面向大数据高通量计算的 CPU/GPU 并行优化技术研究

扶聪

2018 年 3 月

中图分类号 TQ028.1

UDC分类号： 540

面向大数据高通量计算的 CPU/GPU 并行优化技术研究

作 者 姓 名 扶聪

学 院 名 称 计算机学院

指 导 教 师 翟岩龙

答辩委员会主席

申 请 学 位 工学硕士

学 科 专 业 计算机科学与技术

学 位 授 予 单 位 北京理工大学

论 文 答 辩 日 期 2018 年 3 月

Synthesis and Application on textile of the Shape Memory Polyurethane

Candidate Name:	Fu Cong
School or Department:	****
Faculty Mentor:	Prof. YanLong Zhai
Chair, Thesis Committee:	Prof. **
Degree Applied:	****
Major:	****
Degree by:	Beijing Institute of Technology
The Data of Defence:	*, ****

面向大数据高通量计算的CPU/GPU并行优化技术研究

北京理工大学

研究成果声明

本人郑重声明：所提交的学位论文是我本人在指导教师的指导下进行的研究工作获得的研究成果。尽我所知，文中除特别标注和致谢的地方外，学位论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得北京理工大学或其它教育机构的学位或证书所使用过的材料。与我一同工作的合作者对此研究工作所做的任何贡献均已在学位论文中作了明确的说明并表示了谢意。

特此申明。

作者签名：_____　　签字日期：_____

关于学位论文使用权的说明

本人完全了解北京理工大学有关保管、使用学位论文的规定，其中包括：① 学校有权保管、并向有关部门送交学位论文的原件与复印件；② 学校可以采用影印、缩印或其它复制手段复制并保存学位论文；③ 学校可允许学位论文被查阅或借阅；④ 学校可以学术交流为目的, 复制赠送和交换学位论文；⑤ 学校可以公布学位论文的全部或部分内容（保密学位论文在解密后遵守此规定）。

作者签名：_____　　导师签名：_____

签字日期：_____　　签字日期：_____

摘要

本文……。（摘要是一篇具有独立性和完整性的短文，应概括而扼要地反映出本论文的主要内容。包括研究目的、研究方法、研究结果和结论等，特别要突出研究结果和结论。中文摘要力求语言精炼准确，硕士学位论文摘要建议 500~800 字，博士学位论文建议 1000~1200 字。摘要中不可出现参考文献、图、表、化学结构式、非公知公用的符号和术语。英文摘要与中文摘要的内容应一致。）

关键词： 形状记忆；聚氨酯；织物；合成；应用（一般选 3 ～ 8 个单词或专业术语，且中英文关键词必须对应。）

Abstract

In order to exploit ⋯⋯.

Key Words: shape memory properties; polyurethane; textile; synthesis; application

目录

摘要 I

Abstract II

第 1 章 绪论 1

1.1 研究背景 1

1.1.1 大数据高通量仿真计算的研究背景和意义 1

1.1.2 CPU/GPU 异构计算研究背景和意义 2

1.2 CPU/GPU 异构编程简介 3

1.2.1 CUDA 平台简介 3

1.2.2 OpenCL 简介 4

1.2.3 OpenAcc 简介 5

1.3 GPU 结构简介 6

1.3.1 CPU 和 GPU 设计区别 6

1.3.2 GPU 内存结构 7

1.3.3 CPU 与 GPU 内存交互 8

1.4 论文主要工作 9

1.5 论文主要结构 9

第 2 章 相关技术的研究使用 10

2.1 GPU 编程优化方法 10

2.1.1 向量化 10

2.1.2 优化 GPU 内存使用 11

2.1.3 嵌套循环展开 12

2.1.4 线程配置 13

2.2 一般代码优化 15

2.2.1 变量优化 15

2.2.2 循环优化 16

2.3 控制流分析和循环查找 16
2.3.1 基本块和控制流图 17
2.3.2 控制流分析 18
2.4 数据流分析 19
2.4.1 程序中的点和通路 19
2.4.2 到达-定值数据流方程 20
2.4.3 引用-定值链 20
2.4.4 定值-引用链 20
第 3 章 基于优化 CPU 和 GPU 内存交互的设计和实现 21
3.1 迁移数据交换操作 21
3.1.1 迁移限制 22
3.1.2 代码移动 22
3.1.3 异步操作优化 24
结论 26
参考文献 27
附录 A *** 28
附录 B Maxwell Equations 29
攻读学位期间发表论文与研究成果清单 30
致谢 31
作者简介 32

第 1 章　绪论

本章主要分为五部分，分别介绍了大数据高通量仿真和 CPU/GPU 并行计算的研究背景和意义，CPU/GPU 异构编程平台和 GPU 物理特性，最后介绍了本篇论文的主要研究工作和创新点。

1.1　研究背景

互联网技术的飞速发展颠覆了人们的传统生活方式，几乎每一个用户个体每天都会产生数据，如何利用好这些海量数据成为各行各业急需解决的问题。大数据应用正融入生活的方方面面，大数据计算用于建模与仿真学科也是必然的发展趋势。传统的单 CPU 处理器结构计算能力有限，不适用于大数据仿真中，使用 CPU/GPU 异构计算才能快速处理大数据仿真。大数据仿真中 CPU/GPU 异构计算的优化是本文研究的主要内容。

1.1.1　大数据高通量仿真计算的研究背景和意义

建模与仿真技术是以相似理论、模型理论、系统技术、信息技术，以及建模与仿真应用领域的有关专业技术为基础，以计算机系统、与应用相关的物理效应设备及仿真器为工具，根据对系统仿真的目标建立并利用模型对系统进行研究、分析、试验、运行和评估的一门综合性、交叉性技术。建模与仿真技术和计算机科学一起，正成为继理论研究和实验研究之后的第三种认识、改造客观世界的重要手段。大数据技术的出现给仿真技术带来了新的需求与新的挑战，将大数据技术与仿真技术进行结合逐渐成为新的发展趋势。美国国防部高级计划研究局于 2012 年发起 1 亿美元的 XDATA 计划，用于支持国防信息数据处理和分析的软件和技术。军用仿真领域早在 90 年代就已经面临了大数据问题，美英 1997 年联合举行的 STOW97 战争综合演练仿真，由 500 台计算机构成，超过 3700 个仿真实体参与，在三个独立 48 小时阶段仿真中共计产生了大约 1.5TB 的数据。此后越来越多的军用仿真应用不断出现，高解析模型甚至实装模型不断运用到仿真中，仿真产生的数据也在快速增长。但是由于计算能力、计算方法等各种原因，仿真采集的绝大部分数据被丢弃了，仿真数据的使用多处于仿真结果检查或者简单统计分析的级别，没有起到军事分析、决策和预测的作用。国内外

仿真专家近年不断探讨大数据与仿真的结合方式，试图寻求新的建模与仿真方法。有学者提出大数据时代基于仿真的工程科学还需要发展仿真范式, 实现密集计算与密集数据的集成，以实现无组织复杂系统因果规律的发现。数据密集方法可由数据从整体上分阶段发现涌现性、演化机制下的结果, 而计算密集方法在部分时段或部分区域上满足了相似性研究的需要，为实现整体上的可预测性，即通过模型运行来揭示相应复杂性系统的运行规律，必须将数据密集与计算密集集成起来。这就是大数据高通量仿真要解决的核心问题之一。数据量较少的仿真实验很多情况下是不足以描述事物发展趋势的，海量的实验数据可以更好的覆盖到实体各方面运行和结果参数，计算出其中的因果关系。传统的单 CPU 处理器计算能力有限，不具备快速计算大数据仿真的能力。高性能异构集群协同计算方法是解决数据密集且计算密集型大数据高通量仿真的首选方法。随着 GPU 和 Intel 至强 Phi 等协处理器的不断发展，异构系统成为高性能计算的重要方式。将 GPU 与大数据处理框架集成来解决数据密集且计算密集型大数据计算是近年的研究热点。

1.1.2 CPU/GPU 异构计算研究背景和意义

GPU 最初是被设计用于显卡中图形计算的，帮助 CPU 快速处理复杂的图像计算。由于其卓越的并行计算能力，现在 GPU 不再只作用于显卡中，越来越多地作为协处理器帮助 CPU 处理高并行的大数据计算。在一些专业的领域中，比如机器学习和生物研究中，CPU/GPU 异构计算的应用十分广泛。从 CPU 和 GPU 性质上看，CPU 适合处理计算量比较小的逻辑运算，而 GPU 由于其内部有很多个小的计算单元非常适合处理高度并行的复杂计算。单 CPU 处理器结构的计算能力不足以满足海量数据的计算需求，因此业界普遍把 GPU 作为协处理器帮助 CPU 处理高并行的大数据计算。特别是在大数据高通量的仿真程序中，大量的实验参数和仿真计算必须依赖 CPU/GPU 异构计算才能快速得出实验结果。虽然 GPU 编程语言，如 CUDA、OpenAcc 和 C 语言语法规则差别不是很大，语法也比较简单，但是 GPU 编程对于普通程序员来说仍是一个巨大的挑战。设计者只有清楚地了解 GPU 内部结构和 CPU/GPU 通信特点才能编写出高效地 GPU 程序。首先，设计者需要人为判断哪些计算是数据量大，并行程度高，需要放到 GPU 中计算的。其次 CPU 和 GPU 的内存是隔离的，在相应的处理器上计算时必须先把需要的数据拷贝到相应的内存中，这需要程序员手动的设计 CPU 和 GPU 之间的数据交换过程。每一次 GPU kernel 的调用都涉及到数据在 CPU 和 GPU

内存的交换，所以在有多个 kernel 的 GPU 程序中，数据在 CPU 内存和 GPU 内存之间的反复拷贝将成为限制程序运行速度的一大障碍。与此同时，我们也发现编译制导方式的 CPU/GPU 异构编程虽然能够很大程度的降低 GPU 编程难度，但是由于 CPU 与 GPU 体系结构之间的差异和大数据高通量计算的数据密集特点，要实现 CPU/GPU 高性能协同计算，难度仍然不小。正如在 OpenAcc 官方网站中介绍的，由于使用了不同的编译制导语句控制 CPU 与 GPU 之间的数据通信，导致同一个算法性能相差 30 倍。这主要是因为 GPU 内存带宽能够达到 100GB/s~250GB/s，平均为 CPU 内存带宽的 8 倍，但是访问 GPU 显存却需要 400~800 个周期，是访问内存十几倍，PCI-E 总线的实际带宽也只能达到 8GB/s 左右，CPU/GPU 之间的 I/O 瓶颈问题已经是公认的阻碍 CPU/GPU 协同计算性能的关键问题，这一问题对大数据高通量仿真的影响将更加明显。因此本论文拟研究面向大数据高通量仿真的 CPU/GPU 异构计算数据通信模型与优化方法，以提高集群的大数据高通量计算能力。

1.2 CPU/GPU 异构编程简介

大数据时代需要更加快速、更高并行的计算方式，使得 GPU 编程越来越普遍。业界普遍使用的 GPU 编程语言有 CUDA、OpenCL 和 OpenAcc，本节将对 GPU 编程语言做出简单介绍。另外，GPU 编程语言仅仅提供了操作 GPU 的计算语法，如何根据 GPU 物理结构去设计 GPU 计算来充分优化 GPU 计算将在后续章节介绍。

1.2.1 CUDA 平台简介

CUDA(Compute Unified Device Architecture)，是显卡厂商 NVIDIA 推出的运算平台, 一种通用并行计算架构，支持 Linux 和 Windows 平台。该架构仅适用于 NVIDIA 显卡，已应用于 GeForce、ION、Quadro 以及 Tesla GPU 上, 使 GPU 能够快速解决复杂的计算问题。它包含了 CUDA 指令集架构以及 GPU 内部的并行计算引擎, 开发人员现在可以使用 C 语言来为 CUDA 架构编写程序, 所编写出的程序可以在支持 CUDA 的处理器上以超高性能运行。CUDA3.0 已经开始支持 C++ 和 FORTRAN。从 CUDA 体系结构的组成来说，包含了三个部分：开发库、运行期环境和驱动。开发库是基于 CUDA 技术所提供的应用开发库。目前 CUDA 的 1.1 版提供了两个标准的数学运算库——CUFFT （离散快速傅立叶变换）和 CUBLAS（离散基本线性计算）的实现。这两个数学运算库所解决的是典型的大规模的并行计算问题，也是在密集数据计算中非

常常见的计算类型。开发人员在开发库的基础上可以快速、方便的建立起自己的计算应用。此外，开发人员也可以在 CUDA 的技术基础上实现出更多的开发库。运行期环境提供了应用开发接口和运行期组件，包括基本数据类型的定义和各类计算、类型转换、内存管理、设备访问和执行调度等函数。基于 CUDA 开发的程序代码在实际执行中分为两种，一种是运行在 CPU 上的宿主代码（Host Code），一种是运行在 GPU 上的设备代码（Device Code）。不同类型的代码由于其运行的物理位置不同，能够访问到的资源不同，因此对应的运行期组件也分为公共组件、宿主组件和设备组件三个部分，基本上囊括了所有在 GPGPU 开发中所需要的功能和能够使用到的资源接口，开发人员可以通过运行期环境的编程接口实现各种类型的计算。由于目前存在着多种 GPU 版本的 NVidia 显卡，不同版本的 GPU 之间都有不同的差异，因此驱动部分基本上可以理解为是 CUDA-enable 的 GPU 的设备抽象层，提供硬件设备的抽象访问接口。CUDA 提供运行期环境也是通过这一层来实现各种功能的。目前基于 CUDA 开发的应用必须有 NVIDIA CUDA-enable 的硬件支持，NVIDIA 公司 GPU 运算事业部表示：一个充满生命力的技术平台应该是开放的，CUDA 未来也会向这个方向发展。由于 CUDA 的体系结构中有硬件抽象层的存在，因此今后也有可能发展成为一个通用的 GPGPU 标准接口，兼容不同厂商的 GPU 产品。CUDA 相比于其他 GPU 编程语言更加底层，需要程序员手动设计每次 GPU kernel 调用时的数据交换过程，这样可以使设计者根据需求充分利用 GPU 性能，同时也是对程序员的一大挑战，不合理的 GPU 编程设计将会带来巨大的性能丢失。

1.2.2 OpenCL 简介

OpenCL 是第一个面向异构系统通用目的并行编程的开放式、免费标准，也是一个统一的编程环境，便于软件开发人员为高性能计算服务器、桌面计算系统、手持设备编写高效轻便的代码，而且广泛适用于多核心 CPU、GPU、Cell 类型架构以及数字信号处理器等其他并行处理器，在游戏、娱乐、科研、医疗等各种领域被广泛使用。OpenCL 由一门编写 kernels 的语言和一组用于定义并控制平台的 API 组成，提供了基于任务分割和数据分割的并行计算机制，支持 Windows 和 Linux 操作系统。OpenCL 1.1 标准支持三维矢量和图像格式等新数据类型，支持处理多 Host 指令以及跨设备的缓冲区，并且通过链接 OpenCL 和 OpenGL 的事件，高效共享图像和缓冲区，大大改进了与 OpenGL 的互操作性。OpenCL 对图像格式的处理更加快速高效，也是图形学

中常用的 CPU/GPU 异构编程语言之一。OpenCL 2.0 在已有标准上首次引入共享虚拟内存的支持，CPU 和 GPU 内核可以直接共享复杂的，包含指针的数据结构，避免了程序员不熟悉异构编程造成的冗余数据交换，大大提高了并行编程的灵活性。支持动态并行，GPU 内核不需要与主句 CPU 交互情况下进行内核排队，实现灵活的调度方式，减少数据在 CPU 内存和 GPU 内存之间的拷贝，降低 CPU 处理器的负担。该标准改进图像支持，内核可以同时读写同一对象。新增对安卓系统的支持，安卓客户端可以安装驱动扩展将 OpenCL 作为共享对象进行操作。OpenCL 框架由平台 API，运行时 API 和 OpenCL 编程语言组成。平台 API 用于宿主机程序发现 OpenCL 设备的相关函数和为 OpenCL 应用创建上下文的相关函数。运行时 API 是用来管理上下文创建命令队列以及运行时的一些其他操作。OpenCL 语言是用来编写 GPU 内核计算代码的编程语言，与 c 语言相似。目前主流的芯片厂商都已推出支持 OpenCL 的芯片，各大品牌的手机平板几乎全都能支持 OpenCL。移动端的 GPU 通用计算必然成为被广泛使用的技术，推动移动计算进入一个全新时代。

1.2.3 OpenAcc 简介

OpenAcc 是另一种面向 CPU/GPU 异构结构并行编程 GPU 编程语言。2011 年 11 月，Cray、PGI、CAPS 和英伟达 4 家公司联合推出 OpenACC 1.0 编程标准，适用于 NVIDIA 和 AMD 显卡，支持 Windowshe 和 Linux 系统。OpenAcc 与 CUDA 和 OpenCL 相比，不需要程序员手动设计每一块数据在 CPU 和 GPU 内存之间的拷贝细节，而是以一个或者多个在 GPU 上执行的 kernel 为单位，以编译制导语句的方式告诉编译器在当前 kernel 的数据拷贝方式。OpenAcc 大大降低了 GPU 编程的门槛，很多数据的拷贝都是由编译器帮助程序员去完成的。程序设计者一般只需规划好哪些代码是在 GPU 上执行和相应的编译制导语句就可以写出效率较高的 OpenAcc 程序。PGI 编译器对 OpenAcc 的支持非常完善，GCC 编译器也支持大部分 OpenAcc 特性。使用 CUDA 重构 C 语言代码时，需要重写程序，OpenAcc 并行化的方式不是重写程序，而是在串行 C/C++ 或 Fortran 代码上添加一些编译制导标记。支持 OpenACC 的编译器能够看懂这些标记，并根据标记含义将代码编译成并行程序。对英伟达 GPU 来说，编译器将 C/C++/Fortran 代码翻译成 CUDA C/C++ 代码，然后编译链接成并行程序。对 AMD GPU 来说，中间代码是 OpenCL。可见 OpenAcc 语言相比 CUDA 和 OpenCL 更加高级一点，不需要程序设计者具体编写底层的数据拷贝语句，只需要程序员利用编

译制导语句告诉编译器每个 kernel 的执行方式和数据拷贝。程序耗时最多的是循环模块，OpenAcc 把每个并行计算量很大的循环结构，看成是一个可以在 GPU 上执行的 kernel。循环结构中的迭代计算被分散到 GPU 上不同线程执行，GPU 拥有多个核心可以同时快速运行多个线程，大大降低了 CPU 负担。经上述章节介绍可以看出，无论选取何种 GPU 编程语言进行代码编写，都需要设计者规划那些计算需要在 GPU 上执行和每次 GPU kernel 调用 CPU 内存和 GPU 内存之间的数据交换。本文的主要研究方向也是如何优化 CPU 内存 GPU 内存之间的数据交换来优化 GPU 程序。

1.3 GPU 结构简介

GPU 作为协处理器，帮助 CPU 处理图形计算和一些其他的高并行的复杂计算，降低 CPU 负担，提高整个系统运行速度。NVIDIA 和 AMD 是现在最大的两个显卡供应商，本节就 NVIDIA 显卡的物理结构做出简单介绍，并且分析一些在 CPU/GPU 异构计算中的一些瓶颈。

1.3.1 CPU 和 GPU 设计区别

CPU 和 GPU 的设计初衷打不相同，分别针对了两种不同的应用场景。CPU 作为中央处理器需要很强的通用性来处理各种不同数据类型，同时又要有处理判断循环等逻辑的控制结构，这使得 CPU 的内部结构异常复杂。而 GPU 的设计初衷就是解决高度统一的、相互无依赖的大规模数据和不需要被打断的计算环境。图 1.1是来自 NVIDIA 文档关于 CPU 和 GPU 结构的描述图，其中绿色是计算单元（ALU），橙色是控制单元，蓝色是存储单元。从图中可以看出 GPU 内部有大量的计算单元和超长的流水线，只有非常简单的控制逻辑并且 cache，可见 GPU 的设计初衷就是用来处理大规模数据计算的。相比 GPU，CPU 内部计算单元相对较少，被缓存占据了大量空间，拥有复杂的控制逻辑单元和诸多优化电路。所以 CPU 的结构非常适合作为中央处理器来处理很多复杂的控制逻辑，同时大块的缓存也加速了对硬盘的读写，但是由于计算单元有限不擅长计算大规模的并行计算。同时，GPU 拥有很多的核心，可以快速并行运行多个线程，处理循环时，在不破坏数据依赖的情况下可以把每一轮的迭代计算分散到不同线程上。GPU 拥有很少的 cache，GPU 缓存的目的是和 CPU 一样缓存后续可能使用到的数据，而是缓存线程需要的数据。当有多个线程需要一个数据块时，GPU cache 会合并这些请求，从 DRAM 中缓存这些数据到 cache，然后再把 GPU

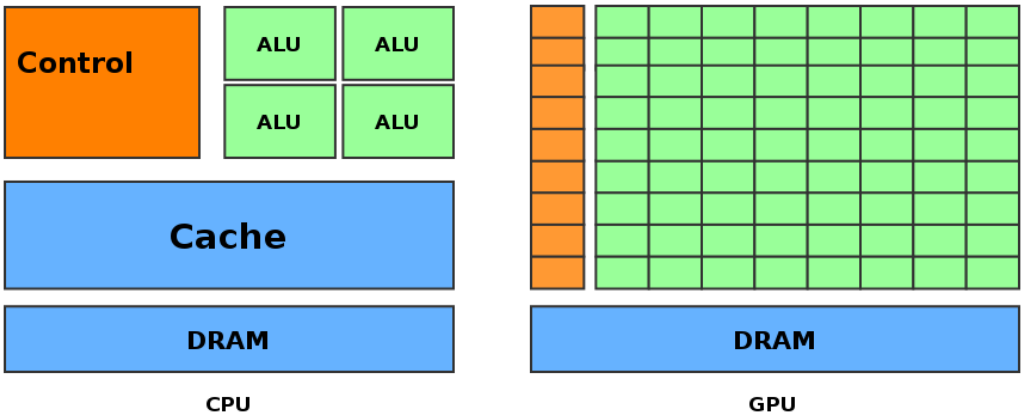

图 1.1　CPU 和 GPU 设计区别

缓存中的数据发送到相应线程。从上述描述可以看出，CPU 作为中央处理器处理通用计算，负责和操作系统、硬盘、网络硬件等交互，拥有复杂的逻辑控制单元可以快速处理每条指令。而 GPU 作为协处理器处理，是用来帮助 CPU 处理专用大规模数据的并行计算，不需要与操作系统和其他硬件交互。

1.3.2　GPU 内存结构

GPU 内存中没有 Cache 缓存，由寄存器、局部存储器、常量内存，共享内存、纹理内存和 DRAM 全局内存等组成。图 1.2描述的是 CPU 和 GPU 内存整体结构。寄存器是 GPU 片内高速缓存器，线程私有，执行单元访问寄存器的延迟非常低。GPU 寄存器单元是由大小为 32bit 的寄存器文件组成，每个线程拥有的寄存器数量有限。我们不应该为单个线程定义过多的私有变量使程序运行时的寄存器数量不足。局部存储器是在 GPU 显存中，也是线程私有。当寄存器数量不足，数据较大或者数据在编译时不能确定大小时，数据就会被缓存到局部存储器中。一个线程的私有变量和中间变量都是存到寄存器和局部内存上，局部存储器不再 GPU 片内，访问延迟较高。共享内存在 GPU 片内，是线程块内所有线程共享存储器，几乎可以达到和寄存器一样低的访问延迟，是线程块内线程通信延迟最低的通信方式。GPU 常量内存是保存核函数执行期间不会变化的数据，NVIDIA 显卡为常量内存分配了 64KB 的空间。常量内存和标准 DRAM 全局内存的设计方式不同，与共享内存相似，在某些情况下常量内存代替全局内存可以有效减少内存带宽。CUDA 架构中有线程束的概念，把 32 个线程看成一个步调一致的集合去执行同一块循环代码，即线程束中的线程以不同的数据执

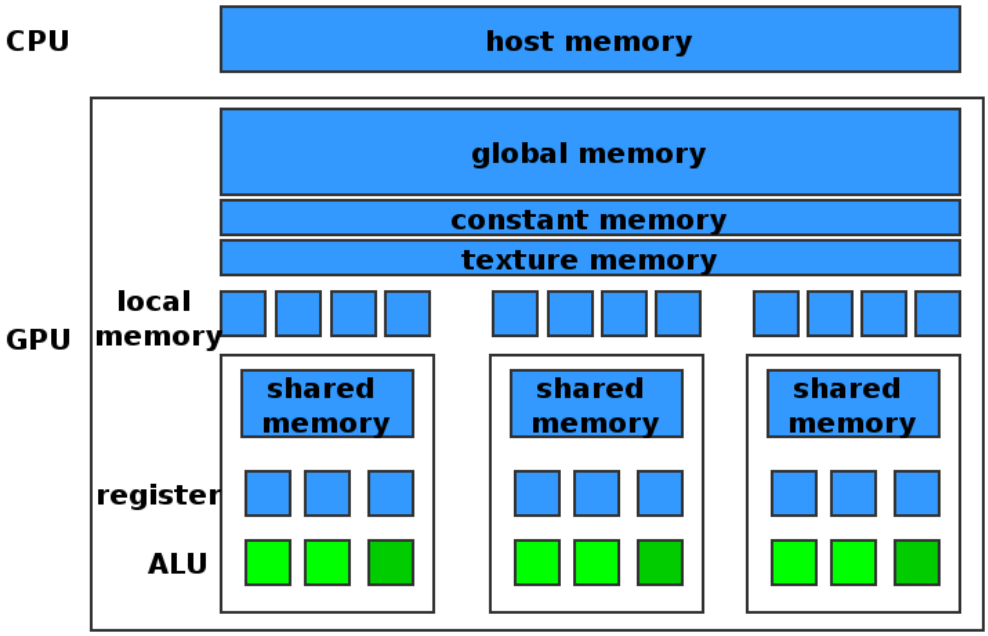

图 1.2　GPU 内存结构

行相同的指令。访问常量内存时，GPU 把改常量数据广播到半线程束中的其他线程，减少全局内存带宽。常量内存的数据不会再次改变，硬件会主动缓存常量内存，减少访问常量内存次数。共享内存是可以被同一线程块中的所有线程访问的可读存储器，该内存在物理 GPU 上，而不是驻留在 GPU 之外的系统内存。GPU 上每新启动一个线程块，CUDA 都会为线程块创建一个块内所有线程共享的变量副本，线程块内所有线程都可以读写该共享变量，但是不可以读写其他线程块的共享内存。使用共享内存作为一个线程块内的全局内存，加速了内存访问速率。纹理内存是另一种只读内存，缓存在芯片上，专门用来计算内存访问模式具有大量空间局部性的图形应用程序。在图形计算时，像素的读取具有很大的空间局部性，使用纹理内存将会缓存本次访问内存的局部内存，加速下次访问。全局内存是所有线程都可以读写的内存，提供很高的带宽，访问延迟较高。

1.3.3　CPU 与 GPU 内存交互

在 CPU/GPU 异构计算中，CPU 内存和 GPU 内存是相互隔离的，执行命令时必须先把计算需要的数据拷贝到相应的处理器上。CPU 和 GPU 内存之间的数据交换是以 PCIe 总线为通道，实际带宽为 8GB/s, 远远低于 GPU 内存带宽。每一次 GPU 核函数的调用都会涉及到数据在 PCIe 总线上传输，频繁的数据交换降低了 GPU 程序的性

能。本文的研究方向正是从 CPU/GPU 内存隔离出发，优化 CPU 和 GPU 内存之间的数据交换。

1.4 论文主要工作

本文在学习已有的 GPU 程序优化策略和 CPU/GPU 隔离内存结构基础上，通过编译原理分析程序中控制逻辑和数据依赖性，拆分和迁移每个核函数的数据交换操作，允许数据交换在程序代码序列中一段区间中执行。迁移之后，可以合并一些核函数的数据交换操作，减少整个 GPU 程序的数据交换大小和数据交换次数，减少 PCIe 总线传输的数据总量以及降低频繁地数据交换带来大量预处理和清理工作的系统消耗。

1.5 论文主要结构

本文共分为四个章节。第一章是绪论，首先介绍了大数据高通量仿真的研究背景和意义以及大数据仿真应用中 CPU/GPU 异构计算的研究意义，并分析了 CPU/GPU 异构计算的体系结构特点，GPU 内存特性，以及 CPU 和 GPU 内存之间的数据交换方式，总结了 GPU 编程一些常见的瓶颈和解决方案。第二章是相关技术的研究使用。介绍了目前已经使用的 GPU 编程优化技术和使用场景，以及编译原理中一些优化程序的常用技巧。第三章是本文的主要工作，首先依据程序中的数据依赖性拆分和迁移每个核函数的数据拷贝操作，再根据具体情景合并一些数据拷贝操作，合并操作可以减少数据拷贝总量和数据拷贝次数来达到优化 GPU 程序的目的。第四章是实验部分，对比了不同数量核函数和不同大小数据拷贝量的 GPU 程序在 CPU 和 GPU 内存通过 PCIe 总线交换数据的时间，验证了本文提出的优化方法确实可以达到优化 GPU 程序的目的。最后对本文的工作做了总结，并对优化 CPU/GPU 异构计算下一步的研究工作提出了展望。

第 2 章　相关技术的研究使用

本章主要介绍了一些已有的 CPU/GPU 异构计算优化方法和编译原理中一些调整代码序列的方法，通过对已有的优化方法进行总结比较，提出本文一种基于代码移动来迁移、合并核函数的数据交换操作的优化方法。

2.1　GPU 编程优化方法

GPU 的物理结构更加适合大数据、高并发的复杂计算，任何优化 GPU 编程的方法都是基于 GPU 物理特性充分发挥 GPU 计算性能来实现的。本节将介绍、对比一些业界主流的 GPU 编程优化方法以及每种方法的使用场景。

2.1.1　向量化

程序员可以在编写代码之前手动分析 GPU 应用程序中数据特点，再根据 GPU 结构特性去确定如何组织、存储、访问数据以及确定程序逻辑可以达到优化程序的目的。向量计算是图像算法中经常使用的计算方法，我们可以使用数组很好地表示一维向量进行计算。但是在向量乘法、加法等计算操作中，向量中每一个维度的操作基本都是相同而且相互不干扰的，具有很高的并行性。核函数中使用数组表示一维向量时，在向量计算中仅仅使用了单个线程循环迭代处理向量中每个维度的计算，没有利用到向量的计算并行性和 GPU 的多核特点，效率较低。主流 GPU 编程语言 CUDA 和 OpenCL 都提供了数据向量化方法，用专有的数据结构去表示向量。如图 2.1所示，向量化后的计算，每一个维度的的操作都会分配到不同核心上的线程执行，不再是整个

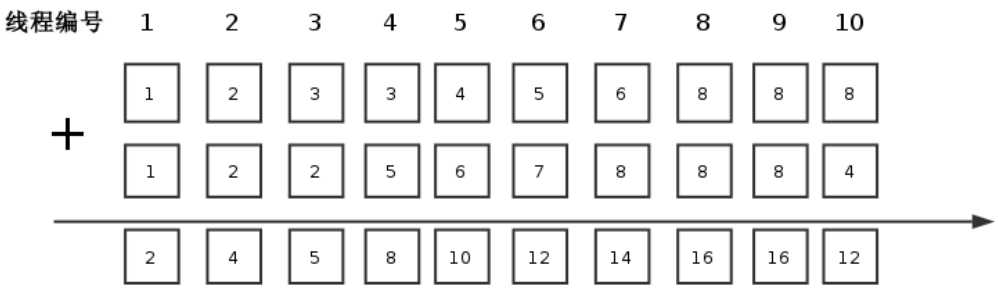

图 2.1　向量化

向量维度的计算都在一个线程上。向量化的数据结构在 GPU 中计算时可以充分利用 GPU 多核特性，每一个维度的计算都放在了不同核心上的线程，可以加速向量计算。

2.1.2 优化 GPU 内存使用

第一章简单描述了 GPU 内存结构，较为复杂，由常量内存、纹理内存、共享内存和全局 DRAM 等组成。不同种类的存储器特点和适用场景差别很大，表 2.1 简单列出了不同 GPU 储存器的一些特点，GPU 程序员必须要清楚地了解 GPU 的内存特点再根据使用场景具体选择。编写代码时，如何设计数据在 GPU 内存中的存储方式也是一种很重要的优化 GPU 程序的方法。寄存器是 GPU 计算单元访问延迟最低的存储器，GPU 程序运行时会自动将一些中间数据存储到寄存器上加速下次读取。但是 GPU 会将一些线程私有变量存到线程的寄存器上，程序员定义了过多的线程私有变量会导致程序运行时一些 GPU 线程内寄存器不足，换而使用显卡中的局部内存，增加了访问延迟。我们可以把 GPU 运行时不发生变化的数据存储到 GPU 常量内存中，单个线程对常量内存中数据的访问会使 GPU 将该数据广播到同一线程束中的其他线程，由此可以减少整个程序对常量内存的读取次数。共享内存是 GPU 片上线程块内所有线程可以读写的共享存储空间，是线程块内线程数据交互延迟最低的通信方式。当程序中一些数据是线程块内所有线程共享的，我们可以使用共享内存代替全局内存，降低线程之间相互通信的延迟。纹理内存也是 GPU 中只读内存，被设计用来存储空间局部性比较高的数据。由于 GPU 结构中没有 cache 来缓存时间局部性的数据，如何利用好数据的空间局部性就是一个很重要的优化方法。特别是在图形计算中，每个像素点的在空间上的联系非常紧密，空间复杂度很高，使用纹理内存的读取特性可以很容易读取空间上联系紧密的数据，加速内存访问速度。综合本小节的描述可以看出，GPU 中不同内存的性质差别很大，根据具体场景选择合适的存储器将会很大程度上优化 GPU 程序。

表 2.1 GPU 内存特性

存储器	位置	访问权限	变量生存周期
寄存器	GPU 片上	GPU 可读/写	与线程相同
局部存储器	板载显卡	GPU 可读/写	与线程相同
共享内存	GPU 片上	GPU 可读/写	与线程块相同
常量内存	板载显卡	CPU/GPU 可读/写	与程序相同
纹理内存	板载显卡	CPU/GPU 可读/写	与程序相同
全局内存	板载显卡	CPU/GPU 可读/写	与程序相同

2.1.3 嵌套循环展开

代码 2.1 嵌套循环

```
1  void addmatrix(vector<vector<int>>& matrix)
2      {
3          int m = matrix.size();
4          int n = matrix[0].size();
5          for(int i=0; i<m; i++)
6              for(int j=0; j<n; j++)
7                  matrix[i][j] += 1;
8          return;
9      }
```

代码 2.2 嵌套循环展开

```
1  void addmatrix(vector<vector<int>>& matrix)
2      {
3          int m = matrix.size();
4          int n = matrix[0].size();
5          for(int i=0; i<m*n; i++)
6              {
7                  int j = i/n;
8                  int k = i%n;
9                  matrix[j][k] += 1;
10             }
11         return;
12     }
```

GPU 之所以能够快速处理大数据并行计算是因为其拥有很多个计算核心，可以同时运行多个线程。GPU 程序中的核函数都是一些计算量很大的循环结构，循环结构中每一次迭代都是独立的，这样 GPU 就可以启动很多个线程去运行循环结构中的每一次迭代计算。由此可见，循环结构的迭代次数越多，加速效果越明显。但是 GPU 程序中循环不都是简单的单层循环，更常见的是逻辑比较复杂的多层嵌套循环。在核函数里的多层嵌套循环结构中，只有最里层的循环结构是同时在不同线程上计算的。这样的计算方式使得嵌套循环结构没有完全利用 GPU 的多核优势，闲置了很多计算资源。嵌套循环展开是指预处理 GPU 程序时把一些核函数中的二层或者多层嵌套循环展开、合并成单个循环。如代码块 2.1、 2.2所示两段代码都是实现了一个矩阵所有元

素加 1 的简单操作。显然两段代码的时间复杂都是相同的, 嵌套展开后的代码段还增加了计算举证行、列索引的操作。如果在普通 CPU 上执行的话，显然代码块 2.1 中的代码段执行的会更快，因为 CPU 是没有足够的计算单元来开启多个线程并行计算该循环的。但是 GPU 的多核特性正是针对处理并行程度高的计算的，代码块 2.2把代码块 2.1中的嵌套循环合并成一个迭代比较大的单个循环，这样允许 GPU 开启更多的线程去计算该核函数，得到更好的优化效果。这里只拿矩阵自加作为例子介绍嵌套循环展开，当计算比较复杂，每一次迭代的计算量都比较大时，嵌套循环展开对 GPU 核函数的优化将是十分明显的。

2.1.4 线程配置

本小节以 CUDA 平台为例简单描述 GPU 编程中的线程配置。CUDA 的线程设计以网格 (Grid)、线程块 (Block) 和线程 (Thread) 组成，具体结构如图 2.2所示。GPU 上所有的计算单元被划分为两到三个网格，每个网格包含一定数量的线程块，目前最多为 65535 个线程块，每个线程块再由一定数量的线程组成，目前主流显卡线程块最多可以包含 1024 个线程。如何配置好如此多的线程对 GPU 程序员将是一个很大的挑战，也是优化 GPU 程序一个很重要的维度。同一线程块内的线程具有相同的指令地址，不但能够并发执行，而且可以通过 GPU 片上共享内存和栅栏实现块内快速通信。这样，程序员在设计 CUDA 程序时需要优先把相互通信量较大、较为频繁或者需要同步的线程分配到同一线程块内，利用共享内存降低线程通信延迟，同时把不需要通信的线程分配到不同线程块内，使其并行粒度更高。CUDA 架构通过块间粗粒度并行，块内细粒度并行两种方式来组织线程。多个计算单元加上一些寄存器和共享存储器组成一个多元处理器。每个线程块最终都是交给一个多元处理器执行的。程序运行时，GPU 任务分配单元将网格分配到 GPU 芯片上。CUDA 平台启动时，需要将网格配置信息从 CPU 下发到 GPU，任务分配单元依据配置信息将线程块块分配到多元处理器上。任务分配单元采用轮询策略：轮询遍历多元处理器有没有足够的资源来执行新的线程块，如果有则给多元处理器分配一个新的线程块。决定是否成功分配的条件有：块运行时需要的寄存器数量，共享存储器数量，以及其它的一些限制条件。任务分配单元在任务分配中保持公平，然而我们可以通过手动编程设置块内线程数量，线程使用的寄存器数和共享存储器数来间接控制，保证处理单元负载均衡。任务以这种方法分配使程序具备了很大的可扩展性：由于每个子任务都能在任意一个多元处理单元上

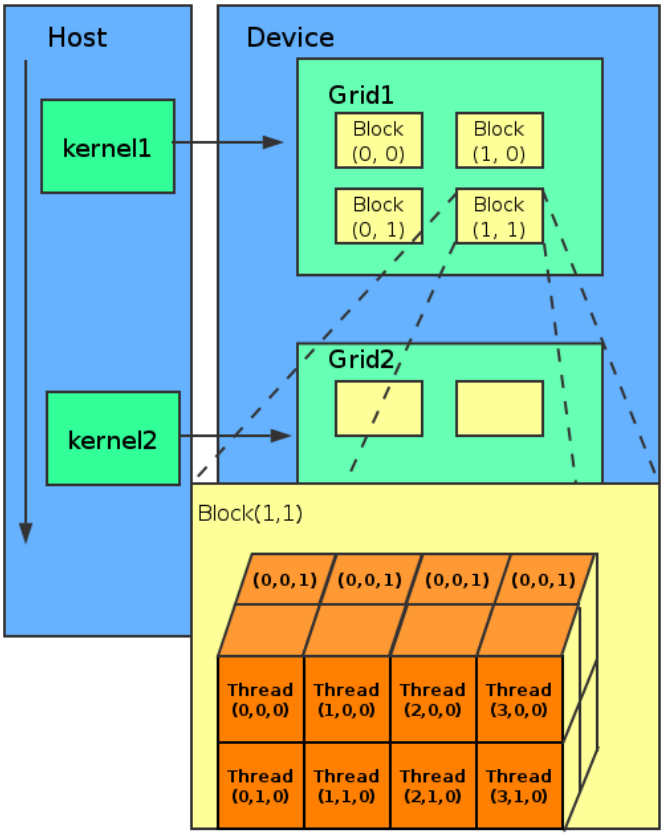

图 2.2　GPU 线程结构

执行，CUDA 程序在核心数量不同的处理器上都能正常运行，隐藏了硬件差异。程序员需要根据具体应用场景将任务拆分为互不相干、可以并行计算的粗粒度子任务，再将每个子任务拆分为能够尽可能利用线程块内计算资源、存储资源的细粒度指令集。现有的显卡同一时刻只能执行一个核函数，但是在 Fermi 架构中可以在同一时刻运行多个核函数。例如程序员可以在一个内核访问 GPU 数据的同时，开启另一个内核进行计算，可以很大程度上提高程序运行效率。线程是串行执行指令的，这就要求每个线程不可以有过多的循环逻辑或者需要过多的存储资源。当然我们也不应该在单个线程块内分配过多的线程，可能由于参数和中间变量过多导致存储资源不足，降低多元处理器计算速度。在问题相对复杂时，程序员也可以手动测试一些配置参数的运行速度，选择一个最合适的线程配置参数。

2.2 一般代码优化

程序员编写的代码不总是十全十美，编译器可以自动在不改变程序运行效果的前提下帮助我们进行一些代码优化，使之运行时的空间效率和时间效率得到提高。编译原理中的代码优化是指通过重排、删除、合并或改变程序片段等策略，使程序代码发生形式上的改变。本节讨论的是编译阶段的一些代码优化方法，不考虑程序本身算法的效率。

2.2.1 变量优化

任何一个程序离不开变量的定义和变量计算，一些基于变量的简单优化方法可以起到一定的优化效果。表 2.2给出了常量合并的传播的优化方法，在编译时就已经可以计算出的常量不需要放到程序执行时计算，减少程序运行时间。表 2.3描述了公共

表 2.2 常量合并与传播优化

优化前代码	优化后代码
X = 3; Y = X + 2; Z = 3 * Y;	X = 3; Y = 5; Z = 15;

子表达式外提的优化实例，把两个表达式中重复出现的部分外提出来单独运算，显然优化前进行了 4 次加法，优化后进行了 3 次加法，可以起到一定的加速作用。还有一

表 2.3 公共字表达式外提优化

优化前代码	优化后代码

x = b+c+d;
y = b+c+e;
	t = b+c;
	x = t+d;
	y = t+e;

些变量优化的场景，比如编译原理中描述的死代码删除。死代码删除是指某个变量的两次定值之间程序没有对该变量的引用，这说明变量的第一次赋值是多余的，可以删除。一些无用转移语句可以导致一些分支在编译的时候就确定了是否执行，这样编译器可以在编译的时候就丢弃这些转移逻辑，提升代码执行速度。

2.2.2 循环优化

循环逻辑是程序中重复执行的代码块，对循环结构的优化将会带来更加显著的优化效果。将循环中的不变量或者不变代码外提是最简单的循环优化方法。表 2.4中将循环中的不变量 2*x+1 提出循环外之后，避免了每次迭代时对 2*x+1 表达式的计算，优化后只需在循环开始之前计算一次，大大减少了程序运行时间。将循环体内的运算

表 2.4 循环不变量外提

优化前代码	优化后代码

x = c;
for(i=0; i<100; i++)
array[i] = 2*x+1;
	x = c;
	t = 2*x+1;
	for(i=0; i<100; i++)
	array[i] = t;

强度削弱也是一种循环优化的方法，在表 2.5中，我们在循环体内使用加法代替乘法并且得到相同的计算结果。这种逻辑上的更改不但没有破坏程序结果，而且将每次迭代中的乘法换成了计算强度相对较小的加法，这种优化的效果将随着循环体执行次数的增加而增加。至于如何确定程序中的循环结构和循环中不变量将在后续章节控制流

表 2.5 循环内运算强度削弱

优化前代码	优化后代码

for(i=0; i<100; i++)
t = i*5;
	t = 0;
	for(i=0; i<100; i++)
	t += 5;

分析和数据流分析中描述。

2.3 控制流分析和循环查找

循环是程序中常见的一种逻辑结构，想要针对循环进行优化必须先确定程序中的循环结构。常见编程语言都有一些用于构造循环的语法，如 FORTRAN 语言中的 DO 语句，C/C++ 等语言中的 for，REPEAT，Do-while 等语句，代码中有循环语句构造的循环结构是比较容易找出的。但是一些条件转移语句和无条件转移语句等形成的循环结构复杂，需要分析程序的控制流才能确定。

2.3.1 基本块和控制流图

编译原理中的基本块是指程序中的一组顺序执行的代码序列，基本块的第一行为唯一入口，最后一行为唯一出口。基本块的特点是只能从入口进入，出口退出，基本块内语句按照顺序无转移地执行。我们可以按照如下方法来确定代码中的基本块，首先找到每个基本块的入口语句，程序的第一行语句，由条件转移或者无条件转移到的语句，以及紧跟在转移语句之后的语句都是基本块入口。然后根据所有基本块的入口构造基本块，由入口语句到下一个入口语句 (不包含该语句) 之间的所有代码构成一个基本块，或者由入口语句到一个最近的转移语句 (包含该语句) 之间的语句构成一个基本块。最后所有没有被包含到基本块中的语句，将是程序中的任何控制语句不能到达的，即一定不会被执行，直接删除。

代码 2.3 基本块划分

```
1  read(input)
2  i = i+1
3  if(i>input) goto(11)
4  read(j)
5  if(i==1) goto(8)
6  sum = sum*2+j
7  goto(9)
8  sum = sum+j
9  i=i+1
10 goto(3)
11 write(sum)
```

代码块 2.3伪代码中，根据上述基本块确定规则可知，(1) 是整个程序入口,(3)(8)(9)(11) 是转移语句转移到的语句，(4)(6)(8)(11) 是转移语句之后的语句，所以 (1)(3)(4)(6)(8)(9)(11) 为入口语句。再根据基本块构造规则可以得到 7 个基本块，分别是语句 (1) 和 (2); 语句 (3); 语句 (4) 和 (5); 语句 (6) 和 (7); 语句 (8); 语句 (9) 和 (10); 语句 (11)。程序被划分成一些基本块之后，可以依据程序中基本块的执行顺序用有向边把基本块链接起来，这就是程序的控制流图。图 2.3中描述的正是代码块 2.3中伪代码的控制流图，是一个唯一首节点的有向图。

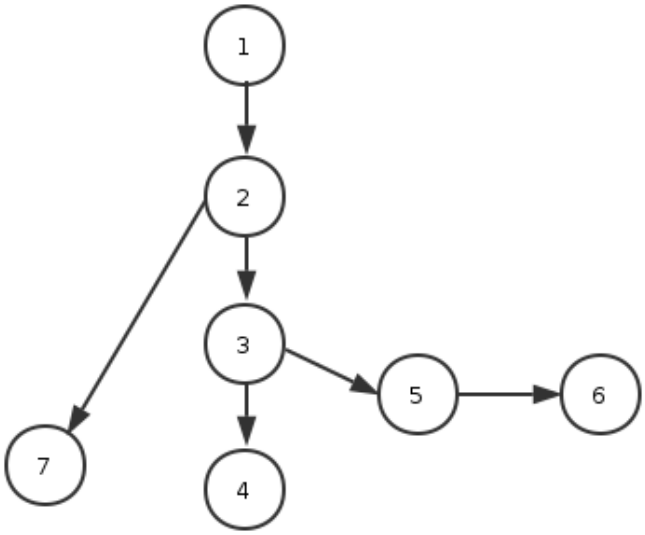

图 2.3　控制流图

2.3.2　控制流分析

在上一章节得到的程序控制流图中，存在具有如下性质的节点序列，则构成一个循环。首先节点序列是强连通的，序列中任意两个节点之间都一定存在一条通路，而且通路上的所有节点都属于该节点序列。特别是节点序列中只包含一个节点，则该节点必须要有一条有向边指向自身。而且节点序列只能有一个入口节点，入口节点一定是从序列外有某个节点指向它或者入口节点就是程序控制流图的首节点。因此，控制流图中的循环一定是强联通性和入口唯一性的节点序列。图 2.4是一个比较复杂程序的控制流图，显然 {5},{4,5,6,7},{2,3,4,5,6,7} 满足循环结构的强联通性和入口节点唯一性，是程序中的循环结构。节点 2,4 虽然也有强联通性，但是节点 (2) 和 (4) 都可以作为这个序列的入口，不具有入口唯一性，因此不是一个循环。编译原理理论提供了如何根据控制流图找出程序中循环结构的方法。首先定义了节点的必经节点，即程序进入该节点之前必须要经过的节点，节点 (1) 是所有节点的必经节点。一个节点的所有必经节点构成必经节点集。图 2.4中，每个节点的必经节点集为：D(1)={1},D(2)={1,2},D(3)={1,2,3},D(4)={1,2,4},D(5)={1,2,4,5},D(6)={1,2,4,6},D(7)={1,2,4,7}。节点 n 的必经节点集为 D，如果在 D 中存在一点 m 而且在程序控制流图中存在有向边 <n, m> , 则称有向边 <n, m> 是一条回边。以节点 (7) 为例，节点 (4) 是它的必经节点而且存在有向边 <7,4>, 所以 <7,4> 为一条回边。则有节点 (4)、节点 (7) 以及所有通路到达 (7) 而且该通路不经过 (4) 的所有节点结合构成一个循环，且节点 (4) 为该循环唯一入口。程序中回边的

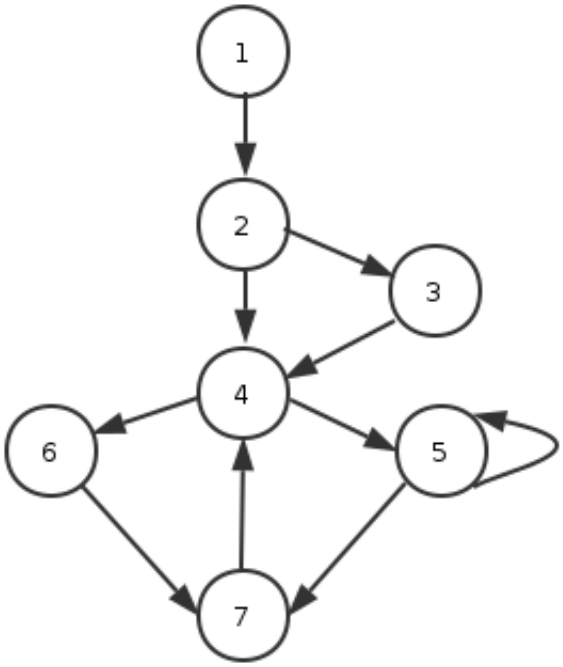

图 2.4　控制流图和循环查找

数量和循环数量一致。

2.4　数据流分析

数据流分析主要是收集、分析整个程序内的数据及控制流相关的基本信息，是实现循环优化和全局优化的必要准备工作。通过变量的引用-定值链和定值-引用链分析，确定一些变量在程序语句中可迁移的范围，这也是本文优化 GPU 程序算法的一个重要支撑。

2.4.1　程序中的点和通路

我们认为控制流图基本块中前后相继的两个语句之间是有一个点的，同样两个相继基本块之间也是有一个点的。从全局代码出发，所有点的序列为 P_1,P_2,$\cdots$,P_n。则对任意的节点 P_i 一定满足如下两个条件: 第一，若 P_i 是语句 d_i 之前最近的一个点，则 P_{i+1} 是紧跟在同一基本块中语句 d_i 之后的点；第二，P_i 若是某个基本块中最后一个点，则 P_{i+1} 是该基本块后记基本块的第一个节点。通路是指一个点可以通过控制流图中的有向边走到另一个点，显然所有点都有到程序中最后一个点的通路。

2.4.2 到达-定值数据流方程

变量 X 在某点 d 的定值到达另一点 u，是指在程序控制流图中存在一条从 d 到 u 的通路，而且在该条通路上没有对变量 X 再定值，则称点 d 为变量 X 在点 u 的定值点。所有程序中的变量都以两种方式出现，一是出现在等号的左边，即定义变量或者改变变量的值，称为定值；二是变量出现在等号的右边作为表达式的一部分或者作为函数的形参，变量本身不会改变，称为变量的引用。值得注意的是定值不仅仅是赋值一种，函数的引用参数和指针都是一种定值的方式。

2.4.3 引用-定值链

程序中某点 P 引用了变量 X 的值，我们把能到达 P 的变量 X 的所有定值点成为变量 X 在引用点 P 的引用-定值链 (ud 链)。在循环优化中，利用变量的引用-定值链可以找出循环中不变的计算。在控制流基本块中，变量 X 的引用点 P 之前有 X 的定值点 d，而且 X 在 d 点的定值可以到达 P，则 X 在点 P 的 ud 链为 {d}。如果基本块中，变量 X 的引用点之前没有 X 的定值点，则在进入基本块时 X 的所有定值点都能到达点 P。在优化循环时，循环中的语句一旦满足该语句引用的所有变量的 ud 链都在循环之外的条件，就可以认为该行语句执行的计算在循环每一次迭代时都是不变的，即可以移出循环之外执行。

2.4.4 定值-引用链

定值-引用链 (du 链) 是和 ud 链相对应的，du 链是指程序在某点 P 对变量 X 的定值可以到达的所有变量 X 的引用的集合。基本块 B 内，若在 P 点之后有对变量 X 的重新定值，则点 P 到距离 P 最近的 X 的定值点之间的所有 X 的引用点集合是变量 X 在点 P 的 du 链。若基本块 B 内不存在对变量 X 的再定值，则 B 内所有对 X 的引用点和 B 之后基本块中对 X 在 P 点处的定值的引用点全体构成这条 du 链。根据 X 的 du 链，我们可以求出在离开基本块 B 时，哪些变量 X 的定值能够到达后续基本块中 X 的引用点。那么根据这些条件可以确定哪些变量的定值在某一区间内不被引用，则在此区间内的定值为无效定值。

第 3 章　基于优化 CPU 和 GPU 内存交互的设计和实现

前两章总结了大数据高通量仿真中 CPU/GPU 异构并行计算的必要性和一些已有的 GPU 异构编程优化方法，并利用编译原理中引用-定值链和定值-引用链分析程序中的数据流，介绍了一些适用于一般程序的代码移动优化方法。程序员只有清楚的了解 CPU/GPU 的体系结构特点，再根据具体的应用场景设计合适的程序逻辑，选择合适的 GPU 编程优化方法才能够编写出充分利用 GPU 计算性能的程序。本章基于 CPU/GPU 异构计算的内存隔离特点，结合编译原理中的代码移动规则新提出了一种通过修改源代码来减少 CPU 和 GPU 内存之间数据交换带来的系统消耗的优化方法，并在第四章通过一些对比实验证明了本文提出的方法确实有优化 GPU 程序的作用。

3.1　迁移数据交换操作

由于 CPU 和 GPU 的内部结构不同，CPU 适合处理顺序计算、程序逻辑以及系统 IO 等任务，而 GPU 适合处理高度并行的复杂计算任务。所以在 CPU/GPU 异构体系结构中运行的程序包含 CPU 代码、GPU 代码以及 CPU 和 GPU 的交互代码。普通的程序逻辑就是 CPU 代码，CPU 和 GPU 内存之间的数据交换操作就是交互代码，核函数就是由 GPU 代码组成的。CPU 内存和 GPU 内存是隔离的，运行 CPU 代码时我们把需要的所有数据拷贝到 GPU 内存中，同样地运行 GPU 代码时，我们需要把所有数据拷贝到 GPU 内存中。所以 GPU 程序中每一次核函数的调用，通常会在 CPU 内存和 GPU 内存之间进行两次数据交换。核函数执行之前，CPU 和 GPU 执行相应的交互代码 API(CudaMalloc、CudaMemcpy) 在 GPU 内存分配相应大小空间并把核函数需要的数据拷贝到 GPU 已分配内存中，这个过程称为数据拷入操作 (data copyin)。data copyin 操作结束之后，GPU 开始启动运行核函数 (execution)。在核函数计算结束之后，我们需要把 GPU 中计算的结果再次拷贝到 CPU 内存, 称为数据拷出 (data copyout)。最后如果 GPU 中的变量不会再被后续核函数使用，GPU 会释放核函数运行之前分配的 GPU 内存，这是数据释放操作 (data freeing)。由此可见数据在 CPU 和 GPU 内存交换的总时间是由一下三个部分组成的：1. 在 GPU 内存上分配空间等预处理操作时间；2. 数据在 PCIe 总线上传输时间；3.GPU 释放内存等清理操作时间。在核函数比较多的程序中，CPU 和 GPU 之间频繁的数据交换将成为整个程序的瓶颈，这也是本文优化方法的出发点。本节主要是根据编译原理中代码移动原理迁移 GPU 程序中核函数

的数据拷贝操作，使原来在程序中固定点执行的数据交换操作可以允许在程序中的一段代码区间执行。

3.1.1 迁移限制

如上所述，一个核函数的执行被拆分成了四个独立的阶段，数据拷入, 执行, 数据拷出和数据释放。显然这四个阶段的执行顺序是不能颠倒的，必须按照数据拷入 $\rightarrow$ 执行 $\rightarrow$ 数据拷出 $\rightarrow$ 数据释放的顺序执行，但是每个操作具体的执行时间确是可以调整的。本文的出发点是优化数据交换操作，所以我们不迁移核函数的执行操作，只迁移数据拷入，数据拷出，数据释放三个数据交换操作。我们可以把数据拷入操作的交互代码可以看作是 CPU 代码中对变量的一个引用，可以沿着代码语句序列向上移动直到程序中某个点改变了该数据，即遇到该变量的定值点就不能再向上移动了，同时我们得保证数据拷入操作在核函数开始执行之前完成。同样地，我们也可以把交互代码中的数据拷出操作看作是 CPU 代码中队变量的定值，可以和数据释放操作一起沿着代码语句序列向下移动，直到程序中出现某条语句引用该变量，即遇到该变量的引用点结束，同时我们必须保证数据拷出和数据释放操作是在核函数执行完成之后才开始执行的。数据拷出操作之后，为了减少 GPU 内存的占用，我们将立刻释放相应的 GPU 内存，除非该内存数据会在后续的核函数中继续用到我们才选择继续保存，这样可以减少冗余的数据拷贝。所以在整个数据交换操作的迁移过程中数据依赖性是迁移的前提，在下面的章节我们将利用编译原理中的引用-定值链和定值-引用链方法来分析整个程序中的控制流和数据流，得到具体的迁移操作。

3.1.2 代码移动

现在我们介绍如何调整核函数执行过程中的四个阶段，以及如何根据代码移动方法迁移核函数的数据交换操作。代码块 3.1描述了原始 CUDA 代码中一个核函数的结构，数据拷入操作，核函数执行，数据拷出操作，以及数据释放操作是一个紧密的整体。但是在代码块 3.2中，数据拷入操作被提前执行，同时数据拷出和数据释放操作被推后执行。显然代码块 3.2中针对数据操作的迁移并没有改变代码块 3.1中的执行结果。本文的 GPU 程序优化算法只针对 CPU 和 GPU 内存之间的数据交换操作，所以我们只迁移数据交换操作。

代码 3.1　原始 CUDA 代码

```
__global__ void kernel(int *device, int i);
int host[1000] = {0}, tmp[1000] = {0};
for (int i = 0; i < 1000; i++)
    host[i] = host[i] + 1;
 .....
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
int *device;
CUDAMalloc(&device, sizeof(int) * 1000);
CUDAMemcpy(device, host, sizeof(int) * 1000,CUDAMemcpyHostToDevice); /*data copyin*/
for (int i = 0; i < 1000; i++) /* execution */
    kernel<<<1,1>>>(device, i);
CUDAMemcpy(host, device, sizeof(int)* 1000,
CUDAMemcpyDeviceToHost); /*data copyout*/
CUDAFree(device); /*data freeing */
 .....
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
```

代码 3.2　迁移数据拷贝操作

```
__global__ void kernel(int *device, int i);
int host[1000] = {0}, tmp[1000] = {0};
for (int i = 0; i < 1000; i++)
    host[i] = host[i]+1;
int *device;
CUDAMalloc(&device, sizeof(int) * 1000);
CUDAMemcpy(device, host, sizeof(int) * 1000, CUDAMemcpyHostToDevice); /*data copyin*/
 .....
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
for (int i = 0; i < 1000; i++) /* execution */
    kernel<<<1,1>>>(device, i);
 .....
for (int i = 0; i < 1000; i++)
    tmp[i] = tmp[i]+1;
CUDAMemcpy(host, device, sizeof(int)* 1000,
CUDAMemcpyDeviceToHost); /*data copyout*/
CUDAFree(device); /*data freeing */
```

核函数的数据拷入操作可以沿着代码语句序列向上移动，我们只要保证数据拷入操作在相应数据的定值点之后和核函数执行之前的代码区间执行就是合法的。同样数据拷出操作可以沿着代码语句序列向下移动，我们只要保证数据拷出操作在相应数据的引用点之前和核函数执行之后的代码区间执行就是合法的至于核函数的数据释放操作，必须保证在数据拷贝操作之后执行。当程序在 GPU 上内存使用量较少而且 GPU 内存足够使用时，我们可以选择让 GPU 内存保持所有的数据直到整个程序执行结束之后释放。但是 GPU 内存在使用时不总是足够的，所以我们选择在数据拷出到 CPU 内存之后立刻让 GPU 内存释放该变量，除非该变量在后续核函数中会再次用到才让 GPU 内存保持。

3.1.3 异步操作优化

图 3.1　初始结构

这里先单独讨论一下单个核函数在数据拷贝操作迁移之后的异步优化。我们发现数据拷入操作沿着代码序列向上移动之后，数据拷入操作和核函数执行的代码语句之间出现了一部分 CPU 代码。同样地，数据拷出操作沿着语句序列向下移动之后，核函数结束执行和数据拷出操作之间也有一部分 CPU 代码。我们再迁移数据交换操作时是以数据依赖性为前提的，所以这部分 CPU 代码和数据交换操作时没有任何依赖的，即可以异步并行执行。图 3.2对图 3.1中单个核函数的执行过程进行了一些异步优化。首先在图 3.1中第一段 CPU 代码执行时，PCIe 和 GPU 都处于空闲状态，数据拷入操作在 CPU 代码完全执行结束之后才开始。我们在代码块 3.2中可以看到，在迁移数据拷入操作之后，又一块 CPU 代码可以和数据拷入操作异步执行。CPU 代码执行时不占用 PCIe 总线和 GPU 资源，GPU 预分配内存和数据拷入操作也不占用 CPU 代

图 3.2　异步优化

码执行时需要的计算资源。图 3.2中的并行策略可以更加充分地利用 CPU/GPU 资源和 PCIe 总线带宽。同样地，我们也可以对数据拷出操作和 CPU 代码进行异步优化，这里就不做赘述。当然如果程序员足够聪明，可以在设计核函数时就选择以异步的方式进行数据拷贝操作。异步优化必须要有一块与数据交换操作无依赖、可并行的 CPU 代码时才能发挥作用，这与我们本文提出的减少数据交换频率和数据交换总量的优化方法不相同。

结论

本文采用……。（结论作为学位论文正文的最后部分单独排写，但不加章号。结论是对整个论文主要结果的总结。在结论中应明确指出本研究的创新点，对其应用前景和社会、经济价值等加以预测和评价，并指出今后进一步在本研究方向进行研究工作的展望与设想。结论部分的撰写应简明扼要，突出创新性。）

参考文献

附录 A　***

附录相关内容…

附录 B Maxwell Equations

因为在柱坐标系下，$\bar{\bar{\mu}}$ 是对角的，所以 Maxwell 方程组中电场 **E** 的旋度

所以 **H** 的各个分量可以写为：

$$H_r = \frac{1}{\mathbf{i}\omega\mu_r}\frac{1}{r}\frac{\partial E_z}{\partial\theta} \tag{B–1a}$$

$$H_\theta = -\frac{1}{\mathbf{i}\omega\mu_\theta}\frac{\partial E_z}{\partial r} \tag{B–1b}$$

同样地，在柱坐标系下，$\bar{\bar{\epsilon}}$ 是对角的，所以 Maxwell 方程组中磁场 **H** 的旋度

$$\nabla\times\mathbf{H} = -\mathbf{i}\omega\mathbf{D} \tag{B–2a}$$

$$\left[\frac{1}{r}\frac{\partial}{\partial r}(rH_\theta) - \frac{1}{r}\frac{\partial H_r}{\partial\theta}\right]\hat{\mathbf{z}} = -\mathbf{i}\omega\bar{\bar{\epsilon}}\mathbf{E} = -\mathbf{i}\omega\epsilon_z E_z\hat{\mathbf{z}} \tag{B–2b}$$

$$\frac{1}{r}\frac{\partial}{\partial r}(rH_\theta) - \frac{1}{r}\frac{\partial H_r}{\partial\theta} = -\mathbf{i}\omega\epsilon_z E_z \tag{B–2c}$$

由此我们可以得到关于 E_z 的波函数方程：

$$\frac{1}{\mu_\theta\epsilon_z}\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial E_z}{\partial r}\right) + \frac{1}{\mu_r\epsilon_z}\frac{1}{r^2}\frac{\partial^2 E_z}{\partial\theta^2} + \omega^2 E_z = 0 \tag{B–3}$$

攻读学位期间发表论文与研究成果清单

[1] 高凌. 交联型与线形水性聚氨酯的形状记忆性能比较 [J]. 化工进展, 2006, 532 一 535.（核心期刊）

致谢

本论文的工作是在导师……。

作者简介

本人…。