仿真	QEMU Xen VGA pass-through	Barra GPGPU-Sim	支持图形渲染
设备独占	VMware Virtual GPU VMware VMDirectPath I/O	Xen VGA pass-through VMware VMDirectPath I/O	不支持
API 虚拟化	VMGL VMware Virtual GPU(3D)	vCUDA、GViM rCUDA、gVirtuS	部分支持

从表 2.1 中可以看出，在三种 GPU 虚拟化方案中，只有设备独占方式可以在虚拟机中执行基于 GPU 的通用计算任务，然而设备独占的解决方案实际上违背了虚拟化技术的本质特点，如不支持实时迁移、快照等虚拟机原有的高级特性，并不适合在实际的虚拟化环境中使用。设备仿真方式虽然在图形渲染方面的技术较为成熟，同时也具备很高的实用价值，但它在基于 GPU 的通用计算领域却基本没有成熟的成果，仅仅开发了模拟器，并没有没有提出针对 GPU 通用计算的虚拟化解决方案。一方面由于设备仿真需要对 GPU 的内部技术细节有较为深入的了解，然而 GPU 生产厂商却将其视为商业机密，对外并不公开；另一方面，目前 GPU 通用计算还没有一个通用的规范标准，无法开发出一个通用的、一致的仿真设备。就当前看来，API 虚拟化方式具备向通用计算扩展的条件，在学术界已经提出了多个可行方案。借鉴与传统的图形渲染方面的 API 虚拟化方案，结合目前主流的通用计算框架，可以发展出可用的通用计算虚拟化框架。

综上所述，近几年来 GPU 虚拟化问题成为了工业界和学术界的一个新热点，并取得了一系列成果，但是仍然有一些问题有待解决：

一、大多数 GPU 虚拟化方法都针对图形渲染，对通用计算的支持基本为零。这是由于通用计算框架提出的时间比较短，并没有太多的时间积累研究经验，另外，通用计算框架大多是私有的标准，其具体的实现细节并不对外开放，这为虚拟化工作带来了很大的难度。

二、GPU 虚拟化面临着可用性问题，在性能上与本地环境下有不小的差距。大多数 I/O 外设设计之初并未考虑对虚拟化平台上的支持，GPU 的特点能够体现在大规模并行计算上，多用于处理数据密集型任务，GPU 在虚拟化平台上带来的性能损失尤为严重。

三、目前存在的 GPU 通用计算虚拟化方案或只针对特定的虚拟化平台，或只支持单对 GPU 使用，在单节点多 GPU 环境下的虚拟化方案并不完善，也不能利用多 GPU 对大规模的计算程

序进行并行加速。

2.3 并行计算框架

2.3.1 MPI

属于消息传递模型的 MPI[35]（Message Passing Interface）是用于并行计算的应用程序接口，基于 MPI 的编程技术常用在服务器集群、集群式高性能计算机等非共享内存环境中，并成为这种编程模型的代表。MPI 标准定义了一组函数并且显得很庞大，但是它的最终目的是服务于进程间通信，使应用程序可以将消息从一个 MPI 进程送到另一个 MPI 进程，因此它需要维护进程 ID、虚拟拓扑结构、同步和进程间通信功能。

2.3.2 OpenMP

OpenMP（Open Multi-Processing）是属于共享内存编程模型的技术，程序员通过在源代码中加入制导性注释（Compile Directive）、称为编译制导指令（#pragma）来指明程序的并发属性。由此，编译器在编译的时候在并行区将程序并行化，并在数据交互的区域通过加入同步函数达到互斥和通信的效果。当选择忽略这些#pragma，或者编译器不支持 OpenMP 时，程序又可退化为串行的程序，大多数代码仍然可以正常、正确运作，但不能利用多线程来加速程序执行。由于它基于制导，具有简单、移植性好、可扩展性高以及支持增量并行化开发等优点，已经成为共享存储系统中的并行编程标准，OpenMP 支持包含 C、C++、Fortran 的主流的编程语言；支持 OpenMP 的商用编译器包括微软的 Visual Studio 和 Intel Compiler 等，以及开放源码的 GCC、Omni、OMPi 编译器等。

由硅谷图形公司领导的工业协会于 1997 年推出了 OpenMP，它提供了一个非正式的并行编程接口来与 Fortran 77 和 C 语言进行绑定，该工业协会后来对这些绑定进行了扩展，同时对 Fortran 95 提供了支持。现在 OpenMP 规范的制定是由 OpenMP Architecture Review Board 组织管理的，其版本从最初的 1.0 到现在已经发布的是 4.0 版本。

2.3.3 CUDA

CUDA 支持超大规模的线程级并行，在硬件中动态地创建、调度和执行这些线程，这些操作在 CPU 中是重量级的，但是在 CUDA 中却是轻量级的。CUDA 计算架构主要将硬件分为两端：CPU 称为主机端（Host），GPU 作为设备端（Device）。其中 CPU 控制程序整体的串行逻辑以及任务调度，GPU 负责处理可以高度并行化的数据部分。基于 CUDA 架构开发的应用程序中分成两类，一是运行在 CPU 上的代码，称为宿主代码（Host Code），另一类运行在 GPU 上，称之为设备代码（Device Code）。在设备代码中的函数被称为 kernel（内核函数）。如图 2.2 所示，内核函数以 Grid（网格）的形式组织，每个 Grid 内又包含了若干个 block（线程块），每

个 block 内又由若干个 thread（线程）组成。如果采用异步并发的方式的话，在整个 GPU 函数执行的过程中存在三个层次的并行：Grid 之间的并行，Grid 内的 block 之间相互并行、block 内的 thread 之间并行。同一个 Grid 内的不同 block 并行执行，block 间无法通信，实现了 Grid 中 block 间的粗粒度并行。同一个 block 内的不同 thread 不仅能够并行执行，而且能够利用共享存储器（shared memory）以及栅栏同步（barrier synchronization）进行通信，实现了 block 中 thread 间的细粒度并行。图 2.2 所示为 CUDA 的线程结构图。

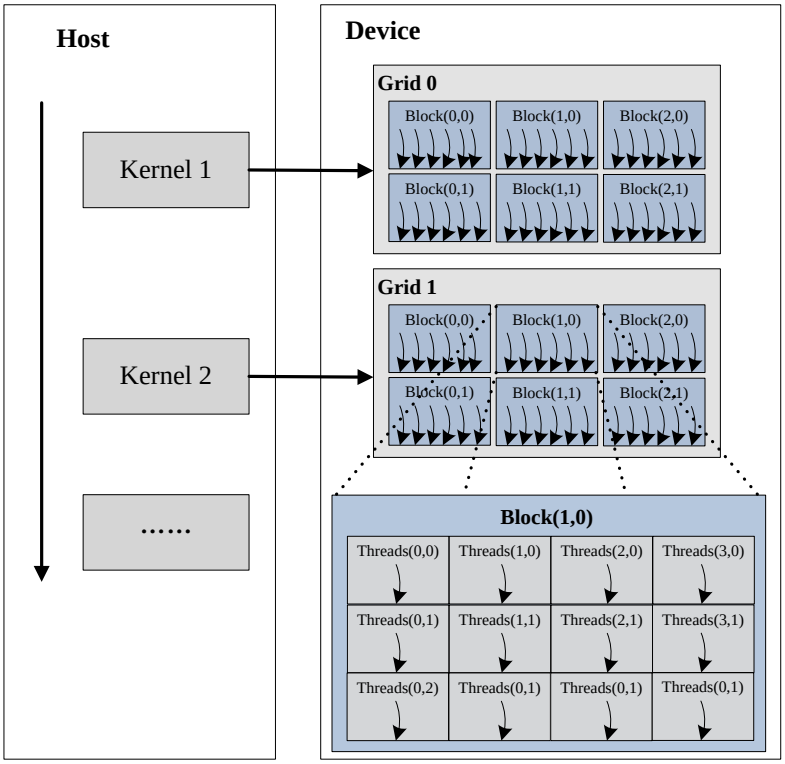

图 2.2 CUDA 线程结构图

CUDA 的语言是 CUDA C 语言，它包含了对 C 语言的最小扩展集和一个运行库，CUDA 的基本软件堆栈由 CUDA 库（CUDA Libraries）、CUDA 运行时（CUDA Runtime）、CUDA 驱动（CUDA Driver）组成，如图 2.3 所示，这三个层次与 CUDA 应用构成了 CUDA 软件堆栈体系。CUDA 库中包含一些常用的较高级别的通用数学库，例如 CUFFT（快速傅里叶变换）、CUDPP（并行操作原语）、CUBLAS（基本线性代数子程序）等。CUDA 提供了两套 API——CUDA runtime API 和 CUDA driver API。CUDA runtime API 和 CUDA driver API 提供了实现设备管理(Device management)、上下文管理(Context management) 、存储器管理(Memory management)、代码块管理（Code Module management）、执行控制（Excution Control）、纹理索引管理（Texture Reference management）、与 OpenGL 和 Direct3D 的互操作性（Interoperity with OpenGL and Direct3D）的应用程序接口。CUDA driver API 是一种基于句柄的底层接口，可以加载二进制或汇编形式的内核函数模块，指定参数，并启动计算。CUDA 驱动程序 API 编程复杂，但是给予

了用户最大的灵活性，能够提供对设备最小粒度的控制。CUDA 运行时 API 是在 CUDA 驱动程序 API 的基础上进行再次封装，使得使用起来更加方便。由于 CUDA runtime API 使用起来方便，编程比较简洁，大部分 CUDA 应用都是使用 CUDA 运行时 API 开发的。CUDA 驱动向 CUDA 运行时提供统一的操作接口，反映了不同架构的 GPU 之间的电气接口的差别，只提供安装包，并不公开源码。

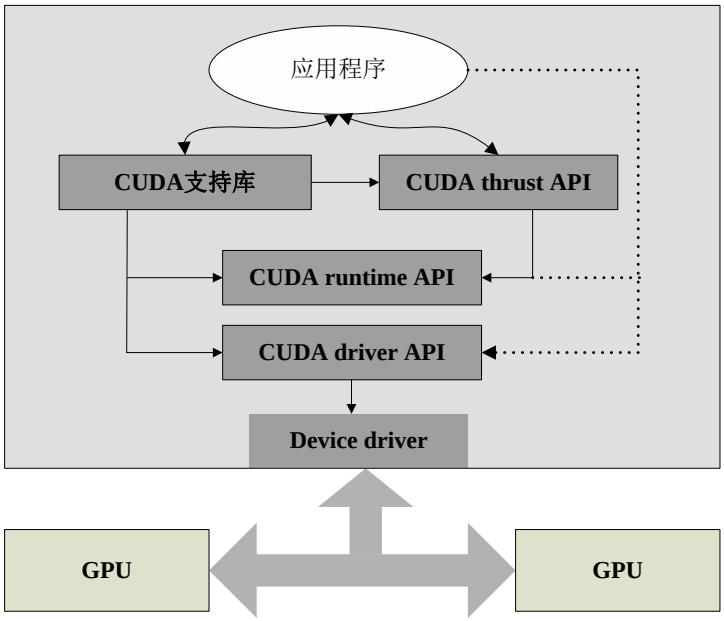

图 2.3 CUDA 软件堆栈

CUDA 程序的编译也有着自己专用的编译器——nvcc。nvcc 首先通过 CUDAfe 分离源文件中的宿主端代码和设备端代码，然后再调用不同的编译器对宿主端和设备端代码分别编译。设备端代码使用扩展部分的语言，由 nvcc 编译成 PTX（Parallel Thread Execution）代码或者二进制代码。宿主端代码由正常的 C 编译器编译，最后将设备端代码链接进来，将 cubin 对象作为全局初始化数据数组嵌入到宿主段代码中。

2.4 本章小结

本章首先在前段介绍了通用计算的由来以及发展历程，分别阐述了经典的 GPGPU 和独立的 GPGPU 技术。接着介绍了 GPU 虚拟化的相关理论基础以及 GPU 虚拟化的几种主要方式，总结了当前 GPU 虚拟化解决方案的缺点，得出 API 虚拟化是目前解决面向 CUDA 的 GPU 虚拟化有效方案。介绍了当前面向 CPU 和 GPU 的几种并行计算框架并重点介绍了 CUDA，为后续章节设计虚拟化环境下的多 GPU 协同计算方法提供了一定的理论基础。

第三章　虚拟化环境下多 GPU 协同计算方法

目前学术界采用 API 重定向的方式将 GPU 虚拟化向通用计算领域扩展，已经提出了多个可行方案。但目前存在的基于通用计算的 GPU 虚拟化解决方案在大部分应用场景下功能过于单一，大多将研究重点放在对用户的透明性以及虚拟机之间的数据传输优化方面。同时目前的 GPU 通用计算虚拟化解决方案均针对单 GPU 环境而设计，在单节点多 GPU 的环境下，不能利用多 GPU 对大规模的计算程序进行加速。

为了使基于 GPU 的通用计算虚拟化方案对多任务环境下有更好的支持，本章对现有虚拟化环境下的 GPU 调度算法进行改进，通过在客户端与服务器中间加入 GPU 注册中心，将客户端的任务请求提交至 GPU 注册中心，达到多任务并发执行的目的。同时，通过引入负载评价值作为调度的依据，将 GPU 自身结构特点以及任务规模及其计算复杂度纳入考量，从而达到降低任务周转时间的目的。

在协同计算方面，学术界将研究重点放在了 CPU 和 GPU 协同计算方面，对于多 GPU 环境下的 GPU 之间协同计算却基本上不涉及。本章通过将 OpenMP 引入到 GPU 中，在虚拟化环境下设计基于 OpenMP 的多 GPU 协同计算方法，针对大规模的计算程序利用多 GPU 对其进行加速。研究虚拟化环境下的多 GPU 协同计算方法对异构平台下的超级计算、云计算、网格计算具有重大理论意义和现实意义。

3.1 研究基础

文献[9]提出了一种 GPU 虚拟化和共享服务的系统——gVirtuS。它通过在用户层拦截并重定向 CUDA API，经由伪库（与 CUDA 运行时 API 具有相同的接口，但将其中的实现改为远程重定向）将 CUDA API 转换成远程调用，最后通过 VMsocket 等域间通信机制传回给用户，对于 CUDA 应用程序而言，所有的操作都像发生在本地。设计的主要目标就是使一个虚拟机运行在高性能计算云上，从而正确地利用 NVIDIA 的 CUDA 系统。在讨论系统结构的框架时，文献着重强调 gVirtuS 的主要特点——平台无关性和完全的透明性，并没有提出支持多用户并发的解决方案。

文献[36]在 CUDA 框架基础上提出了新的 GPU 虚拟化方案，通过引入 GPU 注册中心组件，对 GPU 资源采用集中、灵活的管理机制，并统一调度任务，提升了 GPU 资源的利用率，但是并没有给出在 GPU 上执行任务的复杂度信息，因此只完成了粗粒度的调度。同时，该方案中设置了基于虚拟化平台的通信策略，使得位于同一物理机上的虚拟机可以通过共享内存进行高效的通信，支持不同物理机上的虚拟机间通过网络远程通信。

本章将以上述两篇文献为基础，立足于支持多用户并发及降低任务周转时间，在虚拟化环境下给出一种更细粒度、更灵活的 GPU 资源调度方案。同时，在基于 gVirtuS 的 GPU 虚拟化环境下，将 OpenMP 引入 GPU 通用计算领域，针对大规模的计算程序设计一种基于 OpenMP 的多 GPU 协同计算方法。

3.2 研究目标

为了在 GPU 虚拟化环境下实现多用户并发并降低任务的平均周转时间的目的，本文在文献[9]和文献[36]的基础上提出了一种 GPU 虚拟化环境下的多任务调度算法。该算法充分考虑到在实际环境中多任务并发请求以及任务的周转时间的需求问题，除了考虑 GPU 自身结构配置因素外，还将任务的规模及其计算复杂度纳入负载评价综合考量。这不仅更符合 GPU 虚拟化的实际应用环境，也实现了该环境下的更细粒度的调度，达到降低周转时间的目的。此外，算法中引入任务复杂度的任务信息表，设计基于 GPU 计算能力和全局内存影响因子的反馈算法，使得同一任务对 GPU 配置的需求更接近于实际情况。通过本算法，不仅可以支持复杂环境下多用户多任务的并发请求，同时能更很好地满足用户对周转时间的需求。

为了在虚拟化环境下对大规模的计算程序利用多 GPU 对其进行加速，本章在 GPU 虚拟化环境下引入 OpenMP，使用 OpenMP 在 CPU 端开与 GPU 个数相同的线程数量，线程号与 GPU 的设备号一一对应。对于大规模的计算程序，通过将 CPU 端的线程 id 与 GPU 的设备 id 绑定起来，在 OpenMP 指定的编译制导语句区达到多个 GPU 协同计算的目的。

3.3 面向多任务的 GPU 调度算法改进

文献[36]设计了一个基于 GPU 特征的调度算法，其将 GPU 的计算能力和 GPU 的全局内存引入到负载评价中，同时任务的规模也是负载评价所考虑的元素。但其对 GPU 的计算能力和全局内存的影响因子都设置为固定的 0.5，与实际环境中任务对计算能力和全局内存的要求不尽相符，且并未考虑具有相同规模却有着不同的计算复杂度的任务类型，故此调度的算法较为简单，与实际环境有不小的差别。

本文在此调度算法的基础上，提出一种更细粒度的负载评价方法。通过将任务的计算复杂度和时间复杂度引入负载评价，对 GPU 的计算能力和全局内存的影响因子进行反馈调节，达到更细粒度负载评价。本文基于 GPU 资源的调度算法由§3.3.1 中 GPU 注册中心来完成，采用 gVirtuS 的客户端和服务端组件，并将客户端的任务提交至 GPU 注册中心，由 GPU 注册中心根据负载情况对任务进行统一调度。以下内容为对调度算法的详细描述。

3.3.1 总体架构

本文虚拟化总体框架中的服务端和客户端借鉴文献[9]中的 gVirtuS，同时引入 GPU 注册中

心。以基于 Workstation 的虚拟化平台为例，本文的虚拟化方案整体架构如图 3.1 所示。整个系统可以分为横向和纵向，按横向可以分为应用层和系统层，纵向可以分为物理机和虚拟机。在物理机中的系统层中存在着若干的 GPU，GPU 将自己的资源信息提供给 GPU 注册中心，注册中心对 CUDA 客户端的调用请求进行响应，并将负载最轻的 GPU 信息返回给 CUDA 客户端。

在整个系统的结构中，CUDA 客户端的组件位于虚拟机应用层，CUDA 服务端组件位于物理机中的应用层，GPU 注册中心可位于任意能同 CUDA 客户端和 CUDA 服务端通信的物理机或者虚拟机中，本文将其设立在与 CUDA 客户端处于同一虚拟机监视器的另一虚拟机中。如图 3.1 所示为系统架构图。

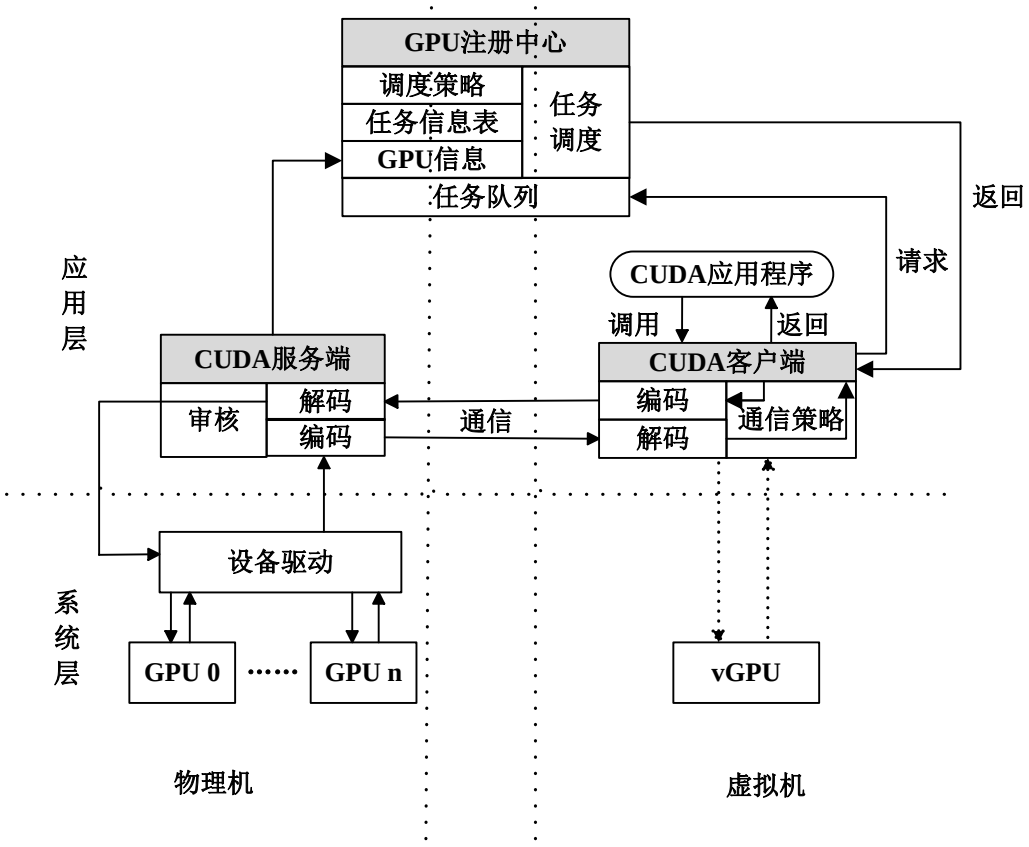

图 3.1 系统架构图

其中，CUDA 客户端面向 CUDA 应用程序，主要完成的功能有：1)对应用程序中 CUDA API 参数进行拦截；2）向 GPU 注册中心索取 GPU 资源并获得 GPU 所在物理机的 IP 号及其设备号；3)根据自己与 CUDA 服务端的位置关系选择与其通信的策略；4)对 CUDA API 的接口和参数进行打包、封装以及编码；5)对 CUDA 服务端返回的数据进行解码，并返回给调用者。

对 CUDA API 参数进行拦截是指对其 CUDA API 的参数进行重定向，使其并不调用 CUDA 的原生库。具体实现是通过在 CUDA 客户端维护一个与原生库同名的伪库——libcudart.so，它具有和原生库完全相同的接口却有不同的实现。当 CUDA 程序使用 CUDA 运行时 API 时，所

有 CUDA 调用都会被拦截，将其重定向至伪库。但是伪库的内部实现却完全不同，一是因为原生库并不开源，无法获取到其具体的内部实现；二是对 API 进行拦截仅仅是为了下一步的再次重定向，从而达到虚拟化的目的，所以内部实现并不需要与原生库相同。此外，CUDA 客户端同时设置了 vGPU 用来维护与 CUDA 相关的软硬件状态。vGPU 本身实质上仅仅是一个键值对的数据结构，在其中存储了当前使用的地址空间、显存对象、内存对象等，同时记录了 API 的调用次序。当计算结果返回时，CUDA 客户端会根据结果更新 vGPU。vGPU 的概念与虚拟机中的 vCPU 完全不同，vGPU 的设置主要是为了支持虚拟机的高级特性。

CUDA 服务端组件位于宿主物理机中的应用层。宿主物理机可以直接与硬件进行交互，因此服务端组件可以直接利用物理 GPU 来完成通用计算任务。CUDA 服务端面向物理 GPU，其功能主要有：1）将 GPU 信息提交至 GPU 注册中心，并定时将 GPU 的负载信息提交至 GPU 注册中心；2）接收 CUDA 客户端的数据报，对其中的参数和调用进行解析；3）对参数和调用进行审核；4）利用本地 CUDA 调用本地 GPU 对审核通过的调用进行计算；5）将结果进行编码，并返回给 CUDA 客户端；6）对系统中支持 CUDA 的 GPU 进行管理。

此外，CUDA 服务端运行的第一个任务是将自身支持 CUDA 的 GPU 设备的信息提交至 GPU 注册中心中。CUDA 服务端收到 CUDA 客户端的请求时，为每个 CUDA 应用分配独立的服务线程。CUDA 服务端采用虚拟机 ID 和进程 ID 进行识别，相应的 CUDA 客户端编码是在数据报之前加上相应的识别信息。CUDA 服务端收到数据报后，解码后对相应的识别信息进行审核，对审核通过的识别信息启动对应的工作线程。工作线程也由 CUDA 服务端创建，它的生存周期和 CUDA 应用的调用周期相同，开始于第一个调用，结束于最后一个调用。工作线程按 FCFS（First Come First Served，先来先服务）的原则处理 CUDA 应用，从数据报中获取 API 和参数，然后调用本地 GPU 资源处理。

GPU 注册中心可以位于任意能同客户端和服务端通信的物理机或者虚拟机中，它将 GPU 资源在逻辑层次上进行隔离并统一管理。GPU 注册中心调度的原则是尽量使在同一个物理机上的 GPU 需求自给，如果该物理机上具备满足条件的 GPU 资源，则将本地 GPU 的负载评价值乘以一个系数（0~1 之间）。在一般情况下，该物理机上的虚拟机的 GPU 需求都重定向到该物理机的 CUDA 服务端。

GPU 注册中心对 GPU 资源进行统一管理，其功能主要有：1）响应服务端 GPU 注册请求，维护所有注册 GPU 的负载信息；2）响应服务端提交的 GPU 信息，更新 GPU 的负载评价值；3）响应客户端 GPU 资源请求，将负载评价值最小的 GPU 所在物理机的 IP 和设备号返回给客户端；4）反馈调节：利用反馈算法对首次提交的任务类型进行调节，找出 GPU 计算能力和全局内存影响因子的最佳配置。

GPU 注册中心的工作流程如下图 3.2 所示。客户端首先向 GPU 注册信息请求 GPU 资源，

GPU 注册信息对于它所维护的 GPU 的 *I* 值进行更新，并将负载值（如图 3.2 中的 *I*）最小的 GPU 信息返回给 CUDA 客户端。客户端向 GPU 注册中心返回的服务端提交任务请求，CUDA 服务端利用本地 GPU 进行 CUDA 计算，将计算结果进行编码返回给 CUDA 客户端。CUDA 每完成一个 CUDA 调用后就将自己的负载情况及时提交给 GPU 注册中心，再由 GPU 注册中心对自己所维护的 GPU 信息进行维护。至此，CUDA 的一次调用完成。

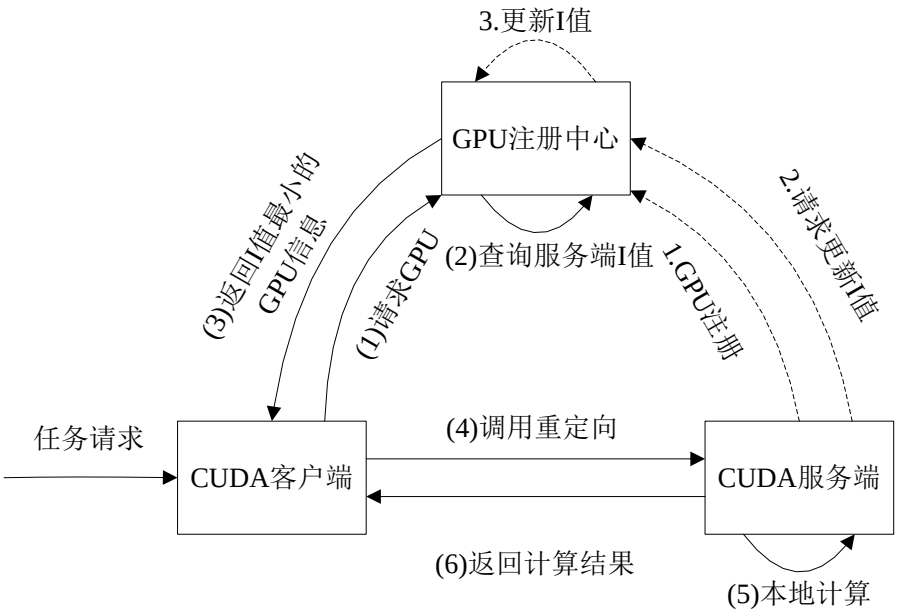

图 3.2 GPU 注册中心工作流程

3.3.2 基础元素

本文的 GPU 调度算法由 GPU 注册中心完成，调度流程如图 3.2 所示。由文献[37]可知，GPU 运算的速度并非取决于 block 数量或者 thread 数量，而是与 SM 数量密切相关，并且运算效率与 GPU 本身的内核数和时钟频率乘积大概成正比，且 GPU 全局内存的大小对计算效率的影响也非常大，如果计算规模大于 GPU 全局内存规模，计算不能在 GPU 上一次完成，会引入额外的通信开销[37]。故注册信息中主要包含处理核心的数量和时钟频率以及全局内存，同时注册中心将任务规模、计算复杂度和时间复杂度纳入负载评价的考量范围，并以此为依据对 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 *I*：$I=\sum_{i=1}^{N}\frac{Scale_i*cmplx_i}{\alpha_i*\mathrm{P}*\mathrm{R}+\beta_i*\mathrm{G}}$，参数定义如表 3.1 所示。

其中，任务规模 $Scale_i$ 由 CUDA 应用程序提供在接口参数中。 GPU 注册中心维护着一张任务信息表，不同任务类型对应于不同的复杂度以及相应的 α_i 、 β_i 值， $cmplx_i$ 由 CUDA 客户

端提供的任务类型所决定，在本文中 $cmplx_i$ 设置成任务的时间复杂度乘以一个多处理器在一个时钟周期下完成任务数的倒数。如单精度浮点加、乘、乘加的任务类型都定义为 1，每个时钟周期操作次数为 8 次，其相应的复杂度为 1/8；单精度浮点数求倒的任务类型为 2，每个时钟周期完成的操作次数为 2，其复杂度为 1/2；除法的任务类型为 3，每个时钟周期完成的操作数为 0.88，其复杂度为 1/0.88 等。P、R、G 等均由 CUDA 服务端的注册信息所提供，由 GPU 注册中心维护。

表 3.2 参数定义

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
$cmplx_i$	第 i 个任务的计算复杂度
P	GPU 的内核数
R	GPU 的时钟频率
G	GPU 的全局内存
α_i	GPU 处理能力对第 i 个任务的影响因子
β_i	GPU 全局内存对第 i 个任务的影响因子

3.3.3 算法流程

步骤 1：调度算法

GPU 注册中心依据 CUDA 任务的调度原则返回注册中心的 GPU 信息中 I 值最小的那个，I 值越小，说明当前 GPU 上的负载越小。另外调度的另一基本原则是 CUDA 任务优先本地处理，这是由 GPU 虚拟化方案的通信机制所决定的，这样可大大降低由于数据传输所带来的性能开销。CUDA 服务端向 GPU 注册中心提交 GPU 信息时，不仅需要提交自己所在的服务器的 IP 地址和设备号，同时需要提供该 GPU 的主频、内核数及全局内存等信息。注册中心则根据负载评价值计算公式计算出负载评价值，并按升序排列。CUDA 客户端向注册中心请求服务时，注册中心首先更新服务端的综合负载评价值，然后将本地服务端的负载评价值乘以一个权值系数，并加入表项中排序，得出一最小的项即负载最低的，并将其分配给 CUDA 客户端。权值系数的大小可以控制本地服务端的优先级，权值系数设置的越小，本地服务端提供服务的可能性越大。权值系数可以看作是本地服务端的优先级和自身负载的一个比值。

具体的调度流程图如下，图 3.3 所示为注册中心的调度流程图。

1） GPU 任务到来，判断注册信息是否为空，是则说明目前没有可用的 GPU 资源，任务失

败，否则转向 2；

2） 信息表中是否有对应的任务类型，有则说明有对应的α和β的值，进入 5，否则进入 3；

3） 任务规模大于全局内存的平均值，进入 4，否则设置α=1，β=0，进入 5；

4） 将α和β均设置为 0.5，记录任务类型，将全局变量 sum 加 1，待反馈调节，进入 5；

5） 更新信息表中的 I 值，如果存在本地 GPU，则将其乘以 0-1 之间的系数，否则转向 7；

6） 按 I 值升序排列，返回最小项的信息给客户端；

7） 结束。

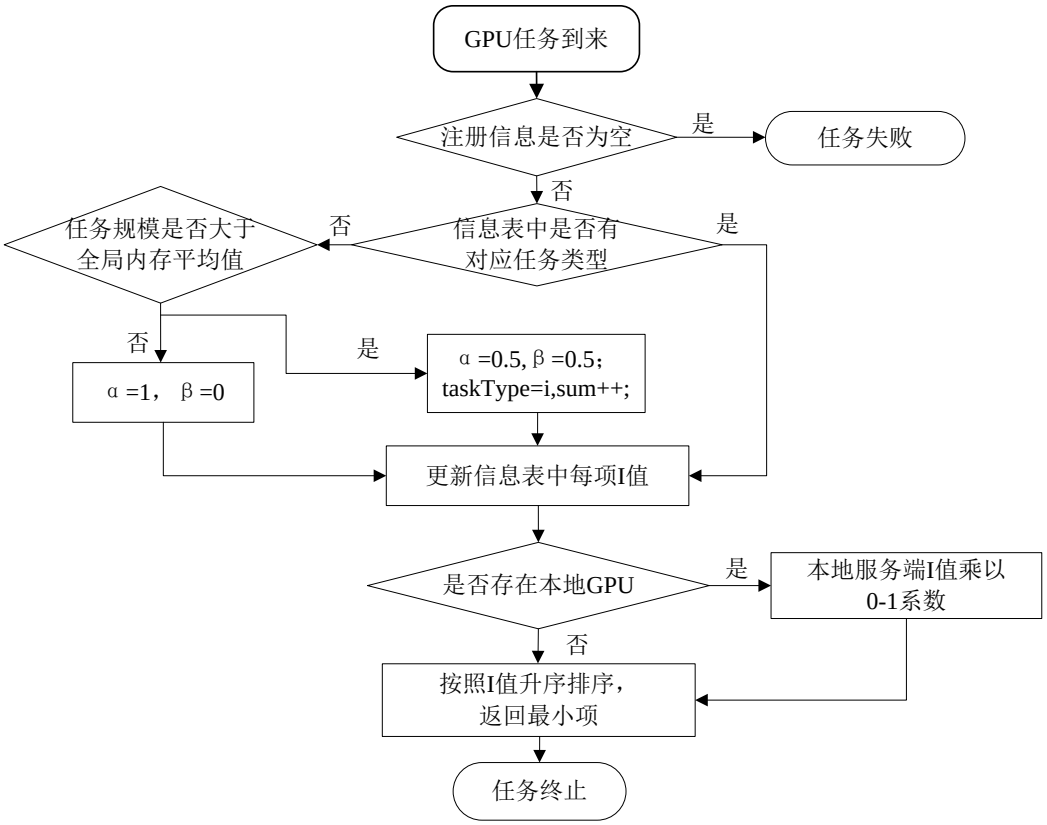

图 3.3 GPU 注册中心调度流程

步骤 2：反馈调节

在初始化时，为 CUDA 中几种常用的一些科学运算通过反馈调节设置了 GPU 处理能力及其全局内存对它们的影响因子，其中运算的规模全部大于全局内存，运算不能一次性在 GPU 中完成。在反馈调节算法中，本文通过客户端随机生成一定数量的任务数提交至 GPU 注册中心，并由注册中心根据调度算法将执行任务的 GPU 信息返回给客户端。

反馈调节的算法流程如下，图 3.4 所谓为反馈调节的流程图：

1）I 之和为零，表示所有 GPU 资源均为空闲，此时可进行反馈调节算法，否则继续等待；

2）sum 设置为全局变量，值等于 0 表示注册中心暂时没有需要进行反馈调节的任务，进入 6，否则进入 3；

3）在客户端随机生成一定数量的任务，且每个任务的规模都大于全局内存，将α和β初始化，进入 4；

4）注册中心根据调度结果返回服务端 IP 及 GPU ID；

5）统计周转时间，α是否等于 0，等于 0 表示调节完成，返回周转时间最小值的α和β，写入任务信息表，转入 1；

6）结束。

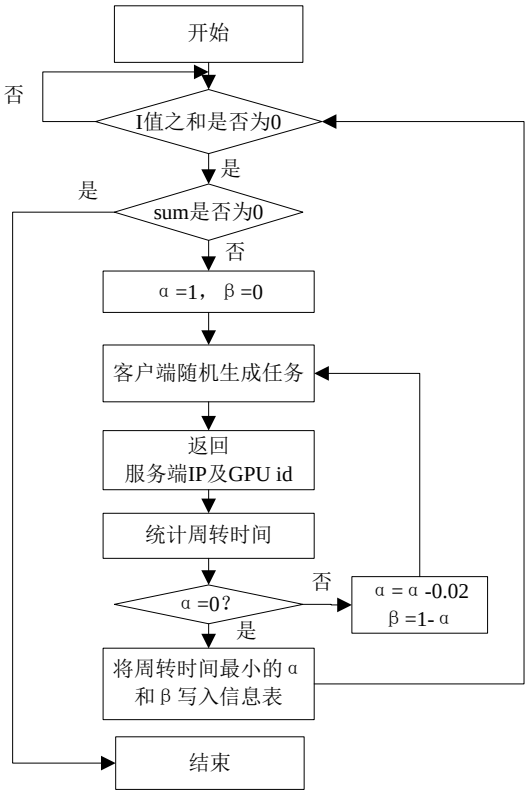

图 3.4 反馈调节算法流程

3.4 面向单任务的多 GPU 协同计算方法

随着 CPU 主频的提高变得越来越困难，各种并行化的软硬件技术应运而生，随之而来的是软件复杂度大大增加。CUDA 技术是 NVIDIA 公司发布在 GPU 上编写通用程序的编程框架，具有高度并行化的程序在 GPU 上执行的性能可达到在 CPU 上执行性能的数十倍以上，一般可达到几百甚至上千。然而，CUDA 并没有针对多 GPU 环境下的 API，国际和国内上对多 GPU 协同计算也仅仅停留在物理机上，对于在虚拟化环境中的多 GPU 研究尚处空白。

本节先从虚拟化环境下多 GPU 协同计算的工作流程开始，详细介绍多 GPU 协同计算过程

中涉及的数据分解（即任务划分）、数据计算以及最后的数据合并。通过在理论和实验中得出多 GPU 的加速比以及影响多 GPU 协作性能的主要因素。

3.4.1 数据分解

数据分解是将大规模的计算程序分解成多个小规模的任务，并将其交给多个 GPU 来并行执行的过程。对于计算能力相同的 GPU 设备，将数据平均分配至每个 GPU 是最理想也是最高效的分配方法。以单节点内有 4 个 GPU 设备为例，本节利用 OpenMP 在主机端开 4 个线程，每个线程控制一个 GPU 设备，并在 OpenMP 编译制导区域保持计算的同步，控制 4 个 GPU 的协同计算。最后用计算复杂度较大的离散傅里叶变换对虚拟化环境下的多 GPU 协同计算解决方案进行验证，得出加速比。

以矩阵的复合运算 A*B+C*D 为例，下图 3.5 所示为将矩阵的数据分解示意图。

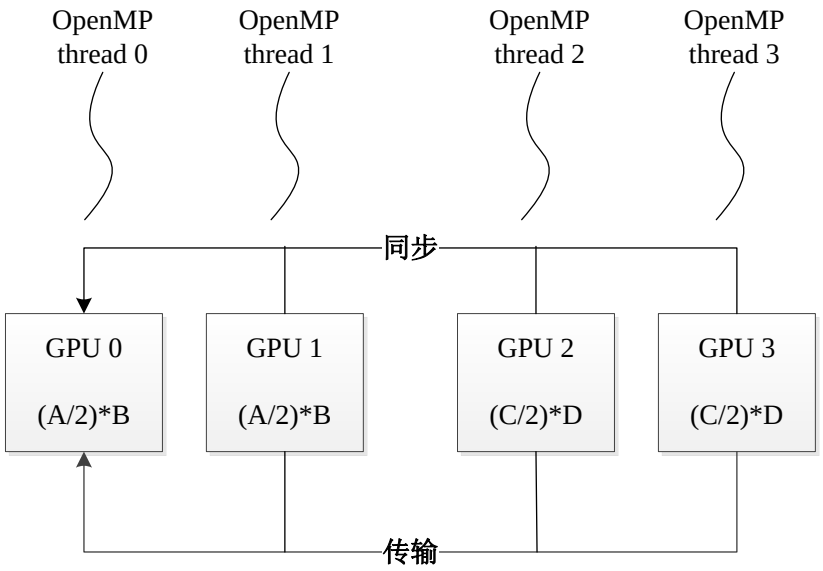

图 3.5 矩阵复合运算数据分解示例

由图 3.5 中可见，GPU 0 完成矩阵 A 的一半乘以 B，GPU 1 完成矩阵 A 的另一半乘以 B；GPU 2 完成矩阵 C 的一半乘以 D，GPU 3 完成矩阵 C 的另一半乘以 D。为保证计算结果的正确性，在做矩阵相乘的过程中保持线程同步，也即每块 GPU 上完成矩阵相乘运算的同时，等待其它 GPU 上的相乘运算全部完成。最后将每块 GPU 计算完成的数据通过 cudaMemcpy()函数拷贝到 GPU 0 上，由于矩阵的加法复杂度相对较低，并行化程度不是太高，所以交由一块 GPU 来完成。最后通过拷贝函数将计算结果传递回主机端并在主机端对计算结果的正确性进行校验。

3.4.2 数据计算与合并

在 3.4 节中，我们的研究目标是对大规模的计算程序利用多 GPU 协同计算与单 GPU 计算

来求出它们的加速比，验证多 GPU 协同计算的有效性和高效性。因此我们在单 GPU 的矩阵运算中并没有采用较前沿的 CUBLAS（CUDA Basic Linear Algebra Subroutines，基本线性代数子例程）库来加速矩阵运算，而是每个线程计算结果矩阵的一个元素，即 A 的一行和 B 的一列（或 C 的一行和 D 的一列）相乘之后再相加。在多 GPU 计算中，通过 OpenMP 在主机端开了和 GPU 个数相同的主机端线程，每个主机端线程负责控制一个 GPU，此时需要在每个设备上为每个线程分配显存，并且各自启动内核函数。矩阵乘法是科学应用中最常见的计算，也是最重要的计算问题之一，许多数值计算的核心都是矩阵乘法，例如 BLAST 测试包中的 GEMM 测试用例。在系统中应用矩阵乘法的解决方案有一定的现实意义和代表意义。

为了简化描述的过程，在本小节以 4 个 GPU 来完成矩阵乘法为例来对数据计算和数据合并进行详细的介绍。基于矩阵乘法的规则，可以容易将矩阵的计算任务按行进行划分。如图 3.6 所示，结果矩阵 C 为一个 N*N 的矩阵，将 C 按行分为 4 个 C/4（N/4*N），则相应的将 A 分为 4 个 A/4（N/4*N），并分别与矩阵 B 相乘，分别得到 4 个 C/4，组合便可得到最后的结果矩阵 C。矩阵乘法的数据划分与合并如图 3.6 所示。

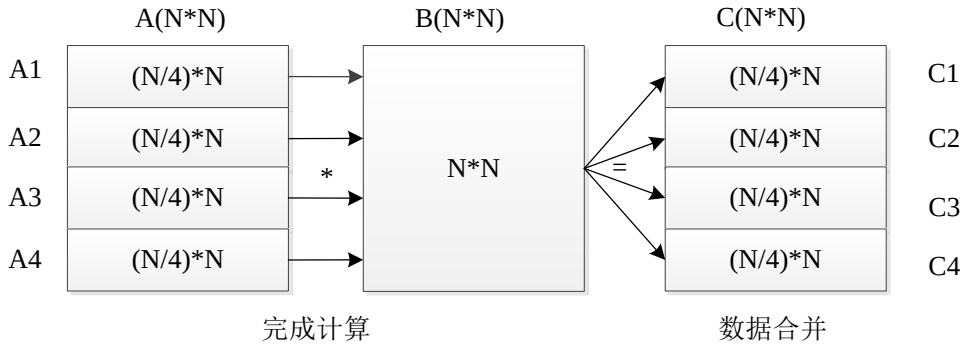

图 3.6 矩阵乘法划分与合并

在具体实现中，核心的算法实现是利用 OpenMP 在主机端开 4 个线程，并在需要控制多 GPU 协同计算的区域通过编译制导语句将其设置成并行区域，通过 cudaSetDevice(cpu_thread_id)函数将主机端线程的 id 号同 GPU 的设备号对应起来，做到一个线程控制一个 GPU。同时在进行规模划分（也即数据分解）的时候，每个线程通过设置自己私有主机端和设备的数据指针，私有主机端线程指向原始指针不同的起始位置，并通过 cudaMemcpy()函数从自己私有主机端线程的位置拷贝 N/4 个数据规模达到规模划分的目的。最后，每个 GPU 将计算完成后的数据从自己私有的设备端数据指针所指向的区域拷贝回主机端指针所指向的区域，这样达到数据合并的目的。

3.5 实验结果及性能分析

3.5.1 实验环境

本文选取科学计算中较为常见的典型应用进行分析，从矩阵乘法方面对于系统的性能进行测试，并对结果进行分析。系统的体系结构由：CUDA 客户端（10 个虚拟机构成）、CUDA 服务端（2 台服务器构成）以及 GPU 注册中心（位于其中一台服务器上的虚拟机中）。系统的软、硬件环境如表 3.2 所示。其中 CUDA 服务端为两台服务器，一台配置为 Intel Xeon E7-4830、Tesla C2050*4、48GB 内存，另一台为 Intel Core i5-2300、Tesla C2050*2、16GB 内存，并在其上部署服务端组件。CUDA 客户端在 10 个虚拟机中，虚拟机则是通过在物理机上的 Workstation 开启，具体配置为两个 vCPU、1G 的内存以及 20G 的外存。GPU 注册中心同 CUDA 客户端处于同一层次的虚拟机中，配置同客户端相同。

表 3.3 系统环境配置

节点名称	数量	硬件环境描述	软件环境描述
CUDA 客户端	10	vCPU*2、1GB 内存	Centos 6.0
CUDA 服务端	1	Intel Xeon E7-4830、Tesla C2050*4、48G 内存	Centos 6.0
CUDA 服务端	1	Intel Core i5-2300、Tesla C2050*2、16G 内存	Centos 6.0
GPU 注册中心	1	vCPU*1、1GB 内存	Centos 6.0

3.5.2 影响因子的获得

本文中α_i、β_i的取值通过加上或减去 0.02 的步长，根据在对给定任务的周转时间的反馈来设定，当周转时间达到最短的时候，对应α_i、β_i的值就为相应的任务类型的值。系统在调度算法中为每个α_i、β_i定义数据结构 struct get_alpha_and_beta{float alpha；float beta；float cycling_time；}state，对每个α_i、β_i的可能取值都进行验证，并将结果按 state->cycling_time 升序排序，返回 state->cycling_time 最小值所对应的α_i、β_i，相同大小则返回平均值。本文以矩阵乘法为例，通过设定不同的α_i、β_i值，将任务发送至不同的 GPU 中，并在服务端生成若干个任务来模拟成实际环境中的负载，最后通过对周转时间的反馈来评价α_i、β_i的最优组合并写入注册信息表。在不同α_i、β_i配置下，对应的周转时间如下图 3.7 所示。当前用户第一次提交矩阵相乘的任务时至不同 GPU 时得到的反馈时间（α和β的取值决定负载评价值 I，进而指定 GPU 完成任务），并由此确定α和β的取值。

结果表明，当α的取值在 0.69~0.83，β在 0.17~0.31 之间时，任务的周转时间最短，则此

时设置α_i=0.76，β_i=0.24，并写入任务信息表。假设 A、B、C 均为 N*N 规模的矩阵，且 C=A*B，则 C 中元素的计算公式为$\sum_{k=1}^{N} A_{ik}B_{kj}$。由此可见，矩阵乘法总计算量的数量级是$O(N^3)$，访问存储量的数量级为$O(N^2)$，计算访存比为$O(N)$，是一个典型的计算密集型任务，因此其对于 GPU 的处理能力相对较高，故而α的取值也相对于β较大。

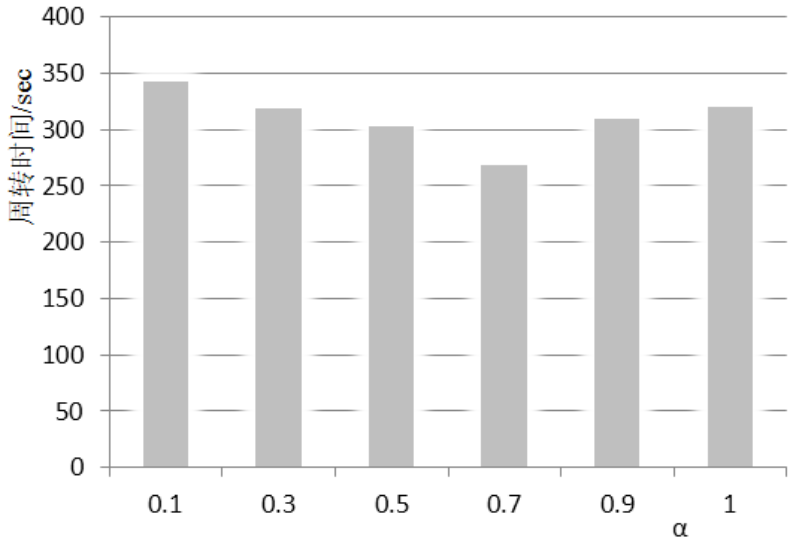

图 3.7 任务在不同α、β配置下的周转时间（β=1−α）

3.5.3 基本性能

针对本章的调度算法，主要从两个方面来验证其可行性和高效性。一是在一定时间内，客户端平均产生任务数服务端的完成情况，另一是系统的平均周转时间。实验环境利用上述 3.4.1 节的实验配置。本节实验利用自己编写的 matrixMul 程序测试，一定时间内在各虚拟机上随机产生 matrixMul 任务，记录任务总共完成次数。实验中的 matrixMul 矩阵参数均设置为 4096*4096。下图 3.8 为本地环境与本文的调度算法和文献[36]中的 GV-GS 的任务调度的比较。图中 native 为本地环境，GV-GS 为文献[36]方案应用基于 GPU 特征的任务调度。native 模式并不能适用于多机环境下，两台服务器并不能在同一时间同时使用，图中得出的数据为两台服务器 native 模式下的平均值。

图 3.8 为在某一台服务器一定时间内平均产生固定任务数的情况。图 3.9 为在某一台服务器上一定时间（时间由任务数决定）内随机产生固定任务数的情况，图中显示的是任务的平均周转时间。

由于 native 模式并不能适用于多机环境下，两台服务器并不能在同一时间同时使用，图 3.8

中 native 得出的数据为两台服务器的平均值。从图 3.8 中可以看出，单独本地环境并不适用在多机环境多任务条件下，在通常情况下只能充分利用一台服务器的处理能力，另外 GV-GS 由于出 GPU 自身特点外，只考虑了任务的规模，并没有将任务的时间复杂度和计算复杂度纳入考量，因此任务调度的粒度相对较粗，经过优化后，本文的任务完成数明显由于 GV-GS。

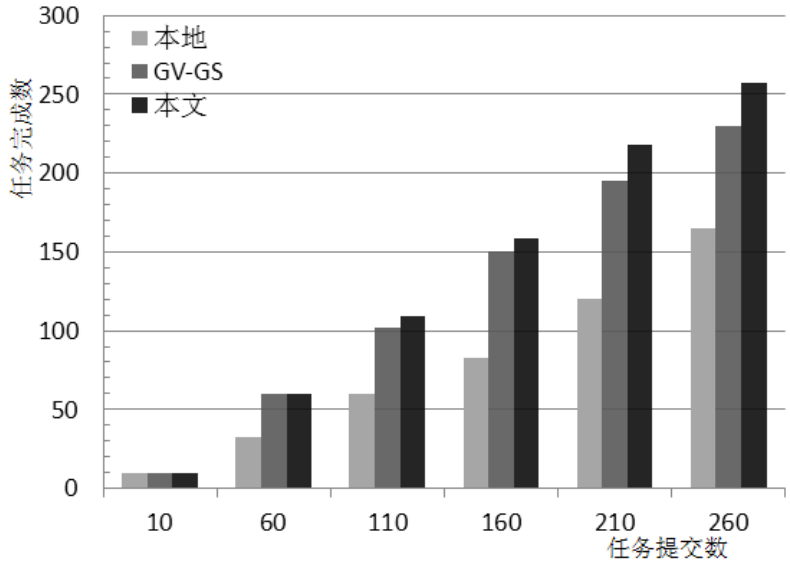

图 3.8 GPU 虚拟化任务平均生成情况下的完成情况

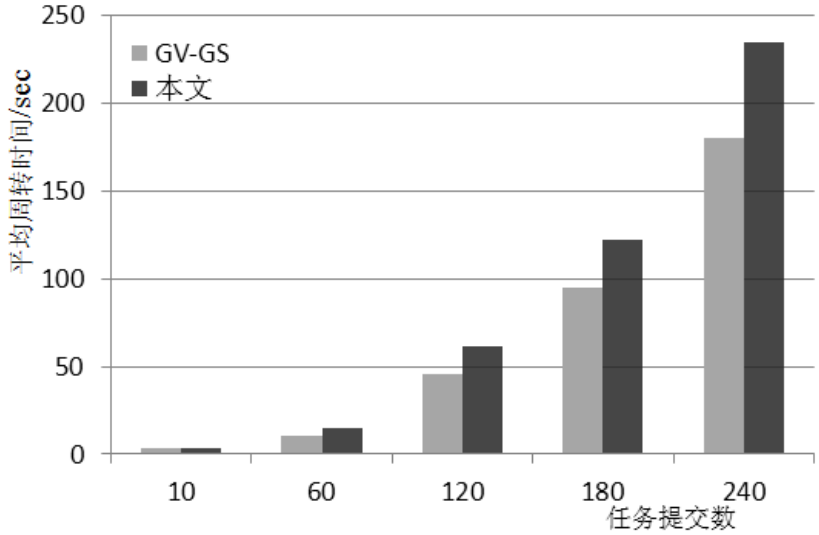

图 3.9 平均周转时间

图 3.9 所示为在客户端生成多个任务，通过 GPU 注册中心对 GPU 资源进行任务调度后整个系统的平均周转时间的情况。从图中可以看出，当任务规模逐渐变大的时候，改进后的调度算法的周转时间明显降低。这是由于在实际情况中，每块 GPU 卡上的任务类型并不完全相同，其计算的时间复杂度和多处理器在每个指令周期完成的操作数也不尽相同，这就导致就算每块卡上的任务数一样，其真实负载却相差很大，采用改进后的调度算法对 GPU 负载的评价不仅能更接近于真实情况，在均衡 GPU 负载的前提下，更加能降低任务的周转时间。

3.5.4 矩阵运算与 DFT 示例

➢ 矩阵运算

在描述了多 GPU 协同计算过程中的数据分解、数据计算与数据合并后，本节就矩阵复合运算和离散傅里叶变换在多 GPU 环境下的基本性能进行实验分析。在程序类型方面，包括较为简单的矩阵相乘的复合运算，也有较为计算复杂度较高的离散傅里叶变换。这两个程序的一些统计特征主要包括它们各自的规模、kernel 执行次数（迭代次数）、使用的显存容量、实验方差，以及在内存和显存之间传递的数据通量。图 3.10 显示的为 A*B+C*D 在多 GPU 与单 GPU 环境下的计算时间，图 3.11 描述的是其加速比。

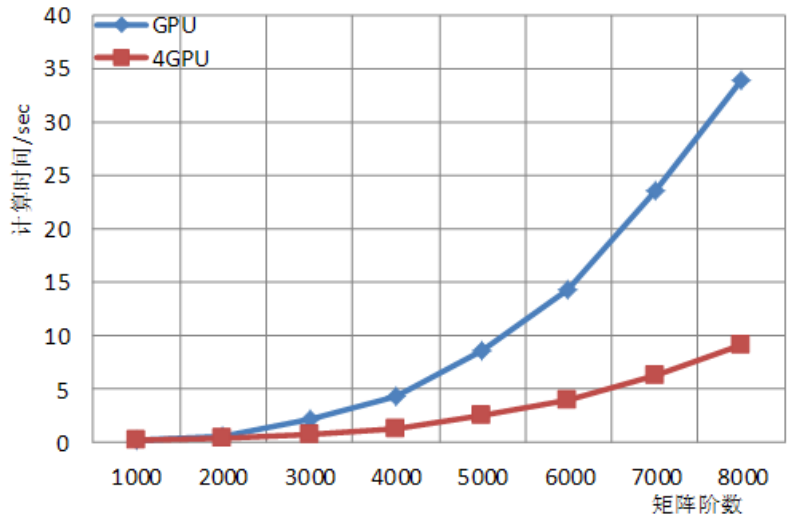

图 3.10 矩阵复合运算多 GPU 与单 GPU 的计算时间

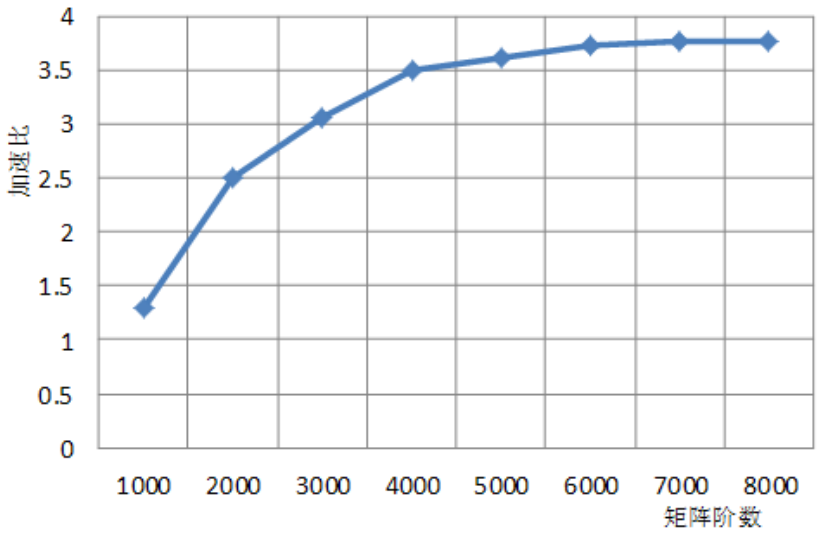

图 3.11 矩阵复合运算多 GPU 与单 GPU 的加速比

为了使矩阵的规模更符合实际环境，实验所用的矩阵规模均为 1000 的整数倍，而并没有采用 2 的 n 次方。如图 3.10，随着矩阵规模逐渐变大，4 个 GPU 所计算的时间呈缓慢增长的态势，

而单 GPU 所计算的时间的增长速度则相对较快。

从图 3.11 可以看出，当矩阵的规模较小的时候，矩阵的计算时间相对于整体的时间（异步数据拷贝+数据计算）的比例较低，因此加速比也相对较低；随着矩阵规模增大时，矩阵的计算时间相对于整体的时间的比例逐渐提高，因此加速比在趋于稳定之前基本呈线性增长的态势，且越来越接近于 GPU 的个数，当加速比达到 3.76 时加速比增长的速度变得非常缓慢。虽然 4 个 GPU 是在矩阵运算的时候是并行工作的，但在进行两次数据拷贝的时候则是异步（对用 PCI-E 总线互斥使用），随着数据规模的增大，异步传输的时间也越来越大，因此数据计算时间占总时间的比例的增长速度也越来越慢，这就使得加速比增长的速度变得非常缓慢。

- 离散傅里叶变换

离散傅里叶变换[38]（Discrete Fourier Transform，DFT），是连续傅里叶变换的延伸，在时域和频域都离散的一种形式，将时域信号的采样变换为在离散时间傅里叶变换（Discrete Time Fourier Transform，DTFT）频域的采样。在形式上，序列在两端（时域和频域）的变换是有限长的，而这两组序列实际上都应被认为是在离散周期信号的主值序列，即使采取对有限长的离散信号作离散傅里叶变换，也应当将其当作经过周期延拓成为周期信号再作变换。

由于离散傅里叶变换在信号处理、图像处理等领域有着广泛的应用，因此选取 DFT 作为验证对象具有一定的理论和现实意义。本文采取计算复杂度较高的离散傅里叶变换来验证在虚拟化环境下多 GPU 相对于单 GPU 的加速比。下面讲述本文采用多 GPU 来加速 DFT 的思路及流程。

离散傅里叶变换的变换对：对于 N 点序列 $\{x[n]\}_{0\le n<N}$（即输入序列有 N 个元素，每个元素皆为复数），它的离散傅里叶变换为：

$$\hat{x}[k]=\sum_{n=0}^{N-1} e^{-i\frac{2\pi}{N}nk}x[n] \quad k=0,1,...,N-1.$$

其中 e 是自然对数的底数，i 是虚数单位。设 $\omega=2\pi kn/N$，采用欧拉函数将上式中 $e^{-i\frac{2\pi}{N}nk}$ 替换成 $\cos(\omega)-i\sin(\omega)$。单 GPU kernel 函数的的核心代码如下所示：

```
{
        int k = (blockDim.x * blockIdx.x + threadIdx.x);
        float tr=0, ti=0; int    n;
        float wn = 2*PI/N*k;
        for (n=0; n<M; n++)
        {
            float w = wn*n;
            tr += ( cin[n].rex*cos(w) + cin[n].imx*sin(w) );
```

```
            ti += ( cin[n].imx*cos(w) - cin[n].rex*sin(w) );
        }
        cout[k].rex = tr;
        cout[k].imx = ti;
    }
```

在采用单 GPU 进行加速的时候，本文采用的方法是去除如上代码的第一重循环，GPU 中的每个线程负责处理一个复数的输出序列。采用多 GPU 的具体方法是每个 GPU 上的输入序列一样，但需要处理的输出序列的规模降为总序列的四分之一。下图 3.12 所示为 DFT 由单个 GPU 和 4 个 GPU 分别执行的计算时间，图 3.13 所示为 4 个 GPU 相对于单个 GPU 的加速比。

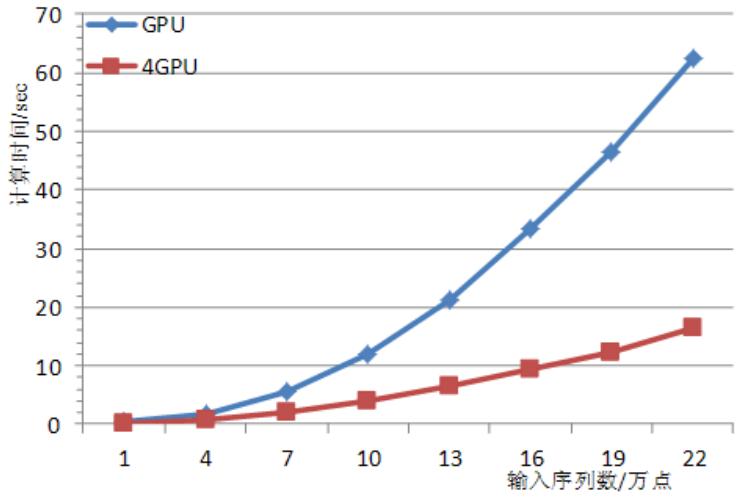

图 3.12 DFT 的 4GPU 和单 GPU 随输入序列数变换的计算时间

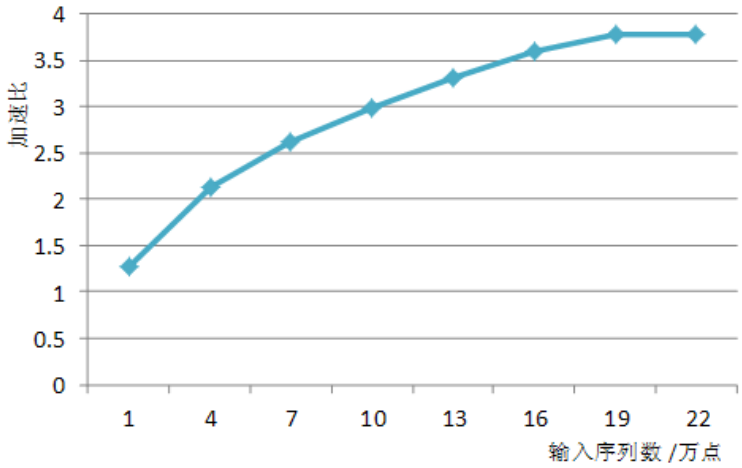

图 3.13 DFT 加速比

由于 DFT 的输入点数的规模为 10000 的整数倍，而并没有采用 2 的 n 次方，为了使 GPU 的资源利用率达到最大化，通过在 NVIDIA 官方的 CUDA_Occupancy_calculator.xls[48]文件设置

不同的线程数对 GPU 的利用率进行计算，经过计算当 GPU 端的线程数设置为 128 时 GPU 的利用率能达到最大。如图 3.12 所示，随着 DFT 输入点数的增大，4 个 GPU 所计算的时间呈缓慢增长的态势，而单 GPU 所计算的时间的增长速度则相对较快。

同理，从图 3.13 可以看出，当输入序列的点数规模较小的时候，其计算时间相对于整体的时间（异步数据拷贝+数据计算）的比例较低，因此加速比也相对较低；随着输入序列点数的增大，加速比越来越接近于 GPU 的个数，当加速比达到 3.784 时基本趋于稳定。这里的加速比变化规律同图 3.11 的相同，另外，DFT 的加速比相对于矩阵运算较高是由于 DFT 本身的数据规模相对于矩阵运算较小，在 GPU 上的主要计算都是三角运算，完成一次计算耗费的指令周期非常大，计算访存比较高，属于计算密集型的算法，所以它由于异步拷贝所带来的时间开销相对较少，加速比也相对来说较高。

3.6 本章小结

本章首先介绍了虚拟化环境下 GPU 的调度算法和多 GPU 协同计算的研究基础，分别对于三个组成部分：CUDA 客户端、CUDA 服务端、GPU 注册中心进行了详细的说明，然后详细地叙述了 GPU 调度算法的整个流程及其子算法。通过在客户端模拟生成固定任务数的任务，测试整个系统在一定时间内的任务完成数以及系统的平均周转时间，实验结果验证了调度算法的可行性与高效性。通过矩阵的复合运算和离散傅里叶变换在多 GPU 上的实现，不仅证明了本文多 GPU 协同计算的解决方案对于虚拟化平台的支持，同时全面地评估了本方案在单 GPU 和多 GPU 环境下的基本性能。

第四章　GPU 虚拟化环境下的数据通信策略研究

由于虚拟化本身的特点，在 GPU 虚拟化环境下进行 CUDA 应用开发在性能上会带来很大的开销。而目前在虚拟化环境下并没有一个通用且高效率的 RPC 系统，传统的如 XMLRPC[39]、CORBA[40]、ICE[41]等应用则只针对开放的网络或分布式环境，当把它们移植到虚拟化环境中时则会在系统吞吐率、延迟以及 CPU 占用方面出现很大的劣势，在实际项目中的应用基本无法使用。与此同时，当采用多 GPU 并行处理大规模的数据时，传统的 GPU 之间的数据交互需要通过 CPU 来中转，不仅会带来“路程”上的开销，同时 PCI-E 相对于 GPU 显存的低带宽更是限制了数据传输的速率。数据通信的选择问题，成为在虚拟化环境下进行 CUDA 应用开发不可越过的难题。

针对以上问题，本章在基于 Xen 和 VMware（Workstation、ESXi）虚拟化平台下下针对 CUDA 应用的延迟和吞吐率找出最优的虚拟机间同时方式。另外通过 NVIDIA 发布的 GPU Direct 2.0[42] 技术使一台服务器内多个 GPU 之间直接点对点通信，不仅绕开了主机端的参与，同时让多 GPU 编程更加轻松且传输效率也大大增强。本章通过采用虚拟机间不同的通信方式以及在 GPU 之间不同的数据传输方式，找出在虚拟化平台下的最优通信方案，同时从理论和实验数据中分析影响多 GPU 协同运算效率的因素。

4.1 研究基础

4.1.1 CUDA 存储器

CUDA 编程模型将 CPU 作为主机端，GPU 作为设备端（协处理器）。在这个模型中，CPU 负责进行逻辑性强的事务处理以及串行的数据计算，如在 kernel 启用前进行数据准备和设备初始化的工作；对应地，GPU 则负责并行化程度很高的并行计算。CPU 和 GPU 都拥有自己独立的存储空间，即主机端的主存和设备端的显存。而主机端内存也分为可分页内存和页锁定内存两种。可分页内存即为通过操作系统 API（malloc()和 new()）分配的存储器空间；页锁定内存不会被分配到虚拟内存中，能够保证存在于物理内存中，并且能通过 DMA 加速与设备端的通信。

而 GPU 中的存储器主要包括寄存器、共享存储器以及设备存储器。在一般情况下，自动变量都存储在寄存器中，寄存器独立地分布于每个标量处理器上，为每个线程提供私有的存储空间。寄存器的特点是容量小、访问速度快。在一般情况下，标量处理器上的寄存器容量大小只有几 KB，当需要对寄存器的空间需求超过寄存器的实际容量大小时，数据就会发生溢出，而此时设备存储器会对溢出的数据进行暂时的保存，对这部分数据的使用会有较长的时间延迟。

共享存储器（share memory）位于每一个多处理器内，在类型上是属于片上存储器的一种，因此共享存储器的访问速度非常高，一般和寄存器的速度差不多，一个时钟周期大约可以读写 2 个字节。共享存储器在一般情况下针对多个线程，同一多处理器（SM）上的线程可以对一片共享存储器进行访问，而一般的寄存器则只是针对单个线程。因此合理充分地使用共享存储器，对于数据之间的高速交换以及线程之间的通信有着很重要的现实意义。

在 GPU 中使用最多的存储器是设备存储器，按其类型可将设备存储器分为全局存储器（即本文之前所述的全局内存）、纹理存储器以及常量存储器。设备存储器有两种分配方式，它既可被分配为线性存储器，也可以被分配为只能读不能写但可以滤波、插值的 CUDA 数组。与共享存储器不一样的是，设备存储器不是安装在多处理器内的存储器，因此设备存储器的访问速度相对于共享存储器则慢了很多，对其直接进行访问需要消耗大量时间。在正常情况下，对设备存储器进行一次访问需要花费数百个时钟周期，这与共享存储器和寄存器相比较而言，差距非常明显。虽然设备存储器并不是安装在多处理器内，但其在多处理器中却分布有片上缓存，相当于 CPU 中 cache 的作用。往往在这种情况下，对缓存内的数据的访问的速度则和共享存储器以及寄存器的速度相当，然而当需要访问的数据在缓存中未被命中时则仍需对设备存储器进行直接访问，此时所需要的时间为两者之和。下图 4.1 所示为 GPU 的多层存储器空间。

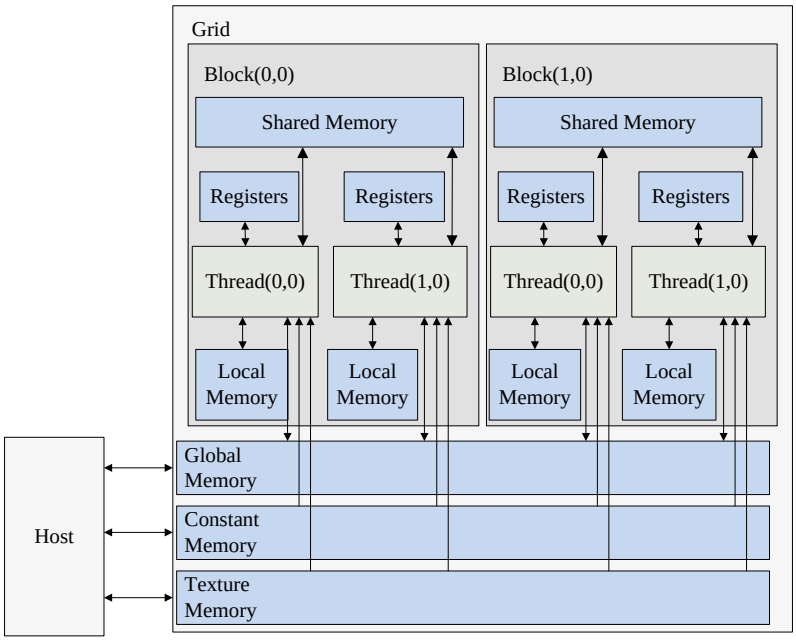

图 4.1 GPU 多层存储器空间

4.1.2 CUDA 计算模式

- 异步并发

基于 GPU 的通用计算是跨越 CPU 端和 GPU 端的并行计算，从步骤上可以将其分为 4 个部

分：CPU 端的初始化与计算、GPU 端的计算、CPU 到 GPU 的数据拷贝以及 GPU 到 CPU 的数据拷贝。而异步并发同样指的是主机端和设备端的并行计算，它在概念上与 GPGPU 有所不同，它是着眼于整个 CPU+GPU 异构计算平台，站在全局的角度来提高 CUDA 计算的效率。

异步并发对于基于 GPU 的通用计算有着重要的意义。首先，同一个流内的数据计算与拷贝是通过串行的方式完成的，但在两个流之间的数据计算与拷贝并没有直接的依赖关系，因此可以将其中一个流内的数据计算和另一个流的数据传输并行执行，而异步并发可以使 GPU 中的存储器控制单元和执行单元并行工作，从而达到提高 GPU 的利用率的目的；此外，当 GPU 在对 kernel 函数进行计算或者在进行数据拷贝的时候返回给主机端线程，使控制 GPU 的主机端线程不必等待 GPU 就可以进行一些其它的 CPU 端的计算或者逻辑处理，从而达到 GPU 和 CPU 并行工作的目的。如果调用了同步版本的 GPU 函数，在设备完成请求任务前，都不会返回主机线程，此时主机端线程将进入让步、阻滞或者自旋状态。

- 流

流（stream）是一种对 CUDA 并发进行管理的一种方式，它指的是一系列完整独立的任务序列，包含 CPU 端到 GPU 端的数据拷贝、kernel 函数的执行以及 GPU 端到 CPU 端的数据拷贝。将不同的流放在同一个时间轴上，不同的流的不同部分可能会相互重叠，从而增加并发性。由于流之间复杂的并发性，使用流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。cudaThreadSynchronize()同步设备上的所有线程，可以保证所有的流在完成了计算后才开始后续计算；cudaStreamSynchronize()只同步某一个流的线程，即等待某一个流中的所有线程都完成计算。

4.2 研究目的

在推出 CUDA Toolkit 3.2 RC 发布候选版进行测试后，NVIDIA 于 2010 年 9 月份发布了最终正式版本的 CUDA 3.2 工具包。该版本在性能上有相对之前的版本有明显的提升，如 CUBLAS 在 Fermi 架构下的矩阵乘法和置换性能提升 50%到 300%，CUFFT 在 Fermi 架构下基数 3、5、7 的转换性能相比 MKL 加速 2 到 10 倍。新增 CUSPARSE GPU 加速稀疏矩阵函数库，性能比 MKL 快 5 到 30 倍。新增 CURAND GPU 加速随机数生成函数库，比 MKL 快 10 到 20 倍。 加入 H.264 编解码库。同时它还扩展了函数库，改进了集群管理特性，包括对新硬件的支持等。

而随着 2011 年 CUDA 4.0 由 NVIDIA 作为一个全新版本发布后，其功能特性大幅度增加，主要涉及应用程序移植的简化、多 GPU 编程的加速、开发工具的增加和改进三个方面。在多 GPU 通信方面，在 CUDA 4.0 之前，多个 GPU 在同一节点内相互访问，需要首先将 GPU 中的数据传输给 CPU，再通过 CPU 的中转完成 GPU 间的数据交互。而在 CUDA 4.0 之后，NVDIA 在其官方 SDK 给出了 simpleP2P 的示例，示例中可以测试自己的显卡是否支持 P2P 以及显卡之

间的数据传输带宽。不同 GPU 可以直接进行传输，且传输可以一次完成。

为了进一步提高在虚拟化环境下多 GPU 协同计算的性能，本章首先在系统层分析由于虚拟化带来性能影响的主要因素，通过对比同一虚拟化平台下的不同数据通信策略得出特定虚拟化平台的最佳通信方式。在应用层方面，通过研究 CUDA 3.2 和 CUDA 4.0 的数据传输机制，采用两种不同的数据传输方式，得到影响多 GPU 协同运算效率的主要因素。

4.3 虚拟机域间通信策略

最好的虚拟机域间通信策略应该兼顾透明性并提供高性能的数据传输，但在实际设计中往往两者之间却不可兼得。目前主流的虚拟化平台上并不存在通用的高效通信方式，为了实现本文 GPU 通用计算虚拟化的可用性和高效性，本章通过对 Xen 和 VMware 系列在不同的虚拟化平台中对不同的特殊通信机制进行全方位的对比，通过对比得出在虚拟化环境下针对 CUDA 应用的最佳通信方式。

4.3.1 Xen

- Xen 体系结构

Xen 是由英国剑桥大学计算机实验室于 2002 年 4 月主持的开源虚拟化项目，在 x86、x86_64、PowerPC 和其他 CPU 架构上都能够提供强大、高效和安全的虚拟化特性。Xen 能够支持广泛的客户操作系统，包括 Windows 和 Linux 的多种发布版本。Xen 系统中的 VMM 采用混合模式，系统中存在一个经过授权的特权虚拟机协助 VMM 对其它虚拟机进行管理，并对其它虚拟机提供硬件设备的原生驱动，尤其其它虚拟机对 I/O 设备的访问。如下图 4.2 所示为 Xen 体系结构[43]。

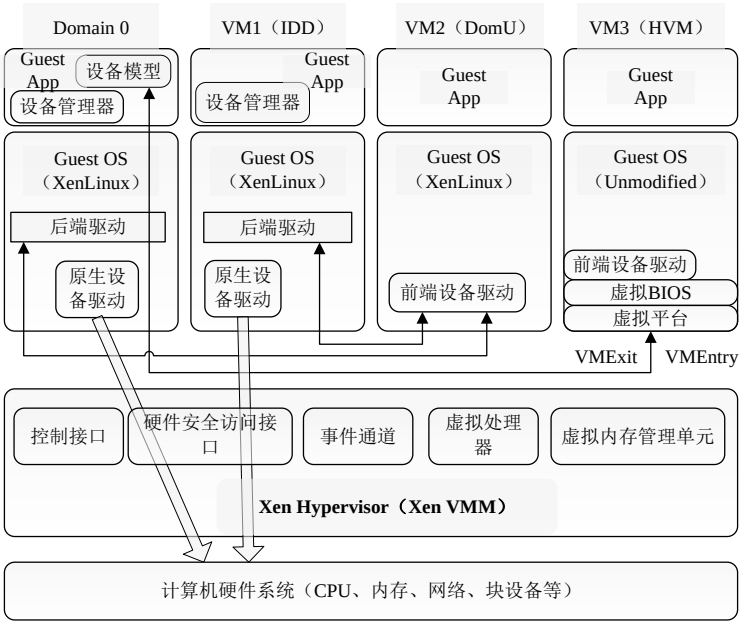

图 4.2 Xen 体系结构

在图 4.2 中，Domain 0 是特权虚拟域，运行在 Xen Hypervisor 之上，可以直接访问底层的物理资源，并可以与其它虚拟域直接进行交互，Domain 0 需要先于 Domain U 启动，其它虚拟域访问 I/O 设备需要经过 Domain 0 授权。在 Xen 系统中，存在一个独立设备驱动域（Isolated Driver Domain，IDD）也提供物理硬件的原生设备驱动，它经过 Domain 0 授权，但除此之外并不提供其它功能。Xen 3.0 支持运行未修改内核的 Guest OS，但这需要特殊硬件技术的支持，例如 AMD-V 或 Intel VT-x，运行这些 Guest OS 的虚拟域称为硬件虚拟域。除了 Domain 和 IDD 外的 Domain 称为非特权域（Unprivilege Domain，DomainU）。

- 基于 Xen 的虚拟机通信方式

本节分别介绍三种 Xen 平台下虚拟机域间通信优化的解决方案，分别从体系结构、工作原理的角度对 XenSocket[44]、XWAY[45]和 XenLoop[46]进行详细的介绍。

XenSocket[44]是在 Xen 中基于 socket 的解决方案，旨在提高虚拟机域间通信的吞吐率。XenSocket 的 API 采用标准的 socket API 接口，在 socket 的接口下，它使用 Xen 提供的共享内存来实现虚拟机之间高速的数据传输。在实现层面，XenSocket 基于 Xen 3.0.2 版本，在无需修改 Xen 和 Linux 内核的前提下编译成一个内核模块。

XenSocket 为通信双方分配了两块共享内存区域，其中一块由 4KB 的内存页组成，这些内存页用来存储通信双方状态和控制变量的信息；第二块是由多个 4KB 缓冲区内存页所形成的一个共享环形缓冲区，每个缓冲区的大小共 128KB。XenSocket 的体系结构如图 4.3 所示。在 XenSocket 中，虚拟机一端的发送方创建一个 socket 对象并通过 send( )函数将数据发送至 socket，这就类似于传统的 TCP/IP 通信协议或 Unix Domain Sockets 中 socket 的创建过程。另一虚拟机的接收方通过相同的方法创建 socket 对象，利用 recv()或者 read()函数从 socket 中取出发送方推送的数据。

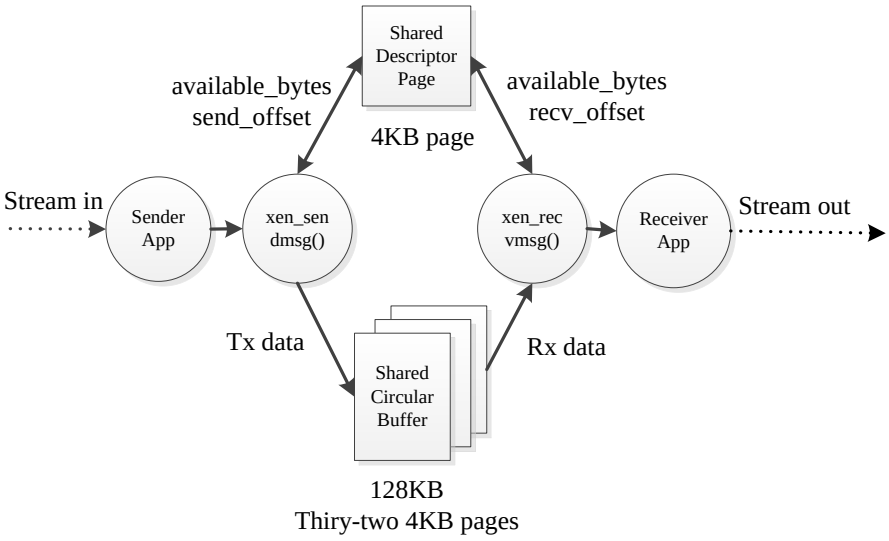

图 4.3 XenSocket 体系结构

对与由于网络通信问题而导致的性能下降三个主要因素，即虚拟机域间通信在对 TCP/IP 处理上所消耗的时间、域间通信之间过长的通信路径以及页面翻转所带来的开销，XWAY[45]机制绕开了基于 TCP/IP 通信标准的协议栈，避免了在页面翻转上的性能损耗，并在同同一物理机的不同虚拟机之间建立一个可以直接到达、达到加速效果的通信路径，以此来实现虚拟机之间的高速通信。对于那些实现标准 socket API 的应用程序，在二进制的兼容性方面有着很好的效果。

XWAY 是作为一个网络模块定制在系统的内核中。如图 4.4 所示为 XWAY 的体系结构，为了充分保证与标准 socket API 的二进制兼容性，XWAY 通过在设备的驱动层上面加入了两个额外的层次——XWAY protocol 层和 XWAY switch 层，对 TCP 和 INET 层之间与 socket 相关的请求。XWAY switch 层用来判断目标虚拟机是否处于同一物理机，如果目标虚拟机和源虚拟机处于同一物理机，XWAY switch 层就会创建一个 XWAY 通道，将其与 XWAY socket 通过 bind() 函数绑定在一起。如果目标虚拟机和源虚拟机处于不同的物理机中，XWAY switch 层则会把 INET 提交的请求传递给 TCP 层。XWAY protocol 层根据现有的 socket 选项，通过 XWAY Device driver 来传输实际的数据。例如，socket 选项中的 MSG-DONTWAIT 表明 XWAY protocol 层可以以以阻塞 I/O 模式或者非阻塞 I/O 模式运行，socket 选项可以来自 INET socket 结构、函数中的参数、TCP socket 结构等不同的地方。XWAY protocol 层对每个套接字选项进行解析，完成实际数据的发送和接收操作。

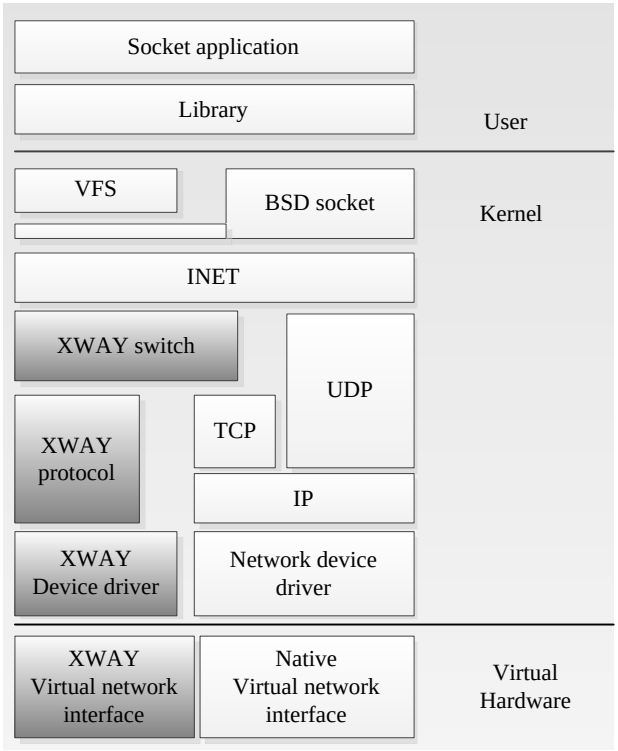

图 4.4 XWAY 体系结构

XWAY 使得在同一物理机不同虚拟机之间运行的程序可以通过标准的 socket API 来进行通信，不仅实现了域间通信的高带宽、低延迟，同时也充分地保证了对二进制兼容性，使任何基于 socket API 的网络程序都可以直接享用 XWAY 带来的高速域间通信的体验。

XenLoop[46]是一个 Xen 平台下另一虚拟机域间通信方案，该方案是一个完全透明的、高性能的虚拟机域间的网络环回通道。Xenloop 可使同一物理机不同虚拟机之间可以直接进行数据通信而不需要第三方软件（如 Domain 0 和）的参与，同时可以不牺牲应用层的透明性，应用程序不用作任何修改便可以无缝地运行。XenLoop 还实现了在 Xen 虚拟化平台下虚拟机域间的高速通信。如图 4.5 所示，XenLoop 在网络层以下对某些端口进行监听，对于收到的数据报首先检查其报头，查看目的虚拟机的地址是否与客户操作系统处于同一物理机，如果不是，则通过通用的 socket 进行通信，如果是，则通过共享内存将数据报传递给目的虚拟机。

XenLoop 支持虚拟化平台的高级特性，可以进行实时迁移，保证正在通信的任务不受影响，无缝地转移到另一个物理机上继续运行。XenLoop 中设计了一个虚拟机发现模块，模块中维护了一个当前物理机上的虚拟机列表，在虚拟机状态发生变化时，模块会相应的更新虚拟机列表。XenLoop 以 Linux 内核模块的方式存在，这样的设计使得它能够在虚拟机上动态加载。目前 XenLoop 已经作为一个 Xen 模块存在，可以在 Xen 开源网站上获取。

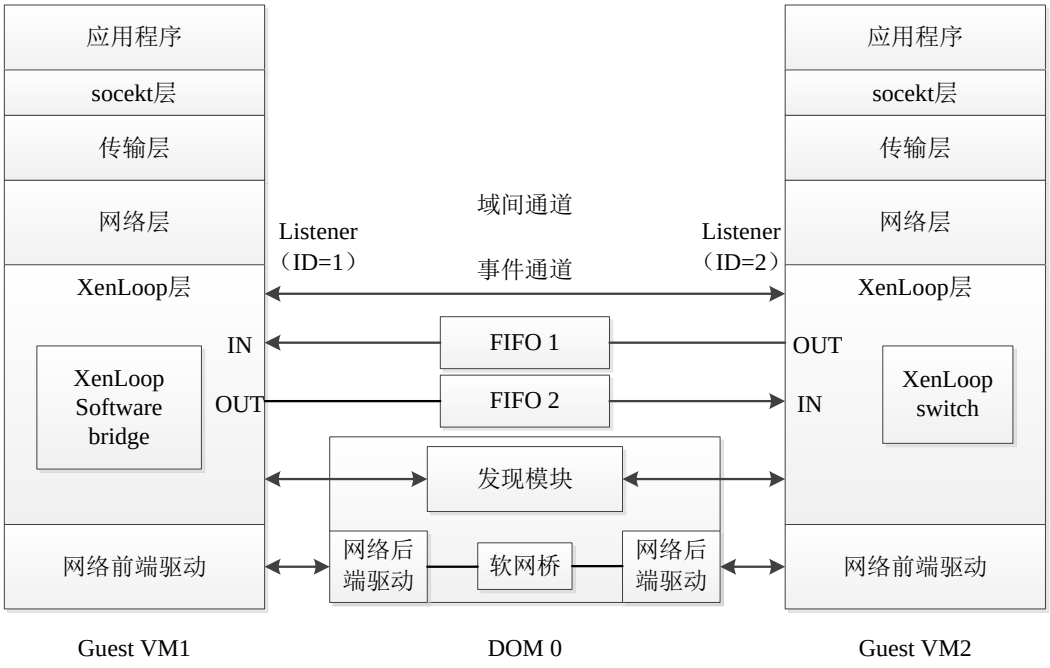

图 4.5 XenLoop 架构

4.3.2 VMware

ESXi 和 Workstation 是 VMware 公司的嵌入式虚拟化平台，它可以实现服务器部署整合，

适合各种要求严格的应用程序的需要，同时为企业未来成长所需扩展空间。对应 VMware 公司的各个虚拟化平台（以 ESXi 和 Workstation 为例），VMware 开发了一套相应的虚拟机通信接口 VMCI（Virtual Machine Communication Interface）。VMCI[47]是一个作用在应用层的方案，提供在 VMware 系列虚拟机平台下的快速、高效的域间通信通道，使得客户端虚拟机与客户端虚拟机、客户端虚拟机与物理机之间能够实现高速的通信。VMCI 实现两类接口，一类是数据报 API，一类是共享内存 API。VMCI 属于 VMware 的商业产品，内部实现细节目前并未披露。下图 4.6 为 VMCI 结构图，图中 ESXi 中虚拟机上的应用程序利用 VMCI 进行通信。

由于目前并不存在适用于虚拟机中的显卡驱动，本文通过在 ESXi 中某一虚拟机中使用 PCI pass-through（也即 VMware 的设备直通）技术，使得该虚拟机获得物理机中 GPU 的访问权，并在此虚拟机中部署 CUDA 服务端组件，使其充当 CUDA 服务端的角色，而在 Workstation 中则直接使用宿主物理机充当 CUDA 服务端的角色。

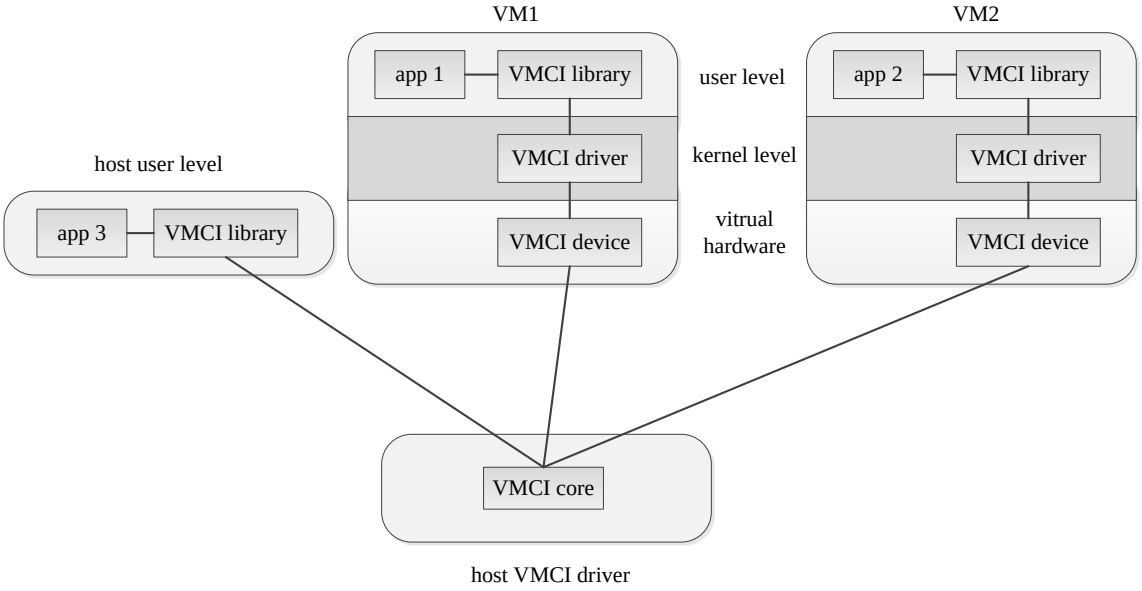

图 4.6 VMCI 体系结构

4.4 GPU 内部数据传输

在 GPU 间的数据传输方式主要分为通过主机端中转以及 GPU 间直接拷贝，通过主机端中转就是 GPU 先通过在 cudaMemcpy()函数里设置数据传输的方向将数据拷贝到主机端内存，再通过相同的函数不同方向将数据从主机端内存拷贝到设备端显存。而在 CUDA 新版本发布后，只要在 cudaMemcpy()函数里面的参数指针处于不同的地址存储器区域，在拷贝方向采用默认的传输方向。

4.4.1 统一虚拟地址空间

统一虚拟地址空间（Unified Virtual Addressing，UVA）模型为主机与系统内所有的设备提供了单一地址空间。这一模型将 CUDA 中包括 CPU 以及 GPU 在内的所有指令执行过程放入同一个地址空间。这之前每个 GPU 和 CPU 都使用它们自己的虚拟地址空间，从而导致许多额外的开销，UVA 的引入为 GPU 之间的点对点数据传输提供了逻辑支持，使得开发者体验到更方便的内存管理。如图 4.7 所示为多存储器模型和统一地址空间模型的结构图。

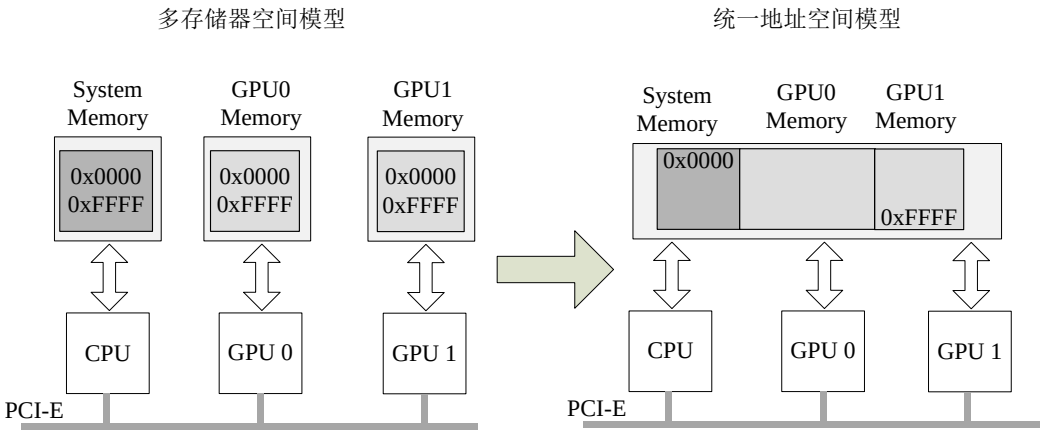

图 4.7 多存储器空间模型和统一地址空间模型

4.4.2 CPU 中转传输

在 CUDA 3.2 及其之前的版本中，多 GPU 的设备存储器和主机端的内存被视为独立的存储器块，各自拥有独立的地址空间。GPU 之间如果需要进行通信的话则必须首先通过 cudaMemcpy()函数将数据从 GPU 端拷贝到主机端内存，再由主机端内存传输至目标 GPU 中的显存。具体的数据传输过程如图 4.8 所示。

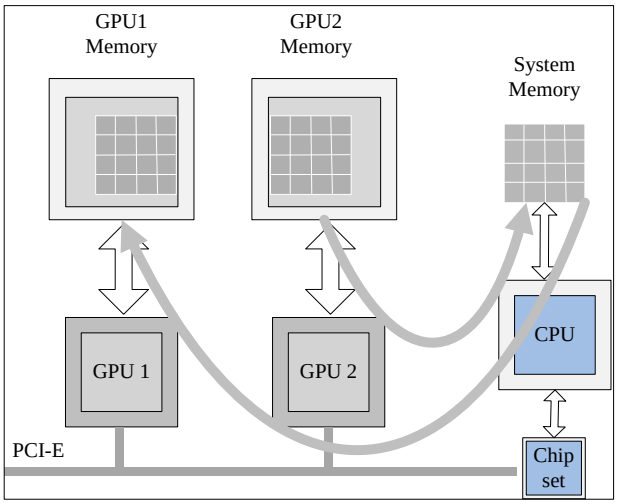

图 4.8 GPU 通过主机端拷贝

4.4.3 GPU 点对点传输

在 CUDA 4.0 之前的多 GPU 程序开发中，每个 GPU 被看成是独立的核心，通过 network 总线进行间接的通信，而在 CUDA 4.0 中，每颗核心以及其具备的各种存储资源被当做一个整体的工作网络中的结点，各个节点之间的通信和同步是对等的（形成了一个共享式内存的 SMP 网络）。host 端的内存资源和 GPU 上的设备存储资源被当做一块统一的存储器池，存储器地址统一编码，多 GPU 之间可以通过 PCI-E 总线直接进行通信，而不再需要通过内存进行中转。GPU 间直接的数据传输过程如图 4.9 所示。

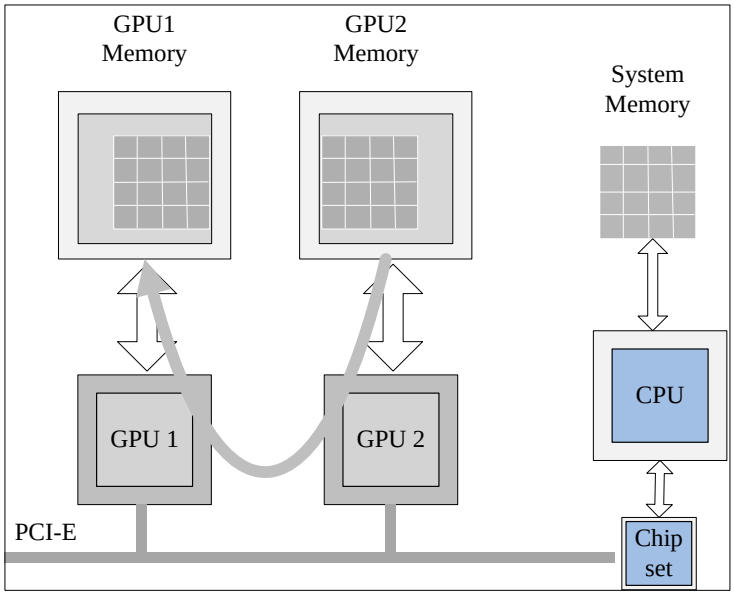

图 4.9 GPU 间点对点通信

4.4.4 结果预测

虽然 GPU 之间的直接数据传输直接绕开了主机端内存的中转，但由于基于 Fermi 架构的显卡没有对 GPU 之间的数据传输提供直接的硬件支持，在将数据传输至 GPU 时仍然需要将数据经由总线，其实质是少了一次经由 PCI-E 总线的数据传输的时间。因此预测随着计算规模的增大，G_G 相对与 G_H 的传输方式的计算时间差应该有所提升，提升的速度应该基本呈线性增长的趋势，其时间差应与数据的规模成正比，与总线带宽成反比。

以矩阵复合运算 A*B+C*D 为例，在一定的数据规模下，设 GPU 的矩阵计算时间为T_0，将四个矩阵拷贝至 4 个 GPU 的总时间为T_1，将计算结果拷贝回 CPU 端的时间为T_2，单 GPU 对中间结果的单向数据传输时间为ΔT，则 G_G 的理论计算时间为$T_0+T_1+T_2+3\Delta T$，G_H 的理论计算时间为$T_0+T_1+T_2+8\Delta T$，时间差为$5\Delta T$。

这是因为，G_H 采用的是 4 块 GPU 全部将数据传输至 CPU，再从 CPU 中转至其中的一块 GPU（如 GPU 0），总计算时间为$T_0+T_1+T_2+8\Delta T$。而 G_G 则不然，这是因为 GPU 0 在计算加

法运算之前需要进行(A/2)*B，而计算结果并不需要进行传输，直接驻留在 GPU 0 中，因此中间结果不仅绕开了主机端的参与，同时也省略了(A/2)*B 的两次数据传输。在本章中，我们设计的矩阵中的数据类型是单精度的浮点数 float，如果在 GPU 中分配和 C/2 和 D 同样大小的数据，采用 CUDA 4.0 SDK 中的 simpleP2P 样例来测试 GPU 之间的数据带宽（实际上就是 PCI-E 的实际带宽），则数据总量与 PCI-E 总线实际带宽的商 ΔT_1 应该与 ΔT 在理论上相等。

4.5 实验结果与性能分析

4.5.1 实验环境

实验所用平台的参数所表 4.1 所示，所用 CPU 为 Intel Xeon E7-4830，搭配 NVIDIA 的 Tesla C2050。

表 4.1 实验参数

实验环境	参数
操作系统	Ubuntu 12.04
虚拟化平台	Xen 4.0.1，ESXi 5.0
GPU 平台	NVIDIA Tesla C2050*4
CPU 平台	Intel Xeon E7-4830
GPU 编译器	nvcc
CPU 编译器	gcc 4.4.7，g++ 4.4.7
CUDA 版本	CUDA 3.2.16，CUDA 4.0
测试数据格式	输入单精度浮点数——输出单精度浮点数

4.5.2 域间通信方式选择

本节从通信策略支持的通信方向、对标准协议（TCP、UDP）的支持、延迟以及对用户的透明性等角度来分析在基于 Xen 的虚拟化环境下不同的通信策略对 CUDA 应用的性能影响。

尽管 XenSocket 和 XWAY 能够在基于 Xen 的虚拟化平台下提供高效率的域间通信，但是这些通信机制都存在某方面的不足。下表分别从客户端和服务的通信方向、对标准协议的支持、TCP 提交的延迟（单位为 TCP 报文的往返时间）以及对用户的透明性对三种通信方式进行比较，具体的对比总结如下表 4.2 所示：

从表 4.2 可见，XenSocket 的功能比较简单，只支持单向通信，而传统的 socket 通信和 RPC 都是双向的，这就使得其无法充分利用 RPC 机制的透明性。由于它的立足点在于增加虚拟机域间的传输带宽，对标准的 TCP 和 UDP 协议的支持也不完善。同时，XenSocket 虽然在数据量少（一般在 16KB 以下）时的单向吞吐量是 XenLoop 的两倍多，但由于其采用固定的缓冲区大小

使得其无法满足更大数据量的传输需求。

表 4.2 Xen 通信方式总结

通信方式	通信方向	标准协议支持	延迟	透明性
XenSocket	半双工	不支持	较长	否
XWAY	全双工	只支持 TCP	较长	是
XenLoop	全双工	TCP 和 UDP	很短	是

XWAY 通过拦截 socket 底层的 TCP 调用来为面向 TCP 连接的应用提供透明的虚拟机域间数据传输。但它同样存在诸多不足，如它需要对网络协议栈进行修改、不支持 UDP 协议、不支持套接字的在线迁移，同时，XWAY 也未对共享内存的安全性进行考虑。

XenLoop 方案对用户和系统都是完全透明的，提供了高性能的虚拟机域间网络回环通道。它支持 TCP 和 UDP 两种协议，同时利用 Xen 提供的授权表机制在数据通道的两端建立共享缓冲，实现全双工通信，并实现了对共享区的同步机制和安全机制。XenLoop 是目前基于 Xen 较为完善的虚拟机通信解决方案。本文将在 Xen 平台下使用 XenLoop 加速虚拟间的通信。

由于 VMware 的闭源性，在学术界很少有基于 VMware 虚拟化平台的通信解决方案。但 VMware 公司开发了一套相应的虚拟机通信接口 VMCI，它是一个工作在应用层的解决方案，提供在 VMware 系列虚拟机平台（Workstation、ESXi 等）下的快速、高效的域间通信通道，使得客户虚拟机与客户虚拟机、客户虚拟机与宿主虚拟机之间能够高效的通信。本文在 VMware 虚拟化系列平台下均采用 VMCI 进行域间加速。

4.5.3 GPU 内部数据传输

由于受计算机计算能力和内存容量方面的限制，有很多并行程序需要将数据整体拆成几个部分，它们之间需要通过数据交换来完成原来程序的功能。在本章中我们通过矩阵复合运算 A*B+C*D 来完成整个实验数据的验证。

A*B+C*D，其中 A*B 分别由 device 0 和 device 1 完成，C*D 分别由 device 2 和 device 3 来完成，乘法完成后通过两种方式分别将计算结果传递给 device 0，由于矩阵加法的复杂度较低，所以在最后所有的加法运算全部交由一块 GPU 来完成。本节给出了 4 个 GPU 在两种不同数据传输方式下的性能比较。下表 4.3 所示为 G_H 和 G_G 随着矩阵规模逐渐增大后的统计特征对比，统计特征包括 kernel 函数的迭代次数、实验方差。

在表 4.3 中，kernel 执行次数即在主函数中执行 kernel 的次数，也即迭代次数。G_H 表示通过主机端中转来完成 GPU 间的数据拷贝，G_G 表示绕开主机端直接通过 GPU 间点对点完成数据拷贝。方差则表示若干次迭代次数的时间的统计方差，单位为秒的平方（s^2）。

表 4.3 两种不同数据传输方式的统计特征

阶数	Kernel 执行次数(G_H)	Kernel 执行次数(G_G)	方差 (G_H)	方差 (G_G)
1000	100	100	0.000020	0.000012
2000	100	100	0.000037	0.000049
3000	50	50	0.000053	0.000066
4000	50	50	0.000194	0.000099
5000	25	25	0.000849	0.000622
6000	25	25	0.001547	0.001124
7000	10	10	0.008421	0.006429
8000	5	5	0.011462	0.009456

图 4.10 所示为通过两种数据传输方式的时间对比，横轴是矩阵的阶数，纵轴是执行的时间。为了使最后统计的数据统一，图中的执行时间均为将迭代次数换算至 100 次后的时间，同时，矩阵复合运算的执行时间是输入数据初始化完毕后开始计时，数据从 GPU 端拷回主机端结束计时，因而未将由于虚拟化带来的性能开销（即访问延迟）考虑在内。

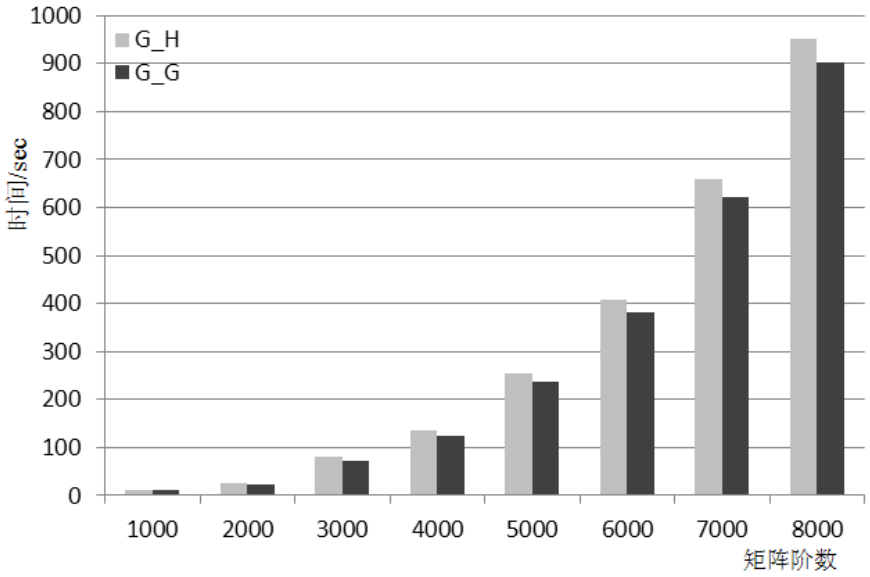

图 4.10 不同数据传输方式的时间对比

从图 4.10 中可以看出，随着矩阵规模的增加，G_H 相对于 G_G 的执行时间差也越来越大，变化速度基本呈线性增长的态势。由 4.4.4 节的结果预测可知，这是由于 G_G 相对于 G_H 少了 $5\Delta T$，其中 ΔT 为单 GPU 对中间结果的单向数据传输时间。G_G 相对于 G_H 不仅少了中间结果在总线上一次数据传输，同时 GPU 0 在计算加法运算之前进行(A/2)*B 的计算结果并不需要进行传输，直接驻留在 GPU 0 中，因此中间结果不仅绕开了主机端的参与，同时也省略了(A/2)*B

的两次数据传输。

下表 4.4 所示为实际测量的时间差与理论计算值对比情况。其中，理论值为单个 GPU 传输(C/2)*D 的结果乘以系数 5 后的结果；测量值为在主机端生成同理论值相同数据量的数据将其传输至 GPU 中所花费的时间乘以系数 5 后的结果；实际值即如图 4.10 中的 G_H 和 G_G 的时间差；实际理论差即为理论计算值与 G_H 和 G_G 时间差的差值；同理，测量实际差为“等价”的测量值与实际的时间差的差值。

由表 4.4 不难看出，理论值、测量值与实际值基本相符，同时随着矩阵的阶数不断增加，实际理论差有着小幅度的增加，但都在允许的测量误差范围之内。测量实际差基本上为 0，这是由于测量的数据和实际传输的数据在格式和规模上完全一致，只是简化了数据传输的过程。图 4.10 和表 4.4 验证了 4.4.4 节结果预测的正确性。

表 4.4 理论值、测量值、实际值对比（单位：秒）

阶数	理论值	测量值	实际值	实际理论差	测量实际差
1000	0.72	0.74	0.74	0.02	0
2000	2.93	2.95	2.96	0.03	0.01
3000	6.67	6.71	6.71	0.04	0
4000	11.80	11.83	11.84	0.04	0.01
5000	18.57	18.60	18.62	0.05	0.02
6000	26.77	26.81	26.82	0.05	0.01
7000	36.45	36.51	36.51	0.06	0
8000	47.62	47.66	47.68	0.06	0.02

4.6本章小结

本章首先引入了所要研究问题的由来，然后对相关的研究基础给出了概要的介绍。接着对本章的研究目的给出了介绍。对现有的虚拟机域间通信的优化方案分别从体系结构以及底层实现方面进行详细介绍，通过对几种优化方案进行详细的对比，在 GPU 虚拟化环境下采用最优的通信方式。在单节点多 GPU 内部通信方面，设计了两种方式（G_G 和 G_H）完成 GPU 间的数据传输，以矩阵复合运算为应用场景设计了实验，结果验证了预测的正确性。

第五章　总结与展望

以 CUDA 为代表的通用并行计算框架自诞生以来，GPU 通用计算在学术界和工业界得到了广泛的应用，如气象预报、石油勘探数据处理、金融工程数据分析、计算结构力学、生命科学计算、计算机视觉、数值分析等领域。虚拟化技术的引入为在基于 CPU+GPU 异构的高性能计算集群中使用 GPU 成为可能，但目前工业界的 GPU 虚拟化解决方案大多只针对图形渲染，对通用计算领域基本没有涉及，学术界对 GPU 虚拟化的研究也刚处于起步阶段。为了在 GPU 虚拟化环境下更充分地利用 GPU 资源，本文选取虚拟化环境下的 GPU 通用计算作为研究课题，对现有的 GPU 调度算法进行改进，提出在虚拟化环境下基于 OpenMP 的多 GPU 协同计算方法，为大规模计算程序在虚拟化环境下使用多 GPU 并行计算提供理论依据。为了降低由于虚拟化本身带来的性能流失，对现有的虚拟机域间通信方式进行总结，采用适合 GPU 虚拟化环境下的最优通信方式，最后通过实验验证了本文提出的方案的正确性和有效性。

5.1 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，在透明性、性能以及对 CUDA 最新特性的支持方面，gVirtuS 是目前较为完整的 GPU 虚拟化解决方案。本文的研究在 gVirtuS 基础上展开，在 Xen 和 VMware 两种虚拟化平台下进行应用。

本文从实现 GPU 负载均衡、降低 CUDA 任务的周转时间出发，借鉴 gVirtuS 的 GPU 虚拟化解决方案，以现有的 GPU 调度算法为基础，设计了基于反馈调节的 GPU 调度算法。该算法不仅基于 GPU 的特征，同时将任务的规模及其计算复杂度纳入考量，实现了在 GPU 虚拟化环境下更细粒度的 GPU 调度算法。为了降低大规模计算程序的执行时间，本文在虚拟化环境下设计并实现了基于 OpenMP 的多 GPU 协同计算方法，该方法通过在主机端开辟与 GPU 个数相同的线程来控制每个 GPU，达到多 GPU 同时计算的目的。在提高 CUDA 应用的效率方面，本文对现有的虚拟机域间通信技术进行总结，采用适合 GPU 虚拟化环境下的最优通信方式，有效地降低了由于虚拟化本身所带来的性能影响。

下面对本文的研究工作进行总结：

（1）对基于 GPU 的通用计算、GPU 虚拟化以及现有的并行计算框架进行了详细的阐述，对各自的优缺点进行总结，概括了目前 GPU 虚拟化技术的发展方向，对现有的图形渲染和通用计算的 GPU 虚拟化解决方案进行了总结。

（2）以现有的 GPU 虚拟化解决方案和 GPU 调度算法为基础，设计了基于 GPU 特征的调度算法，该算法不仅将 GPU 特征作为判断 GPU 负载的依据，同时将 CUDA 任务的任务类型和

计算复杂度纳入考量，使负载评价更加客观。

（3）对于首次提交的任务类型，通过判断所有 GPU 是否均为空闲，设计了反馈调节算法对影响因子进行调节。通过加上固定的步长穷举出所有影响因子配置所得出的任务周转时间，得出 GPU 计算能力和全局内存的最佳影响因子配置。

（4）针对大规模计算程序，本文在虚拟化环境下设计并实现了基于 OpenMP 的多 GPU 协同计算方法，通过主机端的多线程控制 GPU 完成协同计算的目的。采用科学计算中常用的矩阵相乘和相加的复合运算以及运算量较大的一维离散傅里叶变换作为研究对象，验证了方案的有效性和高效性。

（5）为了最大限度地降低由虚拟化本身给 CUDA 任务带来的性能开销，对现有的虚拟机域间通信的优化方式进行总结，采用适合 GPU 虚拟化环境下的最优通信方式。

（6）为了进一步提高多 GPU 协同计算的计算效率，设计了两种不同的 GPU 间数据传输的方式，采用矩阵复合运算对两种不同的方式进行实验验证，找出了影响多 GPU 协同计算效率的主要因素。

5.2 进一步的工作展望

本文对虚拟化环境下 GPU 通用计算的关键技术进行的研究，对现有的 GPU 调度算法进行了改进，针对大规模的计算程序设计并实现了在虚拟化环境下的多 GPU 协同计算方法。找出了在特定虚拟化平台下的最优通信方式以及 GPU 内部数据传输的最优方法。但目前学术界对虚拟化环境下的 GPU 通用计算的研究工作仍处在早起阶段，在可扩展性和性能的提升方面仍有很多可以深入的研究课题。

下面是对未来的工作进行进一步的展望：

（1）对虚拟化平台下的 GPU 调度算法进行进一步的改进。目前的调度算法只针对单 GPU 上的多任务，并没有根据任务的规模情况对任务进行动态的划分，在通用性方面，可以通过进一步完善 GPU 注册中心的功能，使其根据多 GPU 相对于单 GPU 的加速点对任务进行动态评估，实现更细粒度的多 GPU 多任务调度算法。

（2）对虚拟化平台下的多 GPU 协同计算提供统一的接口。目前的多 GPU 协同计算方法只针对特定的 CUDA 应用，对用户来说并不透明。因此接下来可以通过提供的统一接口来将单 GPU 中的程序透明地向多 GPU 移植。

（3）对现有的虚拟机域间通信方式的性能进行进一步的提升。目前的虚拟机域间通信针对的是整个虚拟化平台，并不存在专门针对虚拟化环境下 CUDA 应用进行加速的通用方式，因此针对 CUDA 自身的特点设计适合于 CUDA 应用的通信解决方案也可成为接下来的研究课题。

参考文献

[1] CUDA. http://www.nvidia.cn/object/cuda-cn.html.2013-12-10.

[2] 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.

[3] 王庆波, 金滢, 何乐, 等. 虚拟化与云计算. 北京：电子工业出版社, 2009.

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

[5] Dowty M, Sugerman J. GPU virtualization on VMware's hosted I/O architecture. SIGOPS Operation Systems Review, 2009, 43(3)：73-82.

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

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

[8] Duato J, Pena A, Silla F, et al. rCUDA：Reducing the number of GPU-based accelerators in high performance clusters. In：Proc of International Conference on High Performance Computing and Simulation. Caen, 2010, 224-231.

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

[10] 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.

[11] Takizawa H, Sato K, Komatsu K, et al. CheCUDA：A Checkpoint/Restart Tool for CUDA Applications. In：Proc of International Conference on Parallel and Distributed Computing Applications and Technologies. Higashi Hiroshima, 2009, 408-413.

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

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

[14] Folding@Home. http://folding.stanford.edu/. 2013-12-10.

[15] DirectX 10. http://vga.zol.com.cn/192/1929484.html.2013-12-11.

[16] 仇德元. GPGPU 编程技术—从 GLSL、CUDA 到 OpenCL. 北京:机械工业出版社, 2011.

[17] Brook. http://sourceforge.net/projects/brook/. 2013-12-16.

[18] OpenCL. http://www.khronos.org/opencl/.2013-12-16.

[19] Lahabar S, Agrawal P, Narayanan P J. High Performance Pattern Recognition on GPU. In:Proc of National Conference on Computer Vision Pattern Recognition Image Processing and Graphics. Gandhinagar, 2008, 154-159.

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

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

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

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

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

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

[26] VMware SVGA Device Developer Kit. VMware-svga.sourceforge.net.2013-12-15.

[27] Collange S, Defour D, Parello D.Barra, a Modular Functional Gpu Simulator for GPGPU. Technical Report hal-00359342, 2009.

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

[29] Intel VT-d. http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices.2013-12-16.

[30] Xen VGA Passthrough. wiki. Xensource.com/Xenwiki/XenVGAPassthrough.2013-12-16.

[31] VMware VMDirectPath I/O. communities. VMware.com/docs/DOC-11089.2013-12-06.

[32] VirtualGL. http://www.virtualgl.org/.2013-12-16.

[33] Humphreys G, Houston M, Ng R, et al. Chromium:a Streamprocessing Framework for Interactive Rendering on Clusters. In:Proc of 29th Annual Conference on Computer Graphics and Interactive Techniques. New York, 2002, 693-702.

[34] 何家俊, 廖鸿裕, 陈文智. Kernel 虚拟机的 3D 图形加速方法. 计算机工程, 2010, 36(16):251-253.

[35] MPI. http://zh.wikipedia.org/wiki/MPI.2013-12-17.

[36] 马业. 面向云计算的GPU通用计算虚拟化技术研究, [硕士学位论文]. 南京:南京航空航天大学, 2012.

[37] 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.

[38] DFT. http://blog.csdn.net/seucbh/article/details/8001320.2013-12-18.

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

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

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

[42] Direct 2.0.http://www.valtra.com/news/6568.asp.2013-12-16.

[43] 石磊, 邹德清, 金海. Xen 虚拟化技术. 武汉:华中科技大学出版社, 2009:33-46.

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

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

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

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

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

致谢

时光荏苒，岁月如梭，转眼间两年半的研究生生涯即将进入尾声。从2011年9月进入南京航空航天大学大学开始，到2013年12月坐在南航网格与云计算研究所实验室的今天，回首这段研究生学业生涯，有太多值得回味的点点滴滴。我所做出的每一决定，取得的每一点进步都离不开老师们、同学们以及亲友们真诚的帮助，在此，请接受我最诚挚的感谢！

首先我要感谢我在研究生阶段的导师袁家斌教授！袁老师对本文的研究工作倾入了大量的心血，从论文的开题、课题的研究、实验的设计一直到最后论文的撰写都离不开袁老师悉心指导。在硕士阶段的两年半以来来，袁老师严谨的科研态度、渊博的专业知识、宽阔的视野对我影响深远。袁老师在学术对我们严格要求，在生活上对我们关怀备至，两年半的研究生生涯能够在袁老师的指导下度过非常幸运，同时也非常的有收获。导师高尚的师德、渊博的知识、强烈的责任心以及严谨的治学态度令我终生难忘。

感谢王箭老师不辞辛苦地在每个周六组织论文讨论会，在讨论会中我不仅得到了王箭教授很多宝贵的建议，同时还从和同学们的讨论中找到了论文创作的灵感，您严谨的科研态度时刻指引着我前进的方向。感谢学校网络中心的老师们，张蓝蓝老师、王兴虎老师等。张蓝蓝老师在生活上对我们无微不至的关怀使我得以将自己的全部精力集中在科研中；王兴虎老师严谨的学术态度、高超的专业水平时刻的鞭策我不断地在学业中前进。感谢吕相文博士和马业师兄，吕相文博士对本文的撰写提供了宝贵的建议，马业师兄在本文的前期研究工作中解决了许多技术难点。感谢实验室的全体同学，尤其是张玉洁、张珮、王雪、赵兴方，是大家创造的良好的学术氛围，使得我的论文能够顺利完成。

其次我要感谢我的父母，感谢你们对我在我求学的道路上一直给予的支持。失败的时候你们及时鼓励，成功时你们同我分享喜悦，是你们让我在疲倦的时候有个心灵停靠的港湾，你们的理解和支持就是我不断进步的动力。

最后，感谢各位评审专家和参与答辩的各位老师对本文的评阅。在新春即将来临之际，祝大家新春快乐，身体健康，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. 张云洲，袁家斌，吕相文. 面向多任务的 GPU 通用计算虚拟化技术研究. 计算机工程与科学，2013.11
2. 曾青华，袁家斌，张云洲. 基于 Hadoop 的贝叶斯过滤 MapReduce 模型. 计算机工程，2012.12.

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”