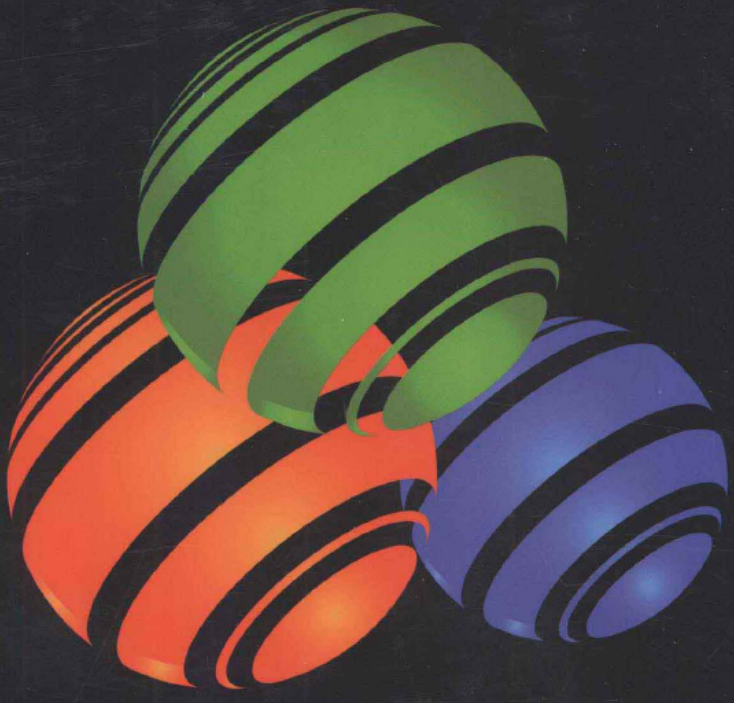
HZ BOOKS
华章科技

CUDA开发者社区技术总监亲自撰写，英伟达中国首批CUDA官方认证工程师翻译，译著双馨

全面、详实地讲解了CUDA并行程序设计的技术知识点和编程方法，包含大量实用代码示例，是目前学习CUDA编程最权威的著作之一

ELSEVIER
爱思唯尔

高性能计算系列丛书



CUDA Programming
A Developer's Guide to Parallel Computing with GPUs

# CUDA并行程序设计
## GPU编程指南

（美）Shane Cook 著
苏统华 李东 李松泽 魏通 译 马培军 审校

机械工业出版社
China Machine Press

# CUDA并行程序设计
## GPU编程指南

**CUDA Programming**
**A Developer's Guide to Parallel Computing with GPUs**

本书旨在为读者学习CUDA打下坚实基础。涵盖如何理解串行程序并把它转化到单指令流多数据流编程模型下，以及如何基于CUDA实现高效的并行程序。本书除了提供多个实例，还深入讨论了GPU程序的优化，同时对共享内存、内存预读和线程的使用均有涉及。

—— Nagarajan Kandasamy博士，德雷塞尔大学电子与计算机工程系副教授

CUDA是一种专门为提高并行程序开发效率而设计的计算架构。在构建高性能应用程序时，CUDA架构可充分发挥GPU的强大计算功能，颇受广大开发者拥趸。本书以并行编程实践者视角，展示了全面、快速提升CUDA程序效能的途径。从并行机制到CUDA开发环境搭建，从GPU高性能计算相关硬件知识到并行计算和CUDA编程技巧，从核心概念到多个热点技术和高级主题，无所不包。本书侧重于CUDA实践应用，特别在CUDA程序优化上下足了工夫，例如发挥多GPU效力。难能可贵的是，作者结合多年实践经验，精心总结出开发者在使用CUDA过程中易犯的错误和有效的规避方法。

本书亮点：

- 全面介绍并行机制和CUDA编程，即便没有CUDA编程经验也能够轻松掌握；
- 细致指导，以帮助读者优化CUDA应用程序；
- 以实用技术演示如何在并行程序中管理内存、线程、算法和各种资源；
- 涵盖英伟达硬件上三大操作系统的CUDA开发；
- 每章附有练习，方便检验读者的掌握程度或为课堂提供讨论要点。

本书译自原版CUDA Programming: A Developer's Guide to Parallel Computing with GPUs并由Elsevier授权出版

ELSEVIER 爱思唯尔

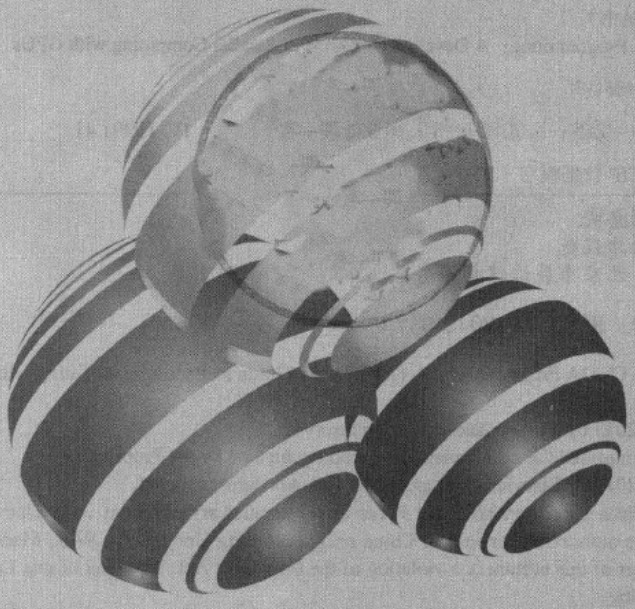
ELSEVIER

投稿热线：(010) 88379604
客服热线：(010) 88378991 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/程序设计
ISBN 978-7-111-44861-7
9 787111 448617 >
定价：99.00元

算系列丛书



CUDA Programming
A Developer's Guide to Parallel Computing with GPUs

# CUDA并行程序设计

## GPU编程指南

（美）Shane Cook 著
苏统华 李东 李松泽 魏通 译 马培军 审校

机械工业出版社
China Machine Press

**图书在版编目（CIP）数据**

CUDA 并行程序设计：GPU 编程指南 /（美）库克（Cook, S.）著；苏统华等译；马培军审校 . —北京：机械工业出版社，2014.1
（高性能计算系列丛书）
书名原文：CUDA Programming：A Developer's Guide to Parallel Computing with GPUs

ISBN 978-7-111-44861-7

I. C… II. ①库… ②苏… ③马… III. 图像处理—程序设计 IV. TP391.41

中国版本图书馆 CIP 数据核字（2013）第 276497 号



本书是 CUDA 并行程序设计领域最全面、最详实和最具权威性的著作之一，由 CUDA 开发者社区技术总监亲自撰写，英伟达中国首批 CUDA 官方认证工程师翻译，详实地讲解了 CUDA 并行程序设计的技术知识点（平台、架构、硬件知识、开发工具和热点技术）和编程方法，包含大量实用代码示例，实践性非常强。

全书共分为 12 章。第 1 章从宏观上介绍流处理器演变历史。第 2 章详解 GPU 并行机制，深入理解串行与并行程序，以辩证地求解问题。第 3 章讲解 CUDA 设备及相关的硬件和体系结构，以实现最优 CUDA 程序性能。第 4 章介绍 CUDA 开发环境搭建和可用调试环境。第 5 章介绍与 CUDA 编程紧密相关的核心概念——网格、线程块与线程，并通过示例说明线程模型与性能的关系。第 6 章借助实例详细讲解了不同类型内存的工作机制，并指出实践中容易出现的误区。第 7 章细述多任务的 CPU 和 GPU 协同，并介绍多个 CPU/GPU 编程秘技。第 8 章介绍如何在应用程序中编写和使用多 GPU。第 9 章详述 CUDA 编程性能限制因素、分析 CUDA 代码的工具和技术。第 10 章介绍编程实践中的库与软件开发工具包。第 11 章讲解如何设计基于 GPU 的系统。第 12 章总结 CUDA 应用中易犯错误以及应对建议。

机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码 100037）
责任编辑：肖晓慧
藁城市京瑞印刷有限公司印刷
2014 年 1 月第 1 版第 1 次印刷
186mm × 240mm • 33.5 印张
标准书号：ISBN 978-7-111-44861-7
定 价：99.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88378991 88361066 投稿热线：（010）88379604
购书热线：（010）68326294 88379649 68995259 读者信箱：hzjsj@hzbook.com

# 致中国读者

获悉本书将译为中文，传播于华夏大地，我欣喜不已。中国已跃居高性能计算的超级大国行列。中国的经济发展与GPU市场的发展有些相似：它没有被过去的束缚所羁绊，而是开启了一个新世界。中国建造高性能计算机的实力，足以问鼎美国大型计算机公司。中国的高性能计算明显优于其他对手的原因在于大量采用了GPU技术。GPU技术在中国广泛传播，主要得益于英伟达硬件和CUDA语言，而后者正是本书竭诚呈现的主题。

与很多国家一样，中国的计算领域也经历了变革。陈旧的串行编程模型正在被抛弃，计算领域正在拥抱并行体系结构。GPU拥有大规模并行的处理器核心，这是这一变革的重要组成部分。CPU领域的编程模型必然发生变化，但仅是微小的调整，以便利用GPU加速器的潜力。GPU目前应用到了各行各业。不论是学生还是专业人士，通过阅读本书均会收获良多。本书将帮助你在GPU和大规模并行处理器编程方面一路前行。

Shane Cook

# 译 者 序

我们正在由单核时代进入多核时代和众核时代。在单核时代，软件行业一直享用着“免费的午餐”。受益于CPU主频的指数级提速，开发软件无须任何代码修改，只要换上新一代的处理器，即可获得性能上的飞速提升。随着汹涌而来的众核时代，这里已经“不再有免费午餐”[⊖]。随着计算架构的不断演进，编程模型也发生着深刻的变化。计算机软件行业面临着最大的变迁问题——从串行、单线程的问题求解方式切换到大规模线程同时执行的问题求解方式。而CUDA提供了非常优秀的可扩展架构，以支持这种大规模并行程序设计需求。

本书是一本很出众的CUDA书籍，内容全面而又不落窠臼。全书可以分成四个部分。第一部分为背景篇，包括前4章。其中前两章简述流处理器历史和并行计算基本原理，第3 ~ 4章分别介绍了CUDA的硬件架构与计算能力和软件开发配置。第二部分为CUDA基本篇，包括第5 ~ 7章。第5、6章依次介绍了CUDA线程抽象模型和内存抽象模型，在此过程中，紧密结合直方图统计实例和样本排序实例进行讨论。为了更好地增进读者的实践经验，第7章全方位剖析了AES加密算法的CUDA实现过程。第三部分为CUDA扩展篇，包括第8 ~ 10章。其中第8、9章面向优化执行性能，而第10章为提升开发生产效率。第8章从充分利用多个硬件设备的角度，讲述了流的使用。相反的，第9章从程序优化角度，给出了CUDA性能调优的全方位指导。第10章介绍了一些常用的函数库和CUDA开发包中提供的优质SDK，为大型软件的快速发布提供了支持。第四部分为CUDA经验篇，包括最后的两章。这两章分别针对硬件系统搭建和软件生产过程中的共性问题提供建议，是作者多年CUDA开发经验的总结。

本书特色鲜明。作者在介绍CUDA时，仿佛在跟朋友聊天论道，谈论家常，讲着故事，娓娓道来。论到关键之处，却又语重心长，体贴备至。在不知不觉中，把CUDA的魅力展示得淋漓尽致，同时把CUDA程序设计的功力传授于你。

本书的翻译工作经过精心的组织，整个过程得到大批专业人士的帮助。在交付出版社之前，译者团队经过了全书讨论、初译、初核、再译、再核、审校等六个环节。很荣幸地邀请到在哈尔滨工

⊖ Herb Sutter在2005年发表的论文“The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software”中对这一问题作了前瞻性讨论。——译者注

业大学长年承担“并行程序设计”和“计算机体系结构”等课程教学工作的李东教授，加盟本书的翻译团队。李东教授翻译了第 1 ~ 3 章，对保证本书的翻译质量起到了重要作用。本书第 5 ~ 7 章的初译以及第 9 章部分小节的初译和再译由李松泽负责。本书第 10 章和第 12 章的初译以及第 9 章部分小节的初译和再译由魏通负责。苏统华负责了本书前言、第 4 章、第 8 章、第 9 章部分小节和第 11 章的初译和再译任务，除此之外，他还负责了全书的初核和再核任务。特别感谢哈尔滨工业大学软件学院院长马培军教授，他应邀审校了全部译文，提出了很多中肯的改进意见。本书在交到出版社之后，又得到了机械工业出版社编辑团队的大力帮助，他们的工作专业而细致，让人钦佩。另外还要感谢哈尔滨工业大学软件学院 2012 级数字媒体方向的硕士研究生，参与了部分内容的初译，特别是王烁行同学做了不少工作。如果没有这么多人的辛勤奉献，这本中译本很难如期呈现。另外，国家自然科学基金（资助号：61203260）对本书的翻译提供了部分资助，哈尔滨工业大学创新实验课《CUDA 高性能并行程序设计》也对本书的翻译提供了大力支持。

由于本书涉及面广，很多术语较新，目前尚无固定译法，翻译难度很大。有时，为一个术语选择一个恰当的中文译法，译者经常反复推敲、讨论。但由于译者水平有限，译文中难免存在一些问题，真诚地希望读者朋友们将您的意见发往 cudabook@gmail.com。

苏统华

哈尔滨工业大学软件学院

# 前　言

过去的五年中，计算领域目睹了英伟达（NVIDIA）公司带来的变革。随后的几年，英伟达公司异军突起，逐渐成长为最知名的游戏硬件制造商之一。计算统一设备架构（Compute Unified Device Architecture，CUDA）编程语言的引入，第一次使这些非常强大的图形协处理器为 C 程序员日常所用，以应对日益复杂的计算工作。从嵌入式设备行业到家庭用户，再到超级计算机，所有的一切都因此而改变。

计算机软件界最大的变迁是从串行编程转向并行编程。其中，CUDA 起到了重要的作用。究其本质，图形处理单元（Graphics Processor Unit，GPU）是为高速图形处理而设计的，它具有天然的并行性。CUDA 采用一种简单的数据并行模型，再结合编程模型，从而无须操纵复杂的图形基元。

实际上，CUDA 与之前的架构不同。它不要求程序员对图形或者图形基元有所了解，也不用程序员有任何这方面的知识。你也不一定要成为游戏开发人员。CUDA 语言使得 GPU 看起来与别的可编程设备一样。

本书并不假定读者有 CUDA 或者并行编程的任何经验，仅假定读者有一定的 C/C++ 语言编程知识。随着本书的不断深入，读者将越来越胜任 CUDA 的编程工作。本书包含更高级的主题，帮助你从不知晓并行编程的程序员成长为能够全方位发掘 CUDA 潜力的专家。

对已经熟悉并行编程概念和 CUDA 的程序员来说，本书包含丰富的学习资料。专设章节详细讨论 GPU 的架构，包括最新的费米（Fermi）和开普勒（Kepler）硬件，以及如何将它们的效能发挥到极致。任何可以编写 C 或 C++ 程序的程序员都可以在经过几个小时的简单训练后编写 CUDA 程序。通过对本书的完整学习，你将从仅能得到数倍程序加速的 CUDA 编程新手成长为能得到数十倍程序加速的高手。

本书特别针对 CUDA 学习者而写。在保证程序正确性的前提下，侧重于程序性能的调优。本书将大大扩展你的技能水平和对编写高性能代码的认识，特别是 GPU 方面。

本书是实践者在实际应用程序中使用 CUDA 编程的实用指南。同时我们将提供所需的理论知识和背景介绍。因此，任何人（不管有无基础）都可以使用本书，从中学习如何进行 CUDA 编程。综上，本书是专业人士和 GPU 或并行编程学习者的理想之选。

本书编排如下：

**第 1 章** 从宏观上介绍流处理器（streaming processor）的演变历史，涉及几个重要的发展历程，正是它们把我们带入今天的 GPU 处理世界。

**第 2 章** 介绍并行编程的概念。例如，串行与并行程序的区别，以及如何采用不同的策略寻找求解问题之道。本章意在为既有串行程序员建立一个基本的认识，这里的概念将在后面进一步展开。

**第 3 章** 详尽地讲解 CUDA 设备及与其紧密相关的硬件和架构。为了编写最优性能的 CUDA 程序，适当了解设备硬件的相关知识是必要的。

**第 4 章** 介绍了如何在 Windows、Mac 和 Linux 等不同操作系统上安装和配置 CUDA 软件开发工具包，另外介绍可用于 CUDA 的主要调试环境。

**第 5 章** 介绍 CUDA 线程模型，并通过一些示例来说明线程模型是如何影响程序性能的。

**第 6 章** 我们需要了解不同的内存类型，它们在 CUDA 中的使用方式是影响性能的最大因素。本章借助实例详细讲解了不同类型内存的工作机制，并指出实践中容易出现的误区。

**第 7 章** 主要详述了如何在若干任务中恰当地协同 CPU 和 GPU，并讨论了几个有关 CPU/GPU 编程的议题。

**第 8 章** 介绍如何在应用程序中编写和使用多 GPU。

**第 9 章** 对 CUDA 编程中限制性能的主要因素予以详解，考察可以用来分析 CUDA 代码的工具和技术。

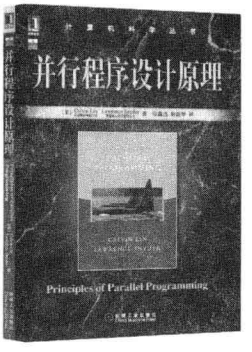
**第 10 章** 介绍了 CUDA 软件开发工具包的示例和 CUDA 提供的库文件，并介绍如何在应用程序中使用它们。

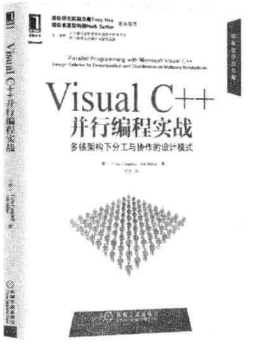
**第 11 章** 关注构建自己的 GPU 服务器或者 GPU 集群时的几个相关议题。

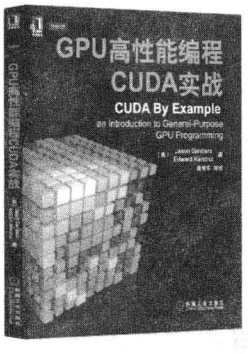
**第 12 章** 检视多数程序员在开发 CUDA 应用程序时易犯的错误类型，并对如何检测和避免这些错误给出了建议。

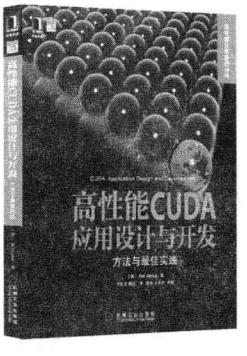
# 推荐阅读

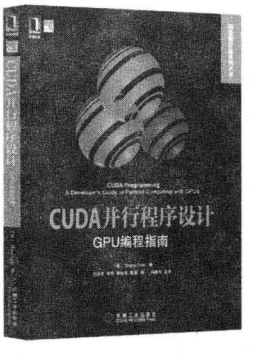

并行程序设计导论
An Introduction to Parallel Programming


并行程序设计原理
Principles of Parallel Programming

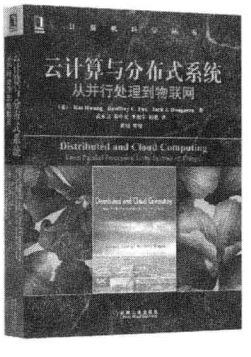

Visual C++
并行编程实战
多核架构下分工与协作的设计模式


GPU高性能编程
CUDA实战
CUDA By Example
an Introduction to General-Purpose
GPU Programming


高性能CUDA
应用设计与开发
方法与最佳实践

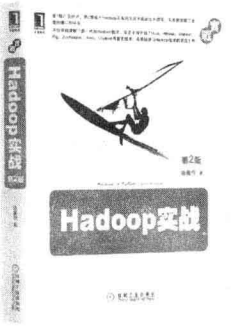

CUDA并行程序设计
GPU编程指南

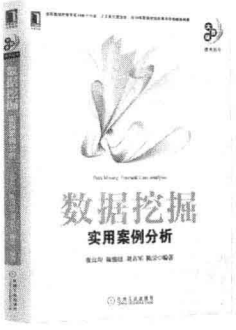

OpenCL编程指南

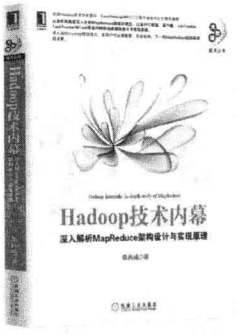

云计算与分布式系统
从并行处理到物联网
Distributed and Cloud Computing

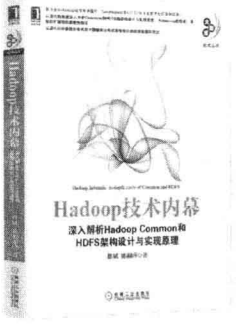

大规模分布式存储系统
原理解析与架构实战

# 推荐阅读

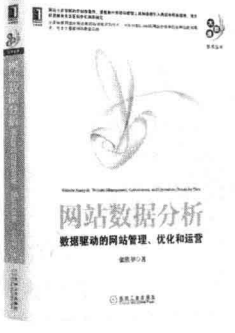







**Hadoop实战（第2版）**
作者：陆嘉恒
ISBN：978-7-111-39583-6
定价：79.00元

**数据挖掘：实用案例分析**
作者：张良均
ISBN：978-7-111-42591-5
定价：79.00元

**Hadoop技术内幕：**
**深入解析MapReduce架构设计与实现原理**
作者：董西成
ISBN：978-7-111-42226-6
定价：69.00元

**Hadoop技术内幕：**
**深入解析Hadoop Common和HDFS架构设计与实现原理**
作者：蔡斌 等
ISBN：978-7-111-41766-8
定价：89.00元

**数据挖掘与数据化运营实战：**
**思路、方法、技巧与应用**
作者：卢辉
ISBN：978-7-111-42650-9
定价：59.00元

**网站数据分析：**
**数据驱动的网站管理、优化和运营**
作者：张洪举
ISBN：978-7-111-43514-3
定价：69.00元

# 目　录

第 1 章

# 超级计算简史

## 1.1 简介

为什么我们会在一本关于 CUDA 的书籍中谈论超级计算机呢？超级计算机通常走在技术发展的最前沿。我们在这里看到的技术，在未来的 5 ~ 10 年内，将是桌面计算机中很普通的技术。2010 年，在德国汉堡举行的一年一度的国际超级计算机大会上宣布，根据 500 强名单（http://www.top500.org），英伟达基于 GPU 的机器在世界最强大的计算机列表中位列第二。从理论上讲，它的峰值性能比强大的 IBM Roadrunner 和当时的第一名 Cray Jaguar 的性能还要高。当时 Cray Jaguar 的性能峰值接近 3 千万亿次。2011 年，采用 CUDA 技术的英伟达 GPU 仍然是世界上最快的超级计算机。这时大家突然清楚地认识到，与简陋的桌面 PC 一起，GPU 已经在高性能计算领域达到了很高的地位。

超级计算是我们在现代处理器中看到的许多技术的发展动力。由于对用更快的处理器来处理更大数据集的需求，工业界不断生产出更快的计算机。正是在这些发展变化中，GPU 的 CUDA 技术走到了今天。

超级计算机和桌面计算正在向着异构计算发展——人们试图通过将中央处理器（Central Processor Unit，CPU）和图形处理器（Graphics Processor Unit，GPU）技术混合在一起来实现更高的性能。使用 GPU 的两个最大的国际项目是 BOINC 和 Folding @ Home，它们都是分布式计算的项目。这两个项目使得普通人也能为具体的科学项目做出真正的贡献。在项目中，采用 GPU 加速器的 CPU/GPU 主机的贡献远远超过了仅装备 CPU 主机的贡献。截至 2011 年 11 月，大约 550 万台主机提供了约 5.3 千万亿次的计算性能，这将近是 2011 年世界上最快的超级计算机（日本富士通的“京（K）计算机”）计算性能的一半。

作为美国最快的超级计算机 Jaguar 的升级换代产品，命名为 Titan 的超级计算机计划于 2013 年问世。它将用近 30 万个 CPU 核和高达 18 000 个 GPU 板卡达到每秒 10 ~ 20 千万亿次的性能。正是由于有像 Titan 这样的来自世界各地的大力支持，无论是在 HPC（高性能计算）行业，还是在桌面电脑领域，GPU 编程已经成为主流。

现在，你可以自己“攒”或者购买一台具有数万亿次运算性能的桌面超级计算机了。在 21 世纪初期，这将会使你跻身 500 强的首位，击败拥有 9632 奔腾处理器的 IBM ASCI Red。

这不仅部分地展现了过去十几年计算机技术取得的巨大进步，更向我们提出了从现在开始的未来十几年，计算机技术将发展到何种水平这个问题。你可以完全相信在未来一段时间内，GPU 将位于技术发展的前沿。因此，掌握 GPU 编程将是任何一个优秀开发人员必备的重要技能。

## 1.2 冯·诺依曼计算机架构

几乎所有处理器都以冯·诺伊曼提出的处理结构为工作基础，冯·诺伊曼被认为是计算之父之一。在该结构中，处理器从存储器中取出指令、解码，然后执行该指令。

现代处理器的运行速度通常高达 4GHz。现代 DDR-3 内存，与标准的英特尔 I7 设备配合使用时，可以在运行任何程序时最高达到 2GHz 的速度。然而，在一个 I7 设备中至少具有四个处理器或内核。如果你认为超线程能力只能作为一个真正的处理器，那么一个 I7 设备中也是两个处理器。

在一个 I7 Nehalem 系统中，三通道 DDR-3 内存具有的理论带宽如表 1-1 所示。受主板和确切的内存模式影响，实际带宽可能要小很多。

表 1-1 I7 Nehalem 处理器带宽

| QPI 时钟频率 | 理论带宽 | 单核带宽 |
|---|---|---|
| 4.8GT/s（标准配置） | 19.2GB/s | 4.8GB/s |
| 6.4GT/s（最大配置） | 25.6GB/s | 6.4GB/s |

注意：QPI 指快速通道互联（Quick Path Interconnect）

当考虑处理器的时钟速度时，你遇到的第一个问题是关于内存带宽的。用速度为 4GHz 的处理器，你可能每个时钟周期需要取来一条指令（操作码）和某个数据（操作数）。

通常，每个指令长 32 位。所以，假设你在每个核上只执行一组不带数据的、顺序执行的指令，则每秒钟你需要取来 4.8GB/s ÷ 4=1.2GB 条指令。这是假设处理器平均每个时钟周期执行一条指令的情况[⊖]。不过，通常你还需要读取和写回数据，这里假设数据与指令的比例是 1：1，那么这意味着我们的实际吞吐量将减少一半。

内存速度和时钟速度的比率是限制 CPU 和 GPU 吞吐量的一个重要因素，这一点我们将在后续的章节中讨论。深入分析你就会发现，除了 CPU 和 GPU 中的一些例外，大多数应用程序属于“内存受限型”[⊜]，而不是“处理器时钟周期或负载受限型”。

CPU 厂商试图通过使用缓存和突发内存访问来解决这个问题，这利用了程序的局部性原理。下面是一个典型的 C 程序，请仔细观察该函数中相关操作的类型：

```
void some_function
{
 int array[100];
 int i = 0;
```

⊖ 实际达到的执行速度可能大于或小于 1，这里我们做了简化处理。

⊜ 即性能提高受到内存速度的限制。——译者注

```
for (i=0; i<100; i++)
{
 array[i] = i * 10;
}
}
```

让我们看看处理器是如何实现的。首先，数组 array 的地址被装入某个内存访问寄存器，然后参数 i 被装入到另一个寄存器。循环退出条件为 100，这个退出条件既可以装入一个寄存器，也可以编码的形式，成为指令流中的一个字面值。这时，计算机将重复执行这几条指令，循环 100 次。对于每一个计算出的值，我们需要取来并执行“控制指令”、“访存指令”和“计算指令”。

这显然是低效的，因为计算机执行的是相同的指令，只是处理的数据不同。因此，硬件设计者就为几乎所有的处理器设置了容量很小的缓存，并且在更复杂的处理器中，设置多级缓存（如图 1-1 所示）。当需要从内存中取数据或指令时，处理器首先查询缓存。如果数据或指令在缓存中，则高速缓存直接将其交给处理器。

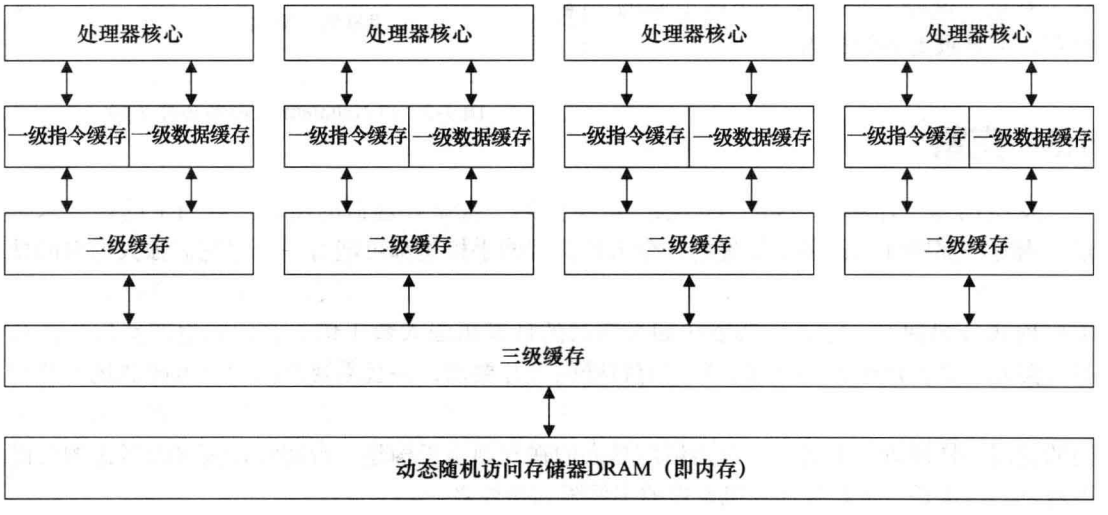



图 1-1 典型的现代 CPU 缓存组成结构

如果数据不在一级缓存（L1）中，则处理器向二级或三级（L2 或 L3）缓存发出读取请求。如果缓存中没有此数据，则需要从主存（即内存）中读取。一级缓存的工作速度通常能达到或接近处理器的时钟速度。因此，假设写入和读取都是在缓存中完成，则循环的执行就很有可能接近处理器全速。然而，这是有一定成本的：一级缓存的大小，通常只有 16K 或 32KB 大小。二级缓存就要慢些，但空间会大些，通常约为 256K。三级缓存则要大得多，通常几兆字节大小，但是比二级缓存要慢得多。

在实际程序中，循环程序往往非常大，可能达几兆字节大小。即便是程序可以存放在缓存中，但数据集往往就存不下了。所以尽管设置了缓存，处理器仍然会因为受到内存吞吐率或带宽的限制，而无法发挥其所具有的处理能力。

当处理器从缓存而不是主存中取来一条指令或一个数据时，称为缓存命中（cache hit）。但是随着缓存容量的不断增大，使用更大容量缓存所带来的增速收益却迅速下降。这意味着，现代处理器中使用大容量缓存并不是提高性能的一个有效办法，除非有办法将问题的整个数据集都装入缓存。

英特尔I7-920处理器有8MB的片内三级缓存。这个缓存的设计是有代价的，看一下英特尔I7处理器的模型，我们会知道三级缓存占用了约30%左右的芯片面积（如图1-2所示）。

随着缓存容量的增长，用于制造处理器的硅片的物理尺寸也逐渐变大。芯片越大，制造成本越昂贵，在制造过程中因发现差错而将芯片丢弃的可能性也就越高。尽管通过屏蔽掉有缺陷的核，这些有生产缺陷的芯片可以当作三核或双核芯片低价售出。但是对最终用户而言，容量不断增大而效率却不断下降的缓存，会导致更高的成本。

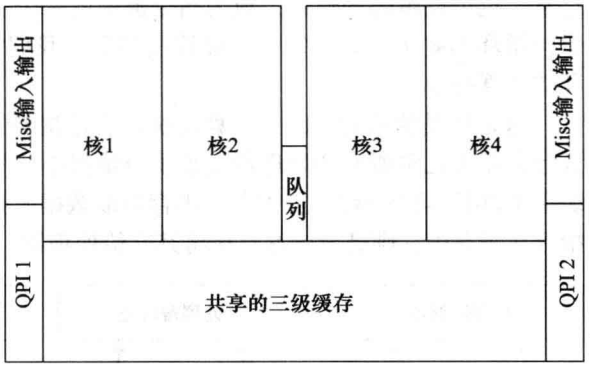



图1-2 I7 Nehalem处理器芯片布局

## 1.3 克雷

众所周知，计算革命始于20世纪50年代第一批微处理器的出现。以当下的标准来衡量，当时的那些计算设备是很慢的，今天你的智能手机里都可能有一个比它们强大得多的处理器。然而，正是它们导致了超级计算机的诞生。这些超级计算机通常为政府部门、大型学术机构或公司拥有。它们的功能比如今普通的计算机强大数千倍。它们的生产费用高达几百万美元，会占用庞大的空间，通常有特殊的冷却要求，并且需要专门的技术团队进行管理和维护。它们的运行需要消耗大量的电能，从某种程度上说，它们每年的运行费用和生产它们的费用一样昂贵。事实上，电力供应是人们在规划是否构建一台超级计算机时考虑的关键因素之一，也是当下超级计算机发展的主要限制条件之一。

现代超级计算机的创始人之一是西蒙·克雷（Seymour Cray），他主持设计的Cray-1超级计算机由克雷研究所（Cray Research）于1976年制造完成。在Cray-1的内部，所有的器件都是通过成千上万条独立的电线连接在一起。为此，他们聘请了许多女工，因为女人的手要比大多数男人的手小些，便于在狭小的空间里将成千上万条电线连接起来。

通常用以小时为单位的正常运行时间（两个停机故障之间的实际运行时间）来评价一台超级计算机。如果能让超级计算机持续运行一整天而不出任何问题，那就是一个了不起的成就。按照今天的标准，这似乎很落后。然而，我们今天拥有的很多产品和技术都要归功于克雷和那个时代的其他人所开展的研究。

克雷持续生产了很多以Cray命名的、极具突破性的超级计算机。最初的Cray-1超级计算机耗资880万美元，计算速度达到了160 MFLOPS（每秒百万次浮点运算）。而今天衡量计

算速度的标准是 TFLOPS（每秒万亿次浮点运算），是原来指标 MFLOPS 的一百万倍（$10^{12}$ 与 $10^6$ 对比）。今天，一个费米（Fermi）型 GPU 卡性能的理论峰值已经超过了 1 TFLOPS。

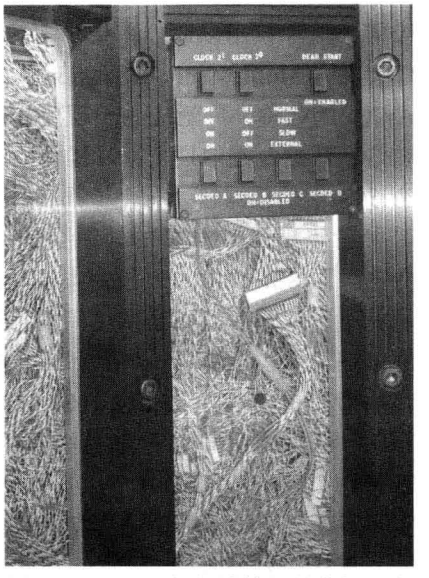


图 1-3 Cray-2 超级计算机的内部连线

Cray-2 在 Cray-1 的基础上有了明显的改进。它采用分成若干内存条的共享内存架构，这些内存条可以与一个、两个或四个处理器相连，进而发展成为今天基于服务器的对称多处理器（Symmetrical MultiProcessor，SMP）系统。在该系统中，多 CPU 共享同一个内存空间。跟那个时代的很多计算机一样，Cray-2 是一台基于向量的计算机（如图 1-3 所示）。在向量机中，一个操作同时处理多个操作数。诸如 MMX、SSE 和 AVX 这样的处理器扩展部分，以及 GPU 设备，它们的核心都是向量处理器。而时至今日，这些向量处理器仍然与早期的超级计算机在设计上有很多相似之处。

Cray 系列超级计算机通过硬件来支持“散布”（scatter）类和“收集”（gather）类操作原语，这些原语在并行计算中非常有用，我们将在后面的章节中介绍。

今天，Cray 系列超级计算机仍然活跃于超级计算机市场。由 Cray 公司研发的、安装在美国橡树岭国家实验室（The Oak Ridge National Laboratory）的超级计算机 Jaguar（http://www.nccs.gov/computingresources/jaguar/），依然位于 2010 年年底发布的世界超级计算机 500 强排行榜中。建议你登录 Cray 公司的网站（http://www.cray.com），了解一下这个伟大公司的历史，它将告诉你计算机发展的内在规律以及我们今天所处的位置。

## 1.4 连接机

早在 1982 年，一家名为 Thinking Machines 的公司提出了一个非常有趣的设计方案，即连接机（Connection Machine，CM）。

这是一个比较简单的概念，却引发了一场现代并行计算机的革命。他们通过一遍又一遍地使用一些简单的零部件创造了一个 16 核的 CPU，并在一台机器上安装了 4096 个这样的设备。这是一个完全不同的概念。它们并不是通过一个高速处理器处理数据集，而是通过 64K 个处理器来完成一个任务。

举一个简单的例子，假如我们要处理一个 RGB（红 Red，绿 Green，蓝 Blue）图像的颜色。其中，每种颜色用单字节表示，用 3 个字节表示 1 个像素的颜色。现在假设我们的问题是要将蓝色值降为零。

假设内存不是交替地存储各个像素点的颜色值，而是被分成红、绿、蓝三条。在传统的处理器中，我们会用一个循环来将蓝色内存条中每个像素的颜色值减 1。这个操作对每个数

据项都是相同的，即每次循环迭代，我们都要对指令流进行取指、译码和执行三个操作。

连接机采用的是单指令多数据（Single Instruction，Multiple Data，SIMD）型并行处理。今天，这种技术以诸如单指令多数据流扩展指令（Streaming SIMD Extensions，SSE）、多媒体扩展（Multi-Media eXtension，MMX）以及高级矢量扩展（Advanced Vector eXtensions，AVX）的名称，广泛应用于现代处理器中。这个技术先定义好一个数据范围，然后让处理器在这个数据范围内进行某种操作。尽管 SSE 和 MMX 是基于一个处理器核的，但连接机却拥有 64K 个处理器核，每个核都在其数据集上执行 SIMD 指令。

诸如 Intel I7 这样的处理器是 64 位的处理器，一次最多可以处理 64 位（即 8 个字节）的数据。而 SSE 的 SIMD 指令集已经扩展到了 128 位。在这样的处理器上运行 SIMD 指令，我们就可以消除所有多余的访问内存取指令的操作，并将内存读 / 写周期变为原来的 1/16，而原来是每次取出和写入 1 个字节。AVX 将这种技术扩展到了 256 位，使其效率更高。

对于分辨率为 1920 × 1080 的高清（High-Definition，HD）视频图像，它的数据大小为 2 073 600 字节，即每种颜色平面约为 2MB。如果使用常规的 SSE/MMX 处理器来处理，将需要约 260 000 个 SIMD 周期。对于 SIMD 周期，我们仅需要一个“读、计算和写”周期。实际需要的处理器时钟数可能会差别很大，这取决于实际的处理器架构。

连接机有 64K 个处理器。因此对于每个处理器，2MB 大小的帧将需要约 32 个 SIMD 的处理周期。显然，这种方法要远远优于现代处理器的 SIMD 方法。不过，有一点需要注意，如果将现在 CPU 采用的粗线程方案迁移至这种采用大规模并行方式处理的机器，处理器之间的同步和通信将是很大的问题。

## 1.5 Cell 处理器

超级计算机的另一个有趣的发展，源于 IBM 公司发明的 Cell 处理器（如图 1-4 所示）。

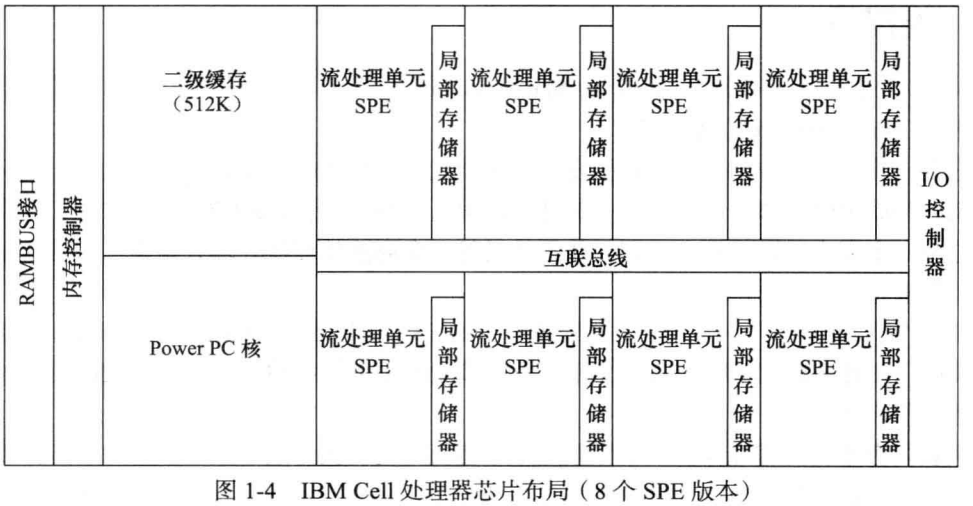



图 1-4 IBM Cell 处理器芯片布局（8 个 SPE 版本）

它的思想是用一个常规处理器作为监管处理器，该处理器与大量的高速流处理器相连。在 Cell 处理器中，常规的 PowerPC（PPC）处理器担任与流处理器和外部世界的接口。而 SIMD 流处理器，IBM 称其为 SPE，则在常规处理器的管理下，处理数据集。

对我们来说，Cell 是一个非常有趣的处理器，因为它与英伟达公司的 G80 和随后生产的 GPU 在设计上很相似。索尼公司也将其应用到游戏产业中的 PS3 控制台机器上，游戏产业是与 GPU 的主要应用领域非常类似的一个领域。

要想为 Cell 处理器编程，你需要写一个在 PowerPC 核心处理器上运行的程序。该程序会用一个互不相同的二进制码，在每个流处理单元 SPE（Stream Processing Element）上，调用执行一个程序。实际上，每个 SPE 本身就是一个核。它可以从自己的本地内存取出一个独立的程序来执行，这个程序与旁边的 SPE 执行的程序是不同的。另外，通过一个共享的互连网络，SPE 之间、SPE 与 PowerPC 核之间可以相互通信。当然，针对这种混合架构进行编程是很困难的。程序员必须从程序和数据两个方面，明确管理 8 个 SPE，以及在 PowerPC 核上运行的串行程序。

由于具有和相应处理器直接对话的能力，因此很多问题的求解就可以通过一系列简单的步骤来实现。例如在前述的 RGB 示例中，PPC 核可以取来一组待处理的数据，然后将其分配给 8 个 SPE 处理。每个 SPE 所做的处理是相同的，即读取一个字节，将其减一，然后存回本地内存。当所有 SPE 完成工作后，PC 核再从每个 SPE 中取回数据，然后将这组数据（或数据条）写入内存区域，整个图像在内存中被组合起来。Cell 处理器被设计为以组的方式工作，这样就重复了我们前面介绍过的连接机的设计。

当连接到一个高速的环形网络时，SPE 还可以按顺序排列起来，以执行一个包括多个步骤的流式操作（如图 1-5 所示）。

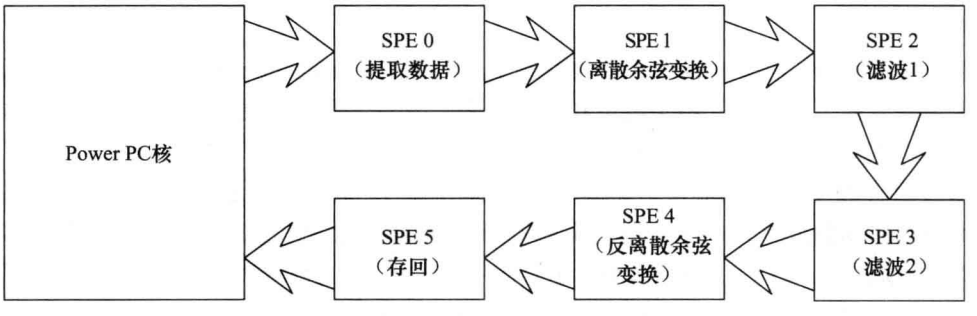



图 1-5　Cell 处理器的流处理器连接路径示例

这种流或流水线处理方法的问题是，系统运行的最快速度等于最慢节点的运行速度。它和工厂里生产线的实际运行情况是完全相同的，整条生产线的运行速度和最慢那个点的运行速度一样。就像装配生产线上的工人一样，每一个 SPE（工人）只执行一小部分任务，所以它可以非常迅速和高效地完成这部分任务。然而，跟其他处理器一样，SPE 也存在带宽限制以及数据传递至下一阶段的开销问题。因此，当通过在每个 SPE 上执行一个一致的程序来提高效率时，我们也会在处理器间通信上有所损失，并最终受到最慢的那个处理步骤的限制。

这是任何基于流水线模型的工作都会遇到的一个共性问题。

还有一种效率更高的方法是，将所有待处理的数据放在一个 SPE 上，然后让其余 SPE 分别处理一小块数据。这相当于培训所有装配线上的工人去装配一个完整的部件。对于简单的任务，这是很容易的，但是每个 SPE 受限于可用的程序和数据内存。这时，PowerPC 核就必须面向 8 个 SPE 进行数据的分发和收集，而不是原先的两个，因此主机和 SPE 间的管理开销及通信量就增加了。

在 2010 年超级计算机 500 强排名中位列第三的 Roadrunner 超级计算机中，IBM 使用了高功率版本的 Cell 处理器。该计算机由 12 960 块 PowerPC 核加上总共 103 680 块流处理器组成。每个 PowerPC 板由一个双核 AMD（Advanced Micro Device）Opteron 处理器监控，总共有 6912 个 PowerPC 板。Opteron 处理器的作用是协调各个节点工作。建造 Roadrunner 超级计算机耗资 1.25 亿美元，占地 560 平方米，理论吞吐率为 1.71 petaflops。工作时需要消耗 2.35 兆瓦的电能！

## 1.6 多点计算

随着对单台计算机硬件（如 CPU、主存、辅存）要求的提升，所需成本快速增加。若购买一个主频为 2.6GHz 的处理器需要花费 250 美元，而购买一个时钟速度增加不到 1GHz 的、主频为 3.4GHz 的类似处理器则需要 1400 美元。在运算速度、主存容量与存储容量上，也能看到类似的关系。

计算性能要求的提高，不仅会带来成本的上升，而且对电力的需求以及相应的散热方面的要求也随之增加。在足够的电力供应和充分的散热条件下，处理器能达到 4 ~ 5GHz 的时钟频率。

在计算领域里，你经常能遇见“收益递减规律”（The law of diminishing returns）。即便你在一个单一方面投入再多，结果也没有太大改变，因为它受限于成本、空间、电力供应、散热等因素。解决办法是在各个影响因素之间选择一个平衡点，然后多次地复制它。

随着时钟频率的不断增长，集群计算在 20 世纪 90 年代开始流行起来，其原理非常简单。用一些成套的或是自己从市场上购买零部件组装的 PC 机，把它们连接到市场上买来的 8 端口、16 端口、24 端口或者 32 端口的以太网交换机上，你得到的性能就比单一主机的性能高 32 倍。与其花费 1600 美元购买一个高性能处理器，不如花 250 美元购买 6 个中等性能的处理器。如果你的应用程序需要巨大的存储容量，则可以先把多台机器上的 DIMM 内存条扩充满，然后互联在一起，这样就可以获得足够的存储容量了。同时工作时，多台机器联合起来达到的计算能力要远远大于花同样的钱购买到的任何一台单机。

一夜之间，大学、中 / 小学校、政府机关和计算机系都能够制造出计算性能比以前强很多的计算机，不再因资金短缺而被排斥在高速计算市场为外。如今天的 GPU 计算一样，集群计算在当年就是一种能改变计算机世界面貌的革命性技术。与不断增加的单核处理器时钟速度结合在一起，这种技术为使用多个单核 CPU 来实现并行处理，提供了一种比较省钱的方式。

PC 集群通常运行 Linux 操作系统的集群版本。在这个版本中，它的每个节点从中央主控节点上获取引导指令和操作系统（Operating System，OS）。例如，在 CudaDeveloper 环境中，我们有一个由低功耗、带嵌入式 CUDA GPU 的独立 PC 机组成的小型集群，很便宜就能买来建立一个集群。有时也可以使用已经被替换掉的旧 PC 机来组建集群，硬件就几乎不花钱了。

然而，集群计算存在的问题是它的速度受到节点之间通信总量的限制，而这些通信是求解问题所必需的。如果你有 32 个运算节点，问题正好被划分成 32 个子任务，并且各个节点之间无须通信，那么你的问题求解是最适合集群计算的。如果每个数据点都要从所有节点上获取数据，那么你就是把一个很糟糕的问题交给了集群。

在现代的 CPU 和 GPU 中都能看到集群技术，比如前面的图 1-1。如果我们把图中的每个 CPU 核作为一个节点，二级缓存作为内存（Dynamic Random Access Memory，DRAM），三级缓存作为网络开关，将内存作为大容量的辅存，我们就得到一个集群的微缩模型（如图 1-6 所示）。

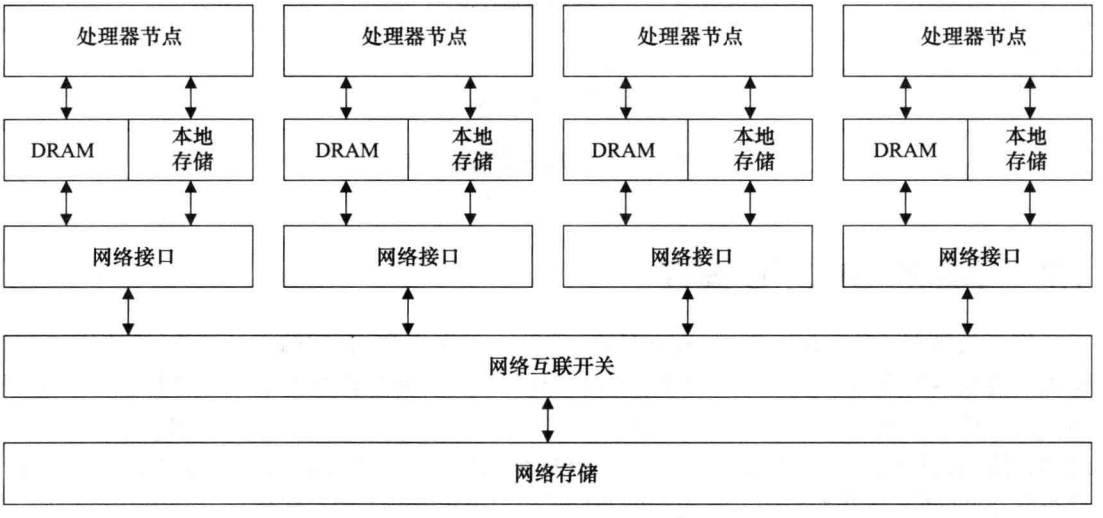



图 1-6 典型的集群层次结构

现代 GPU 的体系结构也完全相同。一个 GPU 内有许多流处理器簇（Streaming Multiprocessor，SM），它们就类似 CPU 的核。这些 SM 与共享存储（一级缓存）连接在一起，然后又与相当于 SM 间互联开关的二级缓存相连。数据先是存储在全局存储中，然后被主机取出并使用。除留一部分自己处理外，主机将剩余的数据通过 PCI-E 互联开关直接送往另一个 GPU 的存储空间。PCI-E 互联开关的传输速度比任何一个互联网络快好多倍。

如图 1-7 所示，节点可以在集群中重复设置。通过在一个可控的环境下，重复设置节点就可以构造一个集群。集群设计的一个进步是分布式应用程序。分布式应用程序可以运行在许多节点上，而每个节点又由包含多个 GPU 的许多处理单元组成。分布式应用程序可以运行在一个受集中控制的集群平台上，也可以运行在随机连接在一起、自主控制的机器上，来求解一个共同的问题。BOINC 和 Folding@Home 是此类应用的两个最大实例。在这两个实例

中，求解问题的计算机是通过因特网（Internet）连接起来的。

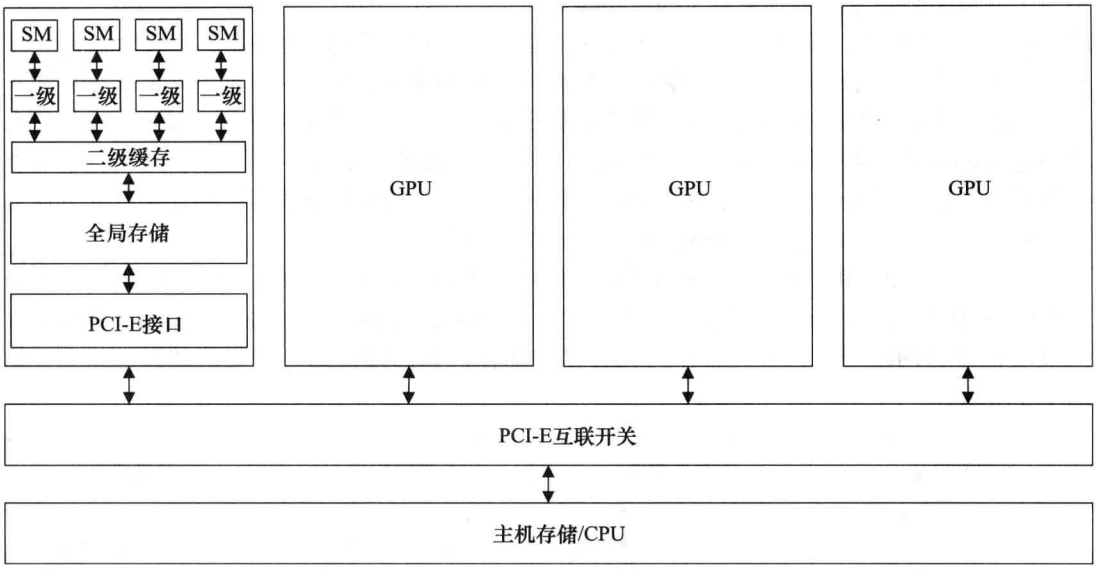



图 1-7 类似集群的 GPU 簇

## 1.7 早期的 GPGPU 编程

图形处理器（Graphics Processing Unit，GPU）是现代 PC 机中的常见设备。它们向 CPU 提供一些基本的操作，比如，对内存中的图像进行着色，然后将其显示在屏幕上。一个 GPU 通常会处理一个复杂的多边形集合，即需要着色的图片映像，然后给这些多边形涂上图片的纹理，进而再做阴影和光照处理。英伟达 5000 系列显卡第一次给我们带来逼真的图片效果，请查看 2003 年英伟达发布的“晨光中的花仙子”（Dawn Fairy）图片示例。

请浏览 http://www.nvidia.com/object/cool_stuff.html#/demos，并下载一些旧的图片示例，从中你可以看出在过去的十几年里 GPU 的发展变化有多大，详见表 1-2。

表 1-2 GPU 技术发展历程表

| 演示 | 显卡 | 年份 |
|---|---|---|
| Dawn | GeForce FX | 2003 |
| Dusk Ultra | GeForce FX | 2003 |
| Nalu | GeForce 6 | 2004 |
| Luna | GeForce 7 | 2005 |
| Froggy | GeForce 8 | 2006 |
| Human Head | GeForce 8 | 2007 |
| Medusa | GeForce 200 | 2008 |
| Supersonic Sled | GeForce 400 | 2010 |
| A New Dawn | GeForce 600 | 2012 |

重要的进步之一就是可编程着色器（programmable shader）的出现。它们是 GPU 运行的一些用来计算各种图片效果的小程序。这样，着色就不必固定在 GPU 中进行。通过可下载的着色器，就可以完成这些操作。这就是最初的通用图形处理器（General-Purpose Graphical Processor Unit，GPGPU）编程。这表示 GPU 设计朝着“处理单元功能不再是固定的”方向迈出了第一步。

然而，这些着色器很自然地取来那些表示一个多边形图像的三维（3D）点集进行处理。着色器以一种高度并行的方式，对许多这样的数据集进行相同的操作，从而提供了巨大的计算能力。

现在即使多边形是用三个点的集合来表示的，其他的一些数据集，比如 RGB 图片，也能够用三个点的集合来表示，但还有许多数据集不是这样表示。因此，有一些勇敢的研究人员就尝试着用 GPU 技术来提高通用计算的速度。这导致开发出了许多新产品（如 Brook GPU、Cg 及 CTM 等），研发这些产品的目标都是将 GPU 变成一个具有和 CPU 一样运行模式的可编程设备。遗憾的是，它们既有优点也存在不足，这些产品都不容易学习和编程使用，而且愿意学习它们的人并不多。一句话，一项不容易学习的技术不可能获得程序员的追捧，也不可能激起程序员广泛的兴趣。这些产品一直没有在市场上获得成功。而 CUDA 可能是首次实现了上述目标，同时向程序员提供了一个真正通用的 GPU 编程语言。

## 1.8 单核解决方案的消亡

现代处理器的问题之一是它们已经达到了 4GHz 左右的时钟速度极限。就目前的技术而言，处理器在这个极限点上工作会产生太多的热量，从而导致需要特殊的、昂贵的冷却措施。产生热量的原因是随着时钟频率的提升，电力功耗增大了。事实上，在电压不变的情况下，一个 CPU 的电力功耗大约是它时钟频率的三次方。更糟糕的是，如果 CPU 产生的热量增加，那么即使时钟频率不变，根据硅材料的性质，CPU 的功耗也会进一步增加。这个电 / 热转换是对能源的一个浪费。这个不断增加的无效的电能消耗，意味着你要么不能充分为处理器提供电力，要么不能够有效地冷却处理器，已经达到了电子设备或芯片封装的散热极限，即所谓的“功耗墙”（power wall）。

面对不能再提高时钟频率的挑战，却还需要制造更快的处理器，处理器制造商只好另辟蹊径。两个主要的 PC 机处理器制造商，Intel 和 AMD，已经采取了另外一种方案。他们已经从持续地提高时钟频率或者通过指令级并行处理技术提高每个时钟周期内执行的指令条数的旧的发展道路，转移到向处理器里添加更多核的新的发展道路。现在已经有了双核、3 核、4 核、6 核、8 核，12 核甚至 16 核和 32 核的处理器，这就是以 CPU 和 GPU 为核心的计算技术未来的发展方向。从 CPU 的角度说，费米（Fermi）型 GPU 已经是一个真正的 16 核处理器了。

然而，这种方法有一个很大的问题——它需要编程者从以前的串行、单线程的问题求解方法切换到多线程同时执行的问题求解方法。现在，编程人员必须考虑 2 个、4 个、6 个或 8 个线程以及线程之间的交互和通信。当双核 CPU 出现时，还相对简单，因为正好有一些程

序需要在后台运行，将这些程序迁移到第二个核上运行即可。当 4 核 CPU 出现时，并没有修改这么多的程序来支持 4 核运算，还是会买 4 核 CPU 以运行单线程的应用程序。甚至游戏厂商也不想很快地转向 4 核编程，而我们通常认为这个领域是最希望采用最新技术的。

在一定程度上，处理器制造商也应该为此负责，因为单核程序可以在 4 核处理器的一个核中运行得很好。当只有一个核在工作时，一些设备甚至会动态地提升时钟频率来提高性能，这就导致编程人员变得懒惰，不愿充分使用已经可用的硬件。

当然，这也有经济上的原因，软件开发公司需要尽早把产品推向市场。开发出一个更好的 4 核程序固然是很好的，但如果竞争对手抢先一步占领市场就不好了。当硬件制造商仍然继续生产单核和双核设备的时候，软件市场自然就稳定在最低配置上，因为这样的软件销售范围是最广的。只有当 4 核 CPU 变成市场上产品的最低配置时，市场才会迫使软件开发朝着多核编程的方向发展。

## 1.9 英伟达和 CUDA

如果留意 GPU 和 CPU 的计算能力，可以得到一张有意思的图（见图 1-8）。我们看到，最初 CPU 和 GPU 计算能力的差距不是很大。但是在 2009 年之后，当 GPU 的性能最终突破了 1 万亿每秒大关后，它们的差距就越来越大了，而在 2009 年，GPU 从 G80 硬件设备发展为 G200 硬件设备。然后在 2010 年，发展到革命性的费米型 GPU。这其中的发展动力是大规模并行硬件的引入。G80 是一个具备 128 个 CUDA 核的设备，G200 是一个具备 256 个 CUDA 核的设备，而 Fermi 则是一个具备 512 个 CUDA 核的设备。

我们看到英伟达的 GPU，从 G200 架构到费米型架构，浮点计算性能实现了每秒 3 千亿次（300 gigaflops）的飞跃，在吞吐量方面提高了接近 30%。相比之下，英特尔公司从 Core 2（酷睿 2）架构升级到 Nehalem 架构仅有小幅的改进。只有改为 Sandy Bridge 架构后，CPU 性能才显著地提升。这并不意味着孰优孰劣，传统 CPU 的目标是执行串行代码，在这方面它们还是做得非常好的。它们包含了一些特殊硬件，例如，分支预测单元、多级缓存等，所有这些都是针对串行代码的执行。但 GPU 并不是为执行串行代码而设计的，且只有完全按照并行模式运行时才能发挥它的峰值性能。

2007 年，英伟达发现了一个能使 GPU 进入主流的契机，那就是为 GPU 增加一个易用的编程接口，也就是所谓的统一计算架构（Compute Unified Device Architecture，CUDA）。这为无须学习复杂的着色语言或者图形处理原语，就能进行 GPU 编程提供了可能。

CUDA 是 C 语言的一种扩展，它允许使用标准 C 来进行 GPU 代码编程。这个代码既适用于主机处理器（CPU），也适用于设备处理器（GPU）。主机处理器负责派生出运行在 GPU 设备处理器上的多线程任务（CUDA 称其为内核程序）。GPU 设有内部调度器来把这些内核程序分配到相应的 GPU 硬件上。调度方法将在本书的后面详细介绍。假设这些任务有足够的并行度，随着 GPU 中流处理器簇数量的增加，程序的运算速度就会提升。

但是，这里仍然有一个大问题：程序中能够并行运行的代码占多大比例呢？可能达到的

最大加速比受限于程序中串行代码的数量。即便你有无限的计算能力，并且使得并行任务的运行时间趋近于零，但你还需要付出运行串行代码的时间。因此，我们必须从一开始就考虑是否能够把大量的工作并行化。

英伟达一直致力于为 CUDA 开发提供支持。在其官方网站（http://www.nvidia.com）上的 CudaZone 栏目中，英伟达提供了大量有助于 CUDA 开发的信息、示例和工具。

与之前的处理器不同，CUDA 已经开始进入发展的快车道，并且成为首个有可能发展成为 GPU 开发的候选编程语言。假如有数以百万计的 GPU 支持 CUDA，那么基于 CUDA 的应用程序将成为一个巨大的市场。

现在有很多基于 CUDA 的应用，并且数量还在逐月递增。英伟达在它的社区网站 http://www.nvidia.com/object/cuda_apps_flash_new.html 中列出了其中的诸多应用。

在程序需要做大量计算工作的领域，例如，从家庭录影带翻录成一个 DVD 盘（视频转码），我们发现大部分主流的视频开发包现在都已支持 CUDA。这个领域的平均加速比达到 5 ~ 10 倍。

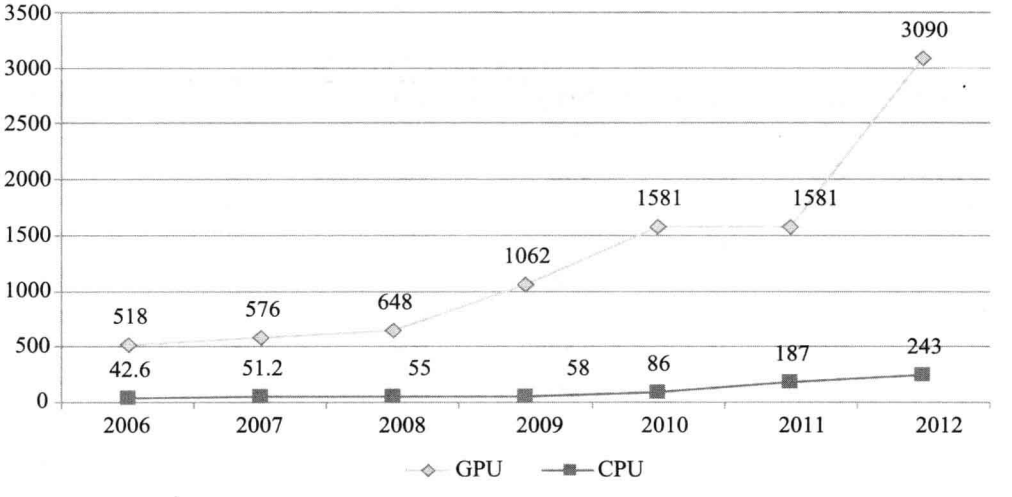



图 1-8 CPU 和 GPU 峰值性能（单位：十亿次浮点操作每秒（gigaflops））

随着 CUDA 一起引入的，是 Tesla 系列板卡。这些并不是图形卡，事实上这些卡既没有 DVI 接口，也没有 VGA 接口。它们是专用于科学计算的计算卡。使用它们可以为科学计算提供很大的加速比。这些卡既可以安装在常规的桌面 PC 上，也可以安装在专用的服务器机架上。在 http://www.nvidia.com/object/preconfigured_clusters.html 网站上，英伟达提供了一个示例系统，据称该系统的计算能力相当于普通集群系统的 30 倍。CUDA 和 GPU 正在改变着高性能计算领域的形式。

## 1.10 GPU 硬件

英伟达 G80 系列处理器及其后续产品是采用类似连接机和 IBM 的 Cell 处理器的设计方

案来实现的。每个图形卡由若干个流处理器簇（SM）组成，每个 SM 配备 8 个或更多的流处理器（Stream Processor，SP）。先前的 9800 GTX 卡有 8 个 SM，总共有 128 个 SP。但是与 IBM 的超级计算机 Roadrunner 不同，购买一块 GPU 板卡只需要几百美元，并且还不需要 2.35 兆瓦的电力去驱动它。后面当我们讲到建立 GPU 服务器时就会看到，电能的问题是不能忽视的。

GPU 卡大致可以看成是一种加速卡或协处理器卡。目前，一个 GPU 卡必须与基于 CPU 的主机相连才能工作。在这方面，它完全遵循 Cell 处理器搭配一个常规的串行处理核和 N 个 SIMD SPE 核的方法。每个 GPU 设备包括一组 SM，每个 SM 又包含一组 SP 或者 CUDA 核。这些 SP 最高可以实现 32 个单元并行工作。它们消除了 CPU 上为实现高速串行执行而设计的大量复杂的指令级并行处理电路。取而代之的是由程序员指定的显式的并行模型，使得同样大小的硅片可以容纳更强的计算能力。

GPU 的总体性能主要由其所具有的 SP 的数量、全局内存的带宽和程序员利用并行架构的充分程度等因素决定。表 1-3 是当前英伟达生产的 GPU 卡的列表。

**表 1-3 当前英伟达生产的 GPU 卡**

| GPU 系列 | 设备 | SP 数目 | 最大存储空间 | GFlops（FMAD） | 带宽（GB/s） | 能耗（瓦） |
|---|---|---|---|---|---|---|
| 9800 GT | G92 | 96 | 2GB | 504 | 57 | 125 |
| 9800 GTX | G92 | 128 | 2GB | 648 | 70 | 140 |
| 9800 GX2 | G92 | 256 | 1GB | 1152 | 2 × 64 | 197 |
| 260 | G200 | 216 | 2GB | 804 | 110 | 182 |
| 285 | G200 | 240 | 2GB | 1062 | 159 | 204 |
| 295 | G200 | 480 | 1.8GB | 1788 | 2 × 110 | 289 |
| 470 | GF100 | 448 | 1.2GB | 1088 | 134 | 215 |
| 480 | GF100 | 448 | 1.5GB | 1344 | 177 | 250 |
| 580 | GF110 | 512 | 1.5GB | 1581 | 152 | 244 |
| 590 | GF110 | 1024 | 3GB | 2488 | 2 × 164 | 365 |
| 680 | GK104 | 1536 | 2GB | 3090 | 192 | 195 |
| 690 | GK104 | 3072 | 4GB | 5620 | 2 × 192 | 300 |
| Tesla C870 | G80 | 128 | 1.5GB | 518 | 77 | 171 |
| Tesla C1060 | G200 | 240 | 4GB | 933 | 102 | 188 |
| Tesla C2070 | GF100 | 448 | 6GB | 1288 | 144 | 247 |
| Tesla K10 | GK104 | 3072 | 8GB | 5184 | 2 × 160 | 250 |

对于一个指定的应用程序，如何选择一块合适的板卡，其实就是在指定的应用程序的内存能耗和 GPU 处理能耗之间找到一个平衡。注意，9800 GX2、295、590、690 和 K10 卡其实是双卡，所以如果要完全利用这些卡，就要把它们当作两个设备来编程而非单个设备。另外需要说明的是，以上描述 GPU 的数据是针对单精度浮点数（32 位）而不是双精度浮点数（64 位）。同样需要注意的是，GF100（Fermi 型）系列，作为 Tesla 的衍生型，双精度运算单元的数量是标准桌面单元的两倍，所以获得了明显更好的双精度输出。即将发布的开普勒

（Kepler）型 GPU K20，与已经发布的兄弟款 K10 相比，双精度运算性能也将有显著改进。

虽然没有提及，但仍应注意到在新一代产品中，每个 SM 的时钟和能耗已经降低了。但是总体能耗却显著增加，这是任何一个基于多 GPU 的解决方案都要考虑的关键问题之一。一个典型的例子是，由于使用了共享电路和降低了时钟频率，基于双 GPU 的卡（9800 GX2、295、590、690）的能耗，比等价的两个单卡的总能耗稍低一些。

为了实现高密度运算，英伟达提供了采用共享 PCI-E 总线相连的、由 2 ~ 4 块 Tesla 卡组成的多种机架（M 系列计算模块）。这样就可以利用标准 PC 部件来建立你自己的 GPU 集群或者微型超级计算机。在本书的后面，我们将介绍如何实现这些想法。

对于 CUDA 来说，一件了不起的事情是，无论硬件设备如何变动，为早期的 CUDA 设备编写的程序依然能够运行在如今的 CUDA 设备上。CUDA 编译模型使用了和 Java 语言一样的编译原则——基于虚拟指令集的运行时编译。这允许现代的 GPU 可以运行即便是最老的 GPU 上的程序代码。当针对新 GPU 而需要修改程序的某些特性时，原先的程序常常可以为程序员的工作提供很多便利。事实上，针对不同硬件版本的更新，可以对程序做很多的优化调整，这些将在本书的最后部分介绍。

## 1.11 CUDA 的替代选择

### 1.11.1 OpenCL

那么其他的 GPU 制造商，如 ATI（现在是 AMD）能够成为主要的厂商吗？从计算能力上看，AMD 的产品和英伟达的产品是旗鼓相当的。但是，在英伟达引入 CUDA 很长时间之后，AMD 才将流计算技术引入市场。从而导致英伟达针对 CUDA 可用的应用程序要远远多于 AMD/ATI 在其技术框架上的应用程序。

OpenCL（Open Computing Language）和“直接计算”（Direct Compute）不是本书详细讨论的内容，但是作为 CUDA 的替代选择，应当提及。目前，CUDA 仅仅能够正式运行于英伟达的硬件产品上。虽然英伟达在 GPU 市场上占有很大的份额，但是其他竞争者所拥有的份额也不小。作为开发者，我们希望开发出的产品能够面向的市场越大越好，尤其是消费者市场。同样的，人们也关心是否有能够同时支持英伟达和其他厂商硬件产品的 CUDA 的替代品。

OpenCL 是一个开放的、免版税的标准，由英伟达、AMD 和其他厂商所支持。OpenCL 的商标持有者是苹果公司，它制定出一个允许使用多种计算设备的开放标准。计算设备可以是 GPU、CPU 或者其他存在 OpenCL 驱动程序的专业设备。截至 2012 年，OpenCL 支持绝大多数品牌的 GPU 设备，包括那些至少支持 SSE3[⊖]的 CPU。

任何熟悉 CUDA 的程序员都可以相对轻松地使用 OpenCL，因为它们的基础概念十分相似。但是，与 CUDA 相比，使用 OpenCL 会复杂一些，因为很多由 CUDA 运行时 API（应用

⊖ SSE3 是 Streaming SIMD Extensions 3 的缩写，表示“单指令多数据流扩展指令 3”。——译者注

程序编程接口）所完成的功能，在 OpenCL 中需要由程序员显式地编程实现。

在 http://www.khronos.org/opencl/ 网站上有更多关于 OpenCL 的内容。而且也有很多关于 OpenCL 的书籍。我个人推荐：在学习 OpenCL 之前，先学习 CUDA。因为在某种意义上讲，CUDA 是一种比 OpenCL 更高级的语言扩展。

### 1.11.2 DirectCompute

DirectCompute 是微软开发的可替代 CUDA 和 OpenCL 的产品。它是集成在 Windows 操作系统，特别是 DirectX 11 API 上的专用产品。对于之前从事显卡编程的人来说，DirectX API 是一个巨大的飞跃。这种产品使得开发者只需掌握一个 API 库，就可以对所有的显卡进行编程，而不必为每个主要的显卡生产商编写或发布驱动程序。

DirectX 11 是最新的标准并且 Windows 7 操作系统能够支持它。由于标准背后的支持厂商是微软，你可以想象得到它会被开发者群体迅速接受。对于已经熟悉 DirectX API 的开发人员，情况更是这样的。如果你熟悉 CUDA 和 DirectCompute，那么将一个 CUDA 应用程序移植到 OpenCL 的确是一项轻松的任务。据微软所言，对同时熟悉两种产品的人来说，这通常仅仅是一个下午的工作量。但是，由于以 Windows 操作系统为核心，DirectCompute 技术被排除在各种版本的 UNIX 占主导地位的高端系统之外。

微软还推出了基于 C++ 的 AMP（加速大规模并行计算）库，它是标准模板库（Standard Template Library，STL）的补充部分。对于熟悉 C++ 风格的 STL 的程序员，它更具有吸引力。

### 1.11.3 CPU 的替代选择

主要的并行程序设计扩展语言有 MPI 和 OpenMP，在 Linux 下开发时，还有 Pthreads。Windows 操作系统下有 Windows 线程模型和 OpenMP。MPI 和 Pthreads 被用作与 UNIX 进行联系的接口。

**MPI**（Message Passing Interface）可能是目前使用最广泛的消息传递接口。它是基于进程的，通常在各个大规模计算实验室中得到应用。它需要一个系统管理员来正确地安装配置，并且它适合于可控的计算环境。它实现的并行处理表现为，在集群的各个节点上，派生出成百上千个进程，通常这些进程通过基于网络的高速通信链路（如，以太网或 InfiniBand）显式地交换消息，以协同完成一个大的任务。MPI 被广泛使用和学习。在可控的集群环境下，它是一个很好的解决方案。

**OpenMP**（Open Multi-Processing）是专门面向单个节点或单个计算机系统而设计的并行计算平台，它的工作方式是完全不同的。在使用 OpenMP 时，程序员需要利用编译器指令精确写出并行运算指令。然后编译器根据可用的处理器核数，自动将问题分为 N 部分。很多编译器对 OpenMP 的支持都是内嵌的，包括用于 CUDA 的 NVCC 编译器。OpenMP 希望根据底层的 CPU 架构，实现对问题的可扩展并行处理。但是，CPU 内的访存带宽常常不够大，满足不了所有核连续将数据写入内存或者从内存中取出数据的要求。

**pthreads** 是一个主要应用于 Linux 上的多线程应用程序库。同 OpenMP 一样，pthreads

使用线程而不是进程，因为它是设计用来在单个节点内实行并行处理的。和 OpenMP 不同的是，线程管理和线程同步由程序员来负责。这提供了更多的灵活性，因此精心设计的程序会带来很好的性能。

**ZeroMQ**（0MQ）也值得一提，这是一个你可以链接的简单的库。在本书的后面，我们将使用它来开发一个多节点、多 GPU 的例子。ZeroMQ 使用一个跨平台的 API 来支持基于线程、基于进程和基于网络的通信模型。Linux 和 Windows 平台都支持 ZeroMQ。它是针对分布式计算而设计的，所以连接是动态的，节点失效不会影响它的工作。

**Hadoop** 也是你应当考虑的技术。Hadoop 是谷歌 MapReduce 框架的一个开源版本。它针对的是 Linux 平台。其概念是你取来一个大数据集然后将其切割或映射（map）成很多小的数据块。然而，并不是将数据发送到各个节点，取而代之的是数据集通过并行文件系统已经被划分给上百个或者上千个节点。因此，归约（reduce）步骤是把程序发送到已经包含数据的节点上。输出结果写入本地节点并保存在那里。后续 MapReduce 程序取得之前的输出并以某种方式对其进行转换。由于数据实际上映射到了多个节点上，因此它可以应用于高容错和高吞吐率系统。

### 1.11.4 编译指令和库

很多编译器厂商，如 PGI、CAPS 以及最著名的 Cray，都支持最近发布的针对 GPU 的 OpenACC 编译器指令集。在本质上，这些是 OpenMP 方法的复制。它们都要求程序员在程序中插入编译器指令来标注出“应该在 GPU 上执行”的代码区域。然后编译器就做些简单的工作，即从 GPU 上移入或移出数据、调用内核程序等。

若使用 pthreads 替代 OpenMP，由于 pthreads 提供更底层的控制，所以你可以获得更高的性能。使用 CUDA 替代 OpenACC 也是这样。但是，对额外层面的控制，要求程序员掌握更高水平的编程知识，也会带来更高的出错风险，进而对开发进度产生影响。目前，OpenACC 不仅要求用指令标注出哪些区域的代码需要运行在 GPU 上，而且还要指明数据要存储在哪种类型的内存上。英伟达宣称使用这类指令可以获得 5 倍以上的速度提升。对于那些想要程序跑得更快的程序员来说，这是一个很好的选择。对于那些把程序设计放在次要位置，仅仅考虑在合理的时间内完成任务的人来说，这也是一个很棒的选择。

库的使用也是很重要的，因为它可以让你的生产效率以及执行程序时间的加速比提高。一些库提供的通用函数的执行效率很高，例如，SDK 提供的 Thrust。诸如 CUBLAS 这样的库是针对线性代数最好的库。很多诸如 Matlab 和 Mathematica 这样著名的应用程序中都有库的存在。在某些语言中，如 Python、Perl、Java 和许多其他的语言中，库是以语言绑定的形式存在的。CUDA 甚至还被集成进 Excel 里。

在现代化软件开发的各个方面，很多你准备开发的东西别人已经做好了。在你准备花费数周时间开发一个库之前，请先去互联网上搜索一下，看看哪些是已经存在的。除非你是一个 CUDA 专家，否则你自己开发不大可能比使用已有的更快。

## 1.12 本章小结

也许你在想，为什么要使用 CUDA 进行开发？答案是，在技术支持、调试工具和驱动程序等方面，CUDA 是目前最容易进行开发的语言。CUDA 在各个方面都领先一步，在成熟性方面更是遥遥领先。即便你的应用程序需要支持非“英伟达”的硬件，但最好的方式也是先在 CUDA 上开发，然后将其移植为其他 API 的应用程序。同理，我们应专注于 CUDA，因为当你成为一个 CUDA 专家后，使用开发所需的其他 API 也是很容易的。理解 CUDA 的工作原理，可以帮助你更好地利用任何一个高层 API，并有助于理解它们的局限性。

我衷心地希望，你能从由单线程 CPU 程序员转变成 GPU 上的并行程序员的旅程中，找到乐趣。即使将来不做 GPU 开发，你获得的知识也会对设计多线程程序有极大帮助。如果你，像我们一样，看到世界正在向并行编程模型转变，那么你将会渴望能够站在技术挑战与创新浪潮的最前沿。单线程工业正在慢慢衰落。要想保持自身价值、成为公司渴望的人才，你需要掌握那些代表计算机世界发展方向的技术，而不是那些日趋没落的技术。

GPU 正在改变计算机世界的面貌。突然之间，十几年前超级计算机的计算能力现在已经能够出现在你的办公桌上。你不再需要：排队、分批提交任务，然后用几个月的时间等待管理员（committee）批准你的请求——在一个超负荷运行的计算系统上使用有限的计算机资源。现在，最多花 5000 ~ 10 000 美元，你就可以在你的办公桌上添置一台超级计算机，或者花一小部分钱买来一台运行 CUDA 的机器。GPU 是一项翻天覆地的技术革新，它使每个人拥有超级计算机级别的计算能力成为可能。

第 2 章

# 使用 GPU 理解并行计算

## 2.1 简介

本章旨在对并行程序设计的基本概念及其与 GPU 技术的联系做一个宽泛的介绍。本章主要面向具有串行程序设计经验，但对并行处理概念缺乏了解的读者。我们将用 GPU 的基本知识来讲解并行程序设计的基本概念。

## 2.2 传统的串行代码

绝大多数程序员是在串行程序占据主导地位而并行程序设计仅吸引极少数技术狂的年代里成长起来的。大多数人是因为对技术感兴趣才到大学去攻读与 IT 有关的学位的。同时他们也会意识到将来还需要一个能够获得可观收入的工作岗位。因此，在选择专业时，他们肯定会考虑毕业后，社会上是否有足够多的就业岗位。即便在并行程序设计最火的年代，除了研究所或学术机构的岗位外，适合并行程序设计的商业岗位总是很少的。绝大多数程序员都是用简单的串行模式来开发应用软件，而这个串行模式主要是基于大学程序设计课程的教学内容，且受市场需求的驱动。

并行程序设计的发展态势一直不太明朗，与它有关的各种技术及程序设计语言一直没能将它变成主流。市场上从来没有出现过对并行硬件的大规模需求。相应地，也没有出现对并行程序员的大规模需求。每一到两年，各个 CPU 厂商都会推出新一代的处理器。相比之前的处理器，新一代处理器执行代码的速度更快。无须改进运行方式就可以有更快的运行效果，因此串行程序的地位稳如泰山。

并行程序通常与硬件的联系更紧密。引入并行程序的目的是为了获得更高的性能，但是这往往需要付出降低可移植性的代价。在新一代并行计算机中，原先计算机的某种特性可能会换种方式实现，甚至被取消。每隔一定的时间，就会出现一种新的革命性的并行计算机架构，从而导致所有的程序代码都要重新编写。作为程序员，如果你的知识仅局限于某一个处理器，那么从商业角度看，这些知识会随着这款处理器的淘汰而失去价值。由此可见，学习 x86 类型架构的程序设计比学习某种并行计算机架构的程序设计具有更大的商业意义，因为

后者很可能仅有几年的使用寿命。

然而有趣的是，多年以来，两个并行程序设计标准，OpenMP和MPI，却通过不断地修改、完善而得以始终采用。面向包含多核处理器的共享存储并行计算机而设计的OpenMP标准，强调的是在单个节点内部实现并行处理。它不涉及节点间并行处理的任何概念。因此，你所能解决的问题只受到单个节点的处理能力、内存容量和辅存空间的限制。但是对于程序员而言，采用OpenMP，程序设计相对容易，因为大多数低层级的线程代码（需要采用Windows线程库函数或POSIX线程库函数编制）都由OpenMP替你完成了。

MPI（Message Passing Interface）标准用于解决节点间的并行处理，常用于定义良好的网络内的计算机集群。它常用于由几千个节点组成的超级计算机系统。每个节点只分担问题的一小部分。因此，公共资源（CPU、缓存、内存、辅存等）的大小就等于单个节点的资源量乘以网络中节点的个数。任何网络的阿喀琉斯之踵（Achilles' heel，要害）就是它的互连结构，即把机器连接在一起的网络结构。通常，节点间通信是任何基于集群的解决方案中决定最大速率的关键因素。

当然，OpenMP和MPI也可以联合使用来实现节点内部的并行处理和集群中的并行处理。但是由于应用程序编程接口（API）库和编程方法完全不同，所以这种情况并不常见。OpenMP的并行指令允许程序员通过定义并行区，从而从一个较高的层次分析算法中的并行性；而MPI却需要程序员做大量的工作来显式地定义节点间的通信模型。

既然已经花很长时间才掌握一种API库，程序员都不愿意再学习另外一种。因此，适合于一台计算机就能解决的问题通常采用OpenMP，而需要用集群来解决的大问题就采用MPI。

本书将要探讨的GPU编程语言CUDA，却能够很好地将OpenMP和MPI联系在一起。例如，CUDA提供的类OpenMP指令（OpenACC）就很方便地供熟悉OpenMP的程序员采用。OpenMP、MPI和CUDA正在越来越多地出现在大学计算机专业本科生或研究生的课堂教学内容中。

然而，绝大多数串行程序设计者第一次接触并行程序是在介绍多核处理器的时候。除了少数技术狂之外，他们往往对眼前的并行处理环境视而不见，因为多核处理器主要用于实现“任务并行”（task parallelism），这是一种基于操作系统的并行处理，后面我们再详细介绍。

显而易见，今天的计算机技术正在向着多核的方向前进。越来越多的程序员开始关注多核处理器。几乎所有商场销售的台式机要么采用双核处理器，要么采用四核处理器。因此，程序员开始利用多线程来发挥处理器中多核的作用。

线程是程序中一个独立的执行流，它可以在主执行流的要求下分出或聚合。通常情况下，CPU程序拥有的活动线程数量，不超过其包含的物理处理核数量的两倍。就像在单核处理器中，操作系统的任务是分时、轮流运行的，每个任务只能运行一小段时间，从而实现数量多于物理CPU核数的众多任务同时运行。

然而，随着线程数量的增加，终端用户也开始感觉到线程的存在。因为在后台，操作系统需要频繁进行“上下文切换”（即对于一组寄存器进行内容的换入、换出）。而“上下文切换”是一项很费时的操作，通常需要几千个时钟周期，所以CPU的应用程序的线程数要尽量

比 GPU 的少。

## 2.3 串行 / 并行问题

线程会引起并行程序设计中的许多问题，例如，资源的共享[⊖]。通常，这个问题用信号灯（semaphore）来解决，而最简单的信号灯就是一个锁或者令牌（token）。只有拥有令牌的那个线程可以使用资源，其他线程只能等待，直到这个线程释放令牌。因为只有一个令牌，所以所有工作都有条不紊地进行。

当同一线程必须拥有两个或者更多的令牌时，就会出现问题。例如，在线程 0 拥有令牌 0 而线程 1 拥有令牌 1 的情况下，如果线程 0 想得到令牌 1 而线程 1 想得到令牌 0，想得到的令牌没有了，线程 0 和线程 1 就只好休眠，直到它们期待的令牌出现。由于没有一个线程会释放掉已经拥有的令牌，所以所有的线程将永远等待下去。这就是所谓的“死锁”（deadlock），如果程序设计不当，死锁将不可避免地发生。

当然，在极偶然的情况下，程序可以共享资源而不发生死锁。无论哪种加锁系统，每一个线程都必须正确地使用资源。也就是说，它们必须先申请令牌，如果没成功则等待。获得令牌后，才执行相应的操作。这就需要程序员定义好共享的资源，并制定恰当的机制来协调多线程对该资源的更新操作[⊜]。然而，在任何一个团队里都有若干个程序员，即便有一个程序员不遵守规则，或者并不知道它是一个共享资源，你的程序多数情况下不会正常工作。

我曾经为一个大公司开发的项目里就出现过这个现象。所有的线程都申请一个锁，如果没成功则等待。获得令牌后，就更新共享资源。一切都运行正常，所有的代码都通过了质量保证检查（Quality Assurance，QA）和所有的测试。然而投入运行后，现场的用户偶尔会报告说某个参数被重置为 0 了，看上去像随机发生的。由于跟踪发现 Bug 的前提往往是它能够持续地引发同一个问题，所以随机的 Bug 总是很难发现的。

最后是公司里的一个实习生发现了问题的原因。在代码中一个与其完全无关的区域里，一个指针在特定的条件下没有被初始化。在程序的运行过程中，当线程以某种特定的顺序执行时，这个指针就会碰巧指向受我们保护的数据。程序中的其他代码会将这个指针所指向变量的值初始化成 0。这样，受我们保护的并被线程共享的参数的值就被清除了。

这是基于线程操作的一个让人为难的地方。线程操作的是一个共享的内存空间，这既可以带来不借助消息就可以完成数据交换的便利，也会引起缺乏对共享数据保护的问题。

可以用进程来替换线程。不过，因为代码和数据的上下文都必须由操作系统保存，所以操作系统装入进程就要吃力得多。相比之下，只有线程代码的上下文（一个程序或指令计数器加上一组寄存器）由操作系统保存，而数据空间是共享的。总之，进程和线程可以在任意时刻、在程序中的不同区域分别执行。

---

⊖ 即对一个公共变量或内存地址进行读 / 写操作。——译者注

⊜ 即对公共变量或内存地址进行写操作。——译者注

默认情况下，进程在一个独立的内存空间内运行。这样就可以确保一个进程不会影响其他进程的数据。因此，一个错误的指针访问将引发“越界访问”异常，或者很容易在特定的进程中找到 Bug。不过，数据传递只能通过进程间消息的发送和接收来完成。

一般说来，线程模型较适合于 OpenMP，而进程模型较适合于 MPI。在 GPU 的环境下，就需要将它们混合在一起。CUDA 使用一个线程块（block）构成的网格（grid）。这可以看成是一个进程（即线程块）组成的队列（即网格），而进程间没有通信。每一个线程块内部有很多线程，这些线程以批处理的方式运行，称为线程束（warp）。后续的章节中我们将进一步介绍。

## 2.4 并发性

并发性的首要内涵是，对于一个特定的问题，无须考虑用哪种并行计算机来求解，而只需关注求解方法中哪些操作是可以并行执行的。

如果可能，请设计出一个公式来把每一个输出数据表示成输入数据的函数。不过，对于某些算法而言，这显得很麻烦，例如，迭代次数很大的算法。对于这些算法，可以单独考虑每一步或每一次迭代。能否将对应每一步的数据表示成输入数据集的一个变换？如果能，则你只需拥有顺序执行的一组内核函数（迭代步）即可解决问题。只需将这些操作压入队列（或者处理流），计算机将依次执行该队列的操作。

很多问题属于“易并行”（embarrassingly parallel）问题，其实这个名称还是相当保守的。如果能够设计出一个公式把每一个输出数据都表示成相互无关的，例如，矩阵乘法，那将是很令人开心的结局。这类问题可以在 GPU 上得到很好的解决而且编程很简单。

尽管算法中可能有一个阶段不是“易并行”，但某一步或某几步还是能够用这种方式表达，这也不错。尽管该阶段会成为一个“瓶颈”（bottleneck），还会让程序员很劳神。但对于问题的其他部分，程序员还是很容易就可以编写出在 GPU 上求解的代码。

如果求解问题的算法在计算每一个点的值的时候，必须知道与其相邻的其他点的值，那么算法的加速比（speedup）最终将很难提高。在这种情况下，对一个点的计算就需要投入多个处理器。在这一点上，计算将会变得很慢，因为处理器（或者线程）需要花费更多的时间来进行通信以实现数据共享，而不做任何有意义的计算。至于你到底会遇到怎样糟糕的情况，则取决于通信的次数和每次通信的开销。

由于“易并行”不需要或者只需少许线程间或线程块间通信，所以 CUDA 是很理想的并行求解平台。它用基于片上资源的、显式的通信原语来支持线程间通信。但是块间通信只有通过按顺序调用多个内核程序才能实现，而且内核间通信需要用到片外的全局内存。块间通信还可以通过对全局内存的原子操作来实现，当然使用这种方法会受到一定的限制。

CUDA 将问题分解成线程块的网格，每块包含多个线程。块可以按任意顺序执行。不过在某个时间点上，只有一部分块处于执行中。一旦被调度到 GPU 包含的 N 个“流处理器簇”（Streaming Multiprocessors，SM）[⊖]中的一个上执行，一个块必须从开始执行到结束。网格中

⊖ 原文误为 symmetrical multiprocessors。——译者注

的块可以被分配到任意一个有空闲槽的 SM 上。起初，可以采用“轮询调度”（round-robin）策略，以确保分配到每个 SM 上的块数基本相同。对绝大多数内核程序而言，分块的数量应该是 GPU 中物理 SM 数量的八倍或者更多倍。

以一个军队比喻，假设有一支由士兵（线程）组成的部队（网格）。部队被分成若干个连队（块），每个连队由一位连长来指挥。按照 32 名士兵一个班（一个线程束），连队又进一步分成若干个班，每个班由一位班长来指挥（参见图 2-1）。

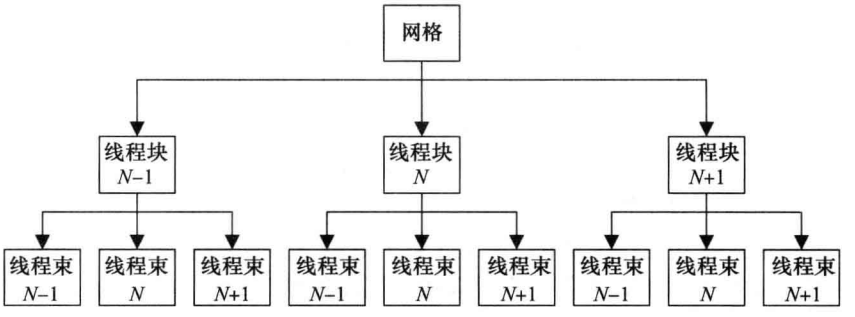



图 2-1 基于 GPU 的线程视图

要执行某个操作，总司令（内核程序 / 主机程序）必须提供操作名称及相应的数据。每个士兵（线程）只处理分配给他的问题中的一小块。在连长（负责一个块）或班长（负责一束）的控制下，束与束之间的线程或者一束内部的线程之间，要经常地交换数据。但是，连队（块）之间的协同就得由总司令（内核函数 / 主机程序）来控制了。

因此，当思考 CUDA 程序如何实现并发处理时，你应该用这种非常层次化的结构来协调几千个线程的工作。尽管一开始听上去很复杂，但是对绝大多数“易并行”程序而言，仅仅需要针对一个线程思考它计算输出数据的那一点。在一个典型的 GPU 上可以运行 24K[⊖]个‘活动”线程。在费米架构 GPU 上，你总共可以定义 65 535 × 65 535 × 1536 个线程，其中 24K 个线程随时都是活动的。这表明一个节点就足够满足单点中绝大多数问题的求解要求了。

## 局部性

在过去的十几年间，计算性能的提高已经从受限于处理器的运算吞吐率的阶段，发展到迁移数据成为首要限制因素的阶段。从设计处理器过程中的成本来讲，计算单元（Algorithmic Logic Unit，ALU）是很便宜的。它们能够以很高的速度运行，而消耗很小的电能并占用很少的物理硅片空间。然而，ALU 的工作却离不开操作数。将操作数送入计算单元，然后从计算单元中取出结果，耗费了大量的电能和时间。

在现代计算机设计中，这个问题是通过使用多级缓存来解决的。缓存的工作基础是空间局部性（地址空间相对簇集）和时间局部性（访问时间的簇集）。因此，之前被访问过的数据，很可能还要被再次访问（时间局部性）；刚刚被访问过的数据附近的数据很可能马上就会被访

⊖ 1K=1024。——译者注

问（空间局部性）。

当任务被多次重复执行时，缓存的作用会充分地发挥。假设一个工人带着一个装有 4 件工具的工具箱（缓存），当分配给他的大量工作都是相同的时候，这 4 件工具将被反复使用（缓存命中）。

反之，大量的工作都需要不同的工具，情况就不一样了。例如，工人来上班时，并不知道今天会分配给他什么工作。简单来说，今天的工作需要一件不同的工具。由于它不在工具箱（一级缓存）中，那么他就到工具柜（二级缓存）里找。

偶尔，工人可能需要一个工具箱和工具柜里都没有的、特殊的工具或零件，这时他就得停下手中的工作，跑到附近的硬件仓库（全局内存）里，取回他想要的东西。由于公路上可能发生拥堵，或者五金商店门前排起了长队（其他进程争相访问主存），因此，无论是工人还是客户都不知道为了拿到工具需要花费多少时间（延迟）。

显然，工人的时间效率很低。因此，每项工作都需要一个新的、不同的工具或零件，工人就只得去工具柜或者五金商店去取。而这个取的过程，工人并不是忙于手里的工作。

与到硬件仓库里去取工具类似，到硬盘或者固态硬盘（Solid-State Drive，SSD）里取数据，也是很糟糕的。尽管固态硬盘比硬盘要快得多，好比一个普通的送货员需要几天才能把硬盘里的数据送到，而送货员一个晚上就能把固态硬盘里的数据送到，但是与访问全局内存相比，它还是很慢的。

某些现代处理器已经支持多线程。如某些英特尔处理器中，每个处理器核支持 2 个硬件线程即“超线程”(hyperthreading)。接着上面的比喻，“超线程”相当于给工人配了一名助手，并分配给他两项任务。在处理某项任务时，每当需要一个新的工具或零件，工人就派他的助手去取。然后工人就切换去处理另一项任务。假定这个助手总是能够在另一项任务也需要新的工具或零件前就能返回，那么工人就始终处于忙碌的工作状态。

尽管改进后时间效率提高了，但是从硬件仓库（全局内存）里取回新的工具或零件到底有多少延迟还是不清楚。通常，访问全局内存的延迟大概有几百个时钟周期。对于传统的处理器设计，这个问题的答案是不断增大缓存的容量。实际上，为了降低访问仓库的次数，必须采用一个更大的工具柜。

然而，这种办法会增加成本。一则更大的工具柜需要投入更多的资金，二则在一个更大的工具柜里查找工具或零件需要更多的时间。因此，在当前绝大多数处理器设计中，这种方法表现为一个工具柜（二级缓存）加一个货车（三级缓存）。在服务器处理器中，这种情况尤为突出，就如同工厂购买了一辆 18 轮大货车来保证工人总是处于忙碌状态。

因为一个基本的原因（局部性），所以上述工作是必须的。CPU 是设计以运行软件的，而编制软件的程序员并不一定关心局部性。无论处理器是否试图向程序员隐藏局部性，局部性是客观存在的。为了降低访存延迟而需要引入大量的硬件并不能否认局部性是客观存在的这个事实。

GPU 的设计则采用一个不同的方法。它让 GPU 的程序员负责处理局部性，并给程序员提供很多小的工具柜而不是一辆 18 轮大货车，同时给他配备很多工人。

对于GPU程序设计，程序员必须处理局部性。对于一个给定的工作，他需要事先思考需要哪些工具或零件（即存储地址或据结构），然后一次性地把它们从硬件仓库（全局内存）取来，在工作开始时就把它们放在正确的工具柜（片内存储器）里。一旦数据被取来，就尽可能把与这些数据相关的不同工作都执行了，避免发生“取来——存回——为了下一个工作再取”。

因此，“工作——等待——从全局内存取”、“工作—等待—从全局内存取……这样连续的周期就被打破了。我们可以用生产流水线来比拟，一次性提供给工人一筐零件，而不是工人需要一个零件就到仓库管理员那里去取。后者导致工人的大多数时间被浪费了。

这个简单的计划使得程序员能够在需要数据前就把它们装入片内存储器。这项工作既适合于诸如GPU内共享内存的显式局部存储模型，也适合于基于CPU的缓存。在共享内存的情况下，你可以通知存储管理单元去取所需的数据，然后就回来处理关于已有的其他数据的实际工作。在缓存的情况下，你可以用特殊的缓存指令来把你认为程序将要使用的数据，先行装入缓存。

与共享内存相比，缓存的麻烦是替换和对“脏”数据的处理。所谓“脏”数据是指缓存中被程序写过的数据。为了获得缓存空间以接纳新的有用数据，“脏”数据必须在新数据装入之前写回到全局内存。这就意味着，对于延迟未知的全局内存访问，我们需要做两次，而不是一次——第一次是写旧的数据，第二次是取新的数据。

引入受程序员控制的片上内存，带来的好处是程序员可以控制发生写操作的时间。如果你正在进行数据的局部变换，可能就没有必要将变换的中间结果写回全局内存。反之用缓存的话，缓存控制器就不知道哪些数据应该写、哪些数据可以抛弃。因此，它全部写入。这势必增加了很多无用的访存操作，甚至会造成内存接口拥塞。

尽管做了很多工作，但是并不是每个算法都具备“事先预知的”内存访问模式，而程序员正是需要对此进行优化。同时，也不是每个程序员都想处理局部性的事务，所以程序的局部性要么是“与生俱有的”，要么就根本没有。要想深入理解局部性，最好、最有效的方法就是开发一个程序，验证概念，然后琢磨如何改进局部性。

为了使这种方法得到更好的应用，并改进那些没有很好定义数据或执行模式的算法的局部性，新一代GPU（计算能力2.x以上）同时设有一级和二级缓存。它们还可以根据需要配置成缓存或者共享内存，这样程序员就可以针对具体的问题灵活地配置硬件条件了。

## 2.5 并行处理的类型

### 2.5.1 基于任务的并行处理

如果仔细分析一个典型的操作系统，我们就会发现它实现的是一种所谓任务并行的并行处理，因为各个进程是不同的、无关的。用户可以在上网阅读文章的同时，在后台播发他音乐库中的音乐。多个CPU核运行不同的应用程序。

就并行程序设计而言，这可以通过编写一个程序来实现，这个程序由多个段组成，这些段将信息从一个应用程序“传递”（通过发送消息）到另一个应用。Linux 操作系统的管道操作符（|）就具有这个功能。一个程序（例如 grep）的输出是另一个程序（例如 sort）的输入。这样，就可以轻松地对一组输入文件进行扫描（通过 grep 程序），以查看是否包含特定的字符，然后输出排好序的结果（通过 sort 程序）。每个程序都分别调度到不同 CPU 核上运行。

一个程序的输出作为下一个程序的输入，这种并行处理称为流水线并行处理（pipeline parallelism）。借助一组不同的程序模块，例如，Linux 操作系统中各种基于文本的软件工具，用户就可以实现很多有用的功能。也许程序员并不知道每个人所需要的输出，但通过共同工作并可以轻松连接在一起的模块，程序员可以为广泛的、各种类型的用户服务。

这种并行处理向着“粗粒度并行处理”（coarse-grained parallelism）发展，即引入许多计算能力很强的处理器，让每个处理器完成一个庞大的任务。

就 GPU 而言，我们看到的“粗粒度并行处理”是由 GPU 卡和 GPU 内核程序来执行的。GPU 有两种方法来支持流水线并行处理模式。一是，若干个内核程序被依次排列成一个执行流，然后不同的执行流并发地执行；二是，多个 GPU 协同工作，要么通过主机来传递数据，要么直接通过 PCI-E 总线，以消息的形式在 GPU 之间直接传递数据。后一种方法，也叫“点对点”（Peer-to-Peer，P2P）机制，是在 CUDA 4.x SDK 中引入的，需要特定的操作系统 / 硬件 / 驱动程序级支持。

和任何生产流水线一样，基于流水线的并行处理模式的一个问题就是，它的运行速度等于其中最慢的部件。因此，若流水线包含 5 个部件，每个部件需要工作 1 秒，那么我们每 1 秒钟就可以产生一个结果。然而，若有一个部件需要工作 2 秒，则整个流水线的吞吐率就降至每 2 秒钟产生一个结果。

解决这个问题的方法是“加倍”(twofold)。让我们用工厂的流水线来打比方。由于工作复杂，Fred 负责的工段需要花费 2 秒钟。如果我们为 Fred 配一名助手 Tim，那么 Fred 就可以把工作一分为二，把一半分给 Tim。这样我们就又回到每个工段只需 1 秒的状态。现在我们拥有了一个 6 段流水线而不是 5 段流水线，但流水线的吞吐率又重新回到每秒钟一个结果。

出于某种考虑，经过精心设计（参见第 11 章），你可以将 4 个 GPU 加到一个桌面 PC 中。这样，若我们仅有一个 GPU，则处理一个特定的工作流花费的时间太长。若我们增加一个 GPU，则这个节点的整体处理能力就提高了。但是我们需要考虑在两个 GPU 之间如何划分工作。简单的 50/50 划分可能不可行。若仅能实现 70/30 划分，则最大收益是当前运行时间的 7/10（70%）。若我们再增加一个 GPU 并能够分给它占总时间 20% 的任务，即按 50/30/20 划分。这样与一个 GPU 相比，加速效果是原来时间的 1/2 或 50%。无论如何，整体的执行时间仍取决于最慢的时间。

为了加速而使用一个 CPU/GPU 组合时，也需要考虑上述问题。如果我们把 80% 的工作从 CPU 移到 GPU 上，而 GPU 计算这些任务仅需原来时间的 10%，那么加速比是多少呢？由于 CPU 花费原来时间的 20%，而 GPU 花费原来时间的 10%，但是它们是并行的，因此决定性因素仍然是 CPU。由于 GPU 是并行地与 CPU 一起工作，且工作时间少于 CPU，所以它

的工作时间就忽略不计了。因此，最大加速比就等于程序执行时间最长那部分占整个程序比例的倒数。

这称为“阿姆达尔法则”（Amdahl’s law），它表示任意加速比的上限。它让我们在一开始，一行代码都还没写的情况下，就知道可能达到的最大加速比。无论如何，你都要做串行操作。即便把所有的计算都移到 GPU 上，你也需要用 CPU 来访问辅存、装入和存回数据。你还需要与 GPU 交换数据以完成输入及输出（I/O）。因此，最大加速比取决于程序中计算或算术部分占整个程序的比例加上剩下的串行部分的比例。

### 2.5.2 基于数据的并行处理

在过去的二十多年间，计算能力不断增长。现在我们已经拥有了运算速度达到每秒万亿次浮点操作的 GPU。然而，跟不上计算能力增长步伐的是数据的访问时间。基于数据的并行处理的思路是首先关注数据及其所需的变换，而不是待执行的任务。

基于任务的并行处理更适合于粗粒度并行处理方法。让我们看一个对 4 个不同且无关的等长数组分别进行不同变换的例子。我们有 4 个 CPU 核以及 1 个带有 4 个 SM 的 GPU。若对这个问题采用基于任务的分解，则 4 个数组将被分别赋给 4 个 CPU 核或者 GPU 中的每个 SM。对问题的这种并行分解是在只考虑任务或变换，而不考虑数据的思路下进行的。

在 CPU 上，我们将产生 4 个线程或进程来完成任务。在 GPU 上，我们将使用 4 个线程块，并把每个数组的地址分别送给 1 个块。在更新的费米架构和开普勒架构 GPU 上，我们也可以产生 4 个并发运行的内核程序，每个内核程序处理 1 个数组。

基于数据的分解则是将第 1 个数组分成 4 数据块，CPU 中的 1 个核或者 GPU 中的 1 个 SM 分别处理数组中的 1 数据块。处理完毕后，再按照相同的方式处理剩下的 3 个数组。对采用 GPU 处理而言，将产生 4 个内核程序，每个内核程序包含 4 个或者更多的数组块。对问题的这种并行分解是在考虑数据后再考虑变换的思路下进行的。

由于我们的 CPU 仅有 4 个核，所以使我们想到将数据分成 4 数据块。我们既可以让线程 0 处理数组元素 0，线程 1 处理数组元素 1，线程 2 处理数组元素 2，线程 3 处理数组元素 3，以此类推。我们还可以把数组分成 4 数据块，每个线程处理数组中对应的一数据块。

在第一种情况下，线程 0 去取元素 0。由于 CPU 包含多级缓存，数据将存入缓存。通常，三级缓存是被所有的核心共享的。因此，第一次内存访问取来的数据需要分发给所有 CPU 核。相反在第二种情况下，需要进行 4 次不同的内存访问，取来的数据分别存入到三级缓存中的不同缓存中。因为 CPU 核心需要把数据写回内存，所以后一种方法通常更好些。第一种情况下，CPU 核交替地使用数据，这意味着，缓存必须协调和组合来自不同 CPU 核心的写操作，这是很糟糕的思路。

如果算法允许，我们还可以探讨另一种类型的数据并行处理——单指令多数据（Single Instruction，Multiple Data，SIMD）模型。这需要特殊的 SIMD 指令，例如，许多基于 x86 型 CPU 提供的 MMX、SSE、AVX 等。这样，线程 0 就可以取出多个相邻的数组元素，并用一条 SIMD 指令来进行处理。

同样的问题，如果我们用GPU来处理，每个数组则需要进行不同的变换，且每一个变换将映射以被看成一个单独的GPU内核程序。与CPU核不同，每个SM可以处理多个数据块，每个数据块的处理被分成多线程来执行。因此为了提高GPU的使用效率，我们需要对问题进行进一步的分解。通常，我们将块和线程做一个组合，使得一个线程处理一个数组元素。使用CPU，让每个线程处理多个数据，会有很多好处。相比之下，由于GPU仅支持“加载”(load)/“存储”(store)/“移动”(move)三条显式的SIMD原语，那么它的应用受到限制。但这反过来促进了GPU“指令级并行处理”(Instruction-Level Parallelism，ILP)功能的增强。在本书的后面，我们将看到ILP给我们带来的好处。

在费米架构和开普勒架构GPU上，我们有一个共享的二级缓存，它的功能与CPU上的三级缓存相同。因此，在CPU上，一个线程访存的结果可以从缓存直接分布给其他线程。早期的硬件处理器，没有缓存。但是在GPU上，相邻的内存单元是通过硬件合并(组合)在一起进行存取的。因此单次访存的效率就更高了。具体细节可查阅第6章关于内存的介绍。

GPU与CPU在缓存上的一个重要差别就是“缓存一致性”(cache coherency)问题。对于“缓存一致”的系统，一个内存的写操作需要通知所有核的各个级别的缓存。因此，无论何时，所有处理器核看到的内存视图是完全一样的。随着处理器中核数量的增多，这个“通知”的开销迅速增大，使得“缓存一致性”成为限制一个处理器中核数不能太多的一个重要因素。“缓存一致”系统中最坏的情况是，一个内存写操作会强迫每个核的缓存都进行更新，进而每个核都要对相邻的内存单元进行写操作。

相比之下，非“缓存一致”系统不会自动地更新其他核的缓存。它需要由程序员写清楚每个处理器核输出的各自不同的目标区域。从程序的视角看，这支持一个核仅负责一个输出或者一个小的输出集。通常，CPU遵循“缓存一致”原则，而GPU则不是。故GPU能够扩展到一个芯片内具有大数量的核心(流处理器簇)。

为了简单起见，我们假设4个线程块构成一个GPU内核程序。这样，GPU上就有4个内核程序，而CPU上有4个进程或线程。CPU也可能支持诸如“超线程”这样的机制，使得发生停顿事件(例如，访问缓存不命中)时，CPU能够处理其他的进程或线程。这样，我们就可以把CPU上进程的数量提高到8，从而得到性能的提升。然而，在某个时间点上，甚至在进程数小于核数的时候，CPU可能会遇到线程过多的情况。

这时，内存带宽就变得很拥挤，缓存的利用率急剧下降，导致性能降低而不是增高。

在GPU上，4个线程块无论如何也不能满足4个SM的处理能力。每个SM最大能处理8个线程块(开普勒架构中为16个线程块)。因此，我们需要$8 \times 4=32$个线程块才能填满4个SM。既然需要完成4个不同的操作，我们就可以借助其流处理功能(参见第8章关于使用多GPU的内容)，在费米架构GPU上同时启动4个内核程序。最终，我们总共可以启动16个线程块来并行处理4个数组。若采用CPU，则一次处理一个数组效率会更高些，因为这会提高缓存的利用率。总之，使用GPU时，我们必须确保总是有足够多的线程块(通常至少是GPU内SM数量的8 ~ 16倍)。

## 2.6 弗林分类法

前面我们用到一个词“SIMD”。它来源于划分不同计算机架构的弗林分类法（Flynn’s taxonomy）。根据弗林分类法，计算机的结构类型有：

- SIMD——单指令，多数据
- MIMD——多指令，多数据
- SISD——单指令，单数据
- MISD——多指令，单数据

绝大多数人熟悉的标准串行程序设计遵循的是SISD模型，即在任何时间点上只有一个指令流在处理一个数据项，这相当于一个单核CPU在一个时刻只能执行一个任务。当然，它也可以通过所谓的“分时”（time-slicing）机制，即在多个任务间迅速切换，达到“同时”执行多个任务的效果。

我们今天看到的双核或4核桌面计算机就是MIMD系统。它具有一个线程或进程的工作池，操作系统负责逐个取出线程或进程，将他们分配到N个CPU核中的一个上执行。每个线程或进程具有一个独立的指令流，CPU内部包含了对不同指令流同时进行解码所需的全部控制逻辑。

SIMD系统尽可能简化了它的实现方法，针对数据并行模型，在任何时间点，只有一个指令流。这样，在CPU内部就只需要一套逻辑来对这个指令流进行解码和执行，而无须多个指令解码通路。由于从芯片内部移除了部分硅实体，因此相比它的MIMD兄弟，SIMD系统就可以做得更小、更便宜、能耗更低，并能够在更高的时钟频率下工作。

很多算法只需要对很少量的数据点进行这样或那样的处理。很多数据点常常被交给一条SIMD指令。例如，所有的数据点可能都是加上一个固定的偏移量，再乘以一个数据，例如放大因子。这就很容易地用SIMD指令来实现。实际上就是你编写的程序，从“对一个数据点进行一个操作”改为“对一组数据进行一个操作”。既然这组数据中每一个元素需要进行的操作或变换是固定的，所以“访问程序存储区取指令然后译码”只需进行一次。由于数据区间是有界且连续的，所以数据可以全部从内存中取出，而不是一次只取一个字。

然而，如果算法是对一个元素进行A变换而对另一个元素进行B变换，而对其他元素进行C变换，那么就很难用SIMD来实现了。除非这个算法由于非常常用而采用硬编码在硬件中实现，例如“先进的加密标准”（Advanced Encryption Standard，AES）”和H.264（一种视频压缩标准）。

与SIMD稍有不同，GPU实现的是被英伟达称为“单指令多线程”（Single Instruction Multiple Thread，SIMT）的模型。在这种模型中，SIMD指令的操作码跟CPU中硬件实现的方式不同，它指示的并不是一个固定的功能。程序员需要通过一个内核程序，指定每个线程的工作内容。因此，内核程序将统一读入数据，程序代码根据需要执行A、B或C变换。实际上，A、B、C是通过重复指令流而顺序执行的，只不过每次执行时屏蔽掉无须参与的线程。与仅支持SIMD的模型相比，从理论上说，这个模型更容易掌握。

## 2.7 常用的并行模式

很多并行处理问题都可以按照某种模式来分析。在很多程序中，尽管并不是每个人都意识到它们的存在，但我们也可以看到不同的模式，按照模式来分析，使得我们能够对问题进行深入的解构或抽象，这样就很容易找到解决问题的办法。

### 2.7.1 基于循环的模式

几乎任何一个编写过程序的人都会对循环很熟悉。不同循环语句（如，for、do...while、while）的主要区别在于入口、退出条件以及两次循环迭代之间是否会产生依赖。

循环的迭代依赖是指循环的一次迭代依赖于之前的一次或多次先前迭代的结果。这些依赖将并行算法的实现变得十分困难，而这是我们希望消除的。如果消除不了，则通常将循环分解成若干个循环块，块内的迭代是可以并行执行的。循环块0执行完后将结果送给循环块1，然后送给循环块2，以此类推。本书的后面有一个例子就是采用这种方法来处理前缀求和（prefix-sum）算法。

基于循环的迭代是实现并行化的模式中最容易的一个。如果循环间的依赖被消除掉了，那么剩下的问题就是在可用的处理器上如何划分工作。划分的原则是让处理器间通信量尽可能少，片内资源（GPU上的寄存器和共享内存，CPU上的一级/二级/三级缓存）的利用率尽可能高。糟糕的是，通信开销通常会随着分块数目的增多而迅速增大，成为提高性能的瓶颈、系统设计的败笔。

对问题的宏观分解应该依据可用的逻辑处理单元的数量。对于CPU，就是可用的逻辑硬件线程的数量；对于GPU，就是流处理器簇（SM）的数量乘以每个SM的最大工作负载。依赖于资源利用率、最大工作负荷和GPU模型，SM的最大工作负载取值范围是1~16块。请注意，我们使用的词是逻辑硬件线程而不是物理硬件线程。某些英特尔CPU采用所谓的“超线程”技术，在一个物理CPU核上支持多个逻辑线程。由于GPU在一个SM内运行多个线程块，所以我们需要用SM的数量乘以每个SM支持的最大块数。

在一个物理设备上支持多个线程可以使设备的吞吐率最大化，也就是说在某线程等待访存或者I/O类型的操作时，设备可以处理其他线程的工作。这个倍数的选择有助于在GPU上实现负载平衡（load balancing），并可以应用于改进新一代GPU。当数据划分导致负载不均时，这一点表现得尤为明显——某些块花费的时间远远大于其他块。这时，可以用几倍于SM数目的数量作为划分数据的基础。当一个SM空闲下来后，它可以去存放待处理块的“池子”里取一个块来处理。

然而对于CPU，过多的线程数量却可能会导致性能下降，这主要是由于上下文切换时，操作系统以软件的形式来完成。对缓存和内存带宽竞争的增多，也要求降低线程的数量。因此对于一个基于多核CPU的解决方案，通常它划分问题的粒度要远大于面向GPU的划分粒度。如果在GPU上解决同一个问题，你则要对数据进行重新划分，把它们划分成更小的数据块。

当考虑采用循环并行来处理一个串行程序时，最关键的是发现隐藏的依赖关系。在循环

体中仔细地查找，确保每一次迭代的计算结果不会被后面的迭代使用。对于绝大多数循环而言，循环计数通常是从 0 ~设置的最大值。当遇到反过来采用递减计数的循环，你就应该小心些。为什么这个程序员采用相反的计数方法呢？很可能就是循环中存在某种依赖。如果不了解这一点就将循环并行化了，很可能把其中的依赖破坏掉。

我们还需要考虑的另外一种情况是有一个内循环和多个外循环。如何将它们并行化呢？对于 CPU，由于你只有有限的线程，所以只能将这些外循环并行化。不过，像前面提到的那样，可以这样处理的前提是不存在循环迭代依赖。

如果分配给 GPU 执行的内循环是很小的，通常用一个线程块内的线程来处理。由于循环迭代是成组进行的，所以相邻的线程通常访问相邻的内存地址，这就有助于我们利用访存的局部性，这一点对 CUDA 程序设计十分重要。外循环的并行处理都是用线程块来实现的，这部分内容将在第 5 章详细介绍。

考虑到大多数循环是可以展开的，因此把内循环和外循环合并成一个循环。例如，图像处理算法中，沿 X 轴的处理是内循环，而沿 Y 轴的处理是外循环。可以通过把所有像素点看成是一个一维数组来展开循环，这样迭代就是沿像素点而不是图像坐标进行。尽管编程时麻烦一些，但是在每次循环包含的迭代次数很小时，收效很大。因为这些小的循环带来的循环开销相对每次迭代完成的有效工作比较大，所以这些循环的效率很低。

## 2.7.2 派生 / 汇集模式

派生 / 汇聚模式是一个在串行程序设计中常见的模式，该模式中包含多个同步点而且仅有一部分内容是可以并行处理的，即首先运行串行代码，当运行到某一点时会遇到一个并行区，这个并行区内的工作可以按某种方式分布到 P 个处理器上。这时，程序就“派生”（fork）出 N 个线程或进程来并行地完成这些工作。N 个线程或进程的执行是独立的、互不相关的，当其工作完成后，则“汇聚”（join）起来。在 OpenMP 中常常可以看见这种处理方法——程序员用编译指令语句定义可并行区，并行区中的代码被分成 N 个线程，随后再汇聚成单个线程。

如图 2-2 所示，有一个输入的数据项队列和三个处理单元（即 CPU 核），输入数据队列被分成三个小的数据队列，一个处理单元处理一个小的数据队列，每个队列的处理是互不相关的，处理结果分别写在结果队列的相应位置。

通常，派生 / 汇聚模式是采用数据的静态划分来实现，即串行代码派生出 N 个线程并把数据集等分到这 N 个线程上。如果每个数据块的处理时间相同的话，这种划分方法是很好的。但是，由于总的执行时间等于最慢线程的执行时间，所

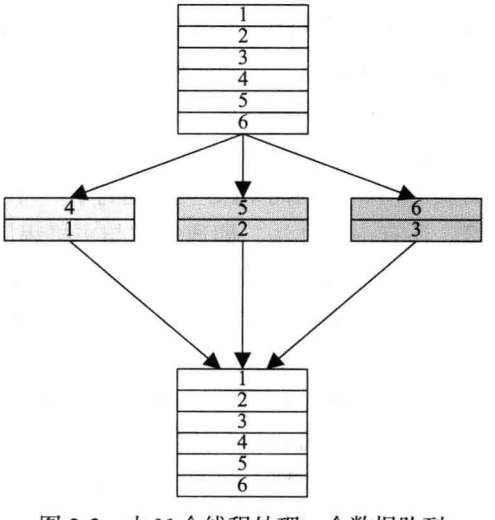



图 2-2 由 N 个线程处理一个数据队列

以如果分配给一个线程太多的工作，它将成为决定总时间的一个因素。

诸如 OpenMP 这样的系统跟 GPU 的方案类似，实现动态的调度分配。具体办法是，先创建一个“线程池”（对 GPU 而言是一个“块池”），然后池中的线程取一个任务执行，执行完后再取下一个。假设有 1 个任务需要 10 个单位时间才能完成，而其余 20 个任务需要 1 个单位时间就能完成，则它们只能分配到空闲的处理器核上执行。现在有一个双核处理器，则把那个需要 10 个单位时间的大任务和 5 个需要 1 个单位时间的小任务分配给核 1，而把其余的 15 个需要 1 个单位时间的小任务分配给核 2。这样，核 1 与核 2 就基本上可以同时完成任务了。

在图 2-2 的例子中，我们选择派生 3 个线程。既然队列中有 6 个数据，为什么不派生 6 个线程呢？这是因为在实际工作中，我们要处理的数据多达好几百万，无论用哪种方法，派生一百万个线程都会使任何一个操作系统以某种方式崩溃掉。

通常，操作系统执行的是一个“公平的”调度策略。因此，每个线程都需要按顺序，分配到 4 个可用的处理器核中的某一个上处理，每个线程都需要一个它自己的内存空间，例如，在 Windows 操作系统中，每个线程需要 1MB 的栈空间。这就意味着，在派生出足够多的线程前，我们已经迅速地用尽了全部的内存空间。

因此，对于 CPU 而言，程序员或者多数多线程库通常是按照处理器的个数来派生相同数目的逻辑处理器线程。由于 CPU 创建或删除一个线程的开销是很大的，而且线程过多也会降低处理器的利用率，所以常常使用一个“工人”线程池，池中的“工人”每次从待处理的任务队列中取一个任务来处理，处理完后再取下一个。

对于 GPU 则相反，我们的确需要成千上万个线程。我们还是使用在很多先进的 CPU 调度程序中使用过的线程池的概念，不过将“线程池”改为“线程块池”更好。GPU 上可以并发执行的“线程块”的数目存在一个上限。每个线程块内包含若干个线程。每个线程块内包含的线程的数目和并发执行的“线程块”的数目会随着不同系列的 GPU 而不同。

派生 / 汇聚模式常常用于并发事件的数目事先并不确定的问题。遍历一个树形结构或者路径搜索这类算法，在遇到另一个节点或路径时，就很可能会派生出额外的线程。当所有的路径都被考查后，这些线程就汇聚回线程池中或者汇聚后又开始新一轮的派生。

由于在启动内核程序时，块 / 线程的数量是固定的，所以 GPU 并不是天生就支持这种模式。额外的块只能由主机程序而不是内核程序启动。因此，在 GPU 上实现这类算法一般都需要启动一系列的 GPU 内核程序，一个内核程序要产生启动下一个内核程序所需的工作环境。还有一种办法，即通知或与主机程序共同，启动额外的并发内核程序。因为 GPU 是被设计来执行固定数目的并发线程，所以无论哪种方法实际效果都不算太好。为了解决这个问题，开普勒架构 GPU 引入了“动态并行性”（dynamic parallelism）的概念。关于这个概念的更多内容，请参见第 12 章。

在求解某些问题时，内核程序内部的并发性会不断变化，内部也会出现一些问题。为此，线程之间需要进行通信与协调。在 GPU 的一个线程块内，线程之间通信与协调可以通过很多方法来实现。例如，假设你有一个 $8 \times 8$ 的块矩阵，很多块仅需要 64 个工作线程。然而，很可能其他块却需要使用 256 个线程。你可以在每个块上同时启动 256 个线程，这时多

数线程处于空闲状态直到需要它们进行工作。由于这些空闲进程占用了一定的资源，会限制整个系统的吞吐率，但它们在空闲时不会消耗 GPU 的任何执行时间。这样就允许线程使用靠近处理器的更快的共享内存，而不是创建一系列需要同步的操作步骤，而同步这些操作步骤需要使用较慢的全局内存并启动多个内核程序。内存的类型将在第 6 章介绍。

最后，新的 GPU 支持更快的原子操作和同步原语。除了可以实现同步外，这些同步原语还可以实现线程间通信，本书的后面部分将给出这方面的例子。

### 2.7.3 分条 / 分块

使用 CUDA 来解决问题，都要求程序员把问题分成若干个小块，即分条 / 分块。绝大多数并行处理方法也是以不同的形式来使用“条 / 块化”的概念。甚至像气候模型这样巨大的超级计算问题也必须分为成千上万个块，每个块送到计算机中的一个处理单元上去处理。这种并行处理方法在可扩展方面具有很大的优势。

在很多方面，GPU 与集成在单个芯片上的对称多处理器系统非常类似。每个流处理器簇（SM）就是一个自主的处理器，能够同时运行多个线程块，每个线程块通常有 256 或者 512 个线程。若干个 SM 集成在一个 GPU 上，共享一个公共的全局内存空间。它们同时工作时，一个 GPU（GTX680）的峰值性能可达 3 Tflops。

尽管峰值性能会给你留下深刻的印象，但是达到这个性能却需要一个精心设计的程序，因为这个峰值性能并不包括诸如访存这样的操作，而这些操作却是影响任何一个实际程序性能的关键因素。无论在什么平台上，为了达到高性能，就必须很好地了解硬件的知识并深刻理解两个重要的概念——并发性和局部性。

许多问题中都存在并发性。可能是由于先前串行程序的背景，你也许不能立刻就看出问题中的并发性。而“条 / 块模型”就很直观地展示了并发性的概念。在二维空间里想象一个问题——数据的一个平面组织，它可以理解为将一个网格覆盖在问题空间上。在三维空间里想象一个问题，就像一个魔方（Rubik's Cube），可以把它理解为把一组块映射到问题空间中。

CUDA 提供的是简单二维网格模型。对于很多问题，这样的模型就足够了。如果在一个块内，你的工作是线性分布的，那么你可以很好地将其分解成 CUDA 块。由于在一个 SM 内，最多可以分配 16 个块，而在一个 GPU 内有 16 个（有些是 32 个）SM，所以把问题分成 256 个甚至更多的块都可以。实际上，我们更倾向于把一个块内的元素总数限制为 128、256 或者 512，这样有助于在一个典型的数据集内划分出更多数量的块。

当考虑并发性时，还可以考虑是否可以采用指令级并行性（ILP）。通常，人们从理论上认为一个线程只提供一个数据输出。但是，如果 GPU 上已经充满了线程，同时还有很多数据需要处理，这时我们能够进一步提高吞吐量吗？答案是肯定的，但只能借助于 ILP。

实现 ILP 的基础是指令流可以在处理器内部以流水线的方式执行。因此，与“顺序执行 4 个加法操作”（压入—等待—压入—等待—压入—等待—压入—等待）相比，“把 4 个加法操作压入流水线队列、等待然后同时收到 4 个结果”（压入—压入—压入—压入—等待）的效率更高。对于绝大多数 GPU，你会发现每个线程采用 4 个 ILP 级操作是最佳的。第 9 章中有更

详细的研究和例子。如果可能的话，我们更愿意让每个线程只处理 N 个元素，这样就不会导致工作线程的总数变少了。

### 2.7.4 分而治之

分而治之模式也是一种把大问题分解成的小问题的模式，其中每个小问题都是可控制的。通过把这些小的、单独的计算汇集在一起，使得一个大的问题得到解决。

常见的分而治之的算法使用“递归”(recursion) 来实现，“快速排序”(quick sort) 就是一个典型的例子。该算法反复递归地把数据一分为二，一部分是位于支点 (pivot point) 之上的那些点，另一部分是位于支点之下的那些点。最后，当某部分仅包含两个数据时，则对它们做“比较和交换”处理。

绝大多数递归算法可以用迭代模型来表示。由于迭代模型较适合于 GPU 基本的条块划分模型，所以该模型易于映射到 GPU 上。

费米架构 GPU 也支持递归算法。尽管使用 CPU 时，你必须了解最大调用深度并将其转换成栈空间使用。所以你可以调用 API cudaDeviceGetLimit() 来查询可用的栈空间，也可以调用 API cudaDeviceSetLimit() 来设置需要的栈空间。如果没有申请到足够的栈空间，CPU 将产生一个软件故障。诸如 Parallel Nsight 和 CUDA-GDB 这样的调试工具可以检测出像“栈溢出”(stack overflow) 这样的错误。

在选择递归算法时，你必须在开发时间与程序性能之间做出一个折中的选择。递归算法较易于理解，与将其转换成一个迭代的方法相比，编码实现递归算法也比较容易。但是所有的递归调用都需要把所有的形参和全部的局部变量压入栈。GPU 和 CPU 实现栈的方法是相同的，都是从全局内存中划出一块存储区间作为栈。尽管 CPU 和费米架构 GPU 都用缓存栈，但与使用寄存器来传递数据相比，这还是很慢的。所以，在可能的情况下最好还是使用迭代的方法，这样可以获得更好的执行性能，并可以在更大范围的 GPU 硬件上运行。

## 2.8 本章小结

至此，我们已经对并行处理的概念及其如何应用到 GPU 工业领域中，有了一个全面的了解。本书的写作目的并不是全面深入地论述并行处理，因为这方面的书籍已经很多了。我们只是希望读者能够感受到并行程序设计的理念，不再按照串行程序设计的思路来考虑程序设计问题。

在后续的章节中，我们将通过分析实际例子，详细介绍上述基本概念。我们还将分析并行前缀求和算法。这个算法允许对一个共享的数组同时进行多个写操作，而不会发生“一个写操作写到另外一个写操作的数据上”这样的错误。这类问题在串行程序设计中不会出现，无须考虑。

随着并行处理带来的复杂度的提高，需要程序员把并发性和局部性当作关键问题事先考虑。在设计面向 GPU 的任何软件时，都应该时刻把这两个概念牢记于心。

第 3 章

# CUDA 硬件概述

## 3.1 PC 架构

首先，我们看看当下许多 PC 中都使用的酷睿 2（Core 2）处理器的典型架构，然后分析一下它是如何影响我们使用 GPU 加速器的（如图 3-1 所示）。

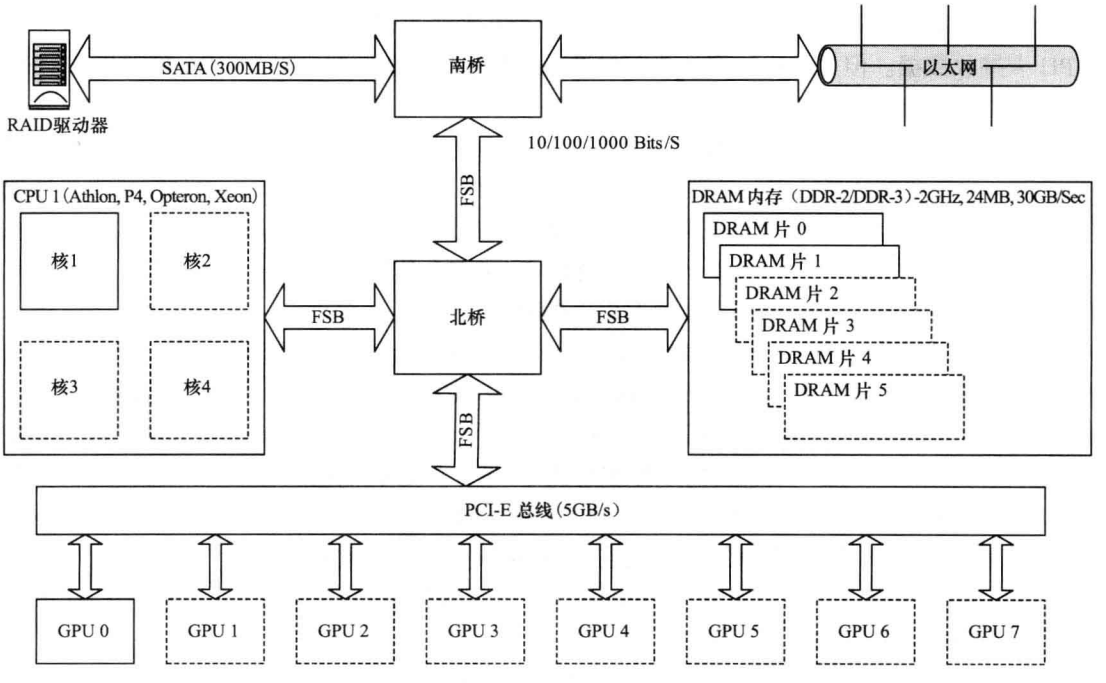



图 3-1　典型的酷睿 2（Core 2）系列处理器的结构图

由于所有的 GPU 设备都是通过 PCI-E（Peripheral Communications Interconnect Express）总线与处理器相连，所以我们以 PCI-E 2.0 总线标准来讨论本章内容。PCI-E 2.0 是目前最快的总线标准，它的传输速率为 5GB/s。在撰写本书的过程中，PCI-E 3.0 已经问世了，它的带

宽明显提高了。

然而，为了从处理器中获取数据，我们需要通过与低速前端总线（Front-Side Bus，FSB）连接的北桥（Northbridge）。理论上，FSB的时钟频率最高只能达到1600 MHz，而很多实际设计的产品就更低了。这通常只是一个高速处理器时钟频率的1/3。

访问内存也需要经过北桥，访问外设则需要经过北桥和南桥（Southbridge）。北桥服务于所有的高速设备，如内存、CPU、PCI-E总线接口等；而“南桥”则服务于低速设备，如硬盘、USB、键盘、网络接口，等等。当然，把硬盘控制器直接连接到PCI-E总线接口上也是可能的。实际上，在这样的系统中，这是要获得高速RAID数据访问的唯一正确方式。

PCI-E是一个很有意思的总线。与其上一代PCI（Peripheral Component Interconnect，外围设备互连）总线不同，PCI-E提供一个确定的带宽。在原先的PCI系统中，每一个设备都可以使用总线的全部带宽，但一次只能让一个设备使用。因此，你增加的PCI卡越多，每个卡能够获得的可用带宽就越少。PCI-E总线通过引入PCI-E通道（lane）解决了这个问题。这些通道是一些高速的串行链路，这些链路组合在一起构成了X1、X2、X4、X8或X16链路。目前，绝大多数GPU使用的至少是如图3-1所示的PCI-E 2.0的X16规范，此配置下提供5 GB/s的全双工总线。这意味着，数据的传入与传出可以同时进行并享有同样的速度。也就是说，我们在以5GB/s的速度向GPU卡传送数据的同时，还能够以5GB/s的速度从GPU卡接收数据。但是，这并不意味着如果不接收数据，我们就可以10GB/s的速度向GPU卡传送数据（即带宽是不可以累加的）。

在一个典型的超级计算机或者一个台式机应用程序中，我们常常需要处理一个很大的数据集。一个超级计算机需要处理上千万亿字节（PB）的数据，而一个桌面计算机也要处理数十亿字节（GB）的高分辨率视频图像。这两种情况都需要从外设取来大量的数据，单块100MB/s的硬盘每分钟只能上传6GB的数据。按照这个速度，读取一个标准的1万亿字节（TB）硬盘上的全部内容需要两个半小时以上的时间。

如果使用集群系统中常用的MPI（Message Passing Interface）作为通信软件，像图3-1这样将以太网（Ethernet）接口连接到南桥芯片而不是PCI-E总线，构成的通信延迟是很大的。因此，诸如InfiniBand这样的专用高速互连设备或者10千兆位（Gigabit）以太网卡常常连接到PCI-E总线上。不过，这就占用了原本可用于GPU的总线插槽。之前，并没有直接用于GPU的MPI接口函数。这类系统的所有通信都需要经过PCI-E总线连通到CPU，然后再原路返回。CUDA 4.0 SDK提供的GPU直连（GPU-Direct）技术就解决了这个问题。借助SDK的支持，InfiniBand卡就可以直接与GPU通信，而无须先经过CPU转发。SDK中的这项升级还支持GPU与GPU的直接通信。

Nehalem架构中有很多新的变化，其中最主要的变化就是用X58芯片组代替了“北桥”和“南桥”芯片。Nehalem架构引入了快速通道互联（Quick Path Interconnect，QPI）技术，该技术明显优于“前端总线”（Front Side Bus，FSB），达到了与AMD公司的超传输（HyperTransport）相当的水平。QPI是一种高速的、可用于与其他设备或CPU直接通信的互连结构。在一个标准的Nehalem系统中，QPI的作用是连接内存子系统，并通过X58芯片组

连接 PCI-E 子系统（如图 3-2 所示）。与 Extreme/Xeon 型号的处理器配合时，QPI 的工作速率要么是 4.8GT/s[⊖]，要么是 6.4GT/s。

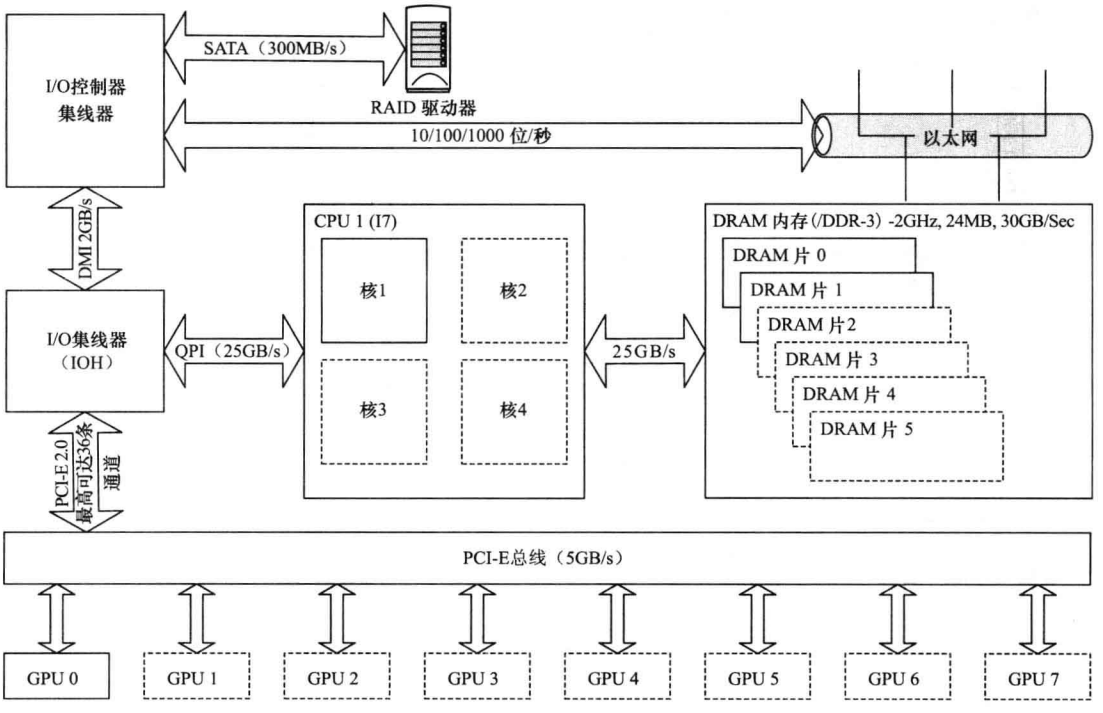



图 3-2 Nehalem/X58 系统

当使用 X58 芯片组和 LGA1366 处理器插槽时，共计有 36 个 PCI-E 通道。这就意味着，配置为 X16 时最多可以支持 2 个 GPU 卡，配置为 X8 时最多可以支持 4 个 GPU 卡。在 LGA2011 插槽出现之前，这是为 GPU 卡数据传输提供的最好的带宽解决方案。

在较小的 P55 芯片组中，也可以使用 X58 方案。不过，这时仅有 16 个 PCI-E 通道。这就意味着，配置为 X16 时仅支持 1 个 GPU 卡，配置为 X8 时最多可以支持 2 个 GPU 卡。

从 I7/X58 芯片组开始，英特尔公司引入了如图 3-3 所示的沙桥（Sandy Bridge）设计方案。其最引人注目的改进之一是支持传输速率可达 600MB/s 的 SATA-3 标准。通过与固态硬盘（SSD）结合，在存 / 取数据时，“沙桥”可以提供很高的输入 / 输出性能。

沙桥的另外一个主要进步是引入了 AVX（Advanced Vector Extensions，高级向量扩展）指令集，该指令集也同时被 AMD 的处理器支持。AVX 允许向量指令最多可以并行处理 4 个双精度浮点数（256 位或 32 字节）。这是一个很有趣的改进，可以使 CPU 上的计算密集型应用程序获得一个很高的加速比。

⊖ GT/s 指 QPI 总线速率，是实际时钟频率的 2 倍。

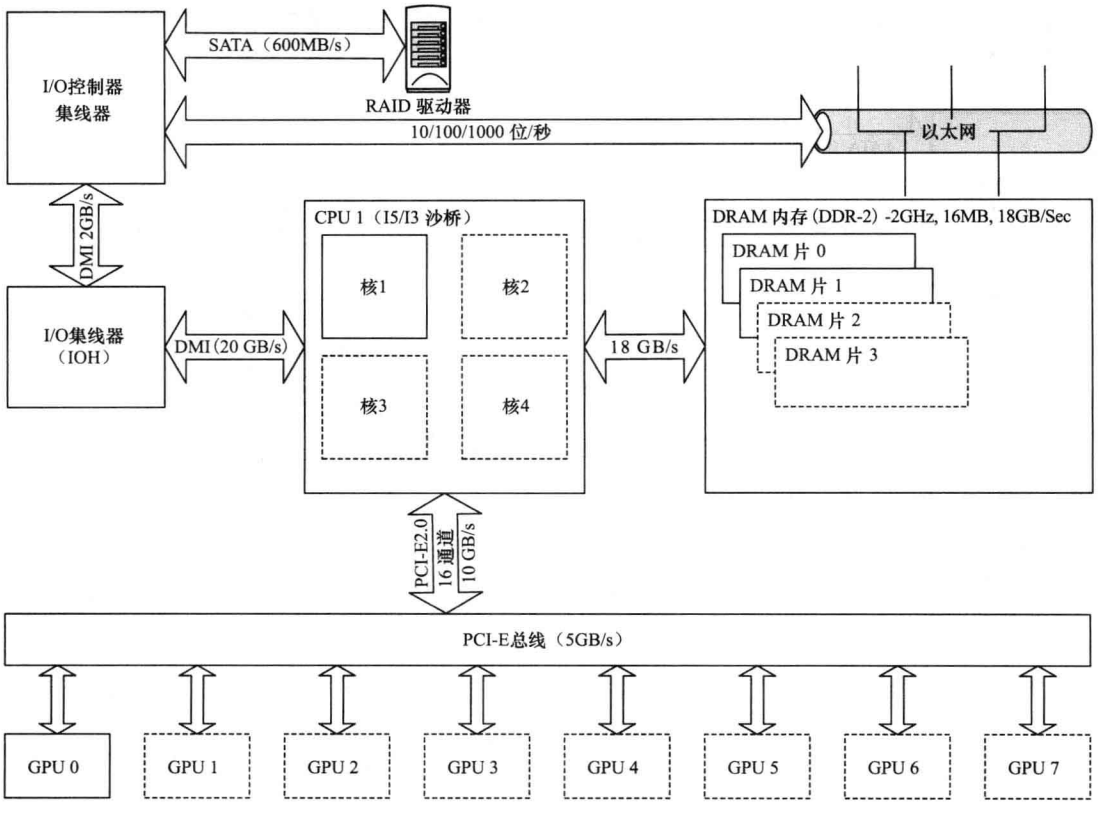



图 3-3 沙桥设计图

然而，需要注意的是，LGA 1155 插槽的沙桥设计方案存在一个很大的问题，就是仅支持 16 个 PCI-E 通道，这将 PCI-E 的理论带宽限制在 16GB/s 以内，而实际带宽为 10GB/s。在桌面处理器上，英特尔公司摒弃了向 CPU 中集成更多 PCI-E 通道的路线。仅仅是面向服务器的 LGA 2011 “沙桥 -E” 插槽，才拥有数量可观的 PCI-E 通道（40 个）。

相比英特尔公司的技术，AMD 的设计方案又是怎样的呢？与英特尔公司不断地减少 PCI-E 通道的数量（除服务器系列产品外）不同，AMD 始终保持一个固定不变的数量。AMD 的 FX 芯片组，要么支持 2 个 X16 设备，要么支持 4 个 X8 PCI-E 设备。AMD 3+ 插槽与 990FX 芯片组一起提供了强劲的工作动力，即 6GB/s 的 SATA 端口以及最多可达 4 个的 X16 PCI-E 插槽（通常以 X8 的速度运行）。

英特尔与 AMD 的主要差别之一是，同一价位对应的处理核数是不同的。如果你考虑的仅仅是实际处理器核而不是逻辑核（如超线程），那么对于同一价位，AMD 的处理器中通常会有较多的核。然而，英特尔处理器核的性能要高一些。因此，选择哪一种处理器主要取决你需要支持的 GPU 的数量以及分配给处理器核的工作负载水平。

在英特尔的设计方案中，你会发现，除了连接主存的带宽有差别外，系统周围的带宽都是基本相同的。在高端系统中，英特尔使用 3 个或者 4 个通道的内存；仅在低端系统中，英

特尔才使用双通道的内存。而 AMD 只使用双通道的内存，这就导致 CPU 与内存之间的可用带宽明显减少（如图 3-4 所示）。

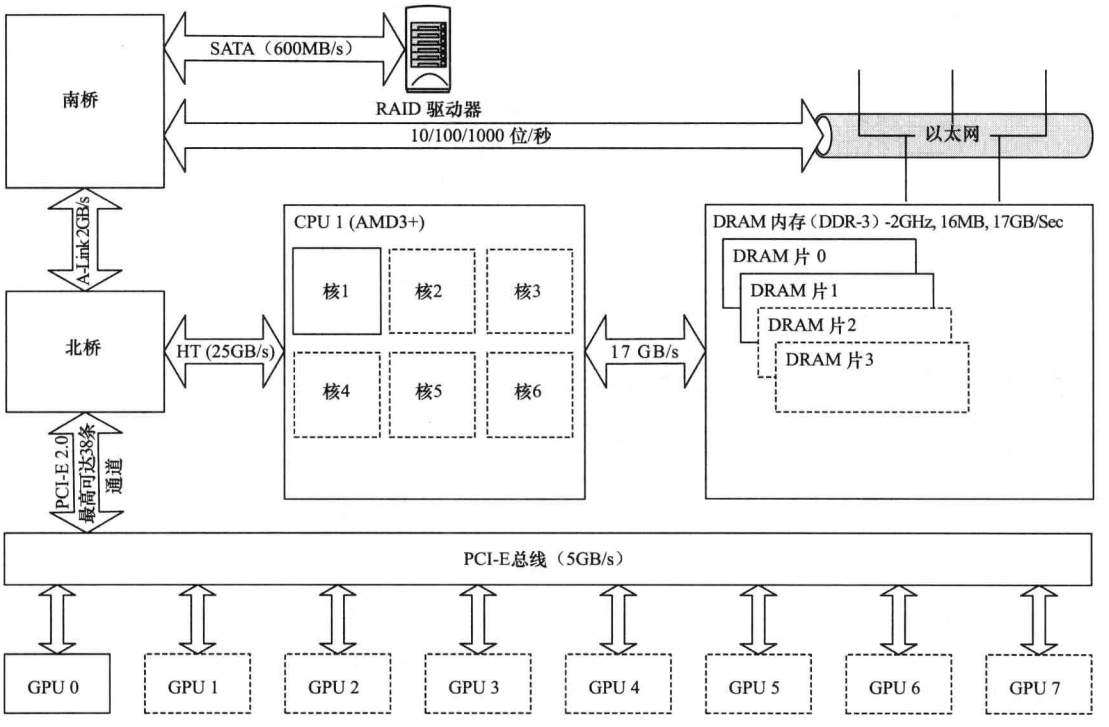



图 3-4 AMD 结构图

相比英特尔的产品，AMD 芯片组的一个重要优点是支持最高可达 6 个的 6GB/s 的 SATA（Serial ATA）端口。如果考虑到系统中最慢的部件通常会限制整个系统的吞吐率，你就需要在这方面好好考虑再做选择。如果系统使用了好几个固态硬盘，SATA3 将很快使“南桥”带宽超载。使用 PCI-E 总线也许是一个更好的选择，但是成本将会显著提高。

## 3.2 GPU 硬件结构

GPU 的硬件结构与 CPU 的硬件结构有着根本的不同，图 3-5 显示了一个位于 PCI-E 总线另一侧的多 GPU 系统。

从图中可以看出，GPU 的硬件由以下几个关键模块组成：

- 内存（全局的、常量的、共享的）
- 流处理器簇
- 流处理器

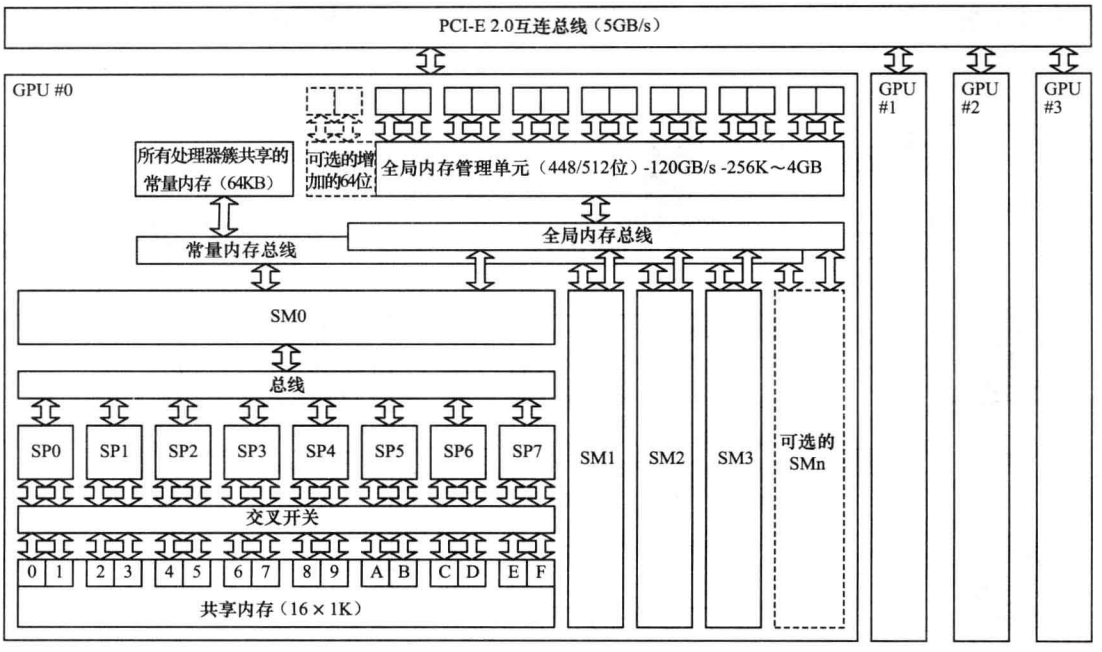



图 3-5 GPU（G80/GT200）卡的组成模块图

这里最值得注意的是，GPU 实际上是一个 SM 的阵列，每个 SM 包含 $N$ 个核（G80 和 GT200 中有 8 个核，费米架构中有 32 ~ 48 个核，开普勒架构中至少再增加 8 个核，如图 3-6 所示）。一个 GPU 设备中包含一个或多个 SM，这是处理器具有可扩展性的关键因素。如果向设备中增加更多的 SM，GPU 就可以在同一时刻处理更多的任务，或者对于同一任务，如果有足够的并行性的话，GPU 可以更快地完成它。

像 CPU 一样，如果程序员编写的代码将处理器使用核的数量限制为 $N$ 个，比如 2 个，那么即便是 CPU 厂商制造出 4 核的设备，用户也不会从中受益。因此，当计算机从双核 CPU 过渡到 4 核 CPU 时，很多软件都必须重写以利用新增加的核。通过不断地增加 SM 的数量以及每个 SM 中的核数，英伟达硬件的性能持续地提高。设计软件时，应该意识到下一代处理器中 SM 的数量或者每个 SM 中的核数可能翻一番。

现在让我们更深入地看看 SM。每个 SM 都是由不同数量的一些关键部件组成，为了简单起见，没有在图中画出。最重要的部分是每个 SM 中有若干个 SP，图中显示的是 8 个 SP，在费米架构中增加至 32 ~ 48 个，在开普勒架构中增加到 192 个。毋庸置疑，下一代产品中每个 SM 中 SP 的数量极有可能继续增加。

每个 SM 都需要访问一个所谓的寄存器文件（register File），这是一组能够以与 SP 相同速度工作的存储单元，所以访问这组存储单元几乎不需要任何等待时间。不同型号 GPU 中，寄存器文件的大小可能是不同的。它用来存储 SP 上运行的线程内部活跃的寄存器。另外，还有一个只供每个 SM 内部访问的共享内存（shared memory），这可以用作“程序可控的”高速缓存。与 CPU 内部的高速缓存不同，它没有自动完成数据替换的硬件逻辑——它完全是由

程序员控制的。

对于纹理内存（texture memory）、常量内存（constant memory）和全局内存（global memory），每一个 SM 都分别设置有独立访问它们的总线。其中，纹理内存是针对全局内存的一个特殊视图，用来存储插值（interpolation）计算所需的数据，例如，显示 2D 或 3D 图像时需要的查找表。它拥有基于硬件进行插值的特性。常量内存用于存储那些只读的数据，所有的 GPU 卡均对其进行缓存。与纹理内存一样，常量内存也是全局内存建立的一个视图。

图形卡通过 GDDR（Graphic Double Data Rate）接口访问全局内存。GDDR 是 DDR（Double Data Rate）内存的一个高速版本，其内存总线宽度最大可达 512 位，提供的带宽是 CPU 对应带宽的 5 ~ 10 倍，在费米架构 GPU 中最高可达 190GB/s。

每个 SM 还有两个甚至更多的专用单元（Special-Purpose Unit，SPU），SPU 专门执行诸如高速的 24 位正弦函数 / 余弦函数 / 指数函数操作等类似的特殊硬件指令。在 GT200 和费米架构 GPU 上还设置有双精度浮点运算单元。

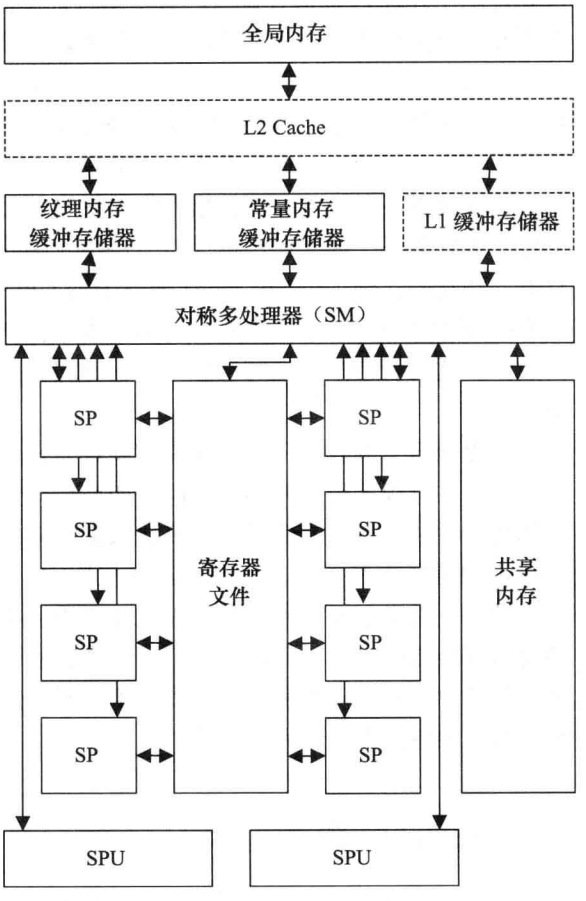



图 3-6 SM 内部组成结构图

## 3.3 CPU 与 GPU

现在你已经对 GPU 的硬件结构有了初步的了解，你也许会认为它很有意思。但对于程序设计，这意味着什么呢？

参加过大型项目开发的人都知道，项目开发分成不同的阶段，每个阶段的任务由不同的工作组来完成。可能有一个软件规格指定组，一个设计组、一个编码组和一个测试组。让团队中每个人完全了解开发链中在他上游和下游的工作，这对高质量地完成项目开发是很有益处的。

以测试为例，如果设计人员不考虑测试，那么他就不会在程序中设置用于检测因具体软件而引发的硬件故障的测试手段。如果只有发生了硬件失效，测试组才能测试出此硬件故障，那么只能修改硬件让它失效。无疑这是很难做到的。相反，编程人员设计一个反映硬件错误标志的软件标志就容易得多，这样硬件故障就很容易检测出来。如果在测试组里工作，你就会看到如果不这么做，测试工作有多难。除非你狭隘地看待你的岗位职责，你可能才会

说测试不关我的事。

最优秀的工程师都是那些关注在他工作流程之前和之后工作的人。作为软件工程师，知道硬件的工作原理总是有益的。对于串行代码的执行，人们可能会关注到它是怎么工作的，但这并没有达到非如此不可的地步。大多数开发人员可能从来没有学习过计算机体系结构的课程或者读过这方面的书籍，这是很令人惭愧的。这就是我们曾经看到有如此多的低效率软件的主要原因之一。以我自己的经历为例，我在 11 岁时开始学习 BASIC 语言，在 14 岁时就使用 Z80 汇编语言，但只有进入大学后才真正开始理解计算机体系结构。

嵌入式领域的工作经历会培养你的硬件方面的动手能力。由于没有 Windows 操作系统帮助你管理处理器，因此程序设计是一件很底层的工作。嵌入式项目中，上市的产品通常数以百万。糟糕的代码（sloppy code）意味着对 CPU 和现有内存的低效使用，它反过来需要更快的 CPU 或更多的内存。对于一百万件产品，每件附加 50 美分的费用，就是增加了 50 万美元的成本。这也会带来增加的设计与编程时间。显而易见，写更好的代码比买更多的硬件要划算得多。

时至今日，并行程序设计还是与硬件紧密相连的。如果你只埋头编程而不关心程序的性能，那么并行程序设计并不难。但是如果想充分发挥硬件的性能，你就需要知道硬件是如何工作的。举个生活中的例子，大多数人都能够在一档的情况下安全、缓慢地驾驶汽车，但是如果不知道还有其他档，或者不具备使用它们的知识，你就永远不能快速地从 A 点到达 B 点。学习硬件类似于学习汽车驾驶时使用手动换档——刚开始有点复杂，但熟能生巧，一段时间后就会变得自然。用相同的比喻，你也可以买一辆带自动档的汽车，就像使用一个由熟悉底层硬件工作机理的程序员开发的函数库。但是，在不了解硬件工作原理的情况下开发软件，其结果往往不是最优的实现。

## 3.4 GPU 计算能力

CUDA 支持多个级别的计算。最早的 G80 系列图形卡就是在配有 CUDA 的第一个版本的情况下上市的。硬件的计算能力是固定的。为了升级到一个新的版本，用户必须升级硬件。听起来虽然像是英伟达公司试图强迫用户购买更多的硬件，但是它确实给用户带来了好处。因为当提升一个计算级别时，你就从一个老的平台迁移到了一个新的平台，新的图形卡与原先的图形卡价格相同，计算能力却翻了一番。英伟达公司至少每隔几年就推出一个新的平台，在 CUDA 出现的短短数年内，人们可以获得的计算能力已经有了巨大的提高。

不同计算能力之间的差别列表，作为本书的一部分，可以在附录 G 中找到。因此，我们仅仅在这里介绍不同计算能力之间的主要差别。作为开发者，这是需要知晓的。

### 3.4.1 计算能力 1.0

计算能力 1.0 出现在早期的图形卡上，例如，最初的 8800 Ultras 和许多 8000 系列卡以及 Tesla C/D/S870s 卡。计算能力 1.0 卡的性能缺陷主要与原子操作有关。原子操作是指那些

必须一次性完成、不会被其他线程中断的操作。要实现这一点，硬件就要在原子函数的入口实现一个栅栏点（barrier point）并确保相应操作（例如，加、减、求最小值、求最大值、逻辑与、逻辑或、逻辑异或等）作为一个整体来完成。计算能力 1.0 现在已经退出市场了。因此，无论为了什么目的和意义，这个限制都可以忽略不计。

### 3.4.2 计算能力 1.1

计算能力 1.1 出现在许多 9000 系列图形卡后期推出的产品上，例如，曾经红极一时的 9800 GTX 卡。相对于计算能力 1.0 的 G80 硬件，这些卡基于 G92 硬件。

计算能力 1.1 带来的最主要的变化是支持数据传送和内核程序的重叠执行。当然，这个变化出现在大多数，但不是所有的计算能力 1.1 卡上。SDK 调用 cudaGetDeviceProperties() 函数返回 deviceOverlap 属性，该属性定义了这项功能是否可用。实现这项功能需要一个很巧妙的重要改进——双缓冲（double buffering），其工作原理如图 3-7 所示。

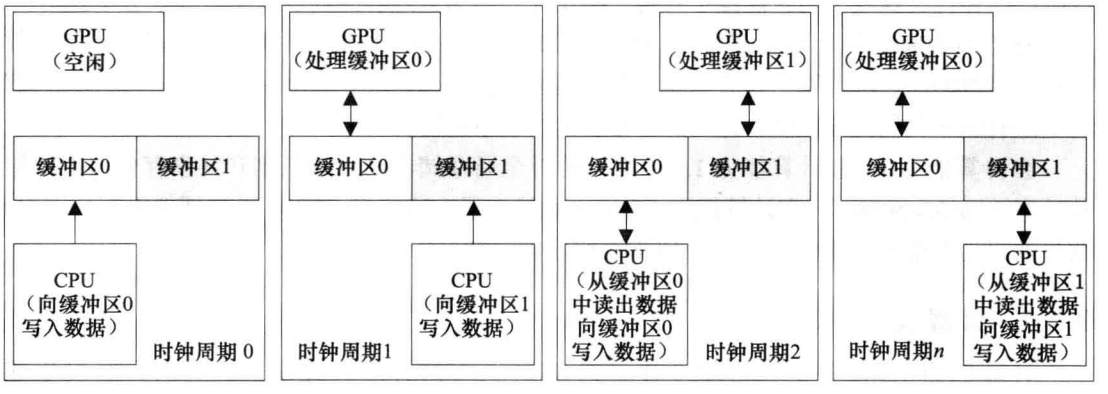



图 3-7 单条 GPU 的双缓冲区技术

要想实现这项功能，我们需要将通常使用的内存空间加倍。如果你的目标市场仅有 512 M 字节显存的卡的话，这一点也许是一个问题。但是，如果使用的是面向科学计算的 Tesla 卡，那么你拥有的 GPU 存储空间最高可达 6GB，实现这项功能就没问题了。让我们看看它的操作步骤，如下所示。

**时钟周期 0**：在 GPU 的存储空间中划分出两块缓冲区，CPU 将一个数据包写入“缓冲区 0”中。

**时钟周期 1**：CPU 调用 GPU 上的一个 CUDA 内核程序（即一个 GPU 任务），然后立即返回（这是一个非阻塞调用）。然后，CPU 从硬盘、网络或者其他地方取来一个数据包。与此同时，GPU 正在后台处理交给它的数据包。当数据包取来后，CPU 开始将其往“缓冲区 1”中写。

**时钟周期 2**：当 CPU 写完后，它又调用一个内核程序来处理“缓冲区 1”。然后检查在时钟周期 1 调用的、处理“缓冲区 0”的内核程序是否完成。如果还没有完成，CPU 则一直

等待直至它完成。完成后，CPU 取走“缓冲区 0”中的计算结果，然后把下一个数据包写入“缓冲区 0”。在这个过程中，本时钟周期一开始就启动的内核程序，则一直在处理 GPU 上“缓冲区 1”中的数据。

**时钟周期 N：**重复时钟周期 2。在 GPU 处理一个缓冲区的同时，让 CPU 选择另外一个缓冲区进行读或写操作。

GPU-CPU 和 CPU-GPU 的数据传送是在相对较慢的 PCI-E 总线（5GB/s）上进行的这个“双缓冲”技术显著地掩盖了通信延迟，使 CPU 和 GPU 都处于忙碌状态。

### 3.4.3 计算能力 1.2

计算能力 1.2 设备是与低端的 GT200 系列硬件一起出现的。最早的产品是 GTX260 和 GTX280 卡。随着 GT200 系列硬件的出现，英伟达公司通过将卡上的多处理器数量增倍，使得单个卡上的 CUDA 核（CUDA core）处理器的数量几乎增加了一倍。因此，与上一代产品 G80/G92 相比，这些卡的性能翻了一番。CUDA 核与多处理器的内容将在后续章节中介绍。

在将多处理器数量增倍的同时，英伟达公司还将一个多处理器中并发执行的线程束的数量从 24 增加到了 32。“束”是在一个多处理器内执行的代码块。每个多处理器内可调度“束”的增加有利于我们提高性能，这一点将在后续章节中介绍。

在计算能力 1.0 和计算能力 1.1 中常见的对全局存储器的访问限制和共享存储器中存储片冲突（bank conflict），在计算能力 1.2 中大大减少了。这使得 GT200 系列硬件更容易编程，明显地提高了很多以前艰难编写的 CUDA 程序的性能。

### 3.4.4 计算能力 1.3

计算能力 1.3 设备是在 GT200 升级到 GT200 a/b 修订版时提出的，这次升级发生在 GT200 系列发布不久。从那时开始，几乎所有的高端卡都兼容计算能力 1.3。

计算能力 1.3 带来的最主要的变化是支持有限的双精度浮点运算。由于 GPU 是针对图形处理的，所以对快速的单精度浮点运算要求很高，但是对双精度浮点运算要求有限。双精度浮点运算性能通常要比单精度浮点运算性能低一个数量级，所以如果程序中只有单精度浮点运算，才能发挥硬件的最大功效。但是在很多情况下，单精度浮点运算和双精度浮点运算会同时出现在程序中，因此硬件中同时设置专用的单精度浮点运算单元和双精度浮点运算单元是最理想的。

### 3.4.5 计算能力 2.0

计算能力 2.0 设备是伴随费米架构硬件出现的。调整应用程序以适应费米架构的最初指导可参见英伟达公司的网站 http://developer.nvidia.com/cuda/nvidia-gpu-computing-documentation。

计算能力 2.x 硬件的主要改进如下：

- 在每个 SP 上引入了 16K ~ 48K 的一级（L1）缓存。
- 在每个 SM 上引入了一个共享的二级（L2）缓存。

- 在基于 Tesla 的设备上支持基于纠错码（Error Correcting Code，ECC）的内存检查和纠错。
- 在基于 Tesla 的设备上支持双复制（dual-copy）引擎。
- 将每个 SM 的共享内存容量从 16K 扩展到 48K。
- 为了优化数据的合并，数据必须以 128 字节对齐。
- 共享内存的片数从 16 增加到 32。

下面让我们选择几个重要的改进，详细分析一下它们的实现。首先，我们分析一下一级缓存以及引入它意味着什么。一级缓存是设置在芯片内的，它是最快的可用存储器。除了纹理和常量缓存外，计算能力 1.x 的硬件中并没有缓存。引入缓存，使程序员更容易编写出适合在 GPU 硬件上工作的程序，还允许应用程序不必遵循在编译时已知的存储器访问模式。但是，为了利用好缓存，应用程序要么需要具有一个顺序的存储器访问模式，要么需要对某些数据反复使用。

费米型硬件上的二级缓存容量最高可达 768K。重要的是，它是一个统一的缓存。这意味着它是一个共享的缓存，对所有的 SM 提供一个一致的视图。通过二级缓存来实现程序块间通信，要比通过全局原子操作实现快得多。访问 GPU 上的全局存储器，需要越过线程块，比较起来，使用共享缓存要快一个数量级。

对于数据中心（data center）而言，支持 ECC 存储是必须的，因为 ECC 存储器具有自动的检错和纠错功能。电子设备会产生少量的电磁辐射。当与其他设备靠得很近时，这个辐射会改变其他设备中存储单元的内容。虽然这种情况发生的概率很小，但是由于数据中心的设备摆放密度不断增加，出错的概率将会增大到无法接受的水平。因此，就需要引入 ECC 来检测和纠正在一个大型数据中心中可能出现的“单个二进制位反转错误”。当然，引入 ECC 技术会减少可用的 RAM 容量并降低访存带宽。这对图形卡而言是一个严重的缺点，所以目前仅有 Tesla 卡采用了 ECC 技术。

“双复制”引擎是将前面介绍过的“双缓冲技术”扩展应用到多“流”处理技术，“流”的概念将在后续章节中详细介绍。简单地说，“流”就是 N 个独立的内核程序以流水线的方式并行执行，如图 3-8 所示。


图 3-8 流的流水线处理技术

请注意图中内核程序段一个接一个执行的过程。每个复制操作被另一个流中执行的内核程序所隐藏。内核程序与复制引擎是并发执行的，因此相关器件的利用率达到最高。

需要说明的是，“双复制”引擎在绝大多数诸如 GTX480 或 GTX580 这样的高端费米型

GPU中是真实存在的。但是，只有Tesla卡中的双引擎对CUDA驱动程序是可见的，即可以由CUDA驱动程序直接操纵。

共享存储器变化较大，它存在于混合的一级缓存中。一级缓存的容量为64KB。但是为了保证向后兼容性（backward compatibility），必须从中至少划分出16KB的存储空间给共享存储器。这就意味着，一级缓存的实际容量至多为48KB。实际工作中还可以通过一个转换开关，将共享存储区和一级缓存区的功能相互转换，即共享存储器的容量为48KB而一级缓存的容量为16KB。共享存储器的容量从16K提升为48K，某些特定的程序会从中获得巨大的好处。

由于新一代GPU引入了一级和二级缓存，为优化访存而提出的对齐（alignment）要求就更加严格了。两级缓存中，缓存存储块（cache line）的大小均为128B。而每次访问缓存，取来的最少数据量就是一个“存储块”。因此，如果你的程序是顺序访问数据元素，那么这个要求会发挥很好的作用。事实上，绝大多数CUDA程序都是这么工作的，即一组线程读取的都是相邻的存储单元。不过这个改进也带来了一个新的限制，就是数据集应该是128B对齐的。

但是，如果你的程序中每个线程的访存模式是稀疏而分散的，那么你就应该屏蔽掉这个“对齐”要求，并转回到32位的缓存操作模式。

最后，我们再看看“共享存储器的片数从16增加到32”这个改进。新一代GPU会从中获得巨大的好处。它允许当前线程束（包含32个线程）中的每一个线程都可以向共享存储器中的某一片写入数据，而不会引起共享存储片冲突。

### 3.4.6 计算能力2.1

计算能力2.1出现在专门面向游戏市场的专用卡上，例如，GTX460和GTX560。这些设备在体系结构方面的改进如下：

- 每个SM中的CUDA核心由原先的32个增为48个。
- 每个SM中面向单精度浮点数的专用超越函数计算部件由4个增至8个。
- 双束调度器代替单束调度器。

x60系列卡对中端的游戏市场一直有很强的渗透能力，所以如果你的应用程序面向这类消费市场，那么了解上述改进的内涵是很重要的。

计算能力2.1硬件中一个值得注意的变化就是牺牲掉双精度浮点数的运算器件来换取CUDA核数量的增加。对于单精度浮点数和整数运算占主要地位的内核程序而言，这是一个很好的取舍。绝大多数游戏主要进行单精度浮点数和整数的数学计算，几乎不处理双精度浮点数。

一个线程束就是一组线程，在后续章节中我们将详细介绍。在计算能力2.0的硬件上，单束调度器需要两个时钟周期来从整个束中取出2条指令执行。在计算能力2.1的硬件上，双束调度器每两个时钟周期分发4条指令，而不是2条。在现在的SM硬件中，有3排、每排16个的CUDA核，共计48个CUDA核心，而不是原先的2排、每排16个的CUDA核。

如果英伟达公司还能再挤进 16 个一排的 CUDA 核，那就更理想了，也许在未来我们能看到这样的硬件[⊖]。

计算能力 2.1 硬件的确采用了类似于从最早的奔腾 CPU 开始就一直采用的超标量（superscalar）技术。要想充分利用所有的核，硬件需要在每一个线程中识别出指令级并行性（Instruction-Level Parallelism，ILP）。这与原先推荐的通用的线程级并行性（Thread-Level Parallelism，TLP）有很大的不同。要想体现出 ILP，指令之间应该是相互无关的。借助专用的向量类库是实现 ILP 最早的方法之一，本书的后续章节将详细介绍。

计算能力 2.1 硬件的性能是变化的。某些著名的应用，如 Folding at Home，使用计算能力 2.1 硬件的性能就非常好。其他诸如视频编码压缩这样的应用，由于很难从中挖掘出 ILP 而且存储器带宽是一个关键因素，所以实际使用性能就很糟糕。

截至撰写本书的时候，开普勒型 GPU 和新的计算能力 3.0 平台的最后细节，还没有公开地发布[⊜]。针对已经公布的开普勒型 GPU 特性，本书在第 12 章中将进行深入的讨论。

⊖ 作者的预言已经实现，比如 Tesla 系列的 K10、K20，GTX 系列的 680、TITAN 等，都是采用这一思路。

⊜ 截至 2013 年 6 月，不仅计算能力 3.0，甚至计算能力 3.5 的细节已经公布。

第 4 章

# CUDA 环境搭建

## 4.1 简介

本章面向从未接触过 CUDA 的初学者。我们将依次介绍如何在不同操作系统上安装 CUDA、有哪些可用的 CUDA 工具以及 CUDA 如何编译代码，最后介绍应用程序接口提供的错误处理手段，并帮助读者识别 CUDA 代码和开发过程中必然碰到的应用程序接口报错。

Windows、Mac 和 Linux 三大主流操作系统均支持 CUDA。最易于使用和学习 CUDA 程序开发的操作系统，应该是你最熟悉的那个。对于零基础的初学者，Windows 加上 Microsoft Visual C++ 可能是最好选择。在 Windows 和 Mac 上安装 CUDA 主要是一些点击操作，它们都提供了非常标准的集成开发环境，很适合 CUDA 程序开发。

## 4.2 在 Windows 下安装软件开发工具包

这里的安装以工具包 4.1 版为例。在基于 Windows 系统的个人计算机上安装 CUDA，需要一些组件，可以到英伟达开发者社区下载，入口在 http://developer.nvidia.com/cuda-toolkit-41。在本书付梓刊印之际，开发包 5.0 版已经处于待发布阶段。请到英伟达官网获取最新版本。

你需要事先安装好 Microsoft Visual Studio 2005、2008 或者 2010。接着首先要下载并安装对应于操作系统的英伟达开发驱动程序，下载地址同上。然后你还要依次下载并安装 32 位或 64 位版本的 CUDA 工具包、GPU 计算软件开发包及其软件开发包的示例程序。要确认你安装的程序版本号与你的操作系统是匹配的。建议的安装次序如下：

1）英伟达开发驱动程序

2）CUDA 工具包

3）CUDA 软件开发工具包

4）GPU 计算软件开发工具包

5）并行 Nsight 调试器

在 Windows 7 系统下，软件开发工具包把全部文件放置在 ProgramData 目录下。这个目录处于 C 盘，是隐藏的。为了查看其中的文件，可以通过桌面上的 CUDA 软件开发工具包图标进行浏览或者进入 Windows Folder Options（文件夹选项）对话框里进行设置，以允许查看隐藏文件，如图 4-1 所示。

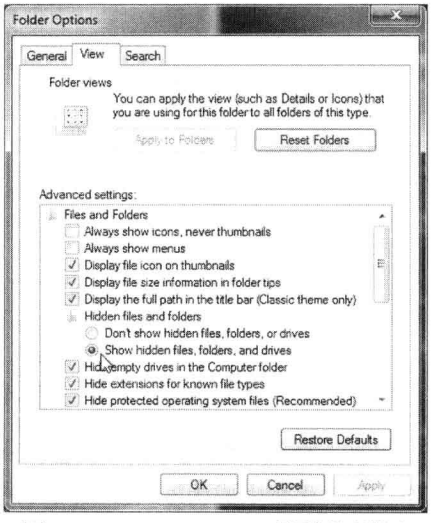



图 4-1 Folder Options 对话框里设置允许查看隐藏文件

## 4.3 Visual Studio

CUDA 支持 Visual Studio 版本的范围为 2005 ~ 2010，也支持大部分的学习版。学习版可以从 Microsoft 免费得到。专业版也能通过 DreamSpark 计划得到，需要在 https://www.dreamspark.com 网站注册为学生身份免费获得。

注册时，只需提供你的学校信息和身份编号。一旦注册成功，就可以下载 Visual Studio 软件以及很多其他开发工具。这个计划不仅面向美国的学术机构，也涵盖全世界的大学生。

综合来看，Visual Studio 2008 对 CUDA 的支持最好，它的编译速度比 Visual Studio 2010 更快。但 Visual Studio 2010 可以实现源代码的自动语法检查，这一特性非常实用。在使用一项未定义的类型时，它能使用红色下划线指明错误所在，与 Microsoft Word 里提示拼写错误是一样的。对于明显的问题，这一特性极其有用，它将大大节省不必要的编译次数。因此，建议使用 2010 版本进行 CUDA 开发，特别是当你可以从 DreamSpark 获取免费版时。

### 4.3.1 工程

为了快速新建一个工程，你可以选择一个软件开发工具包示例作蓝本，然后移除其中不需要的工程文件，并插入自己的源文件。你的 CUDA 源代码，应该包含“.cu”扩展名，这样它的编译会采用英伟达编译器而不是 Visual C 编译器。另一种新建工程的方式，可以通过工程模板向导，方便地建立一个基本的工程框架，细节将在后面看到。

### 4.3.2 64 位用户

当使用 Windows 64 位版本时，要注意一些工程文件默认设置为以 32 位应用程序运行。因此，当尝试生成程序[⊖]时，你可能会收到以下错误消息：“致命错误 LNK1181：无法打开输入文件‘cutil32D.lib’”。

这是因为没有安装该库的缘故。你很有可能只安装了对应 64 位 Windows 的 64 位版本软件开发工具包。要更正此问题，需要把目标平台从 32 位改为 64 位。可以使用 Visual Studio 的 Build（生成）菜单，然后把平台改变为 X64，如图 4-2 所示。

⊖ “生成”这里指通过菜单中的“Build”（生成）菜单条编译并链接程序。——译者注

# 有关此电子图书的说明

本人由于一些便利条件，可以帮您提供各种中文电子图书资料，且质量均为清晰的PDF图片格式，质量要高于网上大量传播的一些超星 PDG 的图书。方便阅读和携带。只要图书不是太新，文学、法律、计算机、人文、经济、医学、工业、学术等方面的图书，我都可以帮您找到电子版本。所以，当你想要看什么图书时，可以联系我。我的 QQ是：**89039855**，大家可以在 QQ上联系我。

此 PDF 文件为本人亲自制作，请各位爱书之人尊重个人劳动，敬请您不要修改此 PDF文件。因为这些图书都是有版权的，请各位怜惜电子图书资源，不要随意传播，否则，这些资源更难以得到。