中国科学技术大学

硕士学位论文

基于CPU-GPU异构平台的性能优化及多核并行编程模型的研究

姓名：陈波

申请学位级别：硕士

专业：计算机应用技术

指导教师：徐云

2011-04-18

摘 要

随着图形处理器（GPU）的计算能力和可编程性的不断提高，利用 GPU 进行通用计算（GPGPU）逐渐成为研究的热点。通常 GPGPU 计算采用 CPU-GPU 的异构模式，虽然这种异构模式能够获得好的性能收益，但其程序开发和性能优化的复杂度要比同构系统大的多。

在 CPU-GPU 异构系统上进行计算会遇到很多性能瓶颈，例如：负载均衡、同步与延迟、数据局部性、任务划分等。这些因素对提高程序的性能至关重要。此外，尽管 CUDA 编程模型极大的降低了 CPU-GPU 异构平台编程的难度，但对于大多数串行程序开发者来说，其开发门槛还是相对较高，而且当底层的硬件平台发生变化时，软件开发者又要学习一种新的编程模型并针对新的硬件平台重新改写已有的程序，这无疑加重了程序员的负担。因此设计一种使用简单、平台无关的多核并行编程模型具有重要意义。

本文主要进行了以下研究工作：

（1）分析了影响 CPU-GPU 异构平台上程序性能的关键因素，全面总结了已有的优化方法并设计了一种使用原子函数实现不同线程块之间同步等自己的优化方法和优化策略。对每一种优化方法都进行了实验验证和理论分析，其中我们设计的使用原子函数实现不同线程块之间同步的方法比现有的重新启动内核函数的方法要快 4~5 倍。

（2）为了进一步验证各种优化方法的效果，也为了完整的介绍在 CPU-GPU 异构上进行程序开发的流程（算法设计、编程实现、性能优化），我们以解决生物信息学中的 DNA 或蛋白质局部序列比对问题为例，在 CPU-GPU 异构平台上设计并实现了基于列并行的 Smith-Waterman 算法，综合运用多种优化方法进行优化后的并行程序获得了平均 37 倍的加速比。

（3）在深入分析了 OpenMM 并行编程框架之后，我们设计了一个基于库的、平台无关的多核并行编程模型。为了验证该模型的可行性和易用性，我们实现了一个面向科学计算的原型系统，通过设计合理的 API 层次结构，对上层用户屏蔽了底层硬件的具体细节，用户只需在编译时根据具体的底层硬件平台选择相应的动态链接库就可以将原来的串行程序变成高效的并行程序。

关键词：GPU，GPGPU，CUDA，异构计算平台，性能优化，并行编程模型

ABSTRACT

With the computing power and programmability of graphics processor uint (GPU) increasing continuously, general purpose computing on GPU (GPGPU) is gradually becoming a research hotspot. Usually the computing with GPGPU utilizes a heterogeneous mode of CPU and GPU. Although the heterogeneous system based on CPU-GPU can achieve good performance gains, program development and performance optimization of it are more complexity compared with the homogeneous system.

Computing on the heterogeneous system based on CPU-GPU will encounter a lot of performance bottlenecks, such as load balancing, synchronization and delay, data locality, task division and so on. These factors are essential to improve the performance of the program. On the other hand, although the programming difficulty of the heterogeneous system based on CPU-GPU reduced greatly due to the CUDA programming model, the development requirement is still high for most of the serial program developers. And when the underlying hardware changes, software developers have to learn a new programming model and rewrite programs for the new hardware platform, which increases the burden on the programmer. So it is very significate to designing a simple and platform-independent multicore parallel programming model.

We mainly did the following researches:

(1) Analyzed the key factors which affect the performance of CUDA programs, summarized the existing optimization methods comprehensively and proposed our new optimization methods and optimization strategy, such as using atomic functions to achieve synchronization between different thread blocks. For each optimization method, we did experiments to verify its effectiveness and theoretical analysis. And our method that exploiting atomic functions to synchronize different thread blocks is 4~5 times faster than existing method that restarting the kernel function.

(2) To further validate the effectiveness of various optimization methods, and also to descripte the development process (algorithm design, programming, performance optimization) of heterogeneous platform based on CPU-GPU, we exploited CPU-GPU heterogeneous computing platform to solve the problem of DNA or protein sequence alignment which is a bioinformatics problem, namely designed and implemented a new column-based parallel Smith-Waterman algorithm based on

CUDA platform. The optimized parallel program is 37 times faster than the serial program.

(3) After analyzing the OpenMM parallel programming framework deep, we proposed a library-based and hardware-independent multicore parallel programming model. In order to verify the feasibility and simplicity of the model, we implemented a prototype system for scientific computing and tested it. It shields the details of underlying hardware for upper users through designing rational hierarchy architecture of APIs. To parallelize the serial programs, programmers only need to select the appropriate dynamic-link library depending on underlying hardware at the compile time.

Key Words: GPU，GPGPU，CUDA，Heterogeneous Computing Platform，Performance Optimization，Parallel Programming Model

中国科学技术大学学位论文原创性声明

本人声明所呈交的学位论文,是本人在导师指导下进行研究工作所取得的成果。除已特别加以标注和致谢的地方外，论文中不包含任何他人已经发表或撰写过的研究成果。与我一同工作的同志对本研究所做的贡献均已在论文中作了明确的说明。

作者签名：_____ 签字日期：_____

中国科学技术大学学位论文授权使用声明

作为申请学位的条件之一，学位论文著作权拥有者授权中国科学技术大学拥有学位论文的部分使用权，即：学校有权按有关规定向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅，可以将学位论文编入《中国学位论文全文数据库》等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。本人提交的电子文档的内容和纸质论文的内容相一致。

保密的学位论文在解密后也遵守此规定。

□公开 □保密（____年）

作者签名：_____ 导师签名：_____

签字日期：_____ 签字日期：_____

第1章 绪论

1.1 研究背景及意义

随着工程和科学计算领域的数据量急剧增长和计算复杂度不断增加，通用 CPU 的计算能力已经无法满足其计算需求，即使是现在的多核处理器也无能为力。尽管通过搭建高性能服务器甚至集群系统的超级计算机可以满足某些领域的计算需求，但这些机器不仅造价昂贵而且维护费用和能量消耗更是惊人，只有少数大公司和科研单位才拥有超级计算机。为了满足大多数中小企业、科研院所以及个人对计算能力的需求，各种专用计算芯片便应运而生了，例如 FPGA（Field Programmable Gate Array）。虽然这些专用计算芯片的计算能力是通用 CPU 十几甚至几十倍，但其功能固定，往往只能对某一特定类型的算法起到加速的效果。此外，其价格较贵、编程复杂，不适合个人和小型实验室使用。

近年来，计算机图形处理器（GPU，Graphics Process Unit）正在以大大超过摩尔定律的速度高速发展（大约每隔半年 GPU 的性能增加一倍），远远超过了 CPU 的发展速度。当前，主流 GPU 的处理器核数达到了数百个，而对应的 CPU 却只有几个处理器核。图 1.1 和图 1.2 分别对 GPU 和 CPU 的浮点计算速度和存储器带宽进行了比较。

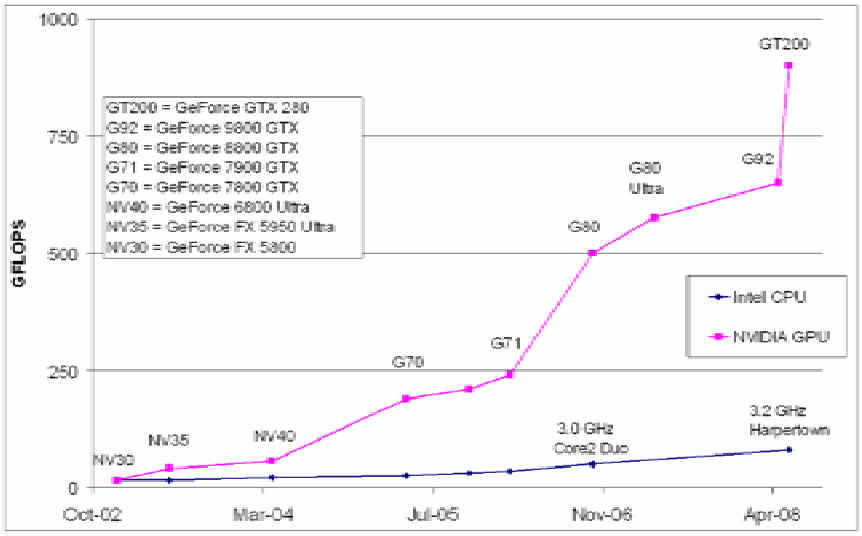

图 1.1 CPU 和 GPU 浮点计算能力的比较[1]

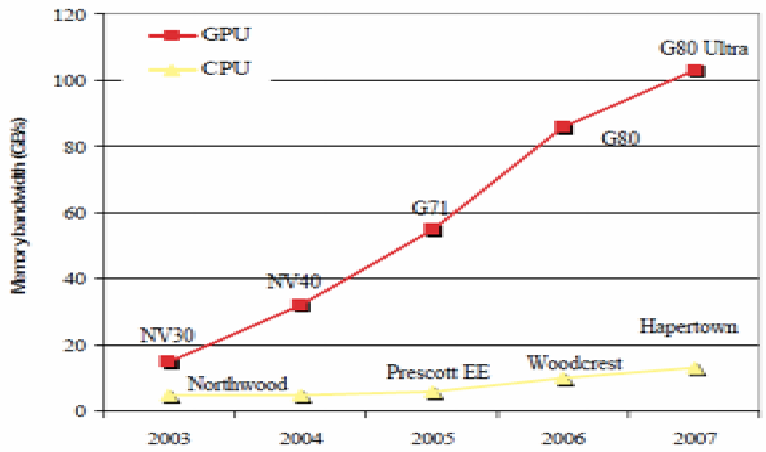

图 1.2 CPU 和 GPU 存储器带宽的比较[1]

从上图不难看出，GPU 的计算能力和存储器带宽都是同一时期 CPU 的十倍左右，而且其性能增长的速度也相当惊人。随着 GPU 处理单元数目的快速增长和可编程性大幅提高，如今的 GPU 不仅在图形图像处理领域大显身手，而且还被广泛用于生物信息学、流体力学、信号处理、信息检索、线性代数等通用计算领域，即利用 GPU 进行通用计算（GPGPU, General Purpose Computing on GPU）。

目前，GPGPU 已经成为工业界和学术界研究的热点。其实 GPGPU 的概念早就有了[7][8][9][10]，只是受当时 GPU 硬件结构的限制，在 GPU 上编程极其困难，使得 GPGPU 技术没有被广泛关注。2007 年，NVIDIA 公司推出了全新硬件架构的 G80 系列 GPU 产品，并提出了在该系列 GPU 上编程的统一设备计算平台（CUDA，Compute Unified Device Architecture）[2]，随后 AMD 公司也为其 GPU 产品提出了 CTM（Close To the Metal）编程环境[3]。这些编程模型的出现，极大的降低了程序员在 GPU 上编程的难度，越来越多的人从此进入了 GPGPU 这一新的研究领域。

GPU 和 CPU 的性能差异如此之大，主要源于二者的硬件设计不同。如图 1.3 所示，GPU 中更多的资源（晶体管）用于计算，缓存和控制部件只占很少一部分，因此 GPU 适合处理计算密集型、高度并行化的计算任务。而 CPU 则恰恰相反，CPU 中大量的资源用于缓存和逻辑控制，只有少部分资源用于计算，因此 CPU 适合运行具有分支密集型、不规则数据结构、递归等特点的串行程序[1]。

CPU 和 GPU 性能特长和适用领域都是互补的，任何一方都存在各自的局限性。为了充分发挥 CPU 和 GPU 各自的优势，搭建一个基于 CPU-GPU 的异构系统是十分必要的。在这样一个异构的系统中，GPU 与 CPU 协同工作，其中少量的 CPU 用于处理串行的任务（如：操作系统）和逻辑控制，大量的 GPU 用于并行计算。这种异构的系统不仅被广泛应用于台式机和服务器，而且已成为搭建超

级计算机的一种新的途径，是高性能计算领域一个新的发展趋势。目前，我国在这一领域已经取得了重大成果，走在世界的前列。在 2010 年 6 月份举办的世界 Top500 超级计算机评选中，我国的“ 星云 ”以 1.27 Pfloap/s 的惊人速度位居第二。该机是世界上第一台使用 CPU 和 GPU 混合搭建的超级计算机。随后，在 10 月份的世界超级计算机大会上，我国的 “ 天河一号 ” 更是以 2.57Pfloap/s 的计算速度排名第一，成为世界上最快的超级计算机，而该机同样也是基于 CPU-GPU 的异构系统[4]。

图 1.3 GPU 和 CPU 内硬件资源的分布[1]

采用 CPU-GPU 的混合结构搭建超级计算机，除了能够获得性能优势之外，在成本和功耗方面也有明显的优势。例如：比利时安特卫普大学有一台超级计算机，该机有 512 颗处理器，占用了几个机柜，成本为 530 万美元。后来换成了一台只有 8 个 GPU 的系统，成本只有 7000 美元，占地面积和功耗都大大降低，性能却和原来的机器相当[5]。日本东京工业大学的超级计算机 Tsubame 2.0 在 Green500（绿色节能 Top500 超级计算机）的测评中位居第二，虽然该机配备了 4200 颗 Tesla GPU，性能达到 Pfloap/s 级别，但功耗却只有 1340 千瓦。因此该机也是当今世界上最节能的 Petafloap 级系统[6]。

高性能计算关系到经济发展和国家安全，是衡量一个国家综合国力和科技核心竞争力的重要指标。高性能计算的广泛应用极大地推动了我国工业制造、生物医药、石油物探、气候气象、信息安全、国防科研等领域的发展。GPU 高度的并行性，惊人的计算能力，低廉的价格，较低的功耗，良好的可编程性以及工业界和学术界的大力支持和推广，使其成为一个新的并行计算平台，而被广泛应用于高性能计算领域前景光明。因此，对 CPU-GPU 异构计算平台的性能优化方法及其应用进行研究有利于推动高性能计算相关领域的发展，具有深远的研究意义。

1.2 研究现状

1.2.1 CPU-GPU 异构计算的研究现状

为了降低 CPU-GPU 异构计算的门槛，2007 年 NVIDIA 与 AMD 公司分别推出了支持各自 GPU 的编程环境：CUDA[2]（Compute Unified Device Architecture）和 CTM[3]（Close To Metal）。借助这些编程环境，程序员可以使用类 C 语言开发出在 GPU 上运行的通用程序，既无需专门学习图形学的知识，也无需熟悉图形 API 的用法。正是由于 GPU 强大的计算能力和可编程性的逐步提高才使得 GPGPU 技术引起了工业界和学术界的广泛关注。

■ GPGPU 技术成为学术界的研究热点

GPGPU 技术自 2006 年出现以后立即成为了学术界的研究热点。从 2008 年到 2011 年，并行计算领域的国际顶级会议 PPOPP（ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming）每年都收录 4 至 6 篇与 GPGPU 相关的论文。并行与分布式计算领域的国际顶级期刊 JPDC（Journal of Parallel and Distributed Computing）每年也收录若干篇与 GPGPU 相关的论文，其中仅 2008 一年就收录了 68 篇。使用 Google 学术搜索能找到 7000 多篇与 NVIDIA CUDA 相关的论文。

此外，GPGPU 技术在高校也迅速流行起来。目前，全世界已有 350 多所大学开设了 GPGPU 编程的课程，其中包括了哈佛大学、剑桥大学等世界顶尖学府，如图 1.4 所示。最值得一提的是伊利诺伊大学厄本那一香槟分校（UIUC），该校率先与 NVIDIA 公司密切合作，成为全球第一个 CUDA 卓越中心，并获得了 NVIDIA 公司捐助的资金和由 16 个节点、64 个 GPU 搭建的集群系统。

在中国，清华大学和中国科学院过程工程研究所在 GPGPU 技术的研究与应用领域走在前列，不仅开设了 CUDA 编程的相关课程，还不定期邀请 NVIDIA 公司的技术专家给学生做学术报告。为了表彰他们为 GPGPU 技术在中国的推广普及所作出的贡献，2009 年 10 月 28 日，NVIDIA 公司分别授予他们 CUDA 卓越中心的荣誉称号。现在，GPGPU 技术对国内的程序员已经不陌生了，在对 1700 名来自 12 所大学的学生进行的调查发现，大约 56%的人听说过利用 GPU 进行通用计算，69%的人对此感兴趣并准备学习 GPGPU 技术[12]。这表明在这场席卷全球的 GPGPU 风暴中，中国的科研机构和高校并没有落后。随着越来越多的学生学习和运用 GPGPU 技术，相信中国在这一领域必将大有作为。

图 1.4 开设 GPGPU 课程的世界知名学府[11]

■ 工业界大力推广 GPGPU 技术

学术界对 GPGPU 的研究热潮离不开工业界的大力支持。由于人们对高质量图形画面的要求越来越高和 3D 游戏市场的巨大需求，三大显卡生产商 NVIDIA、AMD、Intel 纷纷推出各自 GPU 产品，企图尽早占领市场先机，竞争异常激烈。

2007 年 NVIDIA 公司推出了全新架构的 G80 系列 GPU 和 CUDA 编程模型使其在 GPGPU 领域处于明显的领先地位。为了让更多的人了解 GPGPU 技术，NVIDIA 公司十分重视与高校的合作，自 2008 年 6 月授予 UIUC（University of lllinois at Urbana-Champaign）大学全球首家“CUDA 卓越中心”称号以来，已经相继有多家高校获此殊荣，包括哈佛大学、剑桥大学、清华大学等。NVIDIA 公司表示，只要高校开设 CUDA 课程并且在科研中使用 CUDA 技术就能成为 CUDA 卓越中心。NVIDA 公司将对这类高校给与一定的资金和设备支持。此外，NVIDIA 公司还通过“CUDA 校园巡讲”、“举行 CUDA 编程大赛”、“NVIDIA 奖学金计划”等方式提高其知名度。事实证明，这些措施收到了很好的效果，截止到 2011 年，CUDA 软件开发包累计下载次数达到 668000，支持 CUDA 的 GPU 销售数量早已过亿。

2006 年 AMD 公司成功收购了 ATI 公司，成为第一家同时生产 CPU 与 GPU 的芯片制造商。在 NVIDIA 推出 CUDA 计算平台后不久，AMD 也推出了自己的 CTM 编程模型，试图在通用计算领域与 NVIDIA 展开竞争。但由于使用 CTM 进行编程仍然比较复杂，宣传和推广的力度不够，使得 AMD 的 GPU 在通用计算领域远远落后于 NVIDIA。但随着 OpenCL[13][14]（Open Computing Language）的提出，或许会成为 AMD 重整旗鼓的转折点。

看到 NVIDIA GPU 获得的巨大成功，Intel 公司也宣布要生产自己的独立显卡，代号为 Larrabee[8]，并计划在 2009 至 2010 年上市。与 NVIDIA 及 AMD 基

于流处理器的 GPU 不同，Larrabee 的计算核心完全是基于 x86 指令级的通用 CPU。虽然受功耗和散热等因素的限制，Larrabee 的处理器核数比当前的 GPU 少很多（计划第一代 Larrabee 只含 32 个处理器核），并行计算能力也不如前者，但在可编程性方面有更大的优势。可以说，Larrabee 是对当前 GPU 的计算能力和可编程性两方面的折中产物。然而，由于受到制作工艺的限制，Larrabee 的问世被一再推迟，还是拭目以待吧。

■ CPU-GPU 异构计算领域的研究问题

目前，学术界和工业界对 CPU-GPU 异构计算的研究主要集中在应用方面，即如何利用该异构计算平台来加速某个具体的应用问题。例如：斯坦福大学利用 CPU-GPU 集群来加速蛋白质折叠[16][29]，南洋理工大学的 Weiguo Liu 等人利用 CPU-GPU 异构平台加速分子动力学模型[30]，商业上用该异构系统来加速 Google Earth 应用[29]等。

对于计算密集型、并行度高的应用，使用 CPU-GPU 异构平台很容易就能够获得不错的加速比，但要充分发挥该异构系统的计算能力，还需要对程序的性能进行优化。常规的优化方法[1][33][36]有：优化存储器之间的数据传输、利用全局存储器的联合访问、使用高效的共享存储器、避免控制流分支等。这些优化方法具有普适性，即对于几乎所有的 CPU-GPU 异构平台上的程序这些方法都适用而且优化效果比较明显。在此基础上，如果还要进一步提高程序的性能，就需要使用高级的优化方法[45]，例如：计算与通信重叠、主机与设备并行计算、减少同步开销等。这些高级的优化方法并不是对所有的程序都适用而且有一定的难度，需要根据具体问题具体分析。对于一个具体的应用，一般先采用常规方法进行优化，然后再尝试采用高级的优化方法进行优化。

尽管已经有了如此多的优化方法，但至今还没有一种实现不同线程块之间同步的高效方法。现有的采用重新启动内核函数的方法虽然能够实现不同线程块之间的同步，但开销较大，而且对于那些线程块之间需要进行多次同步的应用，这种方法是不可行的。本文设计了一种使用原子函数实现不同线程块之间同步的方法很好的解决了这一难题。

1.2.2 多核并行编程模型的研究现状

目前，几乎所有的计算平台都是多核系统。从高端服务器、普通 PC 机、笔记本到各种手持设备无一不是多核系统。多核芯片种类繁多，包括多核 CPU、GPU、Cell、DSP 等，它们的体系结构相差很大，都有各自应用的领域。

为了更好的利用多核计算平台的计算能力，必须将现有的、大量的串行程序

并行化，即需要软件开发者掌握并行编程的技术。为了降低程序员编写并行程序的难度，体系结构设计者和软件架构设计者们提出了各种各样的并行编程模型。其中传统的并行编程模型有：MPI[46]、OpenMP[46]、Pthreads[47][48]等，新兴的并行编程模型有：TBB[49]、IBM X10[50]、OpenCL[13][14]等，学术界研究中的并行编程模型有：SWARM[41]、Huckleberry[39]、Skandium[51]等，面向特定应用问题的并行编程模型有：Pub/Sub[52]、MapReduce[53]等，面向特定硬件结构的并行编程模型有：CUDA[1][2]、StreamIt[54]、NP-Click[55]等。每一种并行编程模型都与某一特定体系结构的计算机相关。

虽然这些并行编程模型在一定程度上降低了程序员开发并行程序的难度，但仍然没有很好的解决以下两个问题：(1) 如何使程序员完全不用掌握并行编程的技术就可以将串行程序并行化；(2) 如何实现编程模型的平台无关性，即当底层的硬件平台改变时，如何使应用程序无需修改或少做修改即可在新的硬件平台上高效运行。

2010 年 7 月，斯坦福大学的 Eastman 和 Pande 领导的团队设计和实现了一个面向分子动力学模拟计算的并行函数库 OpenMM[25][26][40]。通过采用合理的分层结构，较好的解决了上述两个问题。借鉴 OpenMM 的成功经验，我们设计一个基于库的、平台无关的多核并行编程模型，并实现了一个面向科学计算的原型系统对该模型的可行性、效率和易用性进行了验证。

1.3 本文的研究内容

1.3.1 CPU-GPU 异构平台的性能优化

虽然基于 CPU-GPU 的异构系统能够获得好的性能收益，但相对于同构系统而言，其编程复杂性要大的多。在异构系统上进行计算会遇到很多性能瓶颈，例如：负载均衡、同步与延迟、数据局部性、任务划分等。如果这些问题处理不当就会对程序的整体性能产生重大不利的影响。已有的优化方法主要有：访存优化[36][43][44]、计算与通信重叠以及主机与设备并行计算[45]等，但还没有很好的解决设备端全局同步的问题，现有的方法是通过重新启动内核函数来实现的，但这样做开销太大。本文在第 3 章将会认真分析影响 CPU-GPU 的异构系统上程序性能的关键因素，全面总结了已有的优化方法并提出了使用原子函数实现不同线程块之间的同步等自己的优化方法和优化策略。针对每一种优化方法，我们都设计了测试程序进行实验验证和理论分析。

1.3.2 生物信息学中的问题求解

生物信息学是高性能计算的一个重要应用领域。随着 DNA、蛋白质等生物数据的爆炸式增长，普通计算机的处理能力已经无法满足其计算需求。因此，很多高端服务器、Cluster、甚至超级计算机都被应用于这一领域[15][16]。此外，生物信息学领域的大多数问题都具有很高的并行性和计算密度，而这恰恰是 GPU 所擅长的，所以使用 GPU 来求解这些问题再适合不过了。

生物信息学领域问题众多，比较著名的有斯坦福大学发起的蛋白质折叠项目 Folding@home[16]，DNA 或蛋白质序列比对[17][18]，motif 发现问题[19][20]，单体型分型[21][22]，蛋白质结构预测[23][24]等。本文在第 4 章将以使用 CPU-GPU 的异构系统解决 DNA 局部序列比对问题为例，完整介绍在异构系统上进行程序开发的基本流程（包括算法设计、程序实现、性能优化等），并综合运用多种优化方法对程序进行性能优化。

1.3.3 平台无关的多核并行编程模型

目前，几乎所有的计算平台都是多核的。从高端服务器、普通 PC 机、笔记本到各种手持设备无一不是多核系统。多核芯片种类繁多，包括多核 CPU、GPU、Cell、DSP 等，它们的体系结构相差很大，各有各的应用的领域。

面对种类繁多的计算设备，对应用程序员来说既是机遇也是挑战。一方面我们有更多可供选择的计算平台来提高应用程序的性能；另一方面我们也面临着两个严峻的问题：1）如何使现有的大量串行代码并行化以便充分发挥多核平台的计算能力；2）当底层硬件平台发生变化时，如何使应用程序无需修改或少做修改即可在新的硬件平台上高效运行。为了解决上述两个问题，本文将借鉴 OpenMM[25][26]的成功经验，提出一个平台无关的多核并行编程模型。为了验证模型的可行性和效率，我们将实现一个面向科学计算的原型系统，并进行了测试和验证。

1.4 论文结构

第 1 章绪论中主要介绍了本文的研究背景和研究工作。第 2 章详细介绍了 CUDA 计算平台，主要包括硬件体系结构、软件编程模型，编程规范等内容。第 3 章分析基于 CPU-GPU 的异构系统中影响程序性能的关键因素，全面总结 和提出了相应的优化方法和优化策略并通过实验检验优化效果。第四章详细描述了如何在 CPU-GPU 异构系统上设计并实现基于列并行的 Smith-Waterman 算法[27]以

及相关的优化策略。第 5 章借鉴 OpenMM 的成功经验，设计了一个基于库的、平台无关的多核并行编程模型并实现了一个面向科学计算的原型系统，以及对该原型系统的可行性和效率进行实验验证。第 6 章总结全文。

第2章 CUDA 计算平台

2.1 CUDA 简介

2.1.1 CUDA 理论基础

CUDA 是 NVIDIA 公司 2007 年提出的支持其 GPU 进行通用计算的编程模型和开发环境。在该平台下，软件开发者可以直接使用高级语言（扩展的 C 语言）编写在 GPU 上运行的程序，而不必使用 DirectX 或者 OpenGL 这些图形 API，这就既大大降低了利用 GPU 进行通用计算的难度，也提高了程序的性能。

CUDA 平台为程序开发者提供了所有必要的组件，包括编译器（nvcc）、库函数、运行时环境、显卡驱动等。支持多种操作系统，包括 Windows、Linux 以及 Mac OS。此外，为了方便用户的开发工作，还提供了包含很多程序实例的 SDK（Software Development Kit）、调试器、性能分析器等工具[1][28]。图 2.1 清晰地呈现了 CUDA 软件栈的层次结构以及应用程序调用关系。同时此图也说明了 CUDA 平台是非常灵活的，给用户提供了更多选择的机会。应用程序既可以调用上层 API（CUDA Library 和 CUDA Runtime）也可以直接调用底层 API（CUDA Driver）。后者显然要更快一些但难度也会大一些。

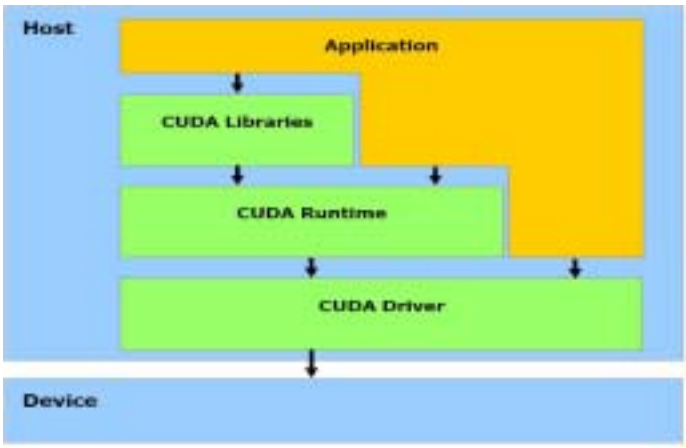

图 2.1 CUDA 软件栈[1]

一个完整的 CUDA 程序分为两部分：主机代码和设备代码。其中前者在主机（通常指 CPU）上运行，后者在设备（GPU 或其他类似的协处理器）上运行（前提是系统中存在支持 CUDA 的 GPU）。此外，对于没有支持 CUDA 的 GPU 的用户，可以选择模拟模式来运行 CUDA 程序，此时系统将会用主机来模拟设

备的运行，但速度很慢。

除了方便性和灵活性之外，CUDA 还具有高度的抽象性和可扩展性。抽象性体现在三个方面：线程层次结构、共享存储器、同步路障。这种机制既提供了细粒度（fine-grained）的数据级并行和线程级并行，也提供了粗粒度（coarse-grained）的任务级并行。一个可执行的 CUDA 程序可以在具有任何数量处理器核的设备上运行，这样用户在进行任务划分时就不必关心底层硬件平台实际所拥有的物理处理器的数量。而且当底层的硬件更新升级后，原有的程序无需做任何改变就可以获得好的性能收益。这就体现了 CUDA 良好的可扩展性[1]。

目前，只有 NVIDIA 自己全新架构的 GPU 支持 CUDA。这些 GPU 几乎涵盖了 NVIDIA 公司的全线产品，包括用于台式机或笔记本的 GeForce 系列（GeForce 8、GeForce 200 等）、面向科学计算领域 Tesla 系列（Tesla C1060、Tesla S1070 等）和针对专业可视化计算领域的 Quadro 系列（Quadro 6000、Quadro 2000M 等）。这些 GPU 为全世界数以百万计的台式机、笔记本、工作站以及超级计算机提供动力支持，为普通消费者、专业技术人员、科学家和研究者加快了处理计算密集型任务的速度。

2.1.2 CUDA 的应用领域及成功案例

GPU 作为众核处理器（拥有 100 个以上的处理器核）具有非常强大的计算能力，特别适合处理高度并行性、计算密集型的大规模数据集。因此 CUDA 被广泛应用于分子动力学、生物信息学、图形图像处理、石油天然气勘探、加密技术、数据挖掘、计算金融学、天文学等高性能计算领域。目前除了可以用 C 语言开发 CUDA 程序以外，还可以使用 C++、Fortran 等语言。

下面列举一些利用支持 CUDA 的 GPU 来加速的典型应用。

- Folding@home：2000 年 Stanford 大学开发了一套命名为 Folding@home 的客户端软件，旨在利用互联网上的计算机进行蛋白质折叠模拟计算，以便研制新药物治疗疾病。目前已开发出了支持 CUDA 版本的 Folding@home，这样就可以利用 NVIDIA 的 GPU 强大的计算能力加速计算。实验表明，使用 GeForce GTX 280 时，显示的速度为 500ns/day 左右，而如果使用 CPU，显示的速度大约只有 4ns/day，可见加速效果非常显著[16][29]。
- Google Earth：这是一款能够将地球上某一区域的地形地貌以 3D 图形的方式显示出来的软件。由于 3D 图形的绘制极其耗时，所以在带有高性能的 GPU 的 PC 上运行该软件，就可以非常流畅的缩放地图并快速得到高质量的 3D 地图场景。而同样的场景、同样的处理器，在没有 GPU 加

速时，速度几乎无法忍受。通过测试表明两种方式的速度落差高达几十倍[29]。

- Adobe PDF：PDF 是一种常用的文件格式，2007 年 Adobe 公司与 NVIDIA 公司合作，开发了利用 GPU 加速的 PDF 版本。当读取一个大小为 50MB，格式为 PDF 的文件时，使用传统方式，需要 8 秒钟的读取时间。相同条件下，如果使用 GPU 加速功能，读取时间不会超过 3 秒钟[29]。
- 分子动力学：2007 年南洋理工大学的 Weiguo Liu 等人利用 CUDA 平台在 G80 GPU 上实现了并行的 MD 分子动力模拟算法，获得了平均 15 倍的加速比[30]。UIUC 大学的理论和计算生物物理组开发的分子动力学软件 VMD/NAMD 在 GPU 上运行获得了 5~44 倍的加速比[15]。
- 数据库操作：数据库的查询等操作具有很高的并行性，2010 年弗吉尼亚大学的 Peter Bakkum 等人将一个小的 SQL 语句集在 NVIDIA Tesla C1060 上实现了，并获得了 20~70 倍的加速比[31]。

此外，我国在 CUDA 应用方面的事例也非常多。例如：在宝钢，使用 CPU-GPU 的异构计算技术使模拟冶金过程的时间从原来的一天缩短到现在的两分钟；在清华大学，使用 GPU 加速使三维图像重构的时间从过去的一个小时减少到几十秒钟甚至几秒钟，实现了三维图像的实时重构；在中国科学院基因研究所，使用 CPU 和 GPU 搭建的混合系统进行基因比对比传统的 CPU 集群系统快 30 倍，而功耗却降低了 5 倍，成本更是降低了 10 倍[32]。

除了上面列举的之外，类似的 CUDA 应用还有很多，这里不再一一列举。可见，CUDA 作为一种简单、高效、新型的并行计算方式必将会受到更多的关注和应用。

2.1.3 OpenCL

OpenCL(Open Computing Language)[13][14]是由苹果公司于 2008 年提出的一个适用于异构计算平台的统一的并行编程模型，也是一种工业标准和规范。一经提出就得到了众多大公司(如：Intel、Nvidia、AMD 等)的支持，发展非常迅速。

众所周知，目前的计算平台种类繁多，既有通用的多核 CPU，又有专用的拥有强大计算能力的众核 GPU、IBM Cell 和基于 ARM 架构的数字信号处理器（DSP）芯片。由于这些硬件平台的特性和架构差别很大，要想编写一个能在所有平台上高效运行的程序几乎是不可能的。因此，程序员不得不针对特定的硬件平台编写一个新的版本的程序，这无疑加重了他们的负担。OpenCL 的出现将会很好的解决这种因底层硬件平台的差异性所带来的软件可移植性的问题。今后，只要程序员编写的程序遵循 OpenCL 标准，就可以不加修改的在那些支持该标准

的硬件平台上高效运行。

OpenCL 不仅仅是一种语言，而是一个并行的编程框架，包括语言、API 和运行时系统，能够与 OpenGL、OpenGL ES 和其他的图形 API 进行高效的互操作。支持的应用非常广泛，从嵌入式应用、消费软件到高性能计算解决方案都能使用它进行开发。OpenCL 的设计初衷是为帮助程序员开发出可移植的、能够高效利用异构系统中所有的硬件资源的程序。基于 CPU-GPU 的异构模式只是众多异构结构中的一种，但也是目前被广泛使用的一种。图 2.2 形象的说明 OpenCL 是专为异构计算而服务的。

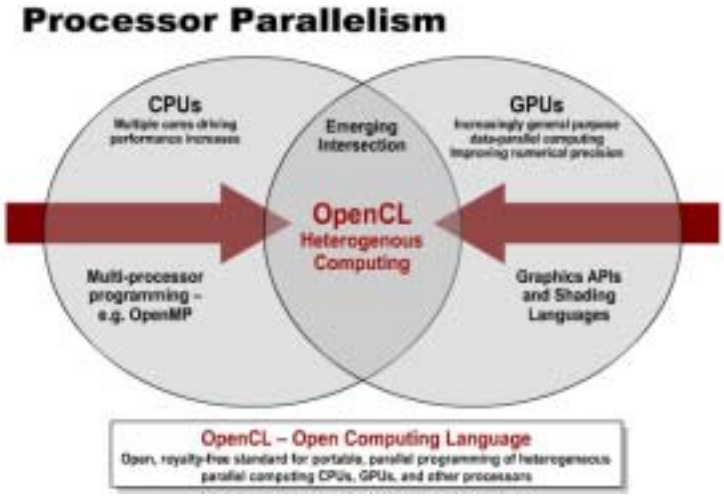

图 2.2 OpenCL 适用于异构平台[13]

那么，CUDA 与 OpenCL 究竟是什么关系呢？OpenCL 作为一个通用的标准规范，要支持各种不同的计算设备，而针对每一个设备，它都可以有一个不同的实现版本。CUDA 只是针对 NVIDIA GPU 的一个 OpenCL 的具体实现版本。类似，其他的厂商也可以为自己的设备开发出一个遵循 OpenCL 标准的编程模型。因此，CUDA 不仅不与 OpenCL 相冲突，还能促进 OpenCL 的发展和普及。

2.2 硬件体系结构

2.2.1 硬件设计

本节以支持 CUDA 的 GPU 为例，介绍其硬件结构。支持 CUDA 的 GPU 主要由一个流多处理器（Stream Multiprocessor，简称 SM）阵列组成，不同的设备所包含的 SM 数量也不同，且每个 SM 拥有各自的资源，相互独立，并行执行所有的计算线程。一个 SM 又包含 8 个标量流处理器（Scalar Stream Processor，简称 SP）、一个指令单元、一个 32 位的寄存器集合、共享存储器（Shared Memory）、

常量缓存（Constant Cache）、纹理缓存（Texture Cache）等硬件资源，如图 2.3 所示。SM 负责线程的创建、调度和执行，由于这些操作都由硬件完成，所以没有时间开销。每个 SP 有自己私有的一组 32 位寄存器，且线程访问寄存器的速度最快。片上的共享存储器由同一个 SM 上的所有 SP 共享，在没有访存冲突的情况下，访问共享存储器的速度与寄存器一样快。常量缓存和纹理缓存都是只读的，且被所有的 SP 共享[1]。

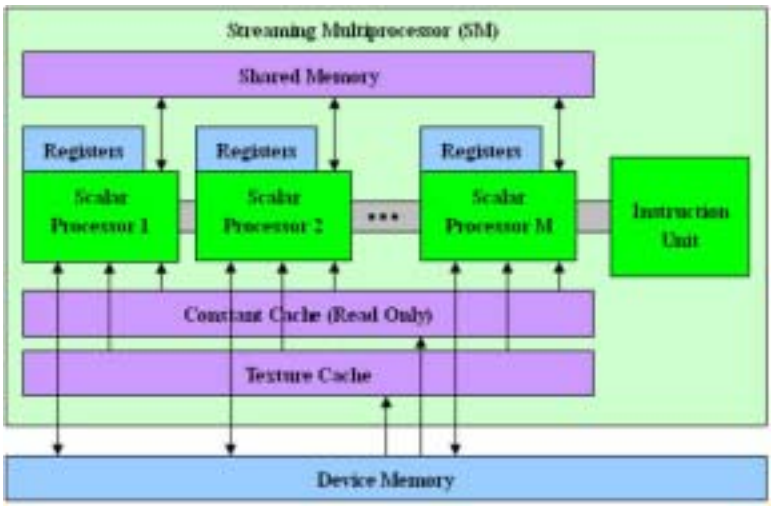

图 2.3 SM 的硬件结构[1]

在执行 CUDA 程序时，每个 SP 对应一个线程，每个 SM 对应一个线程块（Thread Block），但一个 SM 中同时可以有多个活动线程块（Active Block），这样当某个线程块中的线程因访存等操作而阻塞时，可以切换到其他线程块，从而使执行单元保持忙碌。每个 SM 最多能并发执行的线程块的数量取决于给定的内核中每个线程需要使用多少个寄存器以及每个线程块需要使用多少共享存储器。因为 SM 的寄存器和共享存储器是分配给一批线程块的所有线程（线程平均分配这些资源）。如果 SM 连处理一个线程块所需的共享存储器或寄存器都无法满足，内核将无法启动。

SM 采用 SIMT（Single Instruction Multiple Thread，单指令多线程）的执行模型。SIMT 与 SIMD（Single Instruction Multiple Data，单指令多数据）类似，多个线程同时执行相同的指令。当一个线程块被分配到一个 SM 上时，线程控制器将连续的 32 个线程分为一组，称为一个 Warp 块。Warp 块是线程调度的基本单位。每当指令单元要发出一条指令时，线程控制器就会选择一个准备就绪的 Warp 块，并将指令发送给该 Warp 块内的所有线程。如果一个 Warp 块的所有线程执行相同的路径，效率最高。如果一个 Warp 块的线程因执行条件语句而跳转到不同的分支（Divergence），则所有这些分支路径将被依次执行，即执行序列被

串行化。这样总的执行时间就是执行每条分支的时间之和。因此，在开发 CUDA 程序时要尽量避免 Warp 分支，否则程序的性能可能会急剧下降。为了减少出现 Warp 分支的情况，实际执行时，Warp 块中的 32 个线程被分成两组，每组 16 个线程，称为 Half-Warp[1][33]。

支持 CUDA 的 GPU 所拥有的 SM 数量差别很大。例如：GeForce 8400M GS 只有 2 个 SM，GeForce 9800 GT 有 14 个 SM，而 Tesla C1060 有 30 个 SM。SM 数量的多少与 GPU 的计算能力密切相关，这部分内容将在下一节进行介绍。

2.2.2 计算能力

GPU 的技术规范和特性取决于其计算能力（Compute Capability），而计算能力是由主要修订号和次要修订号来表示的。主要修订号相同的 GPU 具有相同的核心架构，次要修订号反映了对核心架构的增量式改进，如引进新的特性或功能。目前支持 CUDA 的 GPU 的计算能力为 1.x（如：1.0、1.1、1.2、1.3），其中 1 为主要修订号，x 为次要修订号，数字越大表示计算能力越高。任何低计算能力支持的特性，高计算能力都会支持，即计算能力向下兼容。GPU 的计算能力和内部硬件资源的相关信息可以使用运行时 API 提供的函数查询或参考相关的技术手册查询。表 2.1 列出了具有不同计算能力的 GPU 主要技术规范的差别[33]。

表 2.1 不同计算能力所支持的技术规范的差别

计算能力 / 主要特性	1.0	1.1	1.2	1.3
SM 数量	12~16	1~16	>24	>24
32 位原子操作	不支持	支持	支持	支持
64 位原子操作	不支持	不支持	支持	支持
SM 上寄存器数量	8192	8192	16384	16384
SM 上最多的活动 Warp 数量	24	24	32	32
SM 上最大的活动线程数量	768	768	1024	1024
双精度浮点计算	不支持	不支持	不支持	支持

除了表中所列举的参数之外，不同计算能力的 GPU 在主频、内存容量、内存带宽等方面也存在一定的差别。表 2.2 列出了具有代表性的 GPU 的相关参数。

表 2.2 典型 GPU 的相关参数[34]

设备型号	多处理器数量	计算能力
GeForce 8800 GTX	16	1.1
GeForce GTX 280	30	1.3
Tesla C1060	30	1.3
Tesla S1070	4×30	1.3
Quadro FX 3700M	16	1.1
Quadro FX 5800	30	1.3

常数 24 表示每个 SM 有 8 个 SP，每个 SP 在一个时钟周期内能够执行 3 条浮点运算指令。以 Tesla C1060 为例，它有 30 个 SM，时钟频率为 1.30GHz，所以它的理论峰值浮点速度为 30*1.3*24=936GFlop/s，几乎达到万亿次的计算能力。当前，只有计算能力为 1.3 的 GPU 才支持双精度浮点运算，其性能大约是单精度的 1/10。

2.3 软件模型

2.3.1 线程组织结构

前面讲过，CUDA 程序分为两部分：主机代码和设备代码。其中设备代码在 GPU 上运行，称为内核函数（Kernel Function）。通常，内核函数由成千上万个线程并发执行，那么 GPU 是如何管理这么多的线程呢？原来 GPU 采用了合理的层次结构来组织线程。如图 2.4 所示，首先若干个线程组成线程块（Block），其次若干个相同大小和维度的线程块组成线程网格（Grid），而每个内核函数就对应一个线程网格。虽然每个线程块最多只能包含 512 个线程，但一个线程网格却能包含 65535 个线程块，这样一个线程网格最多能有 512x65535=33553920 个线程，如此多的线程对于绝大多数程序来说绰绰有余。

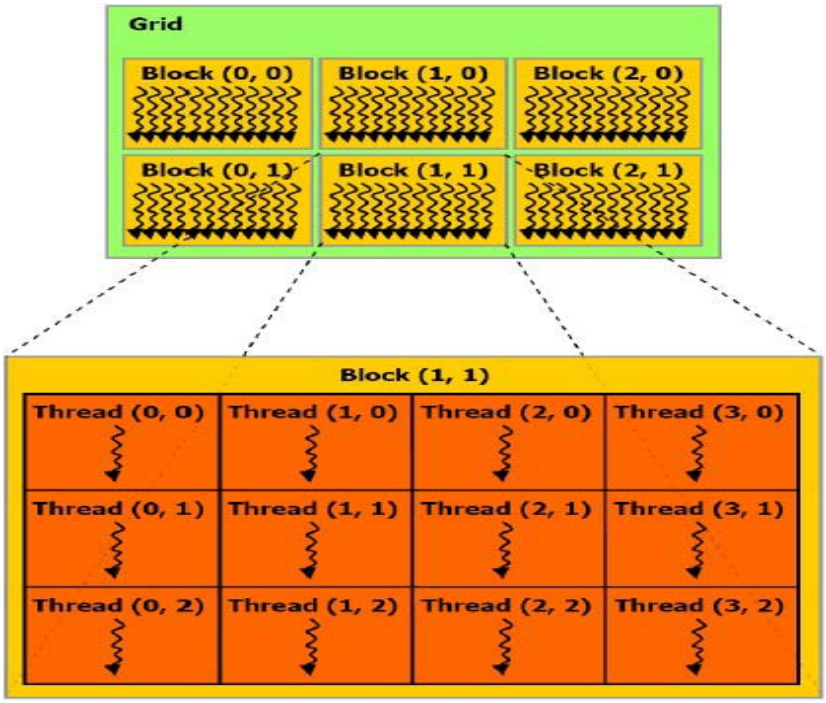

图 2.4 CUDA 线程层次结构[1]

为了方便编程，CUDA 定义了一个新的数据类型：dim3 （dim3 是用 uint3 实现的，相当于 3 个成员变量都是 unsigned int 类型的结构体）。同时还定义了四个内置变量：gridDim、blockDim、blockIdx、threadIdx，其中前两个变量是 dim3 类型，分别表示线程网格和线程块的维度；后两个是 uint3 类型，分别表示网格内线程块的索引和块内线程的索引。这样就可以使用一维、二维、三维的索引来标识线程，并构造一维、二维、三维的线程块。按照这种组织方式，程序员可以很方便的对向量、矩阵或空间中的数据进行直观、自然的划分。根据线程块的维度不同，线程索引与线程下标的对应关系如下[33]：

- 当线程块是一维时，threadIdx.x 就是线程的下标；
- 当线程块是二维且大小为（Dx，Dy）时，如果线程索引为（x，y），则对应的线程下标为（x+y*Dx）；
- 当线程块是三维且大小为（Dx，Dy，Dz）时，如果线程索引为（x，y，z），则对应的线程下标为（x+y*Dx+z*Dx*Dy）。

这种线程组织形式，除了方便管理的好处外，还可以定义某些操作。例如：同一个线程块内的线程既可以使用共享存储器进行通信，也可以调用函数__syncthreads()进行同步。这两种机制对于协调线程的执行是至关重要的。

在编程时，我们都会面临这样两个问题：每个线程网格应该设置多少个线程块以及每个线程块应该包含多少个线程？第一个问题由计算的数据规模和划分方式决定（上限是 65535），虽然没有线程块数量的下限规定，但为了充分利用计算资源，线程块数应大于流多处理器数。而第二个问题比较复杂，需要在线程

块的大小和每个线程的资源占用量之间寻找平衡，可以通过实验来寻找这个最佳的平衡点。但通常将线程块的大小设置为 256 是个不错的选择。

2.3.2 存储器层次结构

相对于 CPU 而言，GPU 的存储器层次结构更加复杂但也给程序员提供了更多的存储数据的方式。不同的存储器在容量和访问速度方面相差很大，这样合理的安排数据和访问数据的方式对提高程序的性能非常重要。图 2.5 清楚地展示了 GPU 的存储器层次结构。

每个线程有自己私有的寄存器和局部存储器（Local Memory）；每个线程块有一个共享存储器（Shared Memory）；Grid 中的所有线程都可以访问全局存储器（Global Memory）。此外，还有两个能够被所有线程以只读方式访问的存储器：常量存储器（Constant Memory）与纹理存储器（Texture Memory）。表 2.3 对这些存储器的位置、有无缓存、访问属性和生存期进行了比较。

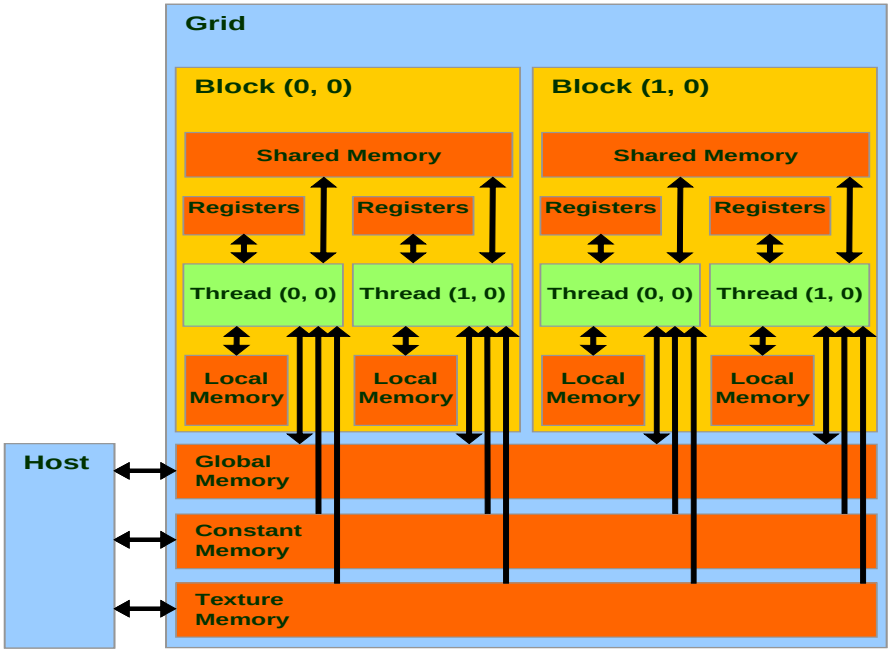

图 2.5 GPU 的存储器层次结构[1]

表 2.3 不同存储器比较

存储器	位置	有无缓存	访问属性	生存期
寄存器（Register）	芯片内	---	GPU 可读/写	与线程相同
共享存储器（Share Memory）	芯片内	---	GPU 可读/写	与线程块相同
常量存储器（Constant Memory）	芯片外	有	GPU 可读 CPU 可读/写	与程序相同
局部存储器（Local Memory）	芯片外	无	GPU 可读/写	与线程相同
纹理存储器（Texture Memory）	芯片外	有	GPU 可读 CPU 可读/写	与程序相同
全局存储器（Global Memory）	芯片外	无	GPU 可读/写 CPU 可读/写	与程序相同

注：寄存器和共享存储器都在芯片内，访问速度最快，没有缓存的必要。常量存储器和纹理存储器的缓存在芯片内。

2.3.3 编程模型

在 CUDA 编程模型[35]中，将 GPU 作为 CPU 的协处理器（Co-processor），当然在一个系统中可以有多个协处理器。在这样一个异构系统中，CPU 和 GPU 协同工作。CPU 负责执行逻辑控制和串行任务，而 GPU 则负责执行高度并行化的任务。对于一个具体的问题，通过分析其数据依赖关系，识别出串行的部分和并行的部分，串行的部分在 CPU 上求解，然后利用 GPU 来计算并行的部分。在 GPU 上运行的函数称为内核函数（Kernel Function），其内部的代码被大量的线程并行的执行。一个完整的 CUDA 程序是由若干个运行在 GPU 上的内核函数和一些在 CPU 上运行的串行代码段组成，如图 2.6 所示。

需要 CPU 完成的串行代码（Serial Code）包括：准备数据，分配设备内存并对设备进行初始化，启动内核函数的执行，将设备内存中的计算结果拷贝到主机内存等。如果程序中有多个内核函数，这一过程可能会重复数次。

如果系统中只有一个 GPU，则同一时间只有一个内核函数在运行；但如果系统中有多个 GPU，则同一时间可以有多个内核函数同时运行（每个 GPU 上运行一个内核函数）。而且内核函数的执行和主机代码的执行是异步的。

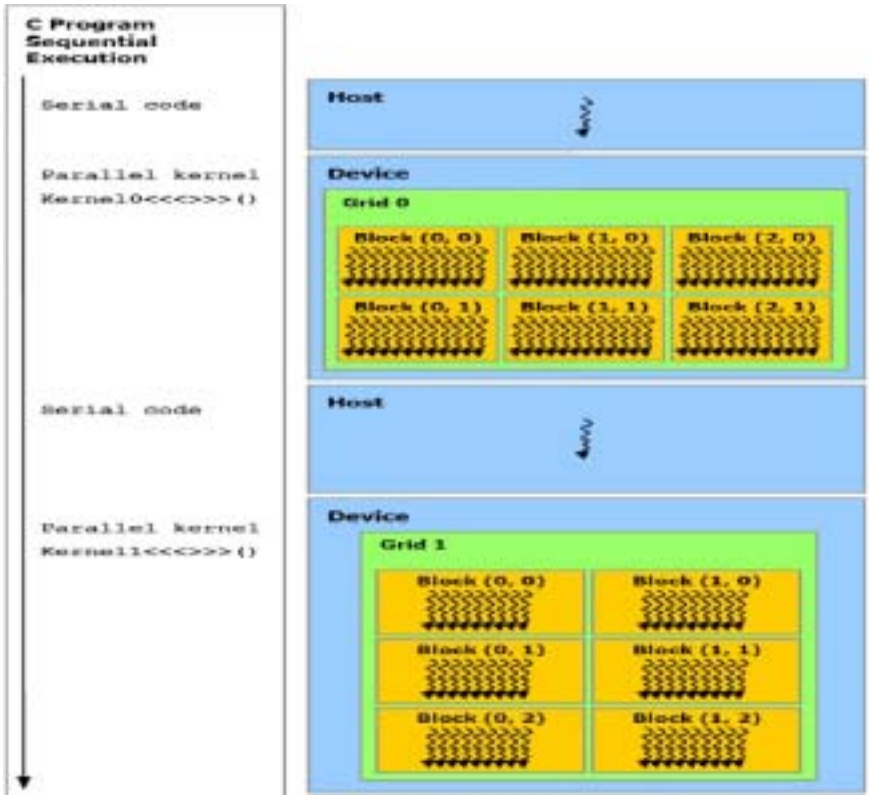

图 2.6 CUDA 编程模型[1]

2.4 编程规范

2.4.1 C 语言扩展

CUDA 编程模型允许程序员使用类 C 语言直接编写在 GPU 上运行的程序。这种类 C 语言是以标准 C 语言为基础，并对其进行了简单的扩展，这些扩展主要包括：引入了函数类型限定符、变量类型限定符、内置矢量类型、内建变量、<<< >>>运算符、运行时 API 以及驱动 API 等方面。图 2.7 反映了 CUDA 类 C 语言的结构。

可见，CUDA 类 C 语言与标准 C 语言非常相似，而 C 语言又是被使用最多的语言之一，这样任何有 C 语言基础的人很容易就能掌握 CUDA 编程技术，这也是 GPGPU 技术得以流行的重要原因。接下来，我们将对这些扩展分别予以介绍。

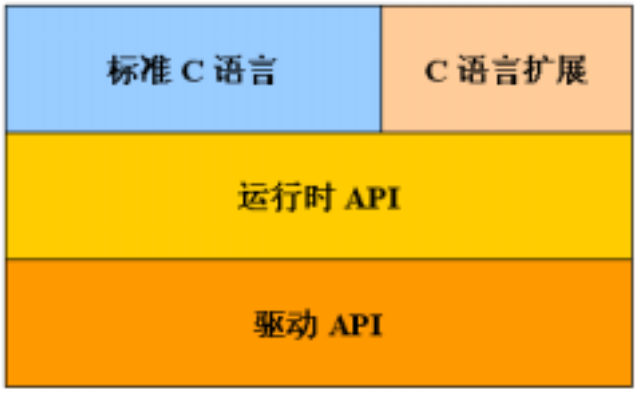

图 2.7 CUDA 类 C 语言结构[1]

2.4.2 函数与变量类型限定符

CUDA 类 C 语言引进了函数类型限定符与变量类型限定符。函数类型限定符用来指定函数是在主机（CPU）还是在设备（GPU）上执行，以及该函数是从主机调用还是从设备调用。函数类型限定符有：__device__、__global__以及__host__。表 2.4 列出了这些限定符的特性。

表 2.4 函数类型限定符的特性

函数类型限定符	在何处执行	从何处调用	特性
__device__	设备	设备	无法获取函数的地址
__global__	设备	主机	返回类型必须为 void
__host__	主机	主机	与不使用任何限定符一样

此外，规范对不同的限定符所修饰的函数有不同的限制。例如：__device__和__global__修饰的函数内不能定义静态变量，不支持递归以及函数的参数个数不允许是变量等。__global__修饰的函数称为内核函数，对其调用是异步的，即主机调用该函数之后立即返回，而不必等待设备执行结束。如果主机调用内核函数之后的操作需要用到设备计算的结果，则可以调用同步函数cudaThreadSynchronize()让主机阻塞以等待设备执行结束。对函数仅使用__host__限定符，与不使用任何限定符效果一样，但对函数同时使用__host__和__device__时，该函数既可以从主机调用也可以从设备调用。

变量类型限定符用来指定变量存储在哪一类存储器上。在 CPU 上执行的程序，通常编译器能够自动决定变量的存储位置（寄存器或内存），但 GPU 上的存储器种类繁多，为了区别不同的存储器，CUDA 类 C 语言引入了 3 个变量类型

限定符：__device__、__constant__以及__shared__。表 2.5 列出了这些限定符的相关特性。

表 2.5 变量类型限定符相关特性

变量类型限定符	变量的存储位置	有权访问的线程	主机访问方式
__device__	全局存储器	线程网格内的所有线程	通过运行时 API 访问
__constant__	常量存储器	线程网格内的所有线程	通过运行时 API 访问
__shared__	共享存储器	线程块内的所有线程	不能从主机访问

此外，规范对这些限定符还有一些限制。例如：__shared__修饰的变量不能在声明时初始化；不可从设备指派__constant__变量，仅可通过运行时 API 从主机指派；__device__和__constant__修饰的变量只能在文件作用域内定义；这 3 个限定符都不能修饰在主机上执行的函数内的局部变量；可以直接获取这 3 个限定符修饰的变量的地址，但只能在设备中使用；通过调用运行时 API 函数 cudaGetSymbolAddress()可以获得__device__和__constant__修饰的变量的地址，但只能在主机中使用。

2.4.3 内置类型及内建变量

CUDA 类 C 语言引入了一些内置向量类型，如：int4、char3、ushort4、float4、double2、dim3 等。这些向量类型都继承自基本的整形和浮点型，均为结构体。末尾的数字表示其分量个数，最多可以有 4 个。第 1、2、3、4 个分量可分别通过 x、y、z、w 字段访问。dim3 与 uint 类似，区别是定义 dim3 类型的变量时，未指定的分量会被默认初始化为 1。此外，还可以通过形式为 make_<type name> 的函数构造向量类型。例如：int3 make_int3(int x，int y，int z)，将会创建一个类型为 int3，值为（x，y，z）的向量。

为了方便程序员获得线程网格和线程块的维度以及线程索引等信息，CUDA 类 C 语言定义了一些内建变量，如表 2.6 所示。

表 2.6 内建变量及其含义

内建变量	类型	含义
gridDim	dim3	线程网格的维度
blockDim	dim3	线程块的维度
blockIdx	uint3	线程网格内线程块的索引
threadIdx	uint3	线程块内线程的索引
warpSize	int	一个 warp 块内包含的线程数目

2.4.4 内核函数和执行配置

使用__global__修饰且在 GPU 上执行的函数称为内核函数。内核函数必须从主机端调用，且调用的方式为 KernelFunction<<<Execution Configuration>>>(Augment List)。其中<<< >>>是 CUDA 类 C 语言引入的运算符，用于为内核函数指定执行配置，如设置线程网格与线程块的维度、共享内存使用量等。通常可这样设置执行配置：<<<GridDim，BlockDim，SharedMemSize>>>，其中最后的 SharedMemSize 用于指定动态分配的共享存储器的大小，如果程序没有进行这一操作，则可以省略此项，缺省值为 0。参数列表（Augment List）中的参数将会通过共享存储器传递给内核函数。

下面给出一个内核函数定义和调用的例子程序。

```
__global__ void KernelFunc(float* d_a, float* d_b, float* d_c)
{
    int tid = threadIdx.x;
    d_c[i] = d_a[i] + d_b[i];
}
int main()
{
    //准备数据，分配设备内存和初始化
    dim3 dimGrid(1, 1, 1);
    dim3 dimBlock(100, 1, 1);
    KernelFunc<<< dimGrid, dimBlock>>>(d_a, d_b, d_c); //  调用内核
}
```

如果执行配置<<< >>>中的选项设置不当，如 dimGrid 或 dimBlock 的设置违反了 GPU 计算能力所约束的技术规范，或者动态分配的共享存储器过大，将会使内核调用失败，即内核不执行而立即返回。

2.4.5 运行时 API 和驱动 API

CUDA 运行时 API 和驱动 API 提供了对设备管理（Device Management）、线程管理（Thread Management）、流管理（Stream Management）、事件管理（Event Management）、存储器管理（Memory Management）、纹理索引管理（Texture Reference Management）、执行控制（Execution Control）、与 OpenGL 互操作（OpenGL Interoperability）、与 Direct3D 互操作（Direct3D Interoperability）、异常处理（Error Handling）的应用程序接口[33]。

虽然两种 API 提供的功能相同，但所处的层次不同。CUDA 运行时 API 是上层 API，是对底层 CUDA 驱动 API 的封装。由于 CUDA 运行时 API 隐藏了一些底层硬件的细节，所以使用更加简单、方便。因此，CUDA 初学者基本上都是使用运行时 API 编程。

CUDA 驱动 API 是一种基于句柄、命令式的 API（大多数对象通过句柄引用），可以加载汇编形式（PTX 代码）或二进制形式（CUBIN 代码）的内核函数模块。虽然使用 CUDA 驱动 API 编程比较复杂，但能够直接控制硬件的执行，从而实现一些非常复杂的功能以及获得更好的性能。

2.5 本章小结

CUDA 是 NVIDIA 公司的 GPGPU 编程模型，旨在帮助软件开发人员使用类 C 语言直接编写在 GPU 上运行的程序。本章从 CUDA 基础知识、软/硬件架构、编程模型和编程规范等方面对 CUDA 计算平台进行了详细的介绍，这些内容主要参考了 CUDA 编程手册 2.0（CUDA Programming Guide 2.0）和 CUDA ZONE 中文站（http://cuda.csdn.net/）。最后，需要强调的是，掌握语言的细节固然重要，但要编写出高质量的程序，还必须对底层的硬件结构有所了解。

第3章 CPU-GPU 异构计算平台的性能优化

3.1 影响程序性能的关键因素

虽然 CUDA 编程模型极大的简化了利用 GPU 进行通用计算的难度，但相对于只含 CPU 的同构系统而言，在基于 CPU-GPU 的异构系统上进行编程要复杂的多，程序的性能优化也更加困难。通常影响 CUDA 程序的性能因素主要包括三个方面：访存延迟、负载均衡以及全局同步开销。虽然这些因素几乎是每个并行程序都会遇到的，但在不同的计算平台上它们产生的原因以及对应的优化方法却不尽相同。本章将首先分析产生这些性能因素的原因，然后针对每个因素提出相应的优化方法并通过实验检验优化的效果。

3.1.1 访存延迟

一直以来，存储器的带宽就是影响计算机性能的主要瓶颈之一，被称为“存储器墙”。无论计算设备是 CPU 还是 GPU，处理器的计算能力都远远高于存储器的访问带宽。例如：C1060 的单精度浮点性能达到 933GFlops，而显存的访问带宽只有 102GB/s，相差 9 倍之多。为了缓解处理器的计算速度和存储器带宽之间的矛盾，几乎所有的系统都使用了高速缓存技术，如 CPU 上的多级 Cache 以及 GPU 上的共享存储器、常数高速缓存和纹理高速缓存等。

在 2.3.2 节中讲过 GPU 的存储器层次结构非常复杂，共有 6 种不同的存储器。与主机端的存储系统类似，不同的存储器在容量、访问速度等方面存在很大差异。例如：片外全局存储器的容量一般都有 1GB 以上，对其访问一次大约需要 400~600 个时钟周期。而片上共享存储器的容量只有 16KB，但访问一次的时间大约只需 4 个时钟周期。此外，即使是同一个存储器，采用不同的访问方式所获得的有效带宽也差别很大。例如：以联合访问方式访问全局存储器所获得的有效带宽比其他方式要高 2~10 倍；在没有存储体冲突（Bank Confict）的情况下访问共享存储器的速度和访问寄存器一样快，而出现存储体冲突时，访问速度要慢数倍。所有的存储器都暴露给用户，即由用户决定数据存储在哪个存储器。这种做法有利有弊：一方面，给程序员提供了更多的灵活性，即程序员可以完全按照自己的意愿来存储数据，而不受编译器的干扰；另一方面，对于 CUDA 初学者或经验尚不丰富的开发者，很容易因为采用了不合理的访存方式或存储器而造成程序的性能大幅下降。因此，选择合适的访存方式以及充分利用片上的高度存储器

能有效地减少访存延迟，进而提高整个程序的性能。

3.1.2 负载分配

任何一个并行程序都存在负载分配的问题。对同构的并行系统而言，需要进行任务划分并将划分后的子任务分配给各个线程或进程处理。理想情况下，应该使每个线程或进程的工作量相同，即负载均衡。这样可以使每个线程的运行时间大致相同，整个程序的执行时间也最短。

这一问题在基于 CPU-GPU 的异构并行系统上更加复杂，需要进行两次任务划分。首先，需要将计算任务分成两部分，一部分交给 CPU 完成，另一部分交给 GPU 完成。其次，由于在 GPU 上执行的内核函数通常由成千上万个线程执行，所以交由 GPU 完成的那部分任务需要进一步划分以分配给这些线程去完成。

负载分配是否均衡关系到计算资源是否充分利用，当然也关系到能够在最短的时间内完成给定的计算任务。

3.1.3 全局同步开销

CUDA 运行时 API 提供了函数 CudaThreadSynchronize()来实现主机和设备的同步，以及函数__syncthreads()来同步属于同一个线程块中的线程，该同步操作由专门的硬件完成，所以基本没有开销。但是却没有提供实现不同线程块同步的函数，即执行内核函数的所有线程的全局同步。

目前，实现不同线程块之间的同步采用的方法是重新启动内核函数，即如果不同线程块处理的任务之间存在数据依赖关系，就需要把原来的内核分解成两个或多个（也可以不分解内核函数但需要重新启动一次），这样做是为了保证当前内核函数中所有的访存事务都完成了并对所有的线程块中的线程可见。

虽然启动一次内核函数的时间开销不大（大约为几个到几十个 ms），但是每次重新启动内核函数时，都需要将当前保存在共享存储器或寄存器中的中间结果保存到全局存储器以供新启动的内核函数使用（因为这些片上存储器的生命周期同内核函数相同），而新启动的内核函数又需要将这些中间结果从全局存储器拷贝到片上存储器，这样就增加了多次访问全局存储器的开销和程序的复杂度。

3.2 访存优化策略

3.2.1 优化存储器之间的数据传输

目前，绝大多数显卡都是通过 PCI Express（PCI-E）总线与主机端相连接。虽然 PCI-E 总线带宽相对于早期的 PCI 以及 AGP 有很大提高，但和显存以及内存带宽相比就逊色的多。例如：PCI-E 1.0 X16 总线的理论带宽为 4GB/s，而 GTX 280 的理论显存带宽达到 141GB/s。因此，虽然 GPU 的显存带宽和 CPU 的内存带宽都很高，但由于使用 PCI-E 总线来连接 GPU 与 CPU，所以主机和设备之间的数据传输率主要受限 PCI-E 总线的带宽。

由于内核函数不能直接访问主机内存，而只能访问显存，所以在启动内核函数之前需要将数据通过 PCI-E 总线从主机内存传输到设备内存。如果在 CUDA 程序中频繁的出现主机和设备之间的数据传输就会增加整个程序的执行时间。

CUDA 运行时提供了多种用来在主机与设备之间传输数据的 API 函数（cudaMemcpy 系列），并根据数据传输方向的不同分为四种：主机到主机（H2H）、主机到设备（H2D）、设备到主机（D2H）以及设备到设备（D2D）。其中只有主机和设备之间的数据传输需要通过 PCI-E 总线。为了测试不同传输方向的有效带宽，我们在 Linux 环境下，使用 Tesla C1060 以不同大小的数据集进行数据传输的实验。主机内存为 15G，显存大小为 4GB，GDDR3 类型，GPU 与主机端的连接总线为 PCI-E 2.0 X16，其理论带宽为 8GB/s，实验结果如表 3.1 所示。

表 3.1 不同数据传输方向的有效带宽（单位：MB/s）

数据大小	主机到设备	设备到主机	设备到设备	主机到主机
32M	3932.170	3472.168	36570.894	3385.741
64M	3974.139	3155.591	36788.693	3719.057
96M	3968.974	3521.172	36868.763	3390.397
128M	3894.758	3172.822	36877.915	2976.271
160M	3919.807	3173.288	36937.568	3389.050
192M	3636.248	3178.562	36921.943	3390.067
224M	3635.830	3197.013	36958.900	3292.375
256M	3639.137	3538.737	36949.729	3714.228

可见，主机与设备之间进行数据传输时，实际获得的带宽（大约 3.4GB/s）距离 PCI-E 总线的理论带宽（8GB/s）还是有很大的差距，这主要是受主机内存带宽的限制。此外，设备与设备之间数据传输的带宽则很高，达到 36GB/s，这是因为 GPU 普遍采用高速的 GDDR3、GDDR4 以及 GDDR5 显存，其位宽高达 512 位，而普通 CPU 的内存位宽仅为 64 位。

另外，在主机端分配的内存有两种类型：可分页的内存（Pageable Memory）和不可分页的内存（Page-locked Memory）。二者的区别如下：

- 使用不同的函数来分配。可分页的内存使用主机端的 Malloc()函数分配，不可分页的内存需要调用 CUDA 运行时提供的 cudaMallocHost()函数进行分配。
- 二者各有利弊。可分页的内存可以被操作系统控制（回收、再分配），因此不会对主机系统的性能产生不利影响。但是在主机与设备进行数据传输之前，需要将该可分页内存块的数据复制到一块专门的内存空间，由于在主机端多了一次数据的拷贝，所以增加了传输时间。相反，不可分页的内存不能够被操作系统控制，如果这种类型的内存分配过大，很可能会造成主机端的可用内存不足，而使主机系统的性能下降。但是不可分页的内存不需要多余的拷贝，所以总的数据传输时间较少。

为了说明这两种不同类型的内存对数据传输性能的影响，我们在相同的平台上做了类似的实验。如表 3.2 所示，使用不可分页的内存与设备进行数据传输所获得的带宽明显高于可分页的内存，大约高出 46%。

表 3.2 可分页内存与不可分页内存对数据传输带宽的影响

数据大小	可分页的主机内存		不可分页的主机内存	
	主机到设备	设备到主机	主机到设备	设备到主机
32M	3932.170	3472.168	5746.252	5537.780
64M	3974.139	3155.591	5755.133	5549.394
96M	3968.974	3521.172	5755.982	5412.616
128M	3894.758	3172.822	5758.235	5414.171
160M	3919.807	3173.288	5758.462	5412.582
192M	3636.248	3178.562	5759.022	5556.369
224M	3635.830	3197.013	5759.236	5557.577
256M	3639.137	3538.737	5759.840	5417.761

通过上面的实验和分析可知，设备与设备之间进行数据传输所获得的带宽远远高于主机与设备之间的数据传输带宽。此外，如果在主机端分配的内存不大或主机内存很大，那么使用不可分页的内存与设备进行数据传输也能提高程序的性能，尤其是对那些主机与设备频繁进行数据传输的程序。

3.2.2 利用全局存储器的联合访问

全局存储器没有 Cache，对其访问一次大约需要 400~600 个时钟周期，很可能成为程序的性能瓶颈[1][36]。对全局存储器的访问是否满足联合访问的条件是影响 CUDA 程序性能的重要因素之一。对于计算能力为 1.0 或 1.1 的设备，是否满足联合访问的条件往往会使 CUDA 程序的性能相差一个数量级。那么实现联合访问需要满足哪些条件呢？不同计算能力的设备有不同的要求，计算能力为 1.2 以上的设备达到这一条件的要求要宽松一些。对于计算能力为 1.0/1.1 的设备需要同时满足以下三个条件才能实现联合访问[1]：

- 半个 warp 中的每个线程访问全局存储器的数据大小必须为 4B、8B 或 16B；
- 被访问的显存块的起始地址必须是该显存块大小的倍数；
- 每个线程必须按顺序访问，即半个 warp 中的第 k 个线程访问第 k 个数据，但并不要求每个线程都实际参与。

针对以上所列的条件，下面给出两种满足联合访问的访问模式，如图 3.1 所示。上下两幅图中每个线程都访问 4B 字节数据；被访问的内存块的起始地址为 128，该内存块的大小为 64B，满足对齐要求；每个线程也都是按顺序访问的。区别仅在于上图中每个线程都进行了访存，而下图中的线程 1 没有访存。但都满足联合访问的条件。

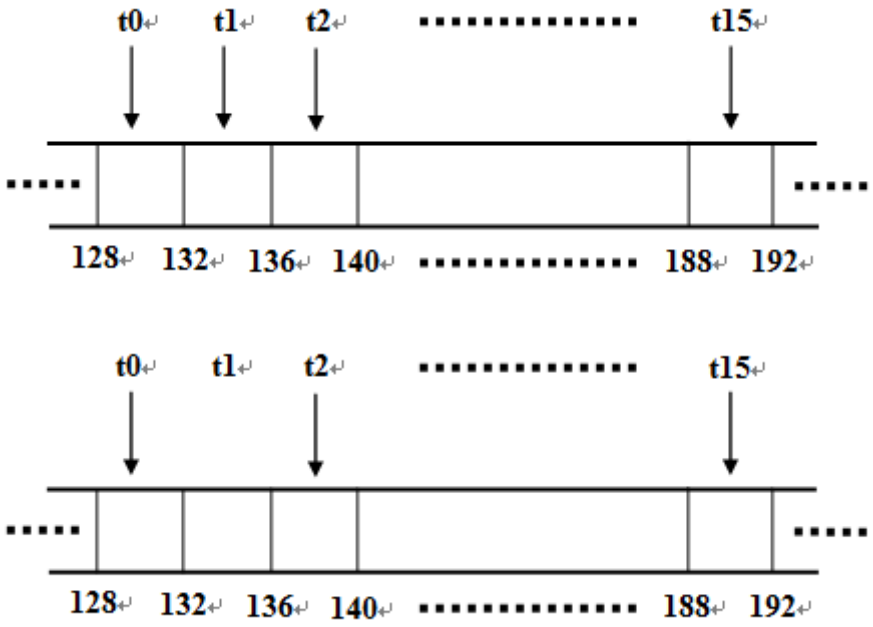

图 3.1 两种常见的联合访问模式

图 3.2 展示了两种常见的非联合访问模式。其中上图中线程 1 和线程 2 没有按顺序访问（交叉访问），违反了条件 3；下图中被访问的内存块的起始地址为 132，不是内存块大小 64 的倍数，违反了条件 2。

如果对全局存储器的访问满足联合访问的条件，则只需执行一次访存操作就可以处理半个 warp 中所有线程的访存请求。否则，需要单独执行每个线程的访存请求，会产生 16 次串行传输。

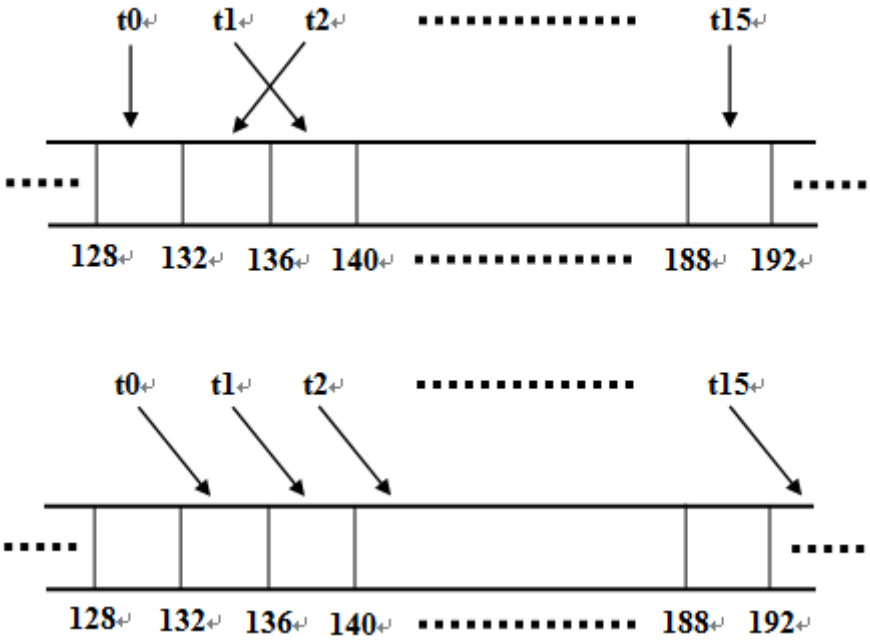

图 3.2 两种常见的非联合访问模式

为了测试这两种访存模式对程序性能的影响，我们在计算能力为 1.1 的 GeForce 9800 GT 上进行试验。测试程序中的内核函数非常简单，每个线程从全局存储器读取一个数，给它加上一个常数值，然后写回。为了排除其他因素的影响，在实验中并没有使用共享存储器。试验结果如表 3.3 所示。

表 3.3 联合访问与非联合访问的性能比较（单位：us）

测试数据大小	联合访问时内核执行时间	非联合访问时内核执行时间	两种方式的时间比值
12MB	613.1	5949.3	9.7
24MB	1177.9	11961.6	10.2
48MB	2299.2	23952.5	10.4

可见，采用联合访问模式访问全局存储器，内核函数的执行时间比非联合访问模式快 10 倍左右。因此，在编写 CUDA 程序时应根据所用设备的计算能力尽

量满足联合访问的条件，实现对全局存储器的联合访问，通常这会使程序的性能有明显的提升。

3.2.3 使用高效的共享存储器

GPU 中的共享存储器相当于 CPU 的 Cache，位于芯片上，访问速度比局部/全局存储器快很多。在没有存储体冲突（bank conflict）的情况下，访问共享存储器的延时几乎只有局部或全局存储器的 1/100，速度和寄存器相当。但是和 CPU 的 Cache 一样，共享存储器容量很小，只有 16KB，通常无法存储全部的输入数据。与 Cache 不同的是，共享存储器完全由程序员手动管理，即由程序员决定是否使用共享存储器，将哪些数据放在共享存储器以及对其采用何种访问方式。不同的策略和用法可能会对程序的性能产生重大影响。

为了提高存储体的访问带宽，共享存储器被划分成大小相等、可以被同时访问的存储块，称为 bank。因此，一个具有 n 个 bank 的存储体所能提供的最大带宽是只有一个 bank 的存储体的 n 倍。但是如果半个 warp 中的多个线程同时访问共享存储器的某一个 bank 就会出现访存冲突，称为 bank conflict。此时，这些访存请求会被串行的完成，有效带宽也会大幅降低。但是如果半个 warp 中所有的线程都访问同一个 bank，则系统将被访问的数据广播给所有的线程，反而使访存速度加快[33]。

共享存储器共有 16 个 bank，且 bank 的划分方式为：以 4B 为单位连续、循环进行划分，即相邻的大小为 4B 的存储空间作为相邻的 bank，第 17 个 4B 的存储空间又被划为第 1 个 bank。如图 3.3 所示。

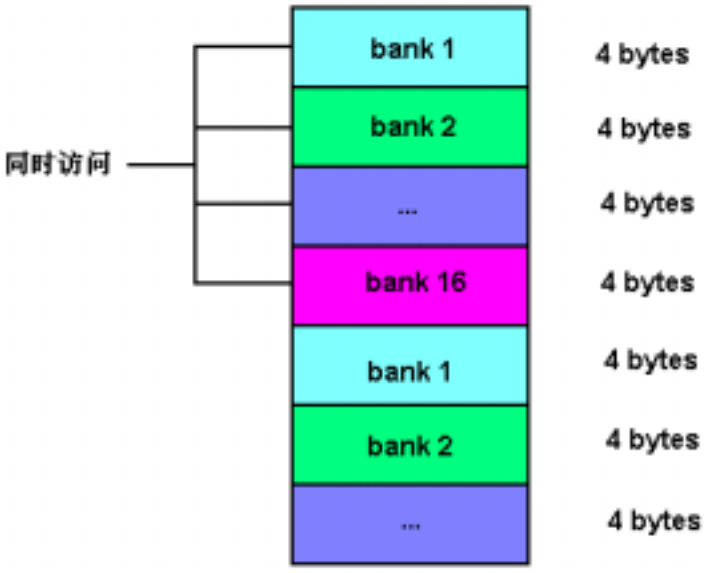

图 3.3 共享存储器的 bank 组织方式

假设有一个共享数组__share__ float s_array[512]，所以元素 s_array[k]就属于 bank k%16。如果每个线程按照这样的方式：float data=s_array[tid]读取数据，则不会产生 bank 冲突，因为半个 warp 中的 16 个线程访问的是不同的 bank。但是如果换成这样：float data=s_array[4*tid]，那么下标为 0、4、8、12 的线程就会都访问第一个 bank，从而产生 4 路 bank 冲突，理论上会带来 4 倍的访存开销。

为了测试访问共享存储器时出现 bank 冲突对程序性能的影响，我们在 GeForce 9800 GT 上进行试验。程序中的内核函数用来计算一组数的平方和。其中一个内核函数没有 bank 冲突，另一个内核函数会出现 16 路 bank 冲突。代码如下：

```
//没有 bank 冲突的内核函数
__global__ void Sum0fQuareKernel
(float *d_data, float *d_result){
    __shared__ float sdata[BLOCK_SIZE];
    int tid=threadIdx.x;
    int id=blockIdx.x*blockDim.x+tid;
    sdata[tid]=d_data[id]*d_data[id];
    __syncthreads();
    if(tid<16){
        for(int i=0;i<blockDim.x/16;i++)
            sdata[tid]+=sdata[16*i+tid];
    }
    __syncthreads();
    if(tid==0){
        for(int i=1;i<16;i++)
            sdata[0]+=sdata[i];
        d_result[blockIdx.x]=sdata[0];
    }
}
```

```
//存在 16 路 bank 冲突的内核函数
__global__ void Sum0fQuareKernel
(float *d_data, float *d_result){
    __shared__ float sdata[BLOCK_SIZE];
    int tid=threadIdx.x;
    int id=blockIdx.x*blockDim.x+tid;
    sdata[tid]=d_data[id]*d_data[id];
    __syncthreads();
    if(tid<16){
        for(int i=0;i<blockDim.x/16;i++)
            sdata[blockDim.x/16*tid]+=
            sdata[blockDim.x/16*tid+i];
    }
    __syncthreads();
    if(tid==0){
        for(int i=1;i<16;i++)
            sdata[0]+=sdata[blockDim.x/16*i];
        d_result[blockIdx.x]=sdata[0];
    }
}
```

实验结果如表 3.4 所示。

表 3.4 访问共享存储器时有无 bank 冲突的性能比较（单位：us）

测试数据大小	没有 bank 冲突时内核执行时间	有 16 路 bank 冲突时内核执行时间	时间比值
12MB	1028.5	2998.4	2.9
24MB	2002.9	5972.3	3.0
48MB	3949.4	11898.2	3.0

可见，存在 16 路 bank 冲突的内核函数的执行时间大约是没有 bank 冲突的内核函数执行时间的 3 倍。因此，在 CUDA 程序中要充分的利用共享存储器，以减少对全局存储器的访问次数，同时在使用共享存储器时还要尽量避免出现 bank 冲突的情况，以减少内核函数的执行时间，提高程序的性能。

3.3 代码优化策略

3.3.1 指令性能和延迟

流多处理器处理一条 warp 指令的过程为：首先，为 warp 中的每个线程读取操作数；其次，执行指令；最后，为每个线程写入计算结果。因此有效地指令吞吐量取决于：名义的指令吞吐量、访存延迟以及访存带宽。关于访存优化和访存带宽的优化已经在前面讨论过了，本节主要讨论指令级优化以便提高指令的吞吐量。

■ 算术指令的优化

GPU 的硬件非常适合进行单精度浮点计算，所以在 CUDA 程序中应尽量将数据的类型定义为单精度浮点型。此外，虽然双精度浮点类型有更高的精度，但只有计算能力 1.3 及以上的设备才支持，而且其执行时间大约是单精度的 10 倍，因此除非必要，否则应尽量避免使用双精度浮点类型。

CUDA 数学函数库包含了绝大多数常用的数学运算函数，而且明确给出了对某种类型的操作数进行某个算术运算所需要的时钟周期数，例如单精度浮点乘法需要占用 4 个时钟周期，整数乘法却需要 16 个时钟周期。这样我们就可以在 CUDA 程序中尽量选择比较高效的指令或者用实现相同功能的高效指令替换低效的指令。例如：整数除法和模运算非常耗时，应该尽量避免使用或用位运算代替：如果 n 是 2 的幂次方，则 i/n 和 i>>log2n 等价，i%n 和 i&(n-1)等价，而位运算和逻辑运算只需 4 个时钟周期。

此外，对某些数学函数 CUDA 提供了一个计算精度稍低一些，但更加快速的版本，这些函数都以“__”开头，例如 sinf()和__sin()，exp()和__exp()等。因此在不影响计算结果精度的情况下，应尽量使用这些快速版本的函数。如果在编译 CUDA 程序时加上-use_fast_math 选项，则程序中使用的所以数学函数都会被替换成与之对应的快速版本[1][35]。

■ 访存指令的优化

访存指令包含读写任何存储器的指令。虽然指令本身只需 4 个时钟周期，但是访问存储器的延迟可能非常大，例如：读写全局存储器还有 400~600 个时钟周期的延迟。对于访存指令的优化策略，除了 3.2 节所讲的之外，提高程序中计算与访存操作的比例以及计算与访存重叠也是隐藏访存延迟的有效措施。这些将在 3.4 节进行讨论。

3.3.2 避免控制流分支

控制流指令，如：if、switch、for、while、do 等，可能会使同一个 warp 内的线程跳转到不同的分支，而一旦 warp 内的线程发生分支，这些不同的执行路径就会被串行的执行，只有等到所有的分支路径都执行结束后，所有的线程才会重新回到同一条执行路径上。因此，warp 分支会严重影响程序的性能。

当分支条件是线程下标的函数时，通过仔细的设置分支条件可以有效地避免发生 warp 分支的情况。例如，当分支条件为：if (threadIdx.x>4) 时就会出现两路 warp 分支，即下标小于 4 的线程走一条路径，下标大于 4 的线程走另外一条路径。但是如果把上面的分支条件改为：if (threadIdx.x/WARP_SIZE>4)，则不会出现 warp 分支[33]。

此外，我们也可以使用制导语句#pragma unroll 来要求编译器对特定的循环进行展开，从而避免出现 warp 分支。

3.4 高级优化策略

3.4.1 计算与通信重叠

前面讲过，主机和设备是通过 PCI-E 总线进行通信的，由于 PCI-E 总线的带宽和显存带宽以及 GPU 的计算速度相比还是有很大差距。因此，如果在 CUDA 程序中主机与设备通信频繁就会使程序的性能下降。对于给定的计算平台，虽然我们无法提高 PCI-E 总线的带宽（除非更换带宽更高的 PCI-E 总线），但是可以使用计算与通信重叠的方法来掩盖通信的开销。而且 CUDA 运行时 API 不仅提供主机与设备进行异步通信的函数，如：cudaMemcpyAsync()，还规定内核的启动是异步的，即主机在启动内核函数后并不等待其执行结束，而是立即返回。这样在 CUDA 程序中就可以很方便的实现计算与通信重叠。具体来说，有以下两种方式：

- 主机端使用异步的通信函数将数据传输到设备内存，然后 CPU 立即返回并执行其他计算任务；

- 当 CUDA 程序含有多个内核函数或需要多次启动同一个内核函数时，主机端可以在启动内核函数后立即返回并与设备进行通信以便为下一次内核函数的运行准备好数据。

3.4.2 主机和设备并行计算

由于内核函数的启动是异步的，如果主机端在启动内核函数后的计算需要用到内核函数的计算结果，则必须阻塞以等待内核函数执行结束。其代码片段如下：

```
Void InvokeKernel(argument list) {
    ``````````````
    KernelFunc<<<GridDim, BlockDim>>>(argument list); //启动内核函数
    cudaThreadSynchronize(); //同步，即主机阻塞等待设备执行结束
    cudaMemcpy(h_mem, d_mem, mem_size, cudaMemcpyDeviceToHost); //将内核函数的计算结果拷贝到主机内存
    ComputeOnHost(h_mem); //主机端对内核函数的结果进行处理
    ``````````````
}
```

主机在调用同步函数 cudaThreadSynchronize()后就处于空闲状态，这实际上是对计算资源的浪费。虽然说 GPU 的计算能力非常强大，但处理大规模数据集时也是很耗时的，而如果我们将一小部分计算任务分配给处于空闲状态的 CPU 来处理，让 CPU 与 GPU 并行的工作，势必会减少内核函数的执行时间，进而减少整个程序的执行时间。当然，如果整个任务的数据依赖关系比较强或者重新组合主机和设备两部分的计算结果比较费时，那么这种优化手段可能就不太适合了。

### 3.4.3 用原子函数实现全局同步

CUDA 运行时 API 提供了用于主机和设备以及同一个线程块内的线程之间的同步函数，但是并没有提供不同线程块之间的同步，即全局线程同步。虽然可以使用重新启内核函数的办法实现全局线程同步，但会带来的许多额外的代价（如：内核启动的开销、多次从全局存储器拷贝数据的开销）以及增加程序的复杂度。

对于大多数并行度较高的计算任务，通过合理的任务划分和分配可以使每个线程块之间没有数据依赖关系，相互独立，这样自然不需要线程块之间的同步操作了，第四章讲述的 DNA 或蛋白质局部序列比对问题就属于这一类型。但是有些计算任务就需要不同线程块之间相互协作、共同完成，例如：在 3.5 节介绍的

基于 CUDA 平台的归并排序算法。

幸好，CUDA 运行时 API 提供了很多原子函数，如：原子自增函数 atomicInc、原子比较交换函数 atomicCAS 等[1]。因此，我们将尝试使用恰当的原子函数来实现全局线程间的同步操作。当然在使用原子函数之前，需要参考相关手册以确定不同计算能力的 GPU 在使用原子函数方面存在的限制。下面的代码片段展示了使用原子函数 atomicInc()实现线程块之间的同步操作。

```
#include <sm_11_atomic_functions.h>//使用原子函数必须包含的头文件
#define UNLIMIT 65536
__device__ int counter=0; //用于记录到达同步点的线程块的个数
__global__ void KernelFunc(argument list) {
    ````````````
    If ( threadIdx.x==0 ) {//每个线程块中的 0 号线程执行原子函数以及测试
        atomicInc(&counter, UNLIMIT);
        while (counter<BLOCK_NUM); //只有当所有的线程块中的 0 号线程都执
        行了原子函数后线程 0 才会退出循环
    }
    __syncthreads(); //线程块内的同步，即其他线程等待 0 号线程
    ````````````
}
```

## 3.5 实例验证

### 3.5.1 异构平台上的并行归并排序算法

归并排序就是将两个或两个以上的有序序列合并成一个新的有序序列。相信大家对串行的归并排序算法已经很熟悉了，所以这里只介绍使用 CUDA 编程模型实现的运行在 CPU-GPU 异构平台上的并行归并排序算法。在实现该程序后我们会使用 3.4 节提出的优化方法对其进行优化并比较优化前后程序的执行时间。

算法描述：首先将输入的无序序列平均分成 N 端，每一段使用一个线程块进行排序。当所有的线程块都执行结束时，每段序列都是；其次，使用一半的线程块对相邻的小段序列进行归并排序，这一过程一直进行下去，直到只剩下一个有序段，此时整个序列已排好序。在两两归并阶段，由于线程块 $b_i$ 需要将上次自己排好序的分段与线程块 $b_i$+1 排好序的分段进行归并，所以线程块之间需要同步，即线程块 $b_i$ 只有等到线程块 $b_i$+1 将它负责的分段排好序后才能执行归并操作。图 3.4 给出了一个计算示例。

图中假设待排序的无序序列被平均分成 8 个分段，每个线程块负责将一个无序分段排好序，这样就得到了图中的第 4 层。从第 3 层到第 1 层，每层相邻的有序分段两两归并成一个新的有序分段，随着分段数量的逐层递减，实际工作的线程块数也逐层减半。对于一个分段数目为 N 的无序序列，只需经过一次排序和 $\log_2 N$ 次归并就可以得到一个有序序列。整个计算过程构成一个树形结构图，其中每层内的线程块并行执行，层间串行执行。

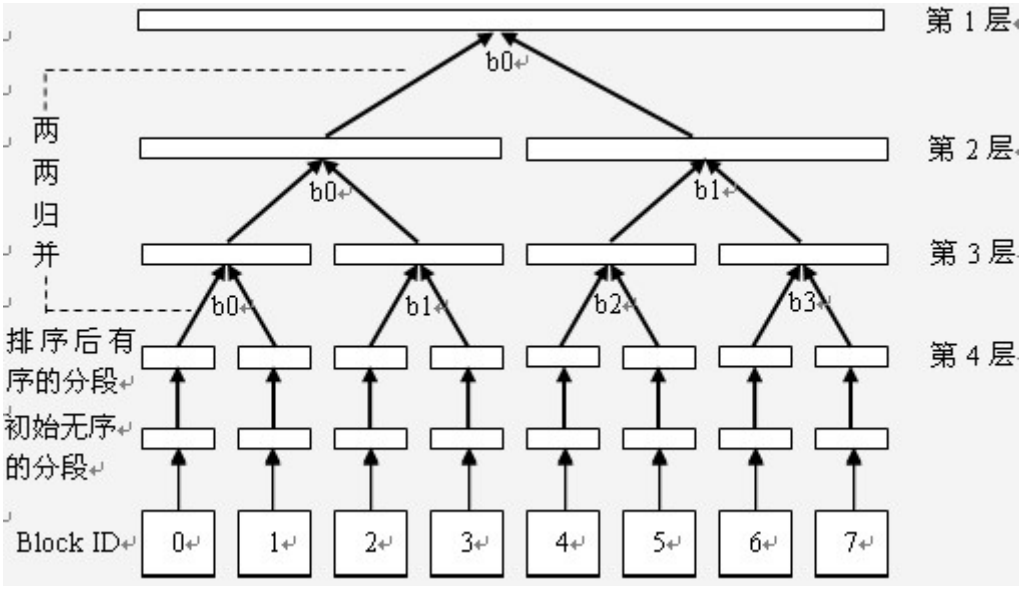


**图 3.4** 并行归并排序计算模型

### 3.5.2 实验结果及理论分析

**表 3.5** 使用一般优化方法后程序的执行时间（单位：ms）

| 数据集大小 | CPU 上串行归并排序的计算时间 | GPU 上并行归并排序的计算时间 | 时间比值 | 整个程序的执行时间 |
|---|---|---|---|---|
| 4M | 2493.0 | 44.2 | 56.4 | 242 |
| 8M | 5369.5 | 93.4 | 57.5 | 350 |
| 12M | 8285.4 | 160.8 | 51.5 | 478 |
| 16M | 11296.9 | 197.5 | 57.2 | 621 |

为了检验上述优化方法的效果，特别是 3.4 节提出的高级优化策略，我们在 CPU-GPU 的异构平台上使用 CUDA 编程模型实现了并行的归并排序算法，并对优化前后程序的性能进行了比较和分析。使用普通优化方法后的实验结果如表 3.5 所示。

需要说明的是，对程序进行性能优化的最终目的是使整个程序的运行时间最短。表 3-5 中我们列出了 CPU 和 GPU 的计算时间以及二者的比值只是为了说明归并排序算法很适合使用 GPU 进行加速。在编写 CUDA 程序时，刚开始为了验证 GPU 计算结果的正确性或者检验 GPU 加速的效果，通常我们都会编写一个在 CPU 上运行的串行版本的程序，并把 CPU 与 GPU 的计算结果或运行时间进行比较。当确定 CUDA 程序正确无误后，就不需要再运行串行版本的程序了，这也是为什么上表中整个程序的运行时间远少于 CPU 上串行程序的执行时间的原因。在使用 3.4 节提出的高级优化策略后，我们仅通过比较整个程序的运行时间来验证各优化方法的效果。实验结果如表 3.6 所示。

**表 3.6** 使用 3.4 节提出的高级优化策略后程序的性能比较（单位：ms）

| 数据集大小 | 优化前程序的执行时间 | 使用计算与通信重叠的优化方法后程序的执行时间 | 使用主机与设备并行计算的优化方法后程序的执行时间 | 两种优化方法同时使用时程序的执行时间 |
|---|---|---|---|---|
| 4M | 242 | 222 | 231 | 212 |
| 8M | 350 | 317 | 332 | 308 |
| 12M | 478 | 432 | 446 | 419 |
| 16M | 621 | 565 | 590 | 548 |

可见，对于并行归并排序算法来说，使用计算与通信重叠的优化方法比使用主机与设备并行计算的优化方法更加有效。前者使程序的性能平均提高了 8.7%左右，而后者只提高了 4.9%左右。这是因为 GPU 在解决这一问题时获得的加速比非常高（平均 55 倍左右），即此时 GPU 的计算速度是 CPU 的 55 倍，所以 CPU 能为 GPU 分担的计算任务就非常有限，导致优化效果不明显。相反，在程序中主机和设备之间还有多次通信，所以利用计算来隐藏通信的延迟还是有一定的优化效果。此外，同时使用这两种优化方法，程序的性能大约提高了 12%，这和两种优化方法分别带来的性能提升之和 13.6%已非常接近了。

对于并行归并排序如果不使用我们所设计的用原子函数实现不同线程块之间的同步，而采用传统的重新启动内核的方法是不切实际的。因为对于分段数目为 N 的并行归并排序，共需要重新启动 N 次内核函数才能完成排序操作。如果 N 值很大，就会给编程带来极大的困难。这也是在本例中我们没有测试使用原子函数实现不同线程块之间同步的优化方法所带来的性能收益的原因。但是为了证明这一优化方法的有效性，我们设计了一个简单示例程序来进行测试。主要代码片段如下所示。

```
//使用原子函数实现不同线程块之间同步
#include <sm_11_atomic_functions.h>
__global__ void KernelAtomicSynchronize
(unsigned int *dsync){
    for(int i=0;i<LOOP_NUM;i++){
        if(threadIdx.x==0)  {
            atomicInc(dsync, 100000);
            while(*dsync<BLOCK_NUM);
        }
        __syncthreads();
    }
}
int main(){
    //分配内存、初始化、执行配置等
    KernelAtomicSynchronize<<<GridDim,
BlockDim>>>(dsync);
    cudaThreadSynchronize();
    //拷贝结果并验证结果的正确性等
}
```

```
//传统的使用重新启动内核函数的方法实现
  不同线程块之间的同步
#include <sm_11_atomic_functions.h>
__global__ void RestartKernelSynchronize(){

   }

int main(){
    //分配内存、初始化、执行配置等
    for(int i=0;i<LOOP_NUM;i++){
        RestartKernelSynchronize<<<GridDim,
        BlockDim>>>();
        cudaThreadSynchronize();
    }
    //其他操作
}
```

上述两个内核函数都非常简单，没有完成实际有用的计算，但左边的内核函数描述了如何使用原子函数 atomicInc 以及块内同步函数__syncthreads()实现不同线程块之间的同步。循环变量 LOOP_NUM 表示需要重新启动内核函数的次数，我们分别测试了不同的 LOOP_NUM 值，两个程序的执行时间，实验结果表 3.7 所示。

**表 3.7** 两种同步方法的性能比较

| LOOP_NUM | 使用传统方法:重新启动内核函数时内核函数的执行时间 | 我们设计的方法：使用原子函数时内核函数的执行时间 | 使用我们的方法所获得的加速比 |
|---|---|---|---|
| 10 | 238.0 | 70.0 | 3.4 |
| 100 | 1997.0 | 406.0 | 4.9 |
| 1000 | 19928.0 | 3750.0 | 5.3 |
| 10000 | 199471.0 | 37121.0 | 5.4 |

由表 3.7 可见，使用我们设计的方法实现不同线程块之间的同步比传统方法平均要快 4.75 倍，而且随着 LOOP_NUM 的增大，获得加速比越高。

## 3.6 本章小结

相对于运行在 CPU 上的程序来说，运行在 CPU-GPU 异构系统上的程序的性能优化更加重要也更加复杂。本章首先分析了影响 CUDA 程序性能的关键因素，然后针对每一种因素提出了相应的优化方法。其中 3.2 和 3.3 两节介绍的优化方法属于常规方法，这些方法在 CUDA 编程手册上也都有介绍，这里只是进行总结和实验验证。但是 3.4 节介绍的优化方法属于高级优化方法，其中使用原子函数实现不同线程块之间的同步方法是我们所设计的，实验表明该方法比现有的重新启动内核函数的方法快 4~5 倍。对于一个给定的 CUDA 程序，通常都是先采用常规方法进行优化，因为这些优化方法具有普适性且能大幅提升程序的性能。此后，如果还想进一步提高程序的性能，就必须使用高级优化方法了，但使用这些方法并不一定能达到目的，甚至还可能适得其反，这需要对具体问题进行具体分析。为了验证这些优化方法的有效性，我们在不同的平台上进行实验并对实验结果给出了必要的理论分析。

# 第4章 CPU-GPU 异构计算平台在生物信息学中的应用

## 4.1 引言

生物信息学是生物学和信息技术科学相交叉的一门新兴学科。研究人员借助计算机来处理生物数据，以发现某些生物学规律，从而指导药物研制和疾病治疗。由于生物数据规模非常庞大并且还在急剧增长，使得该学科需要强大的计算能力，以缩短实验时间。所以一直以来，生物信息学都是高性能计算的一个重要应用领域。

生物信息学研究的问题非常广泛，涵盖了算法设计、机器学习、数据挖掘、生命科学、高性能计算、人工智能等诸多领域。生物信息学的发展离不开上述学科的发展和支持，同时也能推动其他学科的发展、进步。

支持 CUDA 的 GPU 作为一种高效、廉价的计算工具，非常适合用于高性能计算领域，尤其是生物信息学。因为生物信息学中的大多数问题都涉及到大规模数据处理，并行具有较高的并行性和计算密集度，这些恰恰是 GPU 擅长处理的任务类型。

本章我们将以生物信息学中的局部序列比对问题为例，阐述如何设计适合 CPU-GPU 异构系统的并行算法，在实现算法并进行优化后，通过实验说明该异构计算平台在生物信息学领域有广阔的应用前景。

## 4.2 问题描述

### 4.2.1 序列比对问题

序列比对[17][18]是生物信息学中一个常见的问题，用来检测 DNA 或蛋白质序列之间的相似性，分为两种类：全局序列比对和局部序列比对。其中全局序列比对问题定义为：给定 DNA 或蛋白质序列 $A=a_1a_2\ldots a_n$ 和 $B=b_1b_2\ldots b_m$，（n 和 m 分别表示序列 A 和 B 的长度）通过插入或缺失等操作将序列 A 转化成序列 B。而局部序列比对旨在识别出两条序列中最相似的部分。这里的相似性（Similarity）表示在序列比对过程中插入或缺失操作所造成的代价。因此，局部序列比对是全局序列比对的特例，尽管它又分为一对序列的比对（Parallel Sequence Alignment）和多条序列的比对（Multiple Sequences Alignment，MSA）[18]，但本文仅讨论一

对序列的局部比对问题。

通常，给定一条查询序列和一个包含若干条目标序列的数据库，一对序列的局部比对旨在通过将查询序列和数据库中每一条目标序列进行比对，并找出一条和查询序列最相似的目标序列。

### 4.2.2 Smith-Waterman 算法

Smith-Waterman 算法[27]是一个用于解决 DNA 或蛋白质局部序列比对的经典动态规划算法。在计算过程中，该算法使用一个矩阵（称为相似性矩阵）来存储查询序列和目标序列中从第一个位置开始的每一对子序列的相似度值（Similarity Value）。例如：$A=a_1a_2\ldots a_n$ 为给定的一条查询序列，$B=b_1b_2\ldots b_m$ 为目标序列数据库中的某一条序列，矩阵 $H_{(n+1)X(m+1)}$用来保存计算过程中的相似度值（由于串行算法是一个动态规划算法，H 常被称为动态规划矩阵）。则矩阵元素 H(i, j)表示子序列 $a_1a_2\ldots a_i$, (1≤i≤n）和 $b_1b_2\ldots b_j$, (1≤j≤m)的最大相似度值。H(i, j)可用如下公式计算：

$$H(i,j)=\begin{cases} max\{0, H(i,j-1)-1, H(i-1,j)-1, \\ \quad H(i-1,j-1)-1\}, & \text{if } a_i \neq b_j \\ H(i-1,j-1)+2, & \text{if } a_i = b_j \end{cases} \tag{4.1}$$

使用递归公式（4.1），我们可以很容易的计算出查询序列 A 和目标序列 B 的相似矩阵。图 4.1 给出了一个简单的示例。

| | | C | A | C | T | A | T | G | C |
|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 0 | 2 | 1 | 0 | 2 | 1 | 0 | 0 |
| G | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 3 | 2 |
| C | 0 | 2 | 1 | 3 | 2 | 1 | 0 | 2 | 5 |
| C | 0 | 2 | 1 | 3 | 2 | 1 | 0 | 1 | 4 |
| T | 0 | 1 | 1 | 2 | 5 | 4 | 3 | 2 | 3 |
| C | 0 | 2 | 1 | 3 | 4 | 4 | 3 | 2 | 4 |
| A | 0 | 1 | 4 | 3 | 3 | 6 | 5 | 4 | 3 |

the pair of most similar subsequences:

C A C T — A
| | | | |
C — C T C A

**图 4.1** 一对 DNA 序列的局部比对计算示例

图 4.1 中，查询序列 A=AGCCTCA（列方向），目标序列 B=CACTATGC（行方向），首先按照公式（4.1）计算序列 A 和 B 的相似性矩阵，然后从相似性矩

阵中的最大值（A 和 B 的相似度值）开始向前回溯（图中箭头所示），即可得到 A 和 B 中一对最相似的子串：CACTA 和 CCTCA。

此外，从公式（4.1）可以画出相似性矩阵的数据依赖关系图，如图 4.2 所示。

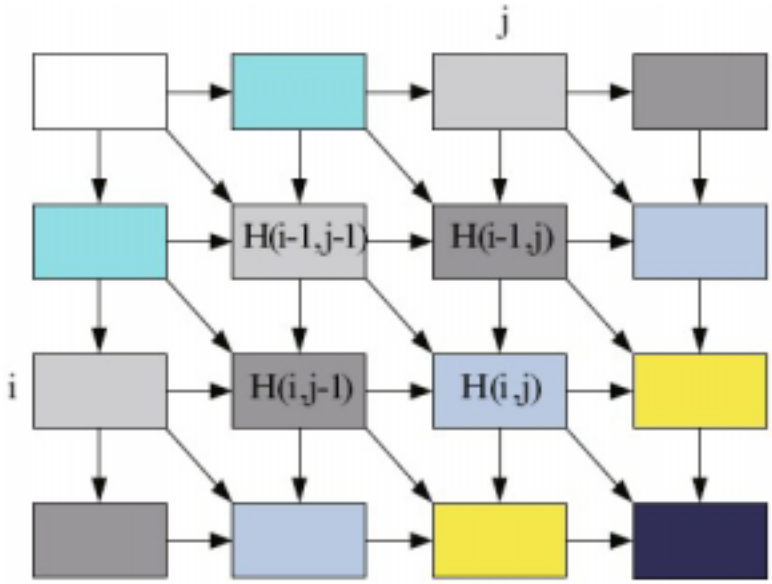


**图 4.2** 相似性矩阵的数据依赖关系图

从图 4.2 中可以看出，元素 H(i, j)的值依赖于元素 H(i, j-1)、H(i-1, j)、H(i-1, j-1)的值。即相似性矩阵的行、列、对角线方向都存在数据依赖关系，但反对角线方向（图中具有相同颜色的方块）没有数据依赖。

可见，查询序列在和数据库中的每一条目标序列进行比对时，都必须计算出一个相似性矩阵。由于每条序列很长（几十到上千个字符），数据库的规模非常大（通常含有十几或几十万条序列）且还在不断增长，使得原始串行的 Smith-Waterman 算法在求解这一问题时非常耗时，因此很多学者开始研究并行的 Smith-Waterman 算法并在多核系统上实现，以便获得好的性能。

## 4.3 Smith-Waterman 算法在异构系统上的实现

### 4.3.1 基于反对角线的并行实现

由于相似性矩阵中反对角线方向没有数据依赖关系，因此位于每条反对角线上的元素可以并行计算，这也是最直观的并行化方式。2007 年，Liu[17]等人按照反对角线方向的并行化方式设计了并行的 Smith-Waterman 算法，并在基于 CPU-GPU 的异构系统上予以实现。实验结果表明，与 CPU 上的串行程序相比，并行的版本获得了 6~13 倍的加速比。据我们所知，这也是第一个尝试用 GPU 来加速序列比对问题的成功实例。2009 年，Fumihiko Ino[37]等人将 Liu 的工作进行

扩展，实现了基于多 GPU 异构系统的并行版本。

尽管上述两个基于CPU-GPU异构系统的并行Smith-Waterman算法获得了不错的加速比，但程序的性能仍然有进一步提升的空间。首先，他们都是采用基于反对角线方向的并行化方式，由于反对角线的长度不一致（45 度方向的对角线最长），会造成执行内核函数的线程间存在负载不均衡的弊端。其次，受当时 GPU 体系结构的限制，Liu 使用片外的纹理存储器来存储相似性矩阵的元素，这比使用当前 GPU 片内的共享存储器效率要低的多。最后，Liu 和 Fumihiko Ino 都是使用图形 API 编程的，这样不仅增加了难度还降低了程序的性能。

鉴于上述原因，本文采取了一些有效地措施，较好的消除了上述因素对程序性能的影响。首先，在认真分析了数据依赖关系后，巧妙的消除了相似性矩阵中同一列上的数据依赖，从而将基于反对角线的并行化方式改为基于列并行。由于矩阵每列的长度相同，从而消除了线程间负载不均衡的问题。其次，我们使用片上的共享存储器来存储相似性矩阵的元素，大大降低了访存的开销。最后，我们使用 CUDA 计算平台编程，既降低了开发难度又提高了程序的效率。下面将对基于列并行的 Smith-Waterman 算法进行详细介绍。

### 4.3.2 基于列的并行实现

#### ■ 基于列并行的理论分析及证明

图 4.2 表明，在相似性矩阵中只有反对角线上的元素没有数据依赖关系，可以并行计算，Liu[17]和 Fumihiko Ino[37]也正是利用这一点才设计出并行的 Smith-Waterman 算法。但是通过展开公式（4.1）发现，相似性矩阵中同一列元素间的数据依赖关系是可以消除的，从而设计成基于列并行的 Smith-Waterman 算法。下面将证明这种做法的正确性和合理性。

根据公式(4.1)，如果条件 $a_i=b_j$ 满足，则 H(i, j)只依赖于 H(i-1, j-1)，而不依赖于任何第 j 列的元素；如果 $a_i\neq b_j$，则 H(i, j)依赖于 H(i, j-1)、H(i-1, j)、H(i-1, j-1)三者中的最大值（暂时不考虑常数 0）。现在我们只需考虑 H(i-1, j)是最大值的情况（只有在这种情况下，相似性矩阵 H 第 j 列元素才存在数据依赖关系），则 H(i, j)=H(i-1, j)-1 (假设 H(i-1, j)-1≥0)，此时我们可以递归的计算 H(i-1, j)直到第 j 列元素没有数据依赖关系或者 i=0（初始化 H(0, j)=0）. 假设在上述递归计算的过程中，$a_k$（k>0）是第一个和 $b_j$ 匹配的元素，并且在每次计算时 H(r, j)-1 (k≤r≤i-1)都是可能存在依赖关系的三个值中的最大值，则有：

$$H(i,j)=\max\{0, H(k-1, j-1)-(i-k)+2\}. \quad (4.2)$$

为了证明公式（4.2），我们首先使用数学归纳法证明引理 1。

**引理 1**：如果元素 $a_i$ 与 $b_j$ 不匹配，$a_k$（$1 \le k < i$）是从元素 $a_i$ 到 $a_1$，第一个与 $b_j$ 匹配的元素，并且在每次递归展开中 $H(r, j)-1$（$k \le r \le i-1$）都是最大值，则有：

$$H( i, j ) = H( k, j ) – ( i – k ) \tag{4.3}$$

证明：（1）当 $k = i-1$ 时，$H(i, j) = H(k, j)-(i–k) = H(i-1, j)-1$，由公式（4.1）知，此时公式（4.3）成立。

（2）假设当 $m = i-r$（$k < m < i-1$）时，公式（4.3）成立，即：

$$H( i, j ) = H( m, j ) – ( i – m ) \tag{4.4}$$

现在我们将证明当 $m = i-r-1$ 时，公式（4.3）仍然成立。

因为 $a_m$ 与 $b_j$ 不匹配，所以

$$H( m, j ) = H( m – 1, j ) – 1 \tag{4.5}$$

将公式（4.5）代入公式（4.4）得到：

$$H( i, j ) = H( m – 1, j ) – ( i – m ) – 1 = H( m –1, j ) – ( i – ( m – 1 ) ),$$

因此假设成立。

（3）由于假设（4.2）成立，所以引理 1 也成立

根据引理 1，可以很容易证明公式（4.2）也是成立的。因为 $a_k$ 与 $b_j$ 匹配，所以：

$$H( k, j ) = H( k – 1, j – 1 ) + 2 \tag{4.6}$$

将公式（4.6）代入公式（4.3），并且考虑常数 0，得到：

$$H( i, j ) = \max\{ 0, H( k – 1, j – 1 ) – ( i – k ) + 2 \},$$

即公式（4.2）成立。

通过上述证明，我们消除了相似性矩阵中同一列数据之间的依赖关系，新的数据依赖关系如图 4.3 所示。

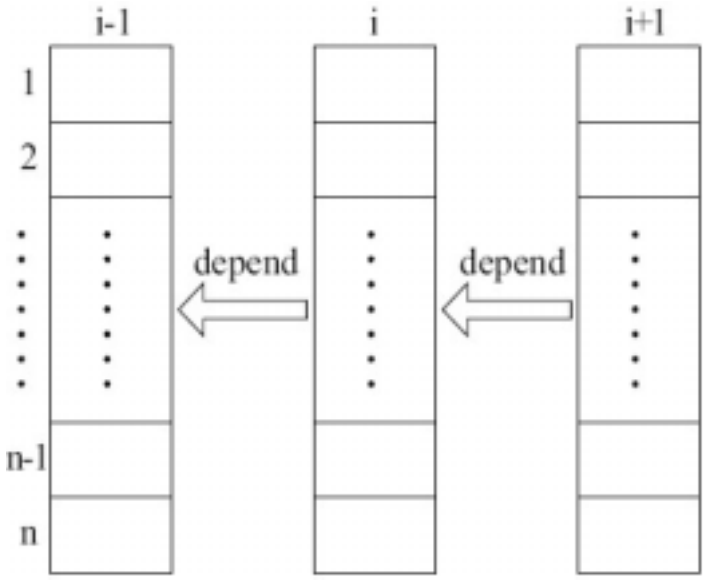


**图 4.3** 基于列并行的数据依赖关系图

现在，我们将计算元素 H(i, j)的公式（4.1）改为

$$H(i,j)=\begin{cases} max\{0, H(i,j-1)-1, H(i-1,j-1)-1, \\ \quad H(k-1,j-1)-(i-k)+2\}, & \text{if } a_i \neq b_j \\ H(i-1,j-1)+2, & \text{if } a_i = b_j \end{cases}$$

$$\text{where} \quad k = max\{n, \text{ for } 0 < n < i \text{ and } a_n = b_j\}. \tag{4.7}$$

在编程时，需要事先计算出查询序列中从当前位置向前推，第一个和目标序列中给定字符相匹配的字符的下标，并将计算结果存储在数组 pre_dependence 中。

## ■ 基于列并行的算法设计及实现

在基于 CPU-GPU 的异构系统中，为了充分发挥 CPU 和 GPU 各自的计算能力，必须进行合理的任务划分。对于一对序列的局部比对问题，在计算相似性矩阵的过程含有较高的并行性，而且它还是程序中最耗时的部分，因此这一部分将作为内核函数在 GPU 上执行。此外，类似于 Liu[17]，为了增加程序的并行度，我们同时实现了任务级并行和数据级并行。其中，任务级并行来自于线程块之间的并行。即每个线程块负责一条目标序列和查询序列的比对，每个流多处理器（SM）最多能并发的执行 8 个线程块，而实验所用的 GeForce 9800 GT 和 Tesla C1060 分别包含 14 和 30 个 SM。这样就会同时有几十条目标序列和查询序列进行比对。数据级并行在于可以并行的计算相似性矩阵中同一列的元素。即线程块中的每个线程负责目标序列中的一个字符与查询序列中相应字符的比对。这两级并行计算的方式如图 4.4 所示。

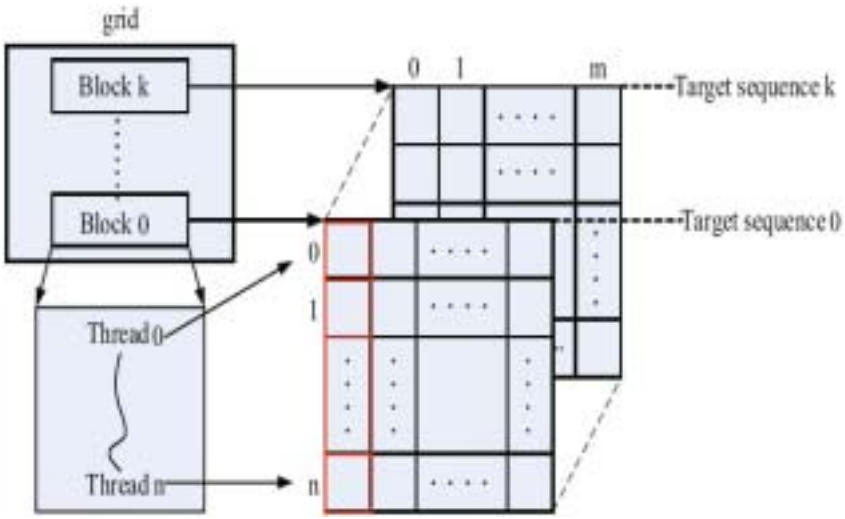


**图 4.4** 内核函数中的两级并行计算

当内核函数执行结束时，查询序列和每一条目标序列的相似度值就计算出来了，并存储在全局存储器中的数组中。然后将该数组从设备内存拷贝到主机内存，最后在主机端从数组中选出一个最大值，并计算具有最大相似度值的两条序列的最相似子串。这一部分在 CPU 上执行，其计算时间相对于内核函数的执行时间可以忽略不计。

## ■ 基于列并行的算法伪代码

按照上面的任务划分策略，我们在基于 CPU-GPU 的异构系统上实现了按列并行的 Smith-Waterman 算法。主机端和设备端的伪代码如下。

主机端的算法伪代码如下：

1. 随机产生查询DNA序列和目标DNA序列的集合
2. 计算查询序列中从当前位置往前第一个和目标序列中某个元素匹配的元素的下标，并存入数组f_index.
3. 拷贝查询DNA序列、数组f_index到gpu的constant memory，拷贝目标DAN序列到GPU的global memory中.
4. 设置执行配置并启动内核函数计算查询DNA序列和每一条目标序列的相似度值，并写入global memory.
5. 从global memory拷贝kernel函数计算的结果到host memory.
6. 从拷贝回的结果中选出最大值，该最大值对应的数组下标即为和查询序列最相似的目标序列标号，然后调用串行的 Smith-Waterman 算法计算出查询序列和目标序列中最相似的字串.

设备端的内核函数伪代码如下：

1．在GPU的shared memory上分配数组seq[],previous_col[],current_col[]分别用来存储一条目标DNA序列，前一次迭代已经计算出来的矩阵前一列的值，本次迭代将要计算的矩阵当前列的值.

2．属于同一个线程块的线程将本块要计算的目标序列从GPU的global memory拷贝到数组seq中，且每个线程只需取序列中的一个元素.

3．初始化相关数据的值

4．For i in m //依次计算矩阵每一列的值

根据数组previous_col[]和cons_pre[]的值计算当前列元素的值，每个线程只计算一个值.

__syncthreads().//同步

拷贝数组current_col[]到数组previous_col[]为下次迭代做准备.

用类似于规约的方法计算当前列的最大值并和当前保存的矩阵列的最大值进行比较，保存较大者.

5. 循环结束后将整个矩阵的最大值写入 global memory.

## 4.4 优化方法及实验结果

### 4.4.1 循环使用共享存储器

为了从数据库中找出一条和查询序列最相似的目标序列，需要计算查询序列和每一条目标序列的相似性矩阵，这是整个程序最耗时的部分，因此我们将启动一个内核函数来完成这部分的计算。但是 GPU 片上的共享存储器只有 16KB，不足以存储整个相似性矩阵，除非使用全局存储器。然而，访问一次全局存储器大约需要 400~600 个时钟周期，而访问一次共享存储器只需 4 个时钟周期（在没有存储体冲突的情况下）。此外，序列比对是一个访存密集型的问题，因此使用全局存储器必将严重影响程序的性能。

在对内核函数的执行过程进行深入分析后发现，其实根本没有必要存储整个相似性矩阵，而只需存储两列矩阵元素即可。原因在于并行的粒度是矩阵的列，即只有当前列的元素能被同时计算，而且当前列仅仅依赖于矩阵前一列元素的值，当前列所有元素的值在计算出来之前，任何后面列的元素都不会被计算。因此，只需在共享存储器中分配两个数组 pre_col 和 cur_col，存储前一次迭代时已经计算出来的前一列矩阵元素的值和本次迭代将要计算的当前列矩阵元素的值。循环使用这两个数组，即可完成整个相似性矩阵的计算。图 4.5 显示了这两个数

组被循环使用的过程。

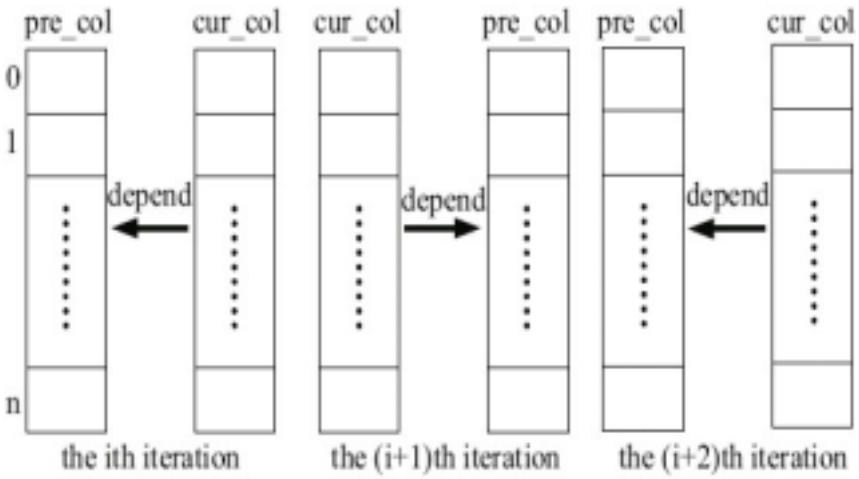


**图 4.5** 循环使用共享存储器中的数组

### 4.4.2 采用类似于规约的方法计算最大值

在计算出当前列矩阵元素的值后，需要计算出该列元素的最大值，以确定它是否为到目前为止所找到的最大值。这里，我们使用一种高效的、类似于规约的方法来计算当前列元素的最大值。图 4.6 给出一个使用该方法的计算示例。从中可以看出，当数组元素的个数等于 $2^n$ 时，只需要 n 此迭代就可以求出最大值，并且第 i 此迭代共有 $2^{(n-i)}$个线程并行的执行。

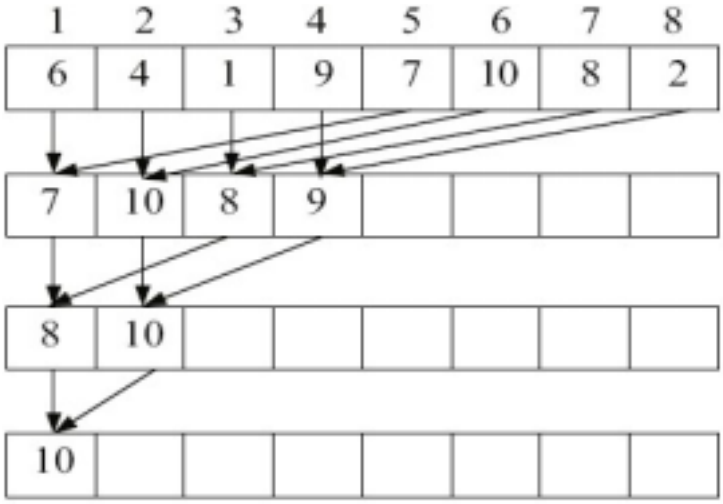


**图 4.6** 一种类似于规约的方法计算数组的最大值

### 4.4.3 实现了对全局存储器的联合访问

全局存储器没有高速缓存（Cache），访问一次全局存储器大约需要 400~600

个时钟周期的开销。因此使用合适的访问模式对最大化内存带宽是非常重要的，因为非联合访问全局存储器的内存带宽比联合访问的要低 2~10 倍。但 GPU 能够仅使用一条指令从全局存储器读取 32 位、64 位或 128 位的数据到寄存器。如果一个 half-warp（半个 warp）中所有线程的访存操作能够联合成一个存储器事务（Memory Transaction），此时可以获得最大的内存带宽。一个存储器事务可以是 32B（计算能力 1.2 以上的设备）、64B 或 128B。但是 DNA 或蛋白质序列是由一些特定的字符组成，且每个字符只需 1B 的存储空间，因此如果序列的数据类型定义成字符型（char），则读/写全局存储器不能实现联合访问。但是我们可以将序列的数据类型定义为整形（int），而一个整形数据需要 4B 的存储空间，且 half-warp 中的每个线程都访问 4B 的数据，这些访存操作会联合成一条存储器事务，从而实现了全局存储器的联合访问。此外，这也避免了访问共享存储器时出现访存冲突（bank 冲突）的情况。

### 4.4.4 实验结果

为了说明基于列的并行 Smith-Waterman 算法比之前基于反对角线的并行方式更加高效，我们分别在两种不同的 CPU-GPU 异构系统上运行优化后的程序并与 Liu[17]和 Fumihiko Ino[37]的实验结果进行比较。表 4.1 显示了实验所用的两种异构平台。

**表 4.1** 实验环境

| 环境 | Liu 的实验平台 | 我们的实验平台 | |
|---|---|---|---|
| | | 平台一 | 平台二 |
| CPU | Pentium 3.0GHz | Intel Pentium Dual E2140 1.6GHz | Intel Xeon CPU E5520 @ 2.27GHz |
| GPU | GeForce 7900 GTX | GeForce 9800 GT | Tesla C1060 |

在实验中，随机生成 176469 条 DNA 目标序列，每条目标序列的长度相同，且为 361。每次随机生成的 DNA 查询序列的长度不同，分别为 63、127、255、361、511 等。实验结果如表 4.2 所示。

**表 4.2** 实验一的结果

| 查询序列长度 | OSEARCH 串行算法 | Liu 的基于反对角线并行的算法 | 加速比 | OSEARCH 串行算法 | 我们的基于列并行的算法 | 加速比 |
|---|---|---|---|---|---|---|
| | Liu 的实验平台 | | | 平台一 | | |
| | CPU 运行时间 | GPU 运行时间 | | CPU 运行时间 | GPU 运行时间 | |
| 63 | 91 | 14 | 6.5 | 90 | 2.3 | 39.1 |
| 127 | 184 | 20 | 9.2 | 183 | 4.9 | 37.3 |
| 255 | 375 | 31 | 12.1 | 375 | 10.7 | 35.0 |
| 361 | 533 | 44 | 12.1 | 537 | 14.3 | 37.6 |
| 511 | 732 | 56 | 13.1 | 768 | 21.6 | 35.6 |

**注：**表中的 OSEARCH 是对原始串行 Smith-Waterman 算法的直接实现，Liu 的实验数据来自[17]。加速比表示串行 OSEARCH 算法在 CPU 上的执行时间与并行 Smith-Waterman 算法在 GPU 上的执行时间的比值。

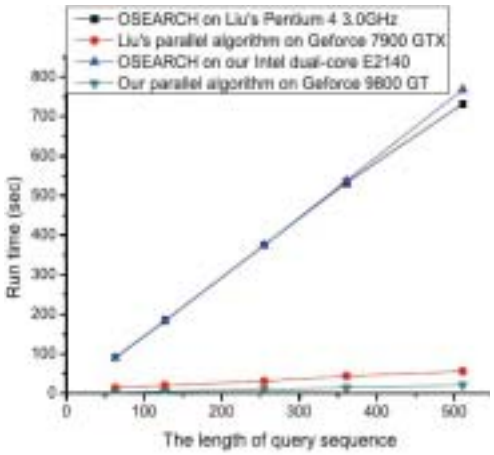


**图 4.7** 不同算法的运行时间

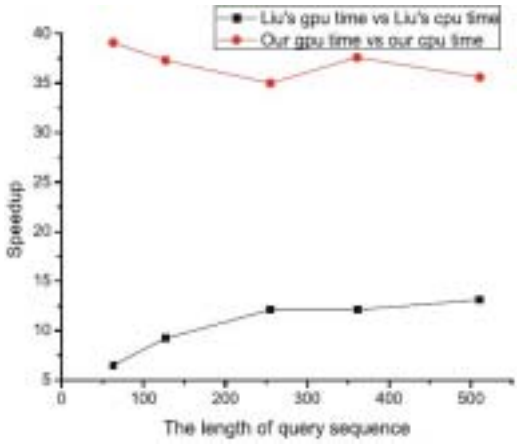


**图 4.8** 两种并行实现的加速比对比

此外，我们也和 Fumihiko Ino[37]的实验结果进行了比较。在 Fumihiko Ino 的试验中，使用的 GPU 是 GeForce 8800 GTX，搜索的数据库是 SWISS-PROT 51.0 版本，该数据库中含有 241242 条 DNA 目标序列，序列的平均长度为 367，单个 GPU 最快的运行时间为 18s。我们也测试了同样规模、随机生成的 DNA 目标序列集合，在平台一的 GPU 上运行时间为 19.12s。虽然我们总的运行时间比 Fumihiko Ino 多，但 GeForce 8800 GTX 有 16 个流多处理器，而 GeForce 9800 GT 只有 14 个流多处理器，如果比较单个标量流处理器的运行时间，则我们所用的

时间为 2141.44s，而 Fumihiko Ino 用时为 2304.0s。

需要说明的是，由于我们、Liu 以及 Fumihiko Ino 所用的实验平台都不相同，所以这种性能的比较只是相对的。

为了进一步验证，基于列并行的 Smith-Waterman 算法的有效性，我们还在实验平台二上运行了优化后的程序。在实验中，随机生成 65000 条 DNA 目标序列和 4 条 DNA 查询序列，且查询序列和目标序列的长度相同，分别取：64、128、256、512。实验结果如表 4.3 所示。

**表 4.3** 实验二的结果

| 序列长度 | 平台二 | | 加速比 |
|---|---|---|---|
| | CPU 运行时间(s) | GPU 运行时间(s) | |
| 64 | 8.67 | 0.13 | 66 |
| 128 | 33.00 | 0.55 | 60 |
| 256 | 129.70 | 2.40 | 54 |
| 512 | 517.16 | 10.47 | 49 |

可见，在实验平台二上，平均加速比达 57 倍。对于单个 GPU 来说这样高的加速比是非常可观的。

## 4.5 本章小结

本章详细介绍了生物信息学中的 DNA 或蛋白质序列比对问题以及用来解决这一问题的 Smith-Waterman 算法。通过分析及证明，我们有效的消除了相似性矩阵中同一列元素之间的数据依赖关系，从而将原来基于反对角线并行的 Smith-Waterman 算法改为基于列并行的算法，并将新的并行算法在两种不同的异构系统上实现及优化。实验结果表明，基于列的并行方式比基于反对角线的并行方式更加高效，在两种计算平台上分别获得了平均 37 倍和 57 倍的加速比。这也反映了基于 CPU-GPU 的异构计算平台在生物信息学和其他高性能计算领域有非常广泛的应用前景和巨大的发展潜力。

# 第5章 平台无关的多核并行编程模型

## 5.1 引言

自从 2005 年初 Herb Sutter 发表名为《免费的午餐已经结束》的著名文章以来，人们就意识到单纯依靠提高处理器时钟频率和增加芯片内晶体管数量的办法已经不能使摩尔定律的传奇得以延续，于是几乎所有的芯片设计者都把目光转向多核技术——通过在单个芯片上集成多个处理核心的办法来提高整个芯片的性能，这也是新世纪延长摩尔定律的最直接、有效的途径。目前，几乎所有的计算平台都是多核系统。从高端服务器、普通 PC 机、笔记本到各种手持设备无一不是多核系统。多核芯片种类繁多，包括多核 CPU、GPU、Cell、DSP 等，它们的体系结构相差很大，都有各自应用的领域。

面对种类繁多的计算设备，对应用程序员来说既是机遇也是挑战。一方面我们有更多可供选择的计算平台来提高应用程序的性能；另一方面我们也面临着两个严峻的问题：1）如何使现有的大量串行代码并行化以便充分发挥出多核平台的计算能力；2）当底层硬件平台发生变化时，如何使应用程序无需修改或少做修改即可在新的硬件平台上高效运行。

对于第一个问题，理想的解决方案是自动并行化，即开发并行编译器或自动并行化工具将串行程序直接并行化。虽然这样的并行编译器和并行化工具目前已经存在，但其并行的效率不高或存在诸多限制（比如要求原有的串行程序具有某些典型的特征），往往达不到人们的要求。例如：斯坦福（Stanford）大学的 SUIF 编译器[38]等，因为要处理多个拆分后的小程序之间的依赖关系，同时要保持原有串行程序语义，导致编译器的策略非常保守，实际效果也不太理想。Huckleberry [39]是一个由哥伦比亚大学（Columbia University）于 2010 年提出的能将串行递归程序针对多核平台自动并行化的工具。Huckleberry 的代码生成器不仅能够将串行的递归程序自动划分成一些小的任务集合，还能进行一些底层的控制操作，如：数据的分布、不同处理器核之间的同步等。虽然 Huckleberry 在自动并行化的研究领域迈出了重要一步，但其局限性也是显而易见的，如：只能并行化采用分治算法实现的递归程序，只能应用在分布存储的多核平台上，并行化后程序的性能还受数据依赖关系、工作负载对局部存储器大小的需求等因素的影响。

而针对某个具体的多核平台，手动的将串行程序改造成并行程序，虽然可以获得较好的性能收益，但是这样不仅对程序员提出了更高的要求（要求他们熟练掌握并行编程）且工作量巨大。

第二个问题也是非常重要的。近年来计算机系统发展迅猛，体系结构朝并行化发展并呈现出多样化。既有通用多核 CPU，也有基于流处理器的 GPU、IBM Cell 以及基于 ARM 架构的 DSP 芯片、FPGA 等。由于这些硬件平台的特性和架构差别很大，要想编写一个能在所有平台上高效运行的程序几乎是不可能的。因此，当底层的硬件平台发生变化时（比如：由多核 CPU 换成 GPU），程序员不得不针对新的硬件平台编写一个新版本的程序，这无疑加重了程序员的负担。因此，我们的应用程序最好是平台独立的，即通过某个抽象的中间层次，对应用程序屏蔽底层硬件平台的差异，实现“Write Program Once，Running on Different Platforms Efficiently”。

2010 年 7 月，斯坦福（Stanford）大学的 Eastman 和 Pande 领导的团队设计和实现了一个面向分子动力学模拟计算的并行函数库 OpenMM。通过采用合理的分层结构，较好的解决了上述两个问题。关于 OpenMM 具体的实现细节请参考文献 [25][26][40]。与 OpenMM 相似，SWARM(Software and Algorithms for Running on Multi-core)[41]也是一个基于库的（Library-based ）并行编程框架。SWARM 是由佐治亚理工学院于 2007 年提出的一个可移植的开源并行编程框架，旨在帮助程序员设计和实现基于多核处理器的并行算法。但 SWARM 只适用于 SMP(Symmetric Multiprocessor)结构的的计算机系统而且尚不成熟，有待进一步的研究。

## 5.2 OpenMM 简介

### 5.2.1 OpenMM 概述

OpenMM 是一个用于在高性能计算机体系结构上执行分子动力学模拟程序的应用程序编程接口（API）。OpenMM 面向的群体是分子动力学模拟软件的开发者，而不是计算生物学家或模拟软件的使用者。它有以下三个主要特点：

（1）平台无关系：OpenMM 计划支持的计算平台包括由大量 CPU 核组成的高并行系统、GPU 以及由网络互连的集群系统；

（2）高效性：OpenMM 提供的 API 能够充分发挥底层硬件的计算能力，从而提高程序的性能；

（3）可扩展性：除了 OpenMM 提供的 API 之外，开发者可以根据自己的需要方便的添加新的 API，从而对其进行扩展。

OpenMM 目前的版本为 2.0，且只支持 GPU 计算平台，采用 C++语言实现，还处在开发之中。

### 5.2.2 OpenMM 设计原则

作为一个并行函数库，OpenMM 的 API 遵循以下五个设计原则[42]：

（1）API 限制在比较窄的应用范围。OpenMM 提供的 API 旨在帮助分子动力学模拟软件的开发者能够有效地利用高性能的计算平台。而与此无关的功能，如文件的读写、分子模型的建立等，并没有对应的 API。

（2）API 必须在各种支持的硬件平台上都能高效地实现。这样在设计 API 时就必须考虑不同硬件体系结构的差异，以避免实现 API 的函数在不同的平台上性能差别太大。例如，OpenMM 不提供直接访问状态信息（粒子的位置、速度等）的 API，因为访问不同体系结构的存储器开销相差很大。

（3）API 应该易于理解和使用。API 除了高效之外，还必须简单易学、好用，这样才不会增加使用者的负担，也才会有更多的人使用它。为了达到这一目的，清晰和简单至关重要。

（4）API 应该具有模块化和可扩展性。任何一个函数库都不可能提供所有用户需要的功能，所以支持扩展性就很重要。当用户需要某个新的功能时，可以很方便的添加一个模块。

（5）API 应该是硬件平台无关的。计算机的体系结构变化很快，很难预测未来会出现什么样的硬件平台，因此将 API 设计成平台独立的就非常重要。当底层硬件平台改变时，应用程序不做修改就可以在新的平台上高效运行，真正实现“Write Program Once，Running on Different Platform Efficiently”。

### 5.2.3 OpenMM 的体系结构

OpenMM 采用的是一种分层的体系结构，如图 5.1 所示。

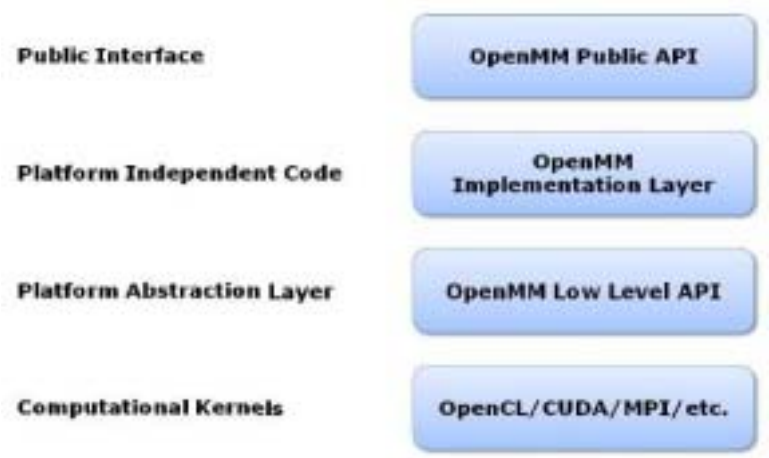


**图 5.1** OpenMM 的体系结构[42]

如图 5-1，处于最上层的是 OpenMM 的公共接口（Public Interface），这一层设计简单、易于理解、完全平台无关。第二层 OpenMM 实现层（Implementation Layer）用来实现上层的 OpenMM Public API，它是上层 API 与底层 API 的接口，该层也是平台独立的。第三层 OpenMM 底层 API（Low Level API）为上层隐藏了底层硬件平台的具体细节，且该层是用与具体硬件平台相关的代码实现，执行实际的计算任务。如果用户需要扩展 OpenMM，需要针对特定的平台实现这一层相关的 API，并且该层被上层的 OpenMM 实现层所调用。

## 5.3 多核并行编程模型的研究意义

随着多核计算平台的广泛应用，很多应用领域都面临着这样一个问题：如何将现有的大量串行代码并行化，以便充分利用新的多核硬件平台提高程序的性能。基于并行函数库的编程模型非常适合解决这个问题，它具有以下优点：

（1）降低了并行编程的难度。应用程序员只需调用并行函数库中相应的函数，就可以轻松的将现有的串行程序在做少量修改甚至不做修改的情况下，变成能够在底层硬件平台上高效运行的并行程序。

（2）应用程序的平台无关性。通过设计合理的 API 层次结构，使得底层硬件平台的细节对上层应用程序是透明的。这样当底层硬件平台发生变化时，应用程序无需做任何修改即可在新的平台上高效运行。

（3）实现串行软件的渐进式并行化。并行函数库中的函数都是从某个应用领域中挑选出来的，这样可以采用逐步往里面添加函数的方法来扩充该并行函数库，从而将大的串行软件逐步并行化。

因此，基于并行函数库的编程模型对某些对性能要求较高的领域具有非常重大的实用价值和研究意义。

## 5.4 平台无关的多核并行编程模型

借鉴 OpenMM 的成功经验，我们设计了一个基于库的、平台无关的多核并行编程模型，如图 5.2 所示。该模型包含三个部分：并行函数库、编程执行模型以及底层硬件平台。

并行函数库由两层 API 组成，即上层 API 和下层 API。其中上层 API 采用与硬件无关的代码实现，可供应用程序员直接调用；下层 API 采用与硬件相关的语言编写并采用相应的优化技术进行优化，执行实际的计算任务，并由上层 API 调用。 例如：如果底层的硬件平台是多核 CPU，则下层 API 会用 OpenMP 或者 Pthreads 开发；如果底层的硬件平台是 GPU，则下层 API 会用 NVIDIA CUDA 或 OpenCL 开发；如果底层的硬件平台是 Cluster，则下层 API 会用 MPI 开发。应当注意的是，下层 API 的不同实现对上层 API 是透明的。如果我们的并行函数库要同时支持多核 CPU、GPU、Cluster 等不同的平台，则针对每个硬件平台，下层 API 都要有一个实现版本，而上层 API 是唯一的且保持不变。这样即使将来出现了某个新的硬件平台，只要我们针对该平台开发出一套下层 API，则上层应用程序无需做任何修改就可以在该平台上高效运行。

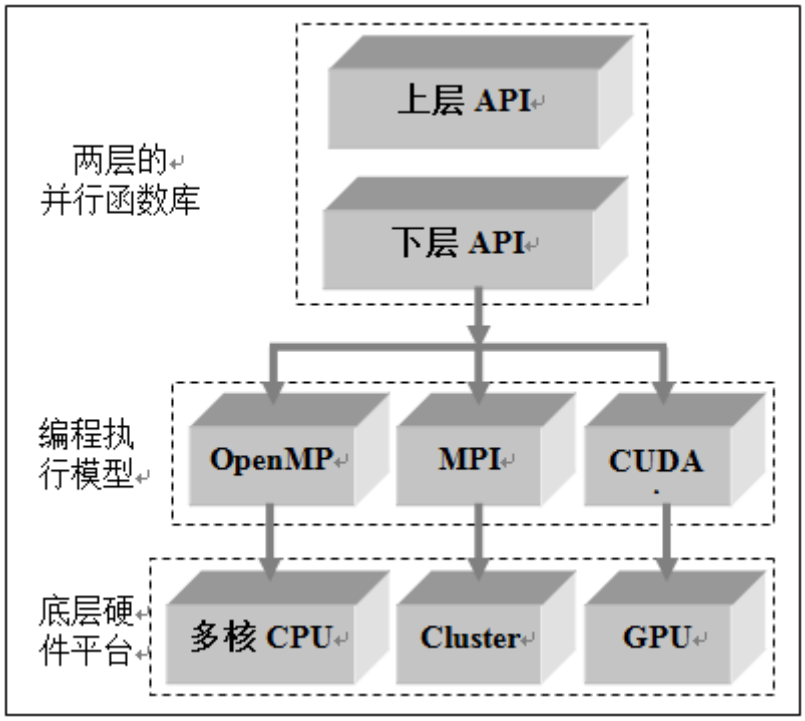


**图 5.2** 平台无关的多核并行编程模型

为了支持这种可扩充性，我们将采用 C++语言来实现并行函数库，将相关的下层 API 子集设计成为一个个抽象类的成员函数（接口），针对每一个硬件平台都有一个继承自抽象类的具体子类，在该子类中实现抽象类中的接口（下层

API)。当上层 API 调用下层 API 时由 C++的多态性在运行时选择与当前硬件平台匹配的下层 API 版本。

针对某个具体的应用领域，可以将某些非常耗时且并行度较高的函数提取出来，设计一个适合的函数接口并针对底层的硬件平台将其并行化，这样程序员只需调用该并行版本的函数就可将原来的串行程序并行化。

## 5.5 原型系统及实验验证

### 5.5.1 面向科学计算的原型系统

为了验证我们提出的平台无关的多核并行编程模型的可行性，我们在 linux 环境下初步实现了一个面向科学计算的并行函数库原型系统。该原型系统包含三个针对不同硬件平台所开发的用于进行矩阵乘法的并行函数库。

目前，该并行函数库只有一个上层 API：

```
void MatrixMul ( float *C, float *A, float *B, int hA, int wA, int wB );
```

该函数用来实现矩阵乘法：C=AxB，参数 hA、wA、wB 分别表示矩阵 A 与 B 的维度。

下层的 API 有三个，即针对每个硬件平台都有一个下层 API：

```
void CPU_MatrixMul ( float *C, float *A, float *B, int hA, int wA, int wB );
void OMP_MatrixMul ( float *C, float *A, float *B, int hA, int wA, int wB );
__global__ void MatrixMulKernel (float *C, float *A, float *B, int hA, int wB);
```

上层 API 独立于具体的硬件平台，而下层 API 则采用与硬件相关的代码实现，且上层 API 需要调用下层 API 来实现其功能。为了对用户完全屏蔽底层 API 的复杂实现，我们将每个底层 API 分别编译成一个动态链接库：libcpu_matrixmul.so、libomp_matrixmul.so、libgpu_matrixmul.so。 其中：

- libcpu_matrixmul.so：是一个串行版本的矩阵乘法，针对单核 CPU 平台，主要用于验证其他并行版本的程序的正确性和性能。
- libomp_matrixmul.so：是一个针对具有共享存储结构的多核 CPU 平台且使用 OpenMP 实现的并行矩阵乘法。
- libgpu_matrixmul.so：是一个用 CUDA 编程模型实现的运行在 GPU 上并行矩阵乘法。

这样用户只需在编译自己的程序时，根据具体的硬件平台来选择对应的动态链接库就可以调用并行版本的矩阵乘法。下面简单介绍一下该并行函数库的使用

方法。假设用户原有的串行源代码 matrixmul.c 或者 matrixmul.cu 如下：

```
#include <stdio.h>
#include “MatrixMul.h”
int main(void) {
    //为矩阵 A、B、C 分配内存空间
    //初始化矩阵 A、B
    MatrixMul(C, A, B, HA, WA, WB); //调用该函数计算 C = A x B
    return 0;
}
```

如果用户的硬件平台是 GPU，则按如下方式进行编译：

nvcc –o gpu_matrixmul matrixmul.cu **./libgpu_matrixmul.so**

就可以使原本串行的程序在执行时变成并行的版本。其他类型的硬件平台的编译过程类似，只是使用的编译器和动态链接库不同罢了。

### 5.5.2 实验结果

为了检验并行函数库中库函数的性能，我们对不同大小的矩阵分别测试了每个版本的库函数的执行时间，实验结果如表 5.1 所示。

**表 5.1** 并行函数库性能测试（单位：ms）

| 矩阵 A 的大小 | 矩阵 B 的大小 | 串行程序的执行时间 | OpenMP 版本的库函数的执行时间 | GPU 版本的库函数的执行时间 |
|---|---|---|---|---|
| 640x480 | 480x800 | 5845.9 | 2783.8 | 120.0 |
| 1280x960 | 960x1600 | 40590.4 | 15611.7 | 165.9 |
| 1600x1280 | 1280x1920 | 140258.6 | 50092.4 | 251.5 |

可见，GPU 版本的矩阵乘法加速效果非常明显，最高加速比达到 557 倍。而 OpenMP 版本的矩阵乘法只比串行程序块 2~3 倍。

## 5.6 本章小结

并行编程模型是软件开发者与多核硬件平台之间的桥梁。一个设计合理、使用方便、清晰高效的并行编程模型能够大大降低并行编程的难度。本章我们首先对 OpenMM 进行了简单介绍，并分析了这种基于库的并行编程模型的优点。然

后借鉴 OpenMM 的成功经验，我们设计了一个基于库的、平台无关的多核并行编程模型。为了验证该模型的可行性和效率，我们实现了一个简单的用于科学计算的原型系统，并对库函数的性能进行了测试。实验表明，我们提出的并行编程模型不仅使用简单、方便，而且还具有良好的可扩展性。编程人员只需调用与硬件无关的上层 API 函数，并在编译时根据具体的硬件平台选择合适的动态链接库就可以将看似串行的程序变成与底层硬件平台相适应的高效的并行程序。此外，当底层的硬件平台发生变化时，原有的程序无需修改，只需要选择一个与新的硬件平台匹配的动态链接库并重新编译一次就可以了。

当然，我们实现的原型系统还很粗糙，比如：只有一个库函数，支持单精度浮点数的矩阵乘法等，有待进一步的完善。

# 第6章 总结

## 6.1 本文工作

本文围绕 CPU-GPU 异构平台的性能优化、应用及多核并行编程模型等方面展开研究，主要做了以下工作：

（1）CPU-GPU 异构平台的性能优化方法的研究

利用 CPU-GPU 异构系统进行计算会遇到很多性能瓶颈，例如：负载均衡、同步与延迟、数据局部性、任务划分等。如果这些问题处理不当，程序的性能可能相差几倍甚至一个数量级。因此采用合适的优化方法对异构系统上的程序进行性能优化至关重要。

在对 CUDA 编程模型从硬件体系结构、软件模型以及编程规范等方面进行详细介绍之后，仔细地分析了影响 CUDA 程序性能的关键因素，包括：访存延迟、负载分配以及全局同步开销。在介绍了已有的优化方法的基础上，我们设计了一种使用原子函数实现不同线程块之间同步的优化方法和优化策略。针对每一种优化方法都进行了实验验证和理论分析，其中我们设计的使用原子函数实现不同线程块之间同步的方法比现有的重新启动内核函数的方法要快 4~5 倍。

（2）CPU-GPU 异构计算平台在生物信息学中的应用

生物信息学领域的许多问题，如：蛋白质折叠、序列比对、蛋白质结构预测等都具有计算规模大、并行度高的特点。这些特点使得 CPU-GPU 异构计算平台在该领域有广泛的应用前景。

为了进一步验证各种优化方法的效果以及完整的介绍在 CPU-GPU 异构上进行程序开发的基本流程（包括：算法设计、编程实现、性能优化等），我们以解决生物信息学中的 DNA 或蛋白质局部序列比对问题为例，在 CPU-GPU 异构平台上设计并实现了基于列并行的 Smith-Waterman 算法，综合运用多种优化方法进行优化后的并行程序获得了平均 37 倍的加速比。

（3）平台无关的并行编程模型的研究

尽管 CUDA 编程模型极大的降低了 CPU-GPU 异构平台编程的难度，但对于大多数串行程序开发者来说，其开发门槛还是相对较高，而且当底层的硬件平台发生变化时，软件开发者又要学习一种新的编程模型并针对新的硬件平台重新改写已有的程序，这无疑加重了程序员的负担。因此设计一种新的、平台无关的多核并行编程模型以降低包括 CPU-GPU 异构计算在内的并行编程的难度，使串行程序开发者也能轻松地利用各种多核硬件平台（多核 CPU、GPU、Cell 等）加速

已有的应用程序具有重要意义。

在深入分析了 OpenMM 并行编程框架之后，我们设计了一个基于库的、平台无关的多核并行编程模型。为了验证该模型的可行性、效率以及易用性，我们实现了一个面向科学计算的原型系统，并进行了测试，其中基于 CPU-GPU 异构平台的矩阵乘法加速比达 557 倍，基于多核 CPU 平台的 OpenMP 版本的矩阵乘法也有 2~3 倍的加速比。通过设计合理的 API 层次结构，对上层用户屏蔽了底层硬件的具体细节，用户只需在编译时根据具体的底层硬件平台选择相应的动态链接库就可以将原来的串行程序变成高效的并行程序。

## 6.2 本文成果

本文的成果包括以下三个方面：

（1）设计了一种使用原子函数实现 CUDA 程序中不同线程块之间同步的方法，该方法比现有的重新启动内核函数的方法快 4~5 倍。系统的介绍和总结了 CPU-GPU 异构计算平台的性能优化方法和优化策略，并进行了实验验证和理论分析。

（2）在 CPU-GPU 异构计算平台上，设计并实现了一种新的、基于列并行的 Smith-Waterman 算法，优化后的并行程序平均加速比达 37 倍。

（3）设计了一种基于库的、平台无关的多核并行编程模型，并实现了一个简单的面向科学计算的原型系统，对该模型的可行性、效率和易用性进行了实验验证。

## 6.3 进一步的工作

接下来的工作可以围绕以下三个方面展开。

（1）进一步探索新的 CUDA 程序性能优化方法。GPU 的硬件体系结构和计算能力发展非常迅速，这样原有的优化方法很可能已经不再适用同时也可能会出现新的性能瓶颈。

（2）进一步推广和扩大 CPU-GPU 异构计算平台在生物信息学以及其他高性能计算领域的应用。例如：多重序列比对问题、单体型推导问题、motif 发现问题以及人工智能领域的 3sat 问题等等。

（3）进一步完善我们提出的基于库的、平台无关的多核并行编程模型。现有的原型系统由于采用 C 语言实现，只能支持一种数据类型，接下来我们将会采用 C++的模板库来实现，以便支持任意的数据类型。我们还会向并行函数库中添加更多的库函数，如 FFT、BLAS 等函数。

# 参考文献

[1] NVIDIA Corporation, CUDA Programming Guide Version 2.0 (July 2008)

[2] http://www.nvidia.com/cuda

[3] Justin Hensley, AMD CTM Overview, ACM SIGGRAPH 2007

[4] http://baike.baidu.com/view/2932264.htm

[5] http://cuda.csdn.net/News.aspx?id=6ab05c8f-c260-4f0a-9e25-663993387e44

[6] http://it.people.com.cn/GB/42891/203889/12982137.html

[7] 王少荣，孙晓鹏，刘丽艳，刁麓弘，李华，基于图形处理器的通用计算技术，中国科学院计算技术研究所智能信息处理实验室

[8] 高小鹏，龙翔，万寒，倪璠，通用计算中的 GPU，中国计算机学会通讯，5（11），2009.11，pp43-48

[9] 吴恩华，柳有权，基于图形处理器(GPU)的通用计算，计算机辅助设计与图形学学报，16(5)，2004，pp601-61

[10] 吴恩华，图形处理器用于通用计算的技术、现状及其挑战，软件学报，2004，15(10)，pp206-214

[11] 邓仰东，NVIDIA CUDA 超大规模并行程序设计训练课程，清华大学微电子学研究所

[12] http://www.iworkstation.com.cn/news/info/2009-10-29/3918.html

[13] OpenCL:Introduction and Overview，June 2010.

[14] A Munshi, Opencl: Parallel computing on the gpu and cpu, ACM SIGGRAPH 2008, 2008

[15] http://www.ks.uiuc.edu/Research/gpu/

[16] http://folding.stanford.edu/

[17] Liu, W., Schmidt, B., Voss, G., Streaming algorithms for biological sequence alignment on GPUs. IEEE Trans. Parallel and Distributed Systems 18(9), 1270–1281 (2007).

[18] Liu, W., Schmidt, B., Voss, G.: GPUClustalW: Using Graphics Hardware to Accelerate Multiple Sequence Alignment. In: Proc. 13th Ann. IEEE Int’l Conf. High Performance Computing (HiPC 2006), pp. 363－374 (2006)

[19] Sinha S, Tompa M, A statistical method for finding transcription factor binding site, Proceedings of the Eighth International Conference on Intelligent Systems on Molecular Biology, San Diego, CA 2000:344-354.

[20] Thijs G, Marchal K, Moreau Y, A Gibbs sampling method to detect over-represented motifs in upstream regions of coexpressed genes. RECOMB 2001, 5:305-312.

[21] C. Lawrence and A. Reilly, An expectation maximization (EM) algorithm for the dentication and characterization of common sites in unaligned biopolymer sequences, Proteins: Structure, Function and Genetics.7: 41-51, 1990.

[22] Yuzhong Zhao, Yun Xu, Zhihao Wang, Hong Zhang, Guoliang Chen, A better block partition and ligation strategy for individual haplotyping, Bioinformatics, 2008, 24(23): pp2720-2725

[23] Chou P Y and Fasman G D. Prediction of protein conformation, Biochemistry, 1974, 13(2): pp222-245

[24] Ding Z, Filkov V and Gusfield D. A linear-time algorithm for perfect phylogeny haplotyping, Journal of Computational Biology, 2006, 13(2): pp522-553

[25] M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, V. S. Pande. “Accelerating Molecular Dynamic Simulation on Graphics Processing Units.” J. Comp. Chem., 30(6): 864-872 (2009)

[26] P. Eastman and V. S. Pande. OpenMM: A Hardware-Independent Framework for Molecular Simulations. Computing in Science and Engineering. Vol. 12, No. 4: 34-39, July/August 2010

[27] Smith, T., Waterman, M.: Identification of Common Molecular Subsequences. J.Molecular Biology 147, 195－197 (1981)

[28] NVIDIA Corporation, The CUDA Compiler Driver NVCC, 2008

[29] http://www.inpai.com.cn/doc/hard/85019_12.htm

[30] Weiguo Liu, Bertil Schmidt, Gerrit Voss and Wolfgang Müller-Wittig，“Molecular Dynamics Simulations on Commodity GPUs with CUDA”，High Performance Computing（HiPC 2007），LNCS 2007, Volume 4873/2007, p185-196

[31] Peter Bakkum and Kevin Skadron，“Accelerating SQL Database Operations On a GPU with CUDA”，GPGPU'10 Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units，ACM New York, NY, USA ©2010

[32] http://tech.163.com/digi/09/1030/11/5MSCN11M001618JV.html

[33] 张舒，褚艳丽，赵开勇，张钰勃，GPU 高性能运算之 CUDA，北京：中国水利出版社，2009.10

[34] http://www.nvidia.com/object/cuda_gpus.html

[35] NVIDIA Corporation, CUDA Programming Model Overview, 2008

[36] Shane Ryoo, Christopher Rodrigues, Sara Baghsorkhi, Sam Stone, David Kirk, Wen-mei Hwu, Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2008.

[37] Ino, F., Kotani, Y., Hagihara, K.: Harnessing the Power of idle GPUs for Acceleration of

Biological Sequence Alignment. In: IEEE International Symposium on Parallel & Distributed Processing, 2009. IPDPS 2009, pp. 1–8 (2009).

[38] http://suif.stanford.edu/

[39] Rebecca L. Collins, Bharadwaj Vellore, and Luca P. Carloni, Recursion-DrivenParallel Code Generation for Multi-Core Platforms. Proceedings of the Conference on Design, Automation and Test in Europe.

[40] https://simtk.org/home/openmm

[41] David A. Bader, Varun Kanade and Kamesh Madduri. SWARM: A Parallel Programming Framework for Multicore Processors. IEEE International Parallel and Distributed Processing Symposium，2007.

[42] OpenMM Project of Stanford University, OpenMM Users Manual and Theory Guide, 2010.7

[43] Stone, S.S., Haldar, J.P., Tsao, S.C, Hwu, W.W., Liang, Z., Sutton, B.P. Accelerating Advanced MRI Reconstructions on GPUs. In: ACM Computing Frontier Conference 2008.

[44] Quintana-Orti, G., Igual, F.D., Quintana-Qrti, E.S., van de Geijn, R., Solving Dense Linear Systems on Platform with Multiple Hardware Accelerators. In: PPoPP, p121-129 (2009).

[45] Anguo Ma, Jing Cai, Yu Cheng, Xiaoqiang Ni, Yuxing Tang, and Zuocheng Xing. Performance Optimization Strategies of High Performance Computing on GPU, The 8th International Conference on Advanced Parallel Processing Technology (APPT 2009), Vol. 5737Springer(2009), p150-164.

[46] 陈国良、安红、陈崚、郑启龙等编著.并行算法实践. 北京：高等教育出版社，2004

[47] 多核系列教材编写组 编著. 多核程序设计. 北京：清华大学出版社，2007

[48] W. Richard Stevens Stephen A. Rago 著，尤晋元、张亚英、戚正伟译. UNIX 环境高级编程（第二版）. 北京：人民邮电出版社，2006

[49] Wooyoung Kim, Michael Voss. “Multicore Desktop Programming with Intel Threading Building Blocks”, IEEE Software, Volume 28 Issue 1, January 2011.

[50] Vijay Saraswat, Vivek Sarkar and Christoph von Praun, “X10: Concurrent Programming for Modern Architectures”，12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP’07)

[51] Mario Leyton and M. Piquer, “Skandium: Multi-core Programming with Algorithmic Skeletons”, 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), p289 – 296

[52] Gregor Hohpe, Bobby Woolf. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison Wesley. October 10, 2003. 0-321-20068-3

[53] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.

Communication of the ACM – 50th anniversary issue: 1958-2008. Volume 51 Issue 1, January 2008.

[54] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02319

[55] Kurt Keutzer Niraj Shah, William Plishker. Np-click: A programming model for the intel ixp1200. In 2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9), Anaheim, CA, February 2003.

# 附录 1　插图索引

# 附录 2 表格索引

# 致　谢

时光飞逝，转眼间三年的研究生生涯即将结束，回首这三年来我在中国科学技术大学的生活和研究工作，感慨万分，那一幕幕感人至深的画面又重新浮现在脑海里。我能顺利的完成硕士研究生阶段的学业与身边的老师、同学以及父母和亲人的帮助和鼓励是密不可分的。在即将毕业离校之际，我要向以下单位和个人表示真诚的感谢。

感谢计算机科学与技术学院三年前录取了我，使我有机会进入中国科学技术大学这个学术圣地，感谢国家高性能计算中心（合肥）为我提供了良好的学习和科研环境。

感谢我的导师徐云副教授三年来在生活上对我的关心和照顾，研究工作上给予的耐心指导和鼓励以及教会了我很多做人的道理。徐老师严谨细致、一丝不苟、坚持不懈的科研精神和学术态度使我深受感染，成为我工作和学习上的榜样。同时他渊博的学识和敏锐的洞察力以及爱岗敬业、平易近人、艰苦奋斗的精神也给我留下了难以磨灭的印象。这些宝贵的精神财富必将使我受益终生。此外，我还要感谢郑启龙副教授、班主任苗付友副教授以及学生工作办公室的钱海老师对我的关心和帮助。

感谢实验室的师兄师姐师弟师妹们在生活和科研上对我的关心、帮助和鼓励。他们是：赵裕众博士、余林彬硕士、张弘硕士、吴晓伟硕士、房明硕士、雷一鸣硕士、邵明芝硕士、汪胜硕士、、王向前硕士、姜海涛博士、王颖硕士、邱鹏飞硕士、李文军硕士、晏涛博士、钱立兵硕士、汪睿硕士、卢世贤硕士、曾中意硕士、杨矫云博士、姚晓辉硕士、滕达硕士、陈元硕士、周寰硕士、聂鹏宇硕士、刘家兵硕士、程文华硕士、陈思灵硕士、冯玉谦硕士、付和平硕士等，与他们的相识与合作将成为我人生的重要经历和美好回忆。

在科大三年的学习、生活中，我还结识了很多来自五湖四海的同学和朋友。我们曾一起畅谈人生理想，一起讨论科研学术问题，也曾一起天南海北、胡乱调侃。他们不仅增添了我生活的乐趣，还给了我很多的信心和帮助。在此我要向闫德莹硕士、艾国红硕士、王峰硕士、曹仁之硕士、周桂芳硕士、李明硕士、张欣硕士、汪志宾硕士等表示感谢。

我还要感谢我的父母和姐姐。父母不仅将我抚养成人还供我读书、教我做人，没有他们就没有我今天的学业和成果。在我成长的过程中，父母和姐姐对我无微不至的关爱、热情鼓励和大力支持给了我无穷的信心和力量，使我能够一直前进。

最后，感谢所有被本文所引用的参考文献的作者，他们的研究成果对我的研究工作提供了重要帮助。

# 在读期间发表的学术论文

## 已发表论文：

[1] Bo Chen, Yun Xu, Jiaoyun Yang, Haitao Jiang. A New Parallel Method of Smith-Waterman Algorithm on a Heterogeneous Platform. In: The 10th International Conference on Algorithms and Architectures for Parallel Processing, Busan, Korea on May 21 ~ 23, 2010. (ICA3PP2010), pp. 79-90

# 在读期间参与的科研项目

[1] 国家 863 重点项目子课题，多核龙芯处理器系统软件移植与开发（2008AA010902），2008-2010.

[2] 华为创新研究计划项目，OpenMM 并行编程框架剖析及其典型行业应用软件并行化研究（IRP-1010-07-03），2011.