

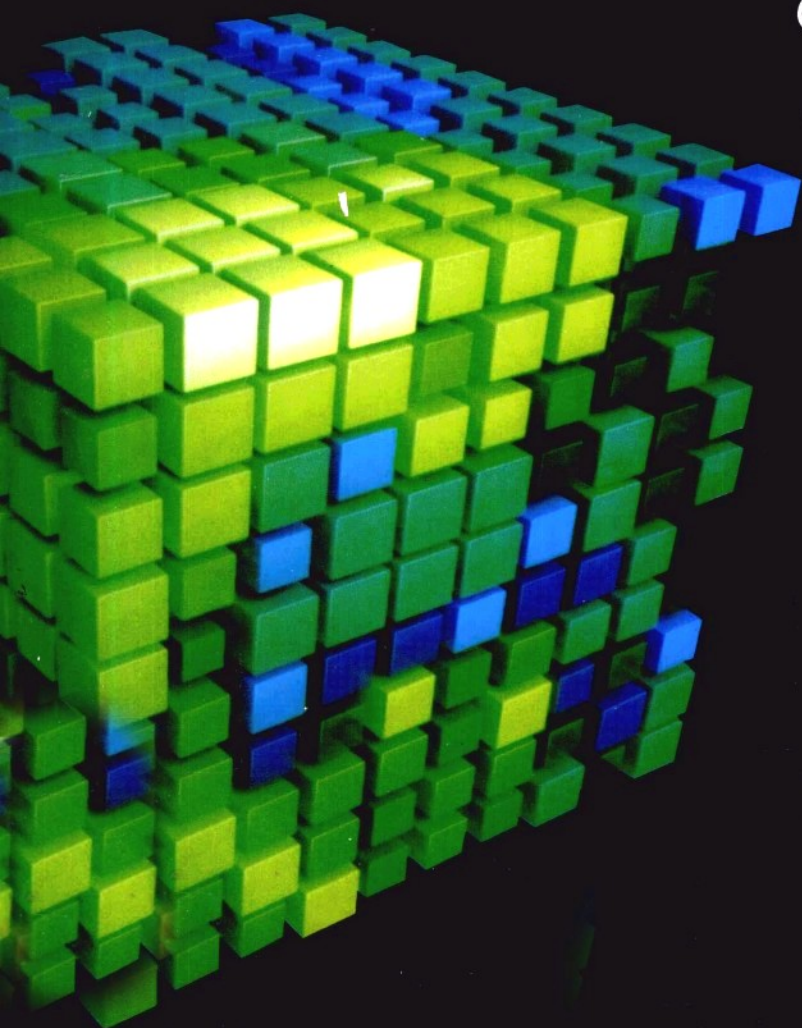
GPU高性能编程 CUDA实战

CUDA By Example

an Introduction to General-Purpose
GPU Programming

(美) Jason Sanders 著
Edward Kandrot

聂雪军 等译



机械工业出版社
China Machine Press

GPU高性能编程 CUDA实战

CUDA By Example

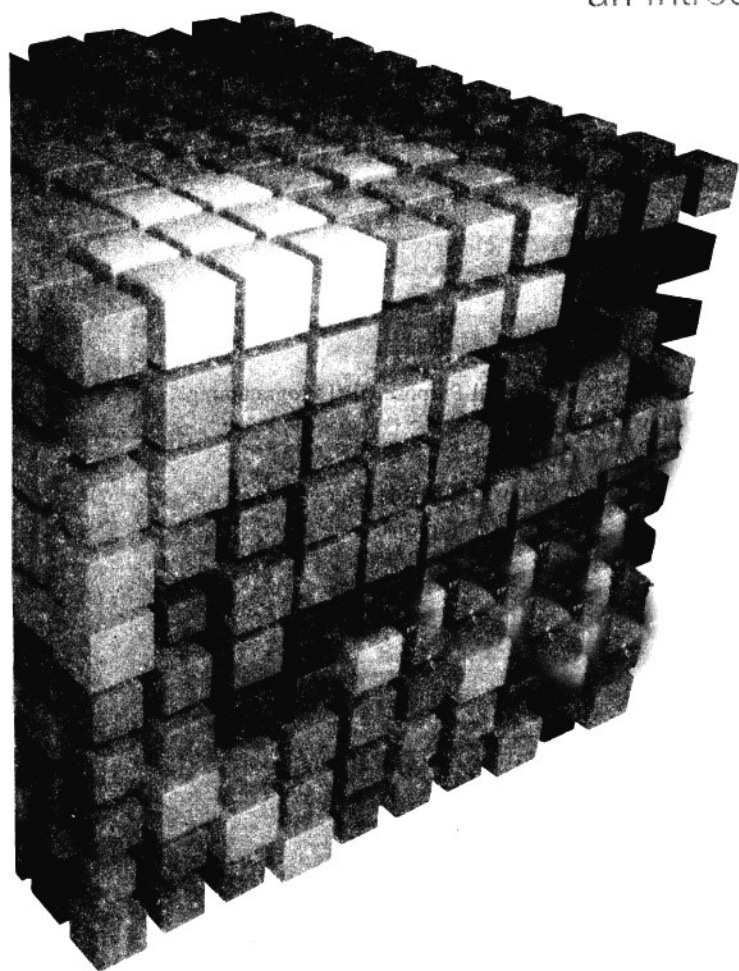
an Introduction to General-Purpose
GPU Programming

(美) Jason Sanders 著
Edward Kandrot

聂雪军 等译



机械工业出版社
China Machine Press



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架构能充分发挥GPU的强大计算功能。本书首先介绍了CUDA架构的应用背景,并给出了如何配置CUDA C的开发环境。然后通过矢量求和运算、矢量点积运算、光线跟踪、热传导模拟等示例详细介绍了CUDA C的基本语法和使用模式。通过学习本书,读者可以清楚了解CUDA C中每个功能的适用场合,并编写出高性能的CUDA软件。

本书适合具备C或者C++知识的应用程序开发人员、数值计算库开发人员等,也可以作为学习并行计算的学生和教师的教辅。

Simplified Chinese edition copyright © 2011 by Pearson Education Asia Limited and China Machine Press.

Original English language title: CUDA by Example: an Introduction to General-Purpose GPU Programming (ISBN 978-0-13-138768-3) by Jason Sanders, Edward Kandrot, Copyright ©2011.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2010-5159

图书在版编目(CIP)数据

GPU高性能编程CUDA实战/(美)桑德斯(Sanders, J.)著;聂雪军等译. —北京:机械工业出版社, 2011.1

书名原文: CUDA by Example: an Introduction to General-Purpose GPU Programming

ISBN 978-7-111-32679-3

I. G… II. ①桑… ②聂… III. 图像处理—程序设计 IV. TP391.41

中国版本图书馆CIP数据核字(2010)第235229号

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:秦健

北京市荣盛彩色印刷有限公司印刷

2011年1月第1版第1次印刷

186mm×240mm·13.25印张

标准书号:ISBN 978-7-111-32679-3

定价:39.00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

购书热线:(010) 68326294; 88379649; 68995259

投稿热线:(010) 88379604

读者信箱:hzjsj@hzbook.com

译者序

并行计算已成为突破摩尔定理局限性的重要研究方向，而GPU强大的并行计算能力也因此吸引了全球广泛的研究兴趣。然而，在实现通用并行计算时，GPU计算模式存在着一些限制。首先，GPU的设计初衷是为了加速应用程序中的图形绘制运算，因此开发人员需要通过OpenGL或者DirectX等API来访问GPU，这不仅要求开发人员掌握一定的图形编程知识，而且要想方设法将通用计算问题转换为图形计算问题。其次，GPU与多核CPU在计算架构上有着很大不同，GPU更侧重于数据并行计算，即在不同的数据上并行执行相同的计算，而对并行计算中的互斥性、同步性以及原子性等方面支持不足。这些因素都限制了GPU在通用并行计算中的应用范围。

CUDA架构的出现解决了上述问题。CUDA架构专门为GPU计算设计了一种全新的结构，目的正是为了减轻GPU计算模型中的这些限制。在CUDA架构下，开发人员可以通过CUDA C对GPU编程。CUDA C是对标准C的一种简单扩展，学习和使用起来都非常容易，并且其最大的优势在于不需要开发人员具备图形学知识。

本书的主要内容是介绍如何通过CUDA C来编写在GPU上运行的并程序。本书首先介绍了CUDA架构的应用背景，并给出了如何配置CUDA C的开发环境。然后，本书通过矢量求和运算、矢量点积运算、光线跟踪、热传导模拟、直方图统计等示例详细介绍了CUDA C的基本语法和使用模式。在这些示例中还穿插介绍了GPU的各种硬件特性及适用环境，例如常量内存、事件、纹理内存、图形互操作性、原子操作、流以及多GPU架构等。

这些示例的构思以及分析过程都很容易理解，它们也是本书最具价值的部分。读者在阅读这些内容时要反复思考，做到融会贯通，举一反三。只要掌握了这些简单的示例，更复杂的问题也能迎刃而解。本书适合所有程序员阅读，只需具备基本的C语言知识即可。最后，本书还给出了CUDA C的其他一些参考资源。

参与本书翻译工作的主要有李杨、吴汉平、徐光景、童胜汉、陈军、胡凯等。由于译者的时间和水平有限，翻译中的疏漏和错误在所难免，还望读者和同行不吝指正。

聂雪军

2010年于武汉

序

主流芯片制造厂商（例如NVIDIA）最近一系列的动作已经表明了，未来的微处理器系统和大型高性能计算系统都将是异构的。在这些异构系统中将同时使用两种不同的技术，二者在不同系统中所占的比例是不同的。

- **多核CPU技术**：CPU核的数量将不断增长，因为人们希望将越来越多的部件封装到芯片上，同时又要避免功耗墙、指令集并行性墙、内存墙等。
- **专用硬件和大规模并行加速器**：例如，近年来NVIDIA公司推出的GPU在浮点运算上的性能已经超过了标准CPU的性能。而且，GPU编程也正变得和多核CPU编程一样简单。

在未来的设计中，这两种技术在系统中所占的比例并不是固定的，并且随着时间的推移很可能发生变化。显然，在未来的计算机系统中，无论是笔记本电脑还是超级计算机，都将包含各种异构的部件。事实上，这种系统的浮点计算能力已经达到了P级（即每秒钟执行 10^{15} 次浮点运算）。

但是，在采用混合处理器的新计算环境中，开发人员面临的问题和挑战仍然很艰难。软件基础架构中的关键部分要赶上这种变革的步伐也同样很困难。在一些情况中，软件的性能无法随着处理器核数量的增加而增加，因为越来越多的时间消耗在数据移动而不是计算上。在其他情况中，软件通常是在硬件出现数年后才发布性能调优后的版本，因此在发布时就已经过时了。还有一些情况，例如在最新的GPU上，软件根本无法运行，因为编程环境已经发生了太大的变化。

本书解决了软件开发中面临的核心问题，它介绍了近年来在编写大规模并行加速器中最具创新性和功能最强大的解决方案之一。

本书介绍了如何使用CUDA C来编写程序，不仅给出了详细的示例，而且还介绍了程序的设计流程以及NVIDIA GPU的高效使用方式。本书介绍了并行计算的一些基本概念，包括简单的示例和调试方法（涵盖逻辑调试和性能调试），此外还介绍了一些高级主题以及在编译和使用应用程序时需要注意的问题，并通过编程示例来进一步强化这些概念。

如果读者需要开发基于加速器的计算系统，那么本书是首选。本书深入分析了并行计算，并为其中的多种问题提供了解决方案。本书对于应用程序开发人员、数值计算库开发人员，以及学习并行计算的学生和教师来说尤为有用。

我很高兴从本书中学到了不少东西，并且我相信你也会有同样的体会。

——Jack Dongarra

田纳西大学杰出教授，美国橡树岭国家实验室杰出研究人员

前 言

本书介绍了如何利用计算机中图形处理器 (Graphics Process Unit, GPU) 的强大计算功能来编写各种高性能的应用软件。虽然GPU的设计初衷是用于在显示器上渲染计算机图形 (现在仍然主要用于这个目的), 但在科学计算、工程、金融以及其他领域中, 人们开始越来越多地使用GPU。我们将解决非图形领域中的问题的GPU程序统称为通用GPU程序。值得高兴的是, 虽然你需要具备C或者C++的知识才能充分理解本书的内容, 但却不需要具备计算机图形学的知识。任何图形学的基础都不要! GPU编程只是使你进一步增强现有的编程技术。

在NVIDIA GPU上编写程序来完成通用计算任务之前, 你需要知道什么是CUDA。NVIDIA GPU是基于CUDA架构而构建的。你可以将CUDA架构视为NVIDIA构建GPU的模式, 其中GPU既可以完成传统的图形渲染任务, 又可以完成通用计算任务。要在CUDA GPU上编程, 我们需要使用CUDA C语言。在本书前面的内容中可以看到, CUDA C本质上是对C进行了一些扩展, 使其能够在像NVIDIA GPU这样的大规模并行机器上进行编程。

我们为经验丰富的C或者C++程序员编写了本书, 这些程序员通常较为熟悉C语言, 因此能很轻松地阅读或者编写C代码。本书不仅将进一步增强你的C语言编程能力, 而且还能作为使用NVIDIA的CUDA C编程语言的一本快速入门书籍。你既不需要具备任何在大规模软件架构上工作的经验, 也不需要编写过C编译器或者操作系统内核的经历, 此外也无需了解ANSI C标准的细枝末节。本书并没有花时间来回顾C语言的语法或者常用的C库函数, 例如malloc()或者memcpy(), 我们假设你对这些概念已经非常熟悉了。

虽然本书的目的并不是介绍通用的并行编程技术, 但你在书中仍将学习到一些通用的并行编模式。此外, 本书并不是一本详细介绍CUDA API的参考书, 也不会详细介绍在开发CUDA C软件时可以使用的各种工具。因此, 我们强烈建议将本书与NVIDIA的免费文档结合起来阅读, 例如《NVIDIA CUDA Programming Guide》和《NVIDIA CUDA Best Practices Guide》等。然而, 你不用费工夫去收集所有这些文档, 因为我们将介绍你需要的所有内容。

不会费太多的周折, CUDA C编程领域欢迎你的到来!

致 谢

任何一本技术书籍的完成都离不开许多人的帮助，本书也不例外。作者要感谢许许多多的人，在这里将列举其中一些。

感谢Ian Buck，NVIDIA GPU计算软件的高级主管，他在本书编写的各个阶段，从本书的构思到最终完成，都给予了极大的帮助。我们还要感谢Tim Murray，他是一位始终保持微笑的审阅者，感谢他保证了本书的技术准确性和可读性。感谢插图设计师Darwin Tat，他在非常紧张的时间中为本书制作了精美的封面和插图。最后，我们要感谢John Park，他帮助本书完成了在出版流程中涉及的复杂法律步骤。

如果没有Addison-Wesley工作人员的帮助，本书将仍然只是作者的一个转瞬即逝的想法而已。感谢Peter Gordon、Kim Boedigheimer、Julie Nahil，感谢他们的耐心、专业，最终使得本书顺利出版。此外，Molly Sharp和Kim Wimpsett的编辑工作帮助本书从一堆充满错误的文档转换成为读者们现在看到的文字。

如果没有其他撰稿人的帮助，读者将无法阅读本书的一些内容。特别是，Nadeem Mohammad收集了第1章中介绍的CUDA案例分析，Nathan Whitehead无私地提供了本书的示例代码。

此外，还要感谢最先阅读本书草稿的一些人，感谢他们提供了有帮助的反馈，他们包括Genevieve Breed和Kurt Wall。在本书的编写过程中，许多NVIDIA的软件工程师都提供了宝贵的技术帮助，包括Mark Hairgrove，他通读了本书，并指出了所有不一致的地方——包括技术上的、拼写上的和语法上的。Steve Hines、Nicholas Wilt和Stephen Jones审阅了一些章节的CUDA API，并补充了作者忽略的许多细节。还要感谢Randima Fernando，感谢他帮助我们启动了本书的编写工作，感谢Michael Schidlowsky在他的书中对Jason表达的谢意。

最后还要感谢我们的家人。我们要感谢我们的家庭，他们陪伴我们经历了每一件事情，并最终完成本书。我们要特别感谢我们两个人的父母，Edward、Kathleen Kandrot、Stephen、Helen Sanders。感谢我们的兄弟，Kenneth Kandrot和Corey Sanders。感谢你们给予的无尽支持。

作者简介

Jason Sanders是NVIDIA公司CUDA平台小组的高级软件工程师。他在NVIDIA的工作包括帮助开发早期的CUDA系统软件，并参与OpenCL 1.0规范的制定，该规范是一个用于异构计算的行业标准。Jason在加州大学伯克利分校获得计算机科学硕士学位，他发表了关于GPU计算的研究论文。此外，他还获得了普林斯顿大学电子工程专业学士学位。在加入NVIDIA公司之前，他曾在ATI技术公司、Apple公司以及Novell公司工作过。

Edward Kandrot是NVIDIA公司CUDA算法小组的高级软件工程师。他在代码优化和提升性能等方面拥有20余年的工作经验，参与过Photoshop和Mozilla等项目。Kandrot曾经在Adobe公司、Microsoft公司工作过，他还是许多公司的咨询师，包括Apple公司和Autodesk公司。

目 录

译者序	
序	
前言	
致谢	
作者简介	
第1章 为什么需要CUDA	1
1.1 本章目标	2
1.2 并行处理的历史	2
1.3 GPU计算的崛起	3
1.4 CUDA	5
1.5 CUDA的应用	6
1.6 本章小结	8
第2章 入门	9
2.1 本章目标	10
2.2 开发环境	10
2.3 本章小结	14
第3章 CUDA C简介	15
3.1 本章目标	16
3.2 第一个程序	16
3.3 查询设备	20
3.4 设备属性的使用	23
3.5 本章小结	24
第4章 CUDA C并行编程	26
4.1 本章目标	27
4.2 CUDA并行编程	27
4.3 本章小结	41
第5章 线程协作	42
5.1 本章目标	43
5.2 并行线程块的分解	43
5.3 共享内存和同步	54
5.4 本章小结	68
第6章 常量内存与事件	69
6.1 本章目标	70
6.2 常量内存	70
6.3 使用事件来测量性能	78
6.4 本章小结	83
第7章 纹理内存	84
7.1 本章目标	85
7.2 纹理内存简介	85
7.3 热传导模拟	86
7.4 本章小结	101
第8章 图形互操作性	102
8.1 本章目标	103
8.2 图形互操作	103
8.3 基于图形互操作性的GPU波纹示例	108
8.4 基于图形互操作性的热传导	113
8.5 DirectX互操作性	118
8.6 本章小结	118
第9章 原子性	119
9.1 本章目标	120
9.2 计算功能集	120
9.3 原子操作简介	122
9.4 计算直方图	124
9.5 本章小结	133
第10章 流	134
10.1 本章目标	135
10.2 页锁定主机内存	135
10.3 CUDA流	139

10.4 使用单个CUDA流	140	11.4 可移动的固定内存	166
10.5 使用多个CUDA流	144	11.5 本章小结	170
10.6 GPU的工作调度机制	149	第12章 后记	171
10.7 高效地使用多个CUDA流	151	12.1 本章目标	172
10.8 本章小结	152	12.2 CUDA工具	172
第11章 多GPU系统上的CUDA C	154	12.3 参考资料	176
11.1 本章目标	155	12.4 代码资源	178
11.2 零拷贝主机内存	155	12.5 本章小结	179
11.3 使用多个GPU	162	附录 高级原子操作	180

第 1 章

为什么需要CUDA

就在前不久，并行计算（Parallel Computing）还被认为是一种“神秘的”技术，属于计算机科学的专业领域之一。然而，这种观念在近几年发生了巨大转变。在计算机业界，并行计算已不再是一个冷门领域，几乎每个程序员都要学习并行计算的知识才能胜任计算机科学领域的工作。当你拿起这本书时，或许还没有意识到并行编程在当前计算机业界有着怎样的重要地位，也没有意识到并行编程在未来数年中将起到如何重要的作用。在本章中，我们将分析近年来硬件领域的发展趋势，以及它们对软件开发的推动作用。我希望通过这些介绍使读者们意识到，并行计算的革命已经开始了，通过学习和使用CUDA C，我们能够在由CPU处理器和GPU构成的异构平台上编写出高性能的应用程序。

1.1 本章目标

通过本章的学习，你可以：

- 了解并行计算正在发挥越来越重要的作用。
- 了解GPU计算和CUDA的发展历程。
- 了解一些通过使用CUDA C而获得成功的应用程序。

1.2 并行处理的历史

近年来，计算机业界正在迅速迈进并行计算时代。在2010年，几乎所有的客户计算机都采用了多核处理器。从入门级的双核低端上网本，到8核或者16核的工作站计算机，并行计算已不再是超级计算机或者大型机的专属技术。此外，一些电子设备，例如手机和便携式音乐播放器等，都开始集成并行计算功能，以提供比早期产品更强大的功能。

随着时间的推移，软件开发人员将看到越来越多的并行计算平台和技术，并需要基于这些平台和技术为不断成熟的用户群提供崭新且丰富的体验。命令提示符的界面已经过时了，现在流行的是多线程图形界面。只能接打电话的手机也已经过时了，现在流行的是能同时播放音乐、浏览网页和提供GPS服务的手机。

中央处理器

在30多年前，要想提升客户计算设备的性能，主要手段之一就是提高处理器的时钟频率。在20世纪80年代早期出现的第一台个人计算机，其中央处理器（CPU）的运行时钟频率为1MHz。30年后，大多数桌面处理器的时钟频率都在1GHz和4GHz之间，这比当初个人计算机的时钟频率要快1 000倍。尽管提高CPU时钟频率并不是提升计算性能的唯一方法，但却是一种相对稳定的提升方法。

然而，近年来计算机制造商们却不得不开始寻求其他的替代方法来提升计算性能。由于在集成电路元器件中存在的各种严重限制，人们已经无法在现有的架构上通过提高处理器时钟频率来提升性能。随着功耗与发热的急剧升高以及晶体管的大小已接近极限，研究人员和制造商们开始寻求其他方式。

除了个人计算机外，超级计算机在过去数十年里也借助相同的方式获得了极大的性能提升。超级计算机中的处理器与个人计算机中CPU一样，在时钟频率上都得到了极大提升。然而，除了在单个处理器上提升性能外，超级计算机的制造商们还通过增加处理器的数量来提升性能。在高速的超级计算机中，几十个、几百个甚至几千个处理器同时运行是很常见的情形。

当人们在探索如何提升个人计算机的性能时，超级计算机中性能提升方式引出了一个很好的问题：为什么不在个人计算机中放置多个处理器，而不是仅提升单个处理器核的性能？这样，在不需要提高处理器运行频率的情况下，个人计算机的性能就能获得持续的提升。

在2005年，当面对竞争日趋激烈的市场以及越来越少的可行方式时，业界一些领先的CPU制造商们开始提供带有两个计算核的处理器。在接下来的几年中，他们延续了这种发展趋势，不断推出3核、4核、6核以及8核的中央处理器。这种趋势也称为多核革命（Multicore Revolution），它标志着在个人计算机上开始发生重大的转变。

当前，要购买一台单核CPU的桌面计算机已经比较困难了。即使在低端、低能耗的中央处理器中，通常都包含有两个或多个计算核。一些业界领先的CPU制造商已经宣布在未来将计划推出12核和16核的CPU，这进一步证明了并行计算已经给人们带来了不可忽视的好处。

1.3 GPU计算的崛起

与中央处理器传统的数据处理流水线相比，在图形处理器（Graphics Processing Unit, GPU）上执行通用计算还是一个新概念。事实上，在计算领域中，GPU本身在很大程度上就是一个新概念。然而，在图形处理器上执行计算却并非新概念。

1.3.1 GPU简史

在前面介绍了中央处理器在时钟频率和处理器核数量上的发展历程。与此同时，图形处理技术同样经历了巨大的变革。在20世纪80年代晚期到90年代早期之间，图形界面操作系统（例如Microsoft公司的Windows）的普及推动了新型处理器的出现。在20世纪90年代早期，用户开始购买配置2D显示加速器卡的个人计算机。这些显卡提供了基于硬件的位图运算功能，能够在图形操作系统的显示和可用性上起到辅助作用。

在专业计算领域，Silicon Graphics在整个20世纪80年代都致力于推动3D图形在各种市场中的应用，包括政府与国防等应用领域以及科学与技术的可视化技术等，此外还提供了各种工具来制作炫目的电影特效。在1992年，SG公司发布了OpenGL库，这是一个对该公司硬件进行编程的接口。SG公司试图将OpenGL作为一种标准化的、与平台无关的3D图形应用程序编写方法。并行计算和GPU的演变情况是类似的，这些技术进入到消费者应用程序中只是时间早晚的问题。

20世纪90年代中期，在消费者应用程序中采用3D图形技术的需求开始快速增长，这两类非常重要的应用程序创造了基础条件。首先，许多第一人称游戏，例如Doom、Duke Nukem 3D和Quake等，都要求为PC游戏创建更真实的3D场景。虽然最终3D图形技术融入几乎所有的计算机游戏中，但第一人称射击类游戏在初期对3D图形技术在消费者应用程序中的普及起到

了极大的推动作用。其次，一些制造商，例如NVIDIA公司、ATI Technologies公司以及3Dfx Interactive公司等，都开始发布一些普通消费者能够买得起的图形加速器卡，以便吸引更多广泛的关注。这些事件都促使3D图形技术成为在未来几年中占据重要地位的技术之一。

NVIDIA公司GeForce 256显卡的发布进一步提升了消费者图形硬件的功能。GeForce 256第一次实现了在图形处理器上直接运行变形与光效（Transform and Lighting）等计算，因此在某些应用程序中能实现更强的视觉效果。由于变形和光效已经成为OpenGL图形流水线功能（Graphics Pipeline）的一部分，因此GeForce 256标志着越来越多的图形流水线功能将开始直接在图形处理器上实现。

从并行计算的角度来看，NVIDIA在2001年发布的GeForce 3代表着GPU技术上的最重要突破。GeForce 3系列是计算工业界的第一块实现Microsoft DirectX 8.0标准的芯片。该标准要求硬件中同时包含可编程的顶点着色功能（Vertex Shading Stage）和像素着色功能（Pixel Shading Stage）。开发人员也正是从GeForce 3系列开始第一次能够对GPU中的计算实现某种程度的控制。

1.3.2 早期的GPU计算

GPU中的可编程流水线吸引了许多研究人员来探索如何在除了OpenGL或者DirectX渲染之外的领域中使用图形硬件。早期的GPU计算使用起来非常复杂。由于标准图形接口，例如OpenGL和DirectX，是与GPU交互的唯一方式，因此要在GPU上执行计算，就必然受限于图形API的编程模型。基于这个原因，研究人员开始探索如何通过图形API来执行通用计算，然而这也使得他们的计算问题在GPU看来仍然是传统的渲染问题。

在2000年早期，GPU的主要目标都是通过可编程计算单元为屏幕上的每个像素计算出一个颜色值，这些计算单元也称为像素着色器（Pixel Shader）。通常，像素着色器根据像素在屏幕上的位置（ x, y ）以及其他一些信息，对各种输入信号进行合成并计算出最终的颜色值。这些信息包括输入颜色、纹理坐标、或其他可以传递给着色器的属性。由于对输入颜色和纹理的计算完全是由程序员控制的，因此研究人员注意到，这些输入的“颜色”实际上可以是任意数据。

因此，如果输入值实际上并不是表示颜色值，那么程序员可以对各个像素着色器进行编程从而对输入值执行任意计算。计算结果将交回GPU作为像素的最终“颜色”，尽管这些颜色值只是程序员通过GPU对输入数据进行计算的结果。研究人员可以获得这些计算结果，而GPU永远都不会知道其中的过程。事实上，GPU能够执行除渲染之外的任务，只要这些任务看起来像是一个标准的渲染任务。虽然这种技术非常聪明，但使用起来却非常复杂。

由于GPU有着很高的计算吞吐量，因此最初从这些实验中得到的结果表明GPU计算有着非常光明的应用前景。然而，这种编程模型对于开发人员来说存在着非常大的局限性。首先，该

模型有着严格的资源限制，因为程序只能以颜色值和纹理单元等形式来输入数据。此外，程序员在将计算结果写入内存的方式以及位置上同样存在着严格限制，如果在算法中需要写入到内存的任意（分散）位置，那么将无法在GPU上运行。而且，我们也无法预测所使用的GPU能否处理浮点数据，如果不能处理浮点数据，那么大多数科学计算都将无法使用GPU。最后，当出现问题时，例如程序计算得到错误结果，程序无法终结，或者使计算机挂起，那么没有任何一种方便的方法能够对GPU上执行的代码进行调试。

除了这些限制因素之外，如果程序员希望通过GPU来执行通用计算，那么他们还需要学习OpenGL或者DirectX，因为这些接口仍然是与GPU交互的唯一方式。这不仅意味着要将数据保存在图形纹理中并调用OpenGL或者DirectX函数来执行计算，而且还意味着要使用特殊的图形编程语言来编写这些计算，这些语言也称为着色语言（Shading Language）。因此，研究人员在开始使用GPU的强大计算功能之前，首先需要考虑严格的资源限制和编程限制，然后还要学习计算机图形学和着色语言，这种负担对于研究人员来说过于沉重，因此GPU计算在早期并没有被广泛的接受。

1.4 CUDA

直到在GeForce 3系列发布五年之后，GPU计算才开始逐渐成为主流技术。在2006年11月，NVIDIA公布了业界的第一个DirectX 10 GPU，即GeForce 8800 GTX。GeForce 8800 GTX也是第一个基于NVIDIA的CUDA架构构建的GPU。CUDA架构专门为GPU计算设计了一种全新的模块，目的是减轻早期GPU计算中存在的一些限制，而正是这些限制使得之前的GPU在通用计算中没有得到广泛应用。

1.4.1 CUDA架构是什么

在之前的图形处理架构中，计算资源划分为顶点着色器和像素着色器，而CUDA架构则不同，它包含了一个统一的着色器流水线，使得执行通用计算的程序能够对芯片上的每个数学逻辑单元（Arithmetic Logic Unit, ALU）进行排列。由于NVIDIA希望使新的图形处理器能适用于通用计算，因此在实现这些ALU时都确保它们满足IEEE单精度浮点数学运算的需求，并且可以使用一个裁剪后的指令集来执行通用计算，而不是仅限于执行图形计算。此外，GPU上的执行单元不仅能任意地读/写内存，同时还能访问由软件管理的缓存，也称为共享内存。CUDA架构的所有这些功能都是为了使GPU不仅能执行传统的图形计算，还能高效地执行通用计算。

1.4.2 CUDA架构的使用

NVIDIA并不局限于通过集成CUDA架构的硬件来为消费者同时提供计算功能和图形功能。尽管NVIDIA在芯片中增加了许多功能来加速计算，但仍然只能通过OpenGL或者DirectX来访

问这些功能。这不仅要求用户仍然要将他们的计算任务伪装为图形问题，而且还需要使用面向图形的着色语言（例如OpenGL的GLSL或者Microsoft的HLSL）来编写计算代码。

为了尽可能地吸引更多的开发人员，NVIDIA采取了工业标准的C语言，并且增加了一小部分关键字来支持CUDA架构的特殊功能。在发布了GeForce 8800 GTX之后的几个月，NVIDIA公布了一款编译器来编译CUDA C语言。这样，CUDA C就成为了第一款专门由GPU公司设计的编程语言，用于在GPU上编写通用计算。

除了专门设计一种语言来为GPU编写代码之外，NVIDIA还提供了专门的硬件驱动程序来发挥CUDA架构的大规模计算功能。现在，用户不再需要了解OpenGL或者DirectX图形编程结构，也不需要将通用计算问题伪装为图形计算问题。

1.5 CUDA的应用

自从CUDA C在2007年首次出现以来，许多企业都尝试以CUDA C为基础来构建应用程序，并获得了极大的成功。基于CUDA C编写的代码比之前的代码在性能上提升了多个数量级。而且，与传统在CPU上运行的应用程序相比，在NVIDIA图形处理器上运行的应用程序的单位成本和单位能耗都要低很多。下面就给出CUDA C以及CUDA架构在一些方面的成功应用。

1.5.1 医学图像

在过去20年中，乳腺癌患者的数量持续在增长。在众多研究人员的不懈努力下，对这种可怕疾病的预防与治疗工作取得了重大进展。人们的研究热点在于如何提前发现乳腺癌，从而有效地防止在辐射和化疗中产生的严重副作用，外科手术造成的永久性后遗症，以及由于治疗无效而导致的死亡等。因此，研究人员一致在努力找到某种快速、精确并且影响最小的方式来找出生乳腺癌的早期症状。

乳房X线照片（Mammogram）是当前识别早期乳腺癌的最好技术之一，然而这种技术在实际应用中存在一些严重的限制。该项技术需要拍摄两张或者多张图像，并且由一位熟练的医生来分析图像以找出潜在的肿瘤。此外，X射线还会造成反复地辐射患者的胸腔。在经过仔细研究后，医生通常要求进一步的、并且更为具体的成像——有时甚至需要进行活体检查，从而判断癌症的可能性。诊断中的误报（False Positive）不仅会导致昂贵的后续诊断工作，而且在最终报告出来之前，会给患者造成过度的压力。

超声波成像技术比X射线成像技术要更为安全，因此医生通常将其与乳房X线照片结合起来使用，以辅助乳腺癌的治疗与诊断。然而，常规的乳房超声波技术仍然有其局限性。因此，人们研发了TechniScan医疗系统。TechniScan采用了一种三维的超声波成像方法，但这种解决方案却由于一个非常简单的问题而无法投入实际使用：计算量的限制。简单来说，在将收集

到的超声波数据转换为3D图像时需要执行非常耗时的计算，因此使得该项技术无法投入实际使用。

NVIDIA的第一款基于CUDA架构的GPU以及CUDA C编程语言为TechniScan由构想变为现实提供了重要的基础平台。TechniScan的Svara超声波成像系统通过超声波对患者的胸腔进行成像。该系统通过两个NVIDIA Tesla C1060处理器来处理在15分钟扫描过程中生成的35GB数据。由于Tesla C1060的强大计算功能，医生在20分钟内就可以获得患者乳房的高清三维成像。TechniScan预期Svara系统将在2010年得到广泛应用。

1.5.2 计算流体动力学

多年来，旋翼桨叶的高效设计始终是一个难题。研究人员很难通过简单的计算公式对这些设备周围的空气和流体的复杂运动进行建模，而精确的模拟却由于需要非常大的计算量而无法实现。只有世界上最大的超级计算机才有希望提供足够的计算资源来开发和验证这些复杂的数值模型。由于很少有人能使用这样的超级计算机，所以支持复杂模拟模型的计算机设计在多年来始终停滞不前。

剑桥大学是研究高级并行计算的发源地。“众核小组 (Many-Core Group)”的Graham Pullan博士和Tobias Brandvik博士敏锐地意识到NVIDIA CUDA架构的巨大潜能：它能将计算流体动力学提升到前所未有的高度。他们最初的研究结果表明，即使个人工作站上的GPU也能够实现显著的性能提升。后来，他们发现只需使用一个小型的GPU集群，就能轻松获得超过超级计算机的计算能力，并且进一步验证了他们最初的假设：NVIDIA GPU的能力非常适合他们想要解决的问题。

对于剑桥大学的研究者们来说，CUDA C带来的大规模性能提升不仅仅是对计算资源的简单提升。强大并且低成本的GPU计算能力使得剑桥大学的研究者们能够进行快速的实验。由于在几秒钟内就可以获得实验结果，因此研究者的实验反馈流程将更为简化。基于GPU集群的实验极大地改变了人们的研究方式。近乎于交互式的模拟实验为之前停滞不前的研究领域带来了新的创新机遇。

1.5.3 环境科学

随着全球经济的快速扩张，人们对消费品的环境无害性需求越来越高。人们关心气候变化，不断上涨的燃料价格，以及空气和水的污染指数，正是这种关心延缓了工业进步所带来的间接破坏。长久以来，洗涤剂 and 清洗剂是最可能在日常使用中对环境造成破坏的消费品。因此，许多科学家都开始研究各种方法来降低这些洗涤剂对环境的影响，同时又要确保它们的洗涤效果。然而，做任何事情都是要付出代价的。

清洁剂的关键成分是表面活性剂。表面活性剂分子决定着清洁剂和洗发剂的清洁能力，但

它们通常也是清洁产品中可能对环境造成最大破坏的成分。这些分子会吸附到脏物上，然后与水混合，这样表面活性剂就和脏东西一起被清洗掉。在传统方式中，测量某种新型表面活性剂的清洁能力需要进行大量的实验，包括将各种材料和需要清洗的杂质混合在一起。显然，这个实验过程非常缓慢，并且成本很高。

美国天普大学与业界的领先企业宝洁公司一起合作，对表面活性剂与脏物、水以及其他材料的交互进行分子模拟。计算机模拟并不仅仅是为了加快传统实验方法的过程，而是还要扩展对不同环境条件的测试广度，这种测试在过去是无法进行的。天普大学的研究者们使用了一款基于GPU加速的HOOMD (Highly Optimized Object-Oriented Many-particle Dynamics) 模拟软件，该软件是由美国Ames国家实验室开发的。HOOMD将模拟实验的计算任务分解到两个NVIDIA Tesla GPU上，并能够获得与128个CPU核的Cray XT3或者1024个CPU的IBMBLunGene/L机器大致相等的性能。通过增加解决方案中Tesla GPU的数量，HOOMD实现的表面活性剂交互模拟性能是之前平台的16倍。由于NVIDIA的CUDA将这种复杂模拟的计算时间从数个星期减少为几个小时，在未来的几年中，将出现大量清洁效率更高且对环境影响更小的产品。

1.6 本章小结

计算机业界正处于并行计算的革命中，NVIDIA的CUDA C已经成为实现并行计算的最成功语言之一。在本书中，我们将帮助你学习如何使用CUDA C编写自己的代码。此外，还将帮助你了解NVIDIA在实现GPU计算时对C语言进行的扩展以及应用程序编程接口。在学习CUDA C时，你不需要具备OpenGL或者DirectX的知识，也不需要了解计算机图形学的背景。

本书并不打算介绍C编程的基础知识，因此对于不具备任何计算机编程知识的读者，我们不建议阅读本书。如果你熟悉并行编程的一些基础知识，那么将带来一定的帮助，但并不需要一定实际做过并行编程。对于任何与并行编程相关的术语或概念，如果读者需要理解，那么我将在本书中给出解释。事实上，在某些情况下，你会发现传统的并行编程知识反而会使你在GPU计算环境下做出错误的假设。事实上，掌握本书内容的唯一前提条件就是具备一定的C或者C++编程经验。

在第2章中，我们将介绍如何配置GPU计算的开发和运行环境，确保在开始学习时就拥有必要的硬件资源和软件资源。之后，你就可以开始学习CUDA C了。如果你曾经使用过CUDA C，或者你的计算机系统已经配置好了CUDA C开发环境，那么可以直接跳到第3章。

第 2 章

入 门

我们希望第1章的内容已经使你迫不及待地想要学习CUDA C了。本书将通过一组示例代码来讲解CUDA C，因此你首先需要一个完备的开发环境。当然，你也可以由其他人来帮你准备好开发环境，但我们认为，如果你能亲自动手实践并尽快地获得CUDA C的一些实际使用经验，那么将能保持更长久的学习兴趣。本章将介绍在开始使用CUDA C之前需要准备的硬件和软件。好消息是，你可以免费获得所有软件，从而可以为你省下更多的钱去做其他事情。

2.1 本章目标

通过本章的学习，你可以：

- 下载本书所需的各种软件。
- 配置好编译CUDA C代码的环境。

2.2 开发环境

在开始学习之前，首先要配置好编写CUDA C代码的开发环境。使用CUDA C来编写代码的前提条件包括：

- 支持CUDA的图形处理器
- NVIDIA设备驱动程序
- CUDA开发工具箱
- 标准C编译器

为了使本章的学习过程尽可能简单，我们首先介绍各个前提条件。

2.2.1 支持CUDA的图形处理器

我们很容易找到一款基于CUDA架构设计的图形处理器，因为自从2006年发布GeForce 8800 GTX以来，NVIDIA推出的每款GPU都能支持CUDA。由于NVIDIA一直在不断地推出基于CUDA架构的新GPU，因此下面给出的只是所有支持CUDA的GPU中的一部分。当然，这些GPU都能支持CUDA。

完整的列表可以参考NVIDIA的网址www.nvidia.com/cuda上给出的信息，不过我们也可以假设所有新推出（从2007年开始）并且显存超过256MB的GPU都可以用于开发和运行基于CUDA C编写的代码。

表2.1 支持CUDA的GPU

GeForce GTX 480	GeForce GT 220	GeForce 9800 GT
GeForce GTX 470	GeForce G210	GeForce 9600 GSO
GeForce GTX 295	GeForce GTS 150	GeForce 9600 GT
GeForce GTX 285	GeForce GT 130	GeForce 9500 GT
GeForce GTX 285 for Mac	GeForce GT 120	GeForce 9400GT
GeForce GTX 280	GeForce G100	GeForce 8800 Ultra
GeForce GTX 275	GeForce 9800 GX2	GeForce 8800 GTX
GeForce GTX 260	GeForce 9800 GTX+	GeForce 8800 GTS
GeForce GTS 250	GeForce 9800 GTX	GeForce 8800 GT

(续)

GeForce 8800 GS	Quadro NVS 320M	GeForce GTX 260M
GeForce 8600 GTS	Quadro NVS 160M	GeForce GTS 260M
GeForce 8600 GT	Quadro NVS 150M	GeForce GTS 250M
GeForce 8500 GT	Quadro NVS 140M	GeForce GTS 160M
GeForce 8400 GS	Quadro NVS 135M	GeForce GTS 150M
GeForce 9400 mGPU	Quadro NVS 130M	GeForce GT 240M
GeForce 9300 mGPU	Quadro FX 5800	GeForce GT 230M
GeForce 8300 mGPU	Quadro FX 5600	GeForce GT 130M
GeForce 8200 mGPU	Quadro FX 4800	GeForce G210M
GeForce 8100 mGPU	Quadro FX 4800 for Mac	GeForce G110M
Tesla S2090	Quadro FX 4700 X2	GeForce G105M
Tesla M2090	Quadro FX 4600	GeForce G102M
Tesla S2070	Quadro FX 3800	GeForce 9800M GTX
Tesla M2070	Quadro FX 3700	GeForce 9800M GT
Tesla C2070	Quadro FX 1800	GeForce 9800M GTS
Tesla S2050	Quadro FX 1700	GeForce 9800M GS
Tesla M2050	Quadro FX 580	GeForce 9700M GTS
Tesla C2050	Quadro FX 570	GeForce 9700M GT
Tesla S1070	Quadro FX 470	GeForce 9650M GS
Tesla C1060	Quadro FX 380	GeForce 9600M GT
Tesla S870	Quadro FX 370	GeForce 9600M GS
Tesla C870	Quadro FX 370 Low Profile	GeForce 9500M GS
Tesla D870	Quadro CX	GeForce 9500M G
QUADRO MOBILE PRODUCTS	Quadro NVS 450	GeForce 9300M GS
Quadro FX 3700M	Quadro NVS 420	GeForce 9300M G
Quadro FX 3600M	Quadro NVS 295	GeForce 9200M GS
Quadro FX 2700M	Quadro NVS 290	GeForce 9100M G
Quadro FX 1700M	Quadro Plex 2100 D4	GeForce 8800M GTS
Quadro FX 1600M	Quadro Plex 2200 D2	GeForce 8700M GT
Quadro FX 770M	Quadro Plex 2100 S4	GeForce 8600M GT
Quadro FX 570M	Quadro Plex 1000 Model IV	GeForce 8600M GS
Quadro FX 370M	GEFORCE MOBILE PRODUCTS	GeForce 8400M GT
Quadro FX 360M	GeForce GTX 280M	GeForce 8400M GS

2.2.2 NVIDIA设备驱动程序

NVIDIA提供了一些系统软件来实现应用程序与支持CUDA的硬件之间的通信。如果正确安装了NVIDIA GPU,那么在机器上将已经安装好了这些软件。要确保安装最新的驱动程序,可以访问网址www.nvidia.com/cuda并点击“Download Drivers”链接,然后选择与开发环境相符的图形卡和操作系统。根据所选平台的安装指令,在机器上将安装最新的NVIDIA系统软件。

2.2.3 CUDA开发工具箱

在有了一款支持CUDA的GPU和相应的NVIDIA设备驱动程序后，就可以运行编译好的CUDA C代码。这意味着，你可以下载基于CUDA编写的应用程序，并且这些应用程序能够在图形处理器上成功执行。然而，我们假设你想要做的不仅仅是运行代码，否则，本书也就没有意义了。如果想要使用CUDA C为NVIDIA GPU开发代码，那么还需要其他一些软件。但正如前面已经提到过的，任何一款软件都不需要花费一分钱。

下一章将介绍这些软件的详细信息，但由于CUDA C应用程序将在两个不同的处理器上执行计算，因此需要两个编译器。其中一个编译器为GPU编译代码，而另一个为CPU编译代码。NVIDIA提供了编译GPU代码的编译器。与NVIDIA设备驱动程序一样，可以在网址<http://developer.nvidia.com/object/gpucomputing.html>下载CUDA工具箱。点击“CUDA Toolkit”链接，将弹出如图2.1所示的下载页面。

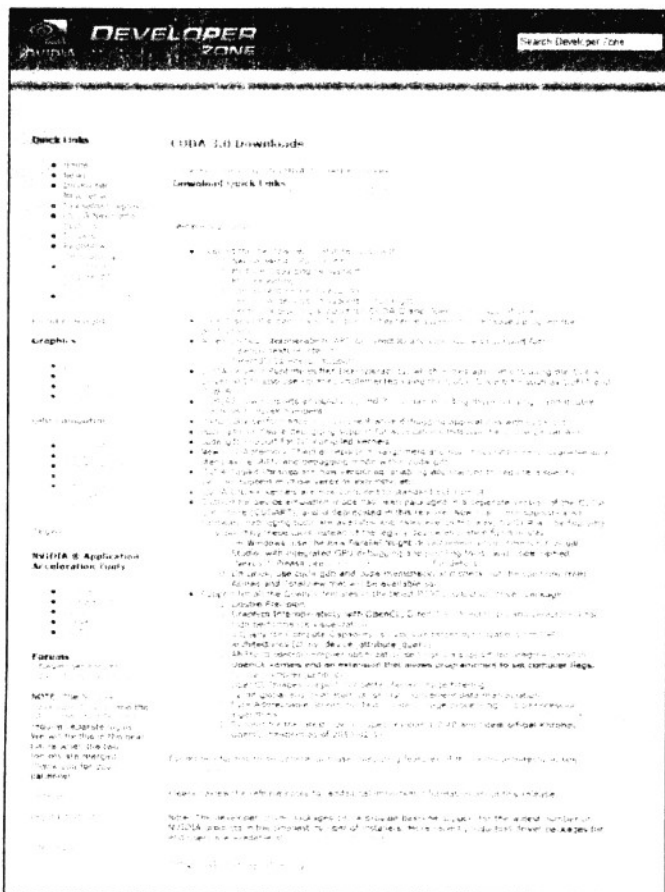


图2.1 CUDA下载页面

在这个页面上选择相应的平台，包括32位/64位版本的Windows XP、Windows Vista、Windows 7、Linux和Mac OS等，然后下载CUDA工具箱来编译在本书中包含的代码示例。此外，你还可以下载GPU Computing SDK代码示例集，其中包含了大量非常有帮助的示例程序。本书不会介绍GPU Computing SDK代码示例集，但它们对于本书内容是一种非常好的补充，从该示例集中不仅可以学习到更多编程风格，还可以阅读更多的示例代码。需要注意的是，虽然本书的所有代码都可以在Linux，Windows以及Mac OS等平台上运行，但我们应用程序的目标运行平台为Linux和Windows。如果你正使用Mac OS X，那么有些代码示例可能得不到支持。

2.2.4 标准的C编译器

前面提到过，需要对GPU和CPU分别采用不同的编译器。如果按照之前的建议下载并安装CUDA工具箱，那么就会获得一个编译GPU代码的编译器。之后，唯一剩下的问题就是还需要一个CPU编译器，我们接下来将解决这个问题。

1. Windows

在Microsoft Windows平台上，包括Windows XP、Windows Vista、Windows Server 2008以及Windows 7，我们推荐使用Microsoft Visual Studio C编译器。NVIDIA当前同时支持Visual Studio 2005和Visual Studio 2008。当Microsoft发布新的版本时，NVIDIA也将增加对Visual Studio新版本的支持，同时去掉对一些旧版本的支持。许多C和C++开发人员都已经在机器上安装了Visual Studio 2005或Visual Studio 2008，因此如果你也是这种情况，那么可以跳过本节内容。

如果没有某个NVIDIA支持的Visual Studio版本，而且你也不准备购买一个新版本，那么可以使用Microsoft在其网址上免费提供的Visual Studio 2008 Express版本。虽然Visual Studio Express版本通常并不适合商业软件的开发，但由于不需要花钱购买软件许可就可以在Windows平台上编写CUDA C代码，因此这是一种非常好的方式。如果你需要Visual Studio 2008，那么直接到网址www.microsoft.com/visualstudio去购买吧。

2. Linux

大多数Linux发行版本通常都带有一个GNU C编译器（gcc）。从CUDA 3.0开始，在以下Linux版本中都包含一个支持的gcc版本：

- Red Hat Enterprise Linux 4.8
- Red Hat Enterprise Linux 5.3
- OpenSUSE 11.1
- SUSE Linux Enterprise Desktop 11
- Ubuntu 9.04
- Fedora 10

如果你是一个忠实的Linux用户，那么可能已经注意到，许多Linux软件包并不仅仅只是在“所支持”的平台上工作。CUDA也不例外，因此即使在这里没有列出你喜欢使用的发行版本，但仍然值得一试。发行版本的内核，gcc以及glibc在很大程度上决定着该发行版本是否兼容。

3. Macintosh OS X

如果希望在Mac OS X上开发，那么需要确保机器上Mac OS X的最低版本不低于10.5.7。这其中包括10.6版本，即Mac OS X“雪豹（Snow Leopard）”。你需要通过下载并安装Apple的开发工具Xcode来安装gcc。这个软件对于苹果开发联盟（Apple Developer Connection，ADC）的成员来说是免费的，可以从<http://developer.apple.com/tools/Xcode>下载。虽然本书的代码都是在Linux和Windows等平台上开发的，但无需任何修改也应该可以在Mac OS X系统上运行。

2.3 本章小结

如果按照本章给出的步骤循序渐进，那么你已经准备好了CUDA C的开发环境。或许你已经开始编译/运行从NVIDIA网址下载的NVIDIA GPU Computing SDK示例代码了。如果情况如此，那么你无疑是值得表扬的！如果不是，也不必担心。本书包含了你需要的所有知识。现在，你可能已经准备好了编写第一个CUDA C程序，让我们开始吧。

第 3 章

CUDA C简介

如果你学习了第1章的内容，那么我们认为你已经了解了图形处理器的强大计算能力，并且需要在程序中使用这种能力。此外，如果你学习了第2章的内容，那么就应该配置好了编译和运行CUDA C代码的开发环境。如果你跳过了这两章的内容，例如只是想浏览代码示例，或者只是在书店里随机地打开这本书翻到这页，或者可能迫不及待地想立即开始动手编写和运行代码，那么也没关系。无论是何种情况，你都应该准备好了编写第一个代码示例，那么让我们开始吧。

3.1 本章目标

通过本章的学习，你可以：

- 编写第一段CUDA C代码。
- 了解为主机（Host）编写的代码与为设备（Device）编写的代码之间的区别。
- 如何从主机上运行设备代码。
- 了解如何在支持CUDA的设备上使用设备内存。
- 了解如何查询系统中支持CUDA的设备的信息。

3.2 第一个程序

我们希望通过示例来学习CUDA C，因此来看第一个CUDA C示例。为了保持计算机编程书籍的行文风格，我们首先给出的是一个“Hello, World!”示例。

3.2.1 Hello, World!

```
#include "../common/book.h"

int main( void ) {
    printf( "Hello, World!\n" );
    return 0;
}
```

当看到这段代码时，你肯定在怀疑本书是不是一个骗局。这不就是C吗？CUDA C是不是真的存在？这些问题的答案都是肯定的。当然，本书也不是一个骗局。这个简单的“Hello, World!”示例只是为了说明，CUDA C与你熟悉的标准C在很大程度上是没有区别的。

这个示例很简单，它能够完全在主机上运行。然而，这个示例引出了本书的一个重要区分：我们将CPU以及系统的内存称为主机，而将GPU及其内存称为设备。这个示例程序与你编写过的代码非常相似，因为它并不考虑主机之外的任何计算设备。

为了避免使你产生一无所获的感觉，我们将逐渐完善这个简单示例。我们来看看如何使用GPU（这就是一个设备）来执行代码。在GPU设备上执行的函数通常称为核函数（Kernel）。

3.2.2 核函数调用

现在，我们在示例程序中添加一些代码，这些代码比最初的“Hello, World!”程序看上去会陌生一些。

```
#include <iostream>

__global__ void kernel( void ) {
}

int main( void ) {
    kernel<<<1,1>>>>();
    printf( "Hello, World!\n" );
    return 0;
}
```

这个程序与最初的“Hello, World!”相比，多了两个值得注意的地方：

- 一个空的函数kernel()，并且带有修饰符__global__。
- 对这个空函数的调用，并且带有修饰字符<<<1,1>>>。

在上一节中看到，代码默认是由系统的标准C编译器来编译的。例如，在Linux操作系统上用GNU gcc来编译主机代码，而在Windows系统上用Microsoft Visual C来编译主机代码。NVIDIA工具只是将代码交给主机编译器，它表现出的行为就好像CUDA不存在一样。

现在，我们看到了CUDA C为标准C增加的__global__修饰符。这个修饰符将告诉编译器，函数应该编译为在设备而不是主机上运行。在这个简单的示例中，函数kernel()将被交给编译设备代码的编译器，而main()函数将被交给主机编译器（与上一个例子一样）。

那么，kernel()的调用究竟代表着什么含义，并且为什么必须加上尖括号和两个数值？注意，这正是使用CUDA C的地方。

我们已经看到，CUDA C需要通过某种语法方法将一个函数标记为“设备代码（Device Code）”。这并没有什么特别之处，而只是一种简单的表示方法，表示将主机代码发送到一个编译器，而将设备代码发送到另一个编译器。事实上，这里的关键在于如何在主机代码中调用设备代码。CUDA C的优势之一在于，它提供了与C在语言级别上的集成，因此这个设备函数调用看上去非常像主机函数调用。在后面将详细介绍在这个函数调用背后发生的动作，但就目前而言，只需知道CUDA编译器和运行时将负责实现从主机代码中调用设备代码。

因此，这个看上去有些奇怪的函数调用实际上表示调用设备代码，但为什么要使用尖括号和数字？尖括号表示要将一些参数传递给运行时系统。这些参数并不是传递给设备代码的参数，而是告诉运行时如何启动设备代码。在第4章中，我们将了解这些参数对运行时的作用。传递给设备代码本身的参数是放在圆括号中传递的，就像标准的函数调用一样。

3.2.3 传递参数

前面提到过可以将参数传递给核函数，现在就来看一个示例。考虑下面对“Hello，

World! ” 应用程序的修改：

```
#include <iostream>
#include "book.h"

__global__ void add( int a, int b, int *c ) {
    *c = a + b;
}

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

    add<<<1,1>>>( 2, 7, dev_c );

    HANDLE_ERROR( cudaMemcpy( &c,
                               dev_c,
                               sizeof(int),
                               cudaMemcpyDeviceToHost ) );

    printf( "2 + 7 = %d\n", c );
    cudaFree( dev_c );

    return 0;
}
```

注意这里增加了多行代码，在这些代码中包含两个概念：

- 可以像调用C函数那样将参数传递给核函数。
- 当设备执行任何有用的操作时，都需要分配内存，例如将计算值返回给主机。

在将参数传递给核函数的过程中没有任何特别之处。除了尖括号语法之外，核函数的外表和行为看上去与标准C中的任何函数调用一样。运行时系统负责处理将参数从主机传递给设备的过程中的所有复杂操作。

更需要注意的地方在于通过cudaMalloc()来分配内存。这个函数调用的行为非常类似于标准的C函数malloc()，但该函数的作用是告诉CUDA运行时在设备上分配内存。第一个参数是一个指针，指向用于保存新分配内存地址的变量，第二个参数是分配内存的大小。除了分配内存的指针不是作为函数的返回值外，这个函数的行为与malloc()是相同的，并且返回类型为void*。函数调用外层的HANDLE_ERROR()是我们定义的一个宏，作为本书辅助代码的一部分。这个宏只是判断函数调用是否返回了一个错误值，如果是的话，那么将输出相应的错误消息，退出应用程序并将退出码设置为EXIT_FAILURE。虽然你也可以在自己的应用程序中使用这个错误处理码，但这种做法在产品级的代码中很可能是不够的。

这段代码引出了一个微妙但却重要的问题。CUDA C的简单性及其强大功能在很大程度上都是来源于它淡化了主机代码和设备代码之间的差异。然而，程序员一定不能在主机代码中对`cudaMalloc()`返回的指针进行解引用（Dereference）。主机代码可以将这个指针作为参数传递，对其执行算术运算，甚至可以将其转换为另一种不同的类型。但是，绝对不可以使用这个指针来读取或者写入内存。

遗憾的是，编译器无法防止这种错误的发生。如果能够在主机代码中对设备指针进行解引用，那么CUDA C将非常完美，因为这看上去就与程序中其他的指针完全一样了。我们可以将设备指针的使用限制总结如下：

可以将`cudaMalloc()`分配的指针传递给在设备上执行的函数。

可以在设备代码中使用`cudaMalloc()`分配的指针进行内存读/写操作。

可以将`cudaMalloc()`分配的指针传递给在主机上执行的函数。

不能在主机代码中使用`cudaMalloc()`分配的指针进行内存读/写操作。

如果你仔细阅读了前面的内容，那么可以得出以下推论：不能使用标准C的`free()`函数来释放`cudaMalloc()`分配的内存。要释放`cudaMalloc()`分配的内存，需要调用`cudaFree()`，这个函数的行为与`free()`的行为非常相似。

我们已经看到了如何在设备上分配内存和释放内存，同时也清楚地看到，在主机上不能对这块内存做任何修改。在示例程序中剩下两行代码给出了访问设备内存的两种最常见方法——在设备代码中使用设备指针以及调用`cudaMemcpy()`。

设备指针的使用方式与标准C中指针的使用方式完全一样。语句`*c = a + b`的含义同样非常简单：将参数a和b相加，并将结果保存在c指向的内存中。这个计算过程非常简单，甚至吸引不了我们的兴趣。

在前面列出了在设备代码和主机代码中可以/不可以使用设备指针的各种情形。在主机指针的使用上有着类似的限制。虽然可以将主机指针传递给设备代码，但如果想通过主机指针来访问设备代码中的内存，那么同样会出现问题。总的来说，主机指针只能访问主机代码中的内存，而设备指针也只能访问设备代码中的内存。

前面曾提到过，在主机代码中可以通过调用`cudaMemcpy()`来访问设备上的内存。这个函数调用的行为类似于标准C中的`memcpy()`，只不过多了一个参数来指定设备内存指针究竟是源指针还是目标指针。在这个示例中，注意`cudaMemcpy()`的最后一个参数为`cudaMemcpyDeviceToHost`，这个参数将告诉运行时源指针是一个设备指针，而目标指针是一个主机指针。

显然，`cudaMemcpyHostToDevice`将告诉运行时相反的含义，即源指针位于主机上，而目标指针是位于设备上。此外还可以通过传递参数`cudaMemcpyDeviceToDevice`来告诉运行时这

两个指针都是位于设备上。如果源指针和目标指针都位于主机上，那么可以直接调用标准C的memcpy()函数。

3.3 查询设备

由于我们希望在设备上分配内存和执行代码，因此如果在程序中能够知道设备拥有多少内存以及具备哪些功能，那么将非常有用。而且，在一台计算机上拥有多个支持CUDA的设备也是很常见的情形。在这些情况中，我们希望通过某种方式来确定使用的是哪一个处理器。

例如，在许多主板中都集成了NVIDIA图形处理器。当计算机生产商或者用户将一块独立的图形处理器添加到计算机时，那么就有了两个支持CUDA的处理器。某些NVIDIA产品，例如GeForce GTX 295，在单块卡上包含了两个GPU，因此使用这类产品的计算机也就拥有了两个支持CUDA的处理器。

在深入研究如何编写设备代码之前，我们需要通过某种机制来判断计算机中当前有哪些设备，以及每个设备都支持哪些功能。幸运的是，可以通过一个非常简单的接口来获得这种信息。首先，我们希望知道在系统中有多少个设备是支持CUDA架构的，并且这些设备能够运行基于CUDA C编写的核函数。要获得CUDA设备的数量，可以调用cudaGetDeviceCount()。这个函数的作用从它的名字就可以看出来。

```
int count;
HANDLE_ERROR( cudaGetDeviceCount( &count ) );
```

在调用cudaGetDeviceCount()后，可以对每个设备进行迭代，并查询各个设备的相关信息。CUDA运行时将返回一个cudaDeviceProp类型的结构，其中包含了设备的相关属性。我们可以获得哪些属性？从CUDA 3.0开始，在cudaDeviceProp结构中包含了以下信息：

```
struct cudaDeviceProp {
    char name[256];
    size_t totalGlobalMem;
    size_t sharedMemPerBlock;
    int regsPerBlock;
    int warpSize;
    size_t memPitch;
    int maxThreadsPerBlock;
    int maxThreadsDim[3];
    int maxGridSize[3];
    size_t totalConstMem;
    int major;
    int minor;
    int clockRate;
    size_t textureAlignment;
```

```

int deviceOverlap;
int multiProcessorCount;
int kernelExecTimeoutEnabled;
int integrated;
int canMapHostMemory;
int computeMode;
int maxTexture1D;
int maxTexture2D[2];
int maxTexture3D[3];
int maxTexture2DArray[3];
int concurrentKernels;
}

```

其中,有些属性的含义是显而易见的,其他属性的含义如下所示(见表3.1)。

表3.1 CUDA设备属性

设备属性	描 述
char name[256];	标识设备的ASCII字符串(例如, "GeForce GTX 280")
size_t totalGlobalMem	设备上全局内存的总量,单位为字节
size_t sharedMemPerBlock	在一个线程块(Block)中可使用的最大共享内存数量,单位为字节
int regsPerBlock	每个线程块中可用的32位寄存器数量
int warpSize	在一个线程束(Warp)中包含的线程数量
size_t memPitch	在内存复制中最大的修正量(Pitch),单位为字节
int maxThreadsPerBlock	在一个线程块中可以包含的最大线程数量
int maxThreadsDim[3]	在多维线程块数组中,每一维可以包含的最大线程数量
int maxGridSize[3]	在一个线程格(Grid)中,每一维可以包含的线程块数量
size_t totalConstMem	常量内存的总量
int major	设备计算功能集(Compute Capability)的主版本号
int minor	设备计算功能集的次版本号
size_t textureAlignment	设备的纹理对齐(Texture Alignment)要求
int deviceOverlap	一个布尔类型值,表示设备是否可以同时执行一个cudaMemory()调用和一个核函数调用
int multiProcessorCount	设备上多处理器的数量
int kernelExecTimeoutEnabled	一个布尔值,表示在该设备上执行的核函数是否存在运行时限制
int integrated	一个布尔值,表示设备是否是一个集成GPU(即该GPU属于芯片组的一部分而非独立的GPU)
int canMapHostMemory	一个布尔类型的值,表示设备是否将主机内存映射到CUDA设备地址空间
int computeMode	表示设备的计算模式:默认(Default),独占(Exclusive),或者禁止(Prohibited)
int maxTexture1D	一维纹理的最大大小
int maxTexture2D[2]	二维纹理的最大维数
int maxTexture3D[3]	三维纹理的最大维数
int maxTexture2DArray[3]	二维纹理数组的最大维数
int concurrentKernels	一个布尔类型值,表示设备是否支持在同一个上下文中同时执行多个核函数

就目前而言，我们不会详细介绍所有这些属性。事实上，在上面的列表中没有给出属性的一些重要细节，因此你需要参考《NVIDIA CUDA Programming Guide》以了解更多的信息。当开始编写应用程序时，这些属性会非常有用。但就目前而言，我们只是给出了如何查询每个设备并且报告设备的相应属性。下面给出了对设备进行查询的代码：

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp prop;

    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );

        //对设备的属性执行某些操作

    }
}
```

在知道了每个可用的属性后，接下来就可以将注释“对设备的属性执行某些操作”替换为一些具体的操作：

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp prop;

    int count;
    HANDLE_ERROR( cudaGetDeviceCount( &count ) );
    for (int i=0; i< count; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        printf( "    --- General Information for device %d ---\n", i );
        printf( "Name:    %s\n", prop.name );
        printf( "Compute capability:  %d.%d\n", prop.major, prop.minor );
        printf( "Clock rate:    %d\n", prop.clockRate );
        printf( "Device copy overlap:  " );
        if (prop.deviceOverlap)
            printf( "Enabled\n" );
        else
            printf( "Disabled\n" );
        printf( "Kernel execution timeout :  " );
        if (prop.kernelExecTimeoutEnabled)
            printf( "Enabled\n" );
```

```

else
    printf( "Disabled\n" );

printf( "    --- Memory Information for device %d ---\n", i );
printf( "Total global mem:  %ld\n", prop.totalGlobalMem );
printf( "Total constant Mem: %ld\n", prop.totalConstMem );
printf( "Max mem pitch:  %ld\n", prop.memPitch );
printf( "Texture Alignment:  %ld\n", prop.textureAlignment );
printf( "    --- MP Information for device %d ---\n", i );
printf( "Multiprocessor count:  %d\n",
        prop.multiProcessorCount );
printf( "Shared mem per mp:  %ld\n", prop.sharedMemPerBlock );
printf( "Registers per mp:  %d\n", prop.regsPerBlock );
printf( "Threads in warp:  %d\n", prop.warpSize );
printf( "Max threads per block:  %d\n",
        prop.maxThreadsPerBlock );
printf( "Max thread dimensions:  (%d, %d, %d)\n",
        prop.maxThreadsDim[0], prop.maxThreadsDim[1],
        prop.maxThreadsDim[2] );
printf( "Max grid dimensions:  (%d, %d, %d)\n",
        prop.maxGridSize[0], prop.maxGridSize[1],
        prop.maxGridSize[2] );
printf( "\n" );
}
}

```

3.4 设备属性的使用

除非是编写一个需要输出每个支持CUDA的显卡的详细属性的应用程序，否则我们是否需要了解系统中每个设备的属性？作为软件开发人员，我们希望编写出的软件是最快的，因此可能需要选择拥有最多处理器的GPU来运行代码。或者，如果核函数与CPU之间需要进行密集交互，那么可能需要在集成的GPU上运行代码，因为它可以与CPU共享内存。这两个属性都可以通过`cudaGetDeviceProperties()`来查询。

假设我们正在编写一个需要使用双精度浮点计算的应用程序。在快速翻阅《NVIDIA CUDA Programming Guide》的附录A后，我们知道计算功能集的版本为1.3或者更高的显卡才能支持双精度浮点数学计算。因此，要想成功地在应用程序中执行双精度浮点运算，GPU设备至少需要支持1.3或者更高版本的计算功能集。

根据在`cudaGetDeviceCount()`和`cudaGetDeviceProperties()`中返回的结果，我们可以对每个设备进行迭代，并且查找主版本号大于1，或者主版本号为1且次版本号大于等于3的设备。但

是，这种迭代操作执行起来有些繁琐，因此CUDA运行时提供了一种自动方式来执行这个迭代操作。首先，找出我们希望设备拥有的属性并将这些属性填充到一个cudaDeviceProp结构。

```
cudaDeviceProp prop;
memset( &prop, 0, sizeof( cudaDeviceProp ) );
prop.major = 1;
prop.minor = 3;
```

在填充完cudaDeviceProp结构后，将其传递给cudaChooseDevice()，这样CUDA运行时将查找是否存在某个设备满足这些条件。cudaChooseDevice()函数将返回一个设备ID，然后我们可以将这个ID传递给cudaSetDevice()。随后，所有的设备操作都将在这个设备上执行。

```
#include "../common/book.h"

int main( void ) {
    cudaDeviceProp prop;
    int dev;

    HANDLE_ERROR( cudaGetDevice( &dev ) );
    printf( "ID of current CUDA device: %d\n", dev );

    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 3;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
    printf( "ID of CUDA device closest to revision 1.3: %d\n", dev );
    HANDLE_ERROR( cudaSetDevice( dev ) );
}
```

当前，在系统中拥有多个GPU已是很常见的情况。例如，许多NVIDIA主板芯片组都包含了集成的并且支持CUDA的GPU。当把一个独立的GPU添加到这些系统中时，那么就形成了一个多GPU的平台。而且，NVIDIA的SLI（Scalable Link Interface，可伸缩链路接口）技术使得多个独立的GPU可以并排排列。无论是哪种情况，应用程序都可以从多个GPU中选择最适合的GPU。如果应用程序依赖于GPU的某些特定属性，或者需要在系统中最快的GPU上运行，那么你就需要熟悉这个API，因为CUDA运行时本身并不能保证为应用程序选择最优或者最合适的GPU。

3.5 本章小结

我们已经开始编写了CUDA C程序，这个过程比你想象的要更轻松。从本质上来说，CUDA C只是对标准C进行了语言级的扩展，通过增加一些修饰符使我们可以指定哪些代码在

设备上运行，以及哪些代码在主机上运行。在函数前面添加关键字`__global__`将告诉编译器把该函数放在GPU上运行。为了使用GPU的专门内存，我们还学习了与C的`malloc()`、`memcpy()`和`free()`等API对应的CUDA API。这些函数的CUDA版本，包括`cudaMalloc()`、`cudaMemcpy()`以及`cudaFree()`，分别实现了分配设备内存，在设备和主机之间复制数据，以及释放设备内存等功能。

在本书的后面还将介绍一些更有趣的示例，这些示例都是关于如何将GPU设备作为一种大规模并行协处理器来使用。现在，你应该知道CUDA C程序的入门阶段是很容易的，在接下来的第4章中，我们将看到在GPU上执行并行代码同样是非常容易的。

第 4 章

CUDA C并行编程

在上一章中，我们看到了使用CUDA C来编写GPU上执行的代码是非常简单的。我们甚至学习了如何将两个数值相加，尽管这两个数值只是2和7。当然，这个示例并没有给我们留下太深的印象，但我们希望通过这个示例使你相信用CUDA C来编写代码是很容易的，并且会激发你去学习更多的知识。GPU计算的应用前景在很大程度上取决于能否从许多问题中发掘出大规模并行性。因此，我们在本章将介绍如何通过CUDA C在GPU上并行执行代码。

4.1 本章目标

通过本章的学习，你可以：

- 了解CUDA在实现并行性时采用的一种重要方式。
- 用CUDA C编写第一段并行代码。

4.2 CUDA并行编程

前面我们看到将一个标准C函数放到GPU设备上运行是很容易的。只需在函数定义前面加上__global__修饰符，并通过一种特殊的尖括号语法来调用它，就可以在GPU上执行这个函数。这种方法虽然非常简单，但同样也是很低效的，因为NVIDIA的硬件工程师们早已对图形处理器进行优化，使其可以同时并行执行数百次的计算。然而，我们在这个示例中只调用了—个核函数，并且该函数在GPU上以串行方式运行。在本章中，我们将看到如何启动一个并行执行的设备核函数。

4.2.1 矢量求和运算

我们通过一个简单示例来说明线程的概念，以及如何使用CUDA C来实现线程。假设有两组数据，我们需要将这两组数据中对应的元素两两相加，并将结果保存在第三个数组中。在图4.1中给出了这个计算过程。如果你学过线性代数，就会发现这实际上是一个矢量求和运算。

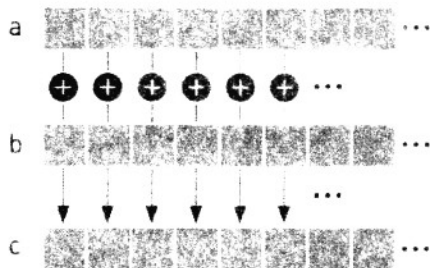


图4.1 两个矢量的求和

1. 基于CPU的矢量求和

首先，我们来看看如何通过传统的C代码来实现这个求和运算：

```
#include "../common/book.h"

#define N 10

void add( int *a, int *b, int *c ) {
```

```

int tid = 0;    //这是第0个CPU. 因此索引从0开始
while (tid < N) {
    c[tid] = a[tid] + b[tid];
    tid += 1;    //由于只有一个CPU, 因此每次递增1
}

int main( void ) {
    int a[N], b[N], c[N];

    //在CPU上为数组 'a' 和 'b' 赋值
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    add( a, b, c );
    //显示结果
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }

    return 0;
}

```

这段示例代码几乎不需要任何解释, 但我们将对函数add()做简要分析, 看看为什么这个函数有些过于复杂。

```

void add( int *a, int *b, int *c ) {
    int tid = 0;    //这是第0个CPU, 因此索引从0开始
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += 1;    //由于只有一个CPU, 因此每次递增1
    }
}

```

函数在一个while循环中计算总和, 其中索引tid的取值范围为0到N-1。我们将a[]和b[]的对应元素相加起来, 并将结果保存在c[]的相应元素中。通常, 可以用更简单的方式来编写这段代码, 例如:

```

void add( int *a, int *b, int *c ) {
    for (i=0; i < N; i++) {
        c[i] = a[i] + b[i];
    }
}

```

上面采用的while循环虽然有些复杂,但这是为了使代码能够在拥有多个CPU或者CPU核的系统上并行运行。例如,在双核处理器上可以将每次递增的大小改为2,这样其中一个核从tid=0开始执行循环,而另一个核从tid=1开始执行循环。第一个核将偶数索引的元素相加,而第二个核则将奇数索引的元素相加。这相当于在每个CPU核上执行以下代码:

第1个CPU核 <pre>void add(int *a, int *b, int *c) { int tid = 0; while (tid < N) { c[tid] = a[tid] + b[tid]; tid += 2; } }</pre>	第2个CPU核 <pre>void add(int *a, int *b, int *c) { int tid = 1; while (tid < N) { c[tid] = a[tid] + b[tid]; tid += 2; } }</pre>
--	--

当然,要在CPU上实际执行这个运算,还需要增加更多的代码。例如,需要编写一定数量的代码来创建工作线程,每个线程都执行函数add(),并假设每个线程都将并行执行。然而,这种假设是一种理想但不实际的想法,线程调度机制的实际运行情况往往并非如此。

2. 基于GPU的矢量求和

我们可以在GPU上实现相同的加法运算,这需要将add()编写为一个设备函数。这段代码与前一章中的代码非常类似。在给出设备代码之前,我们首先给出函数main()。虽然main()在GPU上的实现与在CPU上的实现是不同的,但在下面的代码中并没有包含新内容:

```
#include "../common/book.h"

#define N 10

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    //在GPU上分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    //在CPU上为数组 'a' 和 'b' 赋值
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }
    //将数组 'a' 和 'b' 复制到GPU
```

```

HANDLE_ERROR( cudaMemcpy( dev_a, a, N * sizeof(int),
                          cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMemcpy( dev_b, b, N * sizeof(int),
                          cudaMemcpyHostToDevice ) );

add<<<N,1>>>( dev_a, dev_b, dev_c );

//将数组 'c' 从GPU复制到CPU
HANDLE_ERROR( cudaMemcpy( c, dev_c, N * sizeof(int),
                          cudaMemcpyDeviceToHost ) );

//显示结果
for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}

//释放在GPU上分配的内存
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}

```

需要注意的是，上面的代码中再次使用了一些通用模式：

- 调用cudaMalloc()在设备上为三个数组分配内存：在其中两个数组（dev_a和dev_b）中包含了输入值，而在数组dev_c中包含了计算结果。
- 为了避免内存泄露，在使用完GPU内存后通过cudaFree()释放它们。
- 通过cudaMemcpy()将输入数据复制到设备中，同时指定参数cudaMemcpyHostToDevice，在计算完成后，将计算结果通过参数cudaMemcpyDeviceToHost复制回主机。
- 通过尖括号语法，在主机代码main()中执行add()中的设备代码。

你可能会奇怪为什么要在CPU上对输入数组赋值。其实，这么做并没有什么特殊原因。事实上，如果在GPU上为数组赋值，这个步骤执行得会更快些。但这段代码的目的是说明如何在图形处理器上实现两个矢量的加法运算。因此，我们假设这个加法运算只是其他应用程序中的一个步骤，并且输入数组a[]和b[]可以由其他算法生成，或者由用户从硬盘上读取。我们假设这些数据已经存在了，现在需要对它们执行某种操作。

接下来是add()函数，这个函数看上去非常类似于基于CPU实现的add()。

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    //计算该索引处的数据
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

我们再次看到在函数add()中使用了一种通用模式：

- 编写的一个在设备上执行的函数add()。我们采用C来编写代码，并在函数名字前添加了一个修饰符__global__。

到目前为止，除了执行的计算不是将2和7相加外，在这个示例中没有包含任何新的东西。然而，这个示例中有两个值得注意的地方：尖括号中的参数以及核函数中包含的代码，这两处地方都引入了新的概念。

我们曾看到过通过以下形式启动的核函数：

```
kernel<<<1,1>>>( param1, param2, ... );
```

但在这个示例中，尖括号中的数值并不是1：

```
add<<<N,1>>>( dev_ a, dev_ b, dev_ c );
```

这两个数值代表什么含义？

我们在前面并没有对尖括号中的这两个数值进行解释，而只是很含糊地指出，这两个参数将传递给运行时，作用是告诉运行时如何启动核函数。事实上，在这两个参数中，第一个参数表示设备在执行核函数时使用的并行线程块的数量。在这个示例中，指定这个参数为N。

例如，如果指定的是kernel<<<2,1>>>()，那么可以认为运行时将创建核函数的两个副本，并以并行方式来运行它们。我们将每个并行执行环境都称为一个线程块（Block）。如果指定的是kernel<<<256,1>>>()，那么将有256个线程块在GPU上运行。然而，并行计算从来都不是一个简单的问题。

这种运行方式引出了一个问题：既然GPU将运行核函数的N个副本，那如何在代码中知道当前正在运行的是哪一个线程块？这个问题的答案在于示例中核函数的代码本身。具体来说，是在于变量blockIdx.x：

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;    //计算位于这个索引处的数据
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

乍一看，在编译这个函数时似乎会出现错误，因为代码将变量的值赋给了tid，但之前却没

有定义它。然而，这里不需要定义变量blockIdx，它是一个内置变量，在CUDA运行时中已经预先定义了这个变量。而且，这个变量名字的含义也就是变量的作用。变量中包含的值就是当前执行设备代码的线程块的索引。

你可能会问，为什么不是blockIdx？而是blockIdx.x？事实上，这是因为CUDA支持二维的线程块数组。对于二维空间的计算问题，例如矩阵数学运算或者图像处理，使用二维索引往往会带来很大的便利，因为它可以避免将线性索引转换为矩形索引。即使你不熟悉这些问题类型也不必担心，你只需知道在某些情况下，使用二维索引比使用一位索引要更为方便。当然，你并不是必须使用它。

当启动核函数时，我们将并行线程块的数量指定为N。这个并行线程块集合也称为一个线程格（Grid）。这是告诉运行时，我们想要一个一维的线程格，其中包含N个线程块。每个线程块的blockIdx.x值都是不同的，第一个线程块的blockIdx.x为0，而最后一个线程块的blockIdx.x为N-1。因此，假设有4个线程块，并且所有线程块都运行相同的设备代码，但每个线程块的blockIdx.x的值是不同的。当四个线程块并行执行时，运行时将用相应的线程块索引来替换blockIdx.x，每个线程块实际执行的代码如下所示。

第1个线程块 <pre>__global__ void add(int *a, int *b, int *c) { int tid = 0; if (tid < N) c[tid] = a[tid] + b[tid]; }</pre>	第2个线程块 <pre>__global__ void add(int *a, int *b, int *c) { int tid = 1; if (tid < N) c[tid] = a[tid] + b[tid]; }</pre>
第3个线程块 <pre>__global__ void add(int *a, int *b, int *c) { int tid = 2; if (tid < N) c[tid] = a[tid] + b[tid]; }</pre>	第4个线程块 <pre>__global__ void add(int *a, int *b, int *c) { int tid = 3; if (tid < N) c[tid] = a[tid] + b[tid]; }</pre>

如果回顾前面给出的基于CPU的示例，你会发现在基于CPU的示例中需要遍历从0到N-1的索引从而对两个矢量求和。而在基于GPU的示例中，运行时已经启动了一个核函数，其中每个线程块都拥有0到N-1中的一个索引，因此这几乎将所有的遍历工作都已经完成了。这显然是一件好事，使得我们有更多的时间去写博客，而写的内容或许正是关于如何节约编码的时间。

最后一个需要回答的问题是，为什么要判断tid是否小于N？在通常情况下，tid总是小于N的，因为在核函数中都是这么假设的。然而，我们仍然怀疑是否有人会破坏代码中作出的假设。

破坏假设意味着破坏代码，意味着在代码中出现错误，而我们也不得不工作到深夜去找出代码中的问题，因此也没有时间登录博客。如果没有检查tid是否小于N，并且最终发生了内存非法访问，那么将造成一种糟糕的情况。事实上，这种情况可能会终止核函数的运行，因为GPU有着完善的内存管理机制，它将强行结束所有违反内存访问规则的进程。

如果出现了这类问题，那么代码中的HANDLE_ERROR()宏将检测出这种情况并发出警告信息。与传统的C编程一样，这里的函数会由于某种原因而返回错误码。虽然你通常会忽视这些错误码，但我们还是建议检查每个失败操作的结果，这样就不会像我们曾经一样经历数小时的痛苦时间。由于这种情况经常发生，而这些错误的出现通常也无法阻止应用程序的继续执行，但在大多数情况下，它们将对程序的后续执行造成各种不可预期的糟糕后果。

此时，你或许正在GPU上并行运行代码。你可能听说过在GPU上运行代码是很复杂的，或者有人告诉你必须理解计算机图形学才能在图形处理器上实现通用算法编程。我们希望你开始意识到，CUDA C的出现使得在GPU上编写并行代码变得更加容易。我们在这里给出的示例只是对长度为10的矢量进行相加。如果想编写更大规模的并行应用程序，那么也是非常容易的，只需将#define N 10中的10改为10000或者50000，这样可以启动数万个并行线程块。然而，需要注意的是：在启动线程块数组时，数组每一维的最大数量都不能超过65 535。这是一种硬件限制，如果启动的线程块数量超过了这个限值，那么程序将运行失败。在下一章中，我们将看到如何在这种限制范围内工作。

4.2.2 一个有趣的示例

我们并不认为将矢量相加是一个无趣的示例，但下面这个示例将满足那些希望看到并行CUDA C更有趣应用的读者。

在接下来的示例中将介绍如何绘制Julia集的曲线。对于不熟悉Julia集的读者，可以简单地将Julia集认为是满足某个复数计算函数的所有点构成的边界。显然，这听起来似乎比矢量相加和矩阵乘法更为无趣。然而，对于函数参数的所有取值，生成的边界将形成一种不规则的碎片形状，这是数学中最有趣和最漂亮的形状之一。

生成Julia集的算法非常简单。Julia集的基本算法是，通过一个简单的迭代等式对复平面中的点求值。如果在计算某个点时，迭代等式的计算结果是发散的，那么这个点就不属于Julia集合。也就是说，如果在迭代等式中计算得到的一系列值朝着无穷大的方向增长，那么这个点就被认为不属于Julia集合。相反，如果在迭代等式中计算得到的一系列值都位于某个边界范围之内，那么这个点就属于Julia集合。

从计算上来看，这个迭代等式非常简单，如等式4.1所示：

等式4.1

$$Z_{n+1} = Z_n^2 + C$$

可以看到，计算等式4.1的迭代过程包括，首先计算当前值的平方，然后再加上一个常数以得到等式的下一个值。

1. 基于CPU的Julia集

我们将分析一段计算Julia集的源代码。由于这个程序比到目前为止看到的其他程序都更为复杂，因此我们将其分解为多个代码段。在本章的后面部分将给出完整的源代码。

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *ptr = bitmap.get_ptr();

    kernel( ptr );

    bitmap.display_and_exit();
}
```

main函数非常简单。它通过工具库创建了一个大小合适的位图图像。接下来，它将一个指向位图数据的指针传递给了核函数。

```
void kernel( unsigned char *ptr ){
    for (int y=0; y<DIM; y++) {
        for (int x=0; x<DIM; x++) {
            int offset = x + y * DIM;

            int juliaValue = julia( x, y );
            ptr[offset*4 + 0] = 255 * juliaValue;
            ptr[offset*4 + 1] = 0;
            ptr[offset*4 + 2] = 0;
            ptr[offset*4 + 3] = 255;
        }
    }
}
```

核函数对将要绘制的所有点进行迭代，并在每次迭代时调用julia()来判断该点是否属于Julia集。如果该点位于集合中，那么函数julia()将返回1，否则将返回0。如果julia()返回1，那么将点的颜色设置为红色，如果返回0则设置为黑色。这些颜色是任意的，你可以根据自己的喜好来选择合适的颜色。

```
int julia( int x, int y ) {
```



```

const float scale = 1.5;
float jx = scale * (float)(DIM/2 - x)/(DIM/2);
float jy = scale * (float)(DIM/2 - y)/(DIM/2);

cuComplex c(-0.8, 0.156);
cuComplex a(jx, jy);

int i = 0;
for (i=0; i<200; i++) {
    a = a * a + c;
    if (a.magnitude2() > 1000)
        return 0;
}

return 1;
}

```

这个函数是示例程序的核心。函数首先将像素坐标转换为复数空间的坐标。为了将复平面的原点定位到图像中心，我们将像素位置移动了DIM/2。然后，为了确保图像的范围为-1.0到1.0，我们将图像的坐标缩放为DIM/2倍。这样，给定一个图像点(x,y)，就可以计算并得到复空间中的一个点((DIM/2 - x)/(DIM/2), ((DIM/2 - y)/(DIM/2))。

然后，我们引入了一个scale因数来实现图形的缩放。当前，这个scale被硬编码为1.5，当然你也可以调整这个参数来缩小或者放大图形。更完善的做法是将其作为一个命令行参数。

在计算出复空间中的点之后，需要判断这个点是否属于Julia集。在前面提到过，这是通过计算迭代等式 $Z_{n+1} = z_n^2 + C$ 来判断的。C是一个任意的复数常量，我们选择的值是 $-0.8 + 0.156i$ ，因为这个值刚好能生成一张有趣的图片。如果想看看其他的Julia集，那么可以修改这个常量。

在这个示例中，我们计算了200次迭代。在每次迭代计算完成后，都会判断结果是否超过某个阈值（在这里是1 000）。如果超过了这个阈值，那么等式就是发散的，因此将返回0以表示这个点不属于Julia集合。反之，如果所有的200次迭代都计算完毕后，并且结果仍然小于1 000，那么我们就认为这个点属于该集合，并且给调用者kernel()返回1。

由于所有计算都是在复数上进行的，因此我们定义了一个通用结构来保存复数值。

```

struct cuComplex {
    float  r;
    float  i;
    cuComplex( float a, float b ) : r(a), i(b) {}
    float magnitude2( void ) { return r * r + i * i; }
    cuComplex operator*(const cuComplex& a) {

```

```

        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};

```

这个类表示一个复数值，包含两个成员：单精度的实部 r 和单精度的虚部 i 。在这个类中定义了复数的加法运算符和乘法运算符。（如果你完全不熟悉复数值，那么可以在网上快速学习一本入门书）。最后，我们还定义了一个方法来返回复数的模。

2. 基于GPU的Julia集

在GPU设备上计算Julia集的代码与CPU版本的代码非常类似。

```

int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char    *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() ) );

    dim3    grid(DIM,DIM);
    kernel<<<grid,1>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                               dev_bitmap,
                               bitmap.image_size(),
                               cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    cudaFree( dev_bitmap );
}

```

这个版本的main()看上去比基于CPU的版本要更复杂，但实际上程序的执行流程是相同的。与CPU版本一样，首先通过工具库创建一个DIM x DIM大小的位图图像。然而，由于接下来将在GPU上执行计算，因此在代码中声明了一个指针dev_bitmap，用来保存设备上数据的副本。同样，为了保存数据，我们需要通过cudaMalloc()来分配内存。

然后，我们像在CPU版本中那样来运行Kernel()函数，但是它现在是一个__global__函数，这意味着将在GPU上运行。与CPU示例一样，在前面代码中分配的指针会传递给kernel()来保存计算结果。唯一的差异在于这线程块内存驻留在GPU上，而不是在主机上。

这里最需要关注的是，在程序中指定了多个并行线程块来执行函数kernel()。由于每个点的计算与其他点的计算都是相互独立的，因此可以为每个需要计算的点都执行该函数的一个副本。之前提到过，在某些情况中使用二维索引会带来一定的帮助。显然，在二维空间（例如复平面）中计算函数值正属于这种情况。因此，在下面这行代码中声明了一个二维的线程格：

```
dim3 grid(DIM,DIM);
```

类型dim3并不是标准C定义的类型。在CUDA头文件中定义了一些辅助类型来封装多维数组。类型dim3表示一个三维数组，可以用于指定启动的线程块的数量。然而，为什么要使用三维数组，而我们实际需要的只是一个二维线程格？

坦白地说，这么做是因为CUDA运行时希望得到一个三维的dim3值。虽然当前并不支持三维的线程格，但CUDA运行时仍然希望得到一个dim3类型的参数，只不过最后一维的大小为1。当仅用两个值来初始化dim3类型的变量时，例如在语句dim3 grid(DIM,DIM)中，CUDA运行时将自动把第3维的大小指定为1。虽然NVIDIA在将来或许会支持3维的线程格，但就目前而言，我们只能暂且满足这个API的要求，因为每当开发人员和API做斗争时，胜利的一方总是API。

然后，在下面这行代码中将dim3变量grid传递给CUDA运行时：

```
kernel<<<grid,1>>>>( dev_bitmap );
```

最后，在执行完kernel()之后，在设备上会生成计算结果，我们需要将这些结果复制回主机。和前面一样，通过cudaMemcpy()来实现这个操作，并将复制方向指定为cudaMemcpyDeviceToHost。

```
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(),
                           dev_bitmap,
                           bitmap.image_size(),
                           cudaMemcpyDeviceToHost ) );
```

在基于CPU的版本与基于GPU的版本之间，关键差异之一在于kernel()的实现。

```
__global__ void kernel( unsigned char *ptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // 现在计算这个位置上的值
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}
```

首先，我们要将Kernel()声明为一个__global__函数，从而可以从主机上调用并在设备上运行。与CPU版本不同的是，我们不需要通过嵌套的for()循环来生成像素索引以传递给julia()。与矢量相加示例一样，CUDA运行时将在变量blockIdx中包含这些索引。这种方式是可行的，因为在声明线程格时，线程格每一维的大小与图像每一维的大小是相等的，因此在(0,0)和(DIM-1,DIM-1)之间的每个像素点(x,y)都能获得一个线程块。

接下来，唯一需要的信息就是要得到输出缓冲区ptr中的线性偏移。这个偏移值是通过另一个内置变量gridDim来计算的。对所有的线程块来说，gridDim是一个常数，用来保存线程格每一维的大小。在来示例中，gridDim的值是(DIM, DIM)。因此，将行索引乘以线程格的宽度，再加上列索引，就得到了ptr中的唯一索引，其取值范围为(DIM*DIM-1)。

```
int offset = x + y * gridDim.x;
```

最后，我们来分析判断某个点是否属于Julia集的代码。这段代码看上去与CPU版本是相同的。

```
__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}
```

我们定义了一个cuComplex结构，在结构中同样使用单精度浮点数来保存复数的实部和虚部。这个结构也定义了加法运算符和乘法运算符，此外还定义了一个函数来返回复数值的模。

```
struct cuComplex {
    float r;
    float i;
    cuComplex( float a, float b ) : r(a), i(b) {}
    __device__ float magnitude2( void ) {
```

```

        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};

```

注意，我们使用了在CPU版本中已经用过的相同的语言结构。二者差异之一在于修饰符__device__，这表示代码将在GPU而不是主机上运行。由于这些函数已声明为__device__函数，因此只能从其他__device__函数或者从__global__函数中调用它们。

综上所述，以下给出了完整的源代码清单：

```

#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 1000
struct cuComplex {
    float    r;
    float    i;
    cuComplex( float a, float b ) : r(a), i(b) {}
    __device__ float magnitude2( void ) {
        return r * r + i * i;
    }
    __device__ cuComplex operator*(const cuComplex& a) {
        return cuComplex(r*a.r - i*a.i, i*a.r + r*a.i);
    }
    __device__ cuComplex operator+(const cuComplex& a) {
        return cuComplex(r+a.r, i+a.i);
    }
};

__device__ int julia( int x, int y ) {
    const float scale = 1.5;
    float jx = scale * (float)(DIM/2 - x)/(DIM/2);
    float jy = scale * (float)(DIM/2 - y)/(DIM/2);

    cuComplex c(-0.8, 0.156);
    cuComplex a(jx, jy);

```

```

    int i = 0;
    for (i=0; i<200; i++) {
        a = a * a + c;
        if (a.magnitude2() > 1000)
            return 0;
    }

    return 1;
}

__global__ void kernel( unsigned char *ptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = blockIdx.x;
    int y = blockIdx.y;
    int offset = x + y * gridDim.x;

    // 现在计算这个位置上的值
    int juliaValue = julia( x, y );
    ptr[offset*4 + 0] = 255 * juliaValue;
    ptr[offset*4 + 1] = 0;
    ptr[offset*4 + 2] = 0;
    ptr[offset*4 + 3] = 255;
}

int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() ) );

    dim3 grid(DIM,DIM);
    kernel<<<grid,1>>>>( dev_bitmap );

    HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                               bitmap.image_size(),
                               cudaMemcpyDeviceToHost ) );
    bitmap.display_and_exit();

    HANDLE_ERROR( cudaFree( dev_bitmap ) );
}

```

当运行这个应用程序时，你将看到Julia集的动画演示。为了使你相信它确实称得上是“一个有趣的示例”，图4.2给出了这个应用程序的运行截屏。

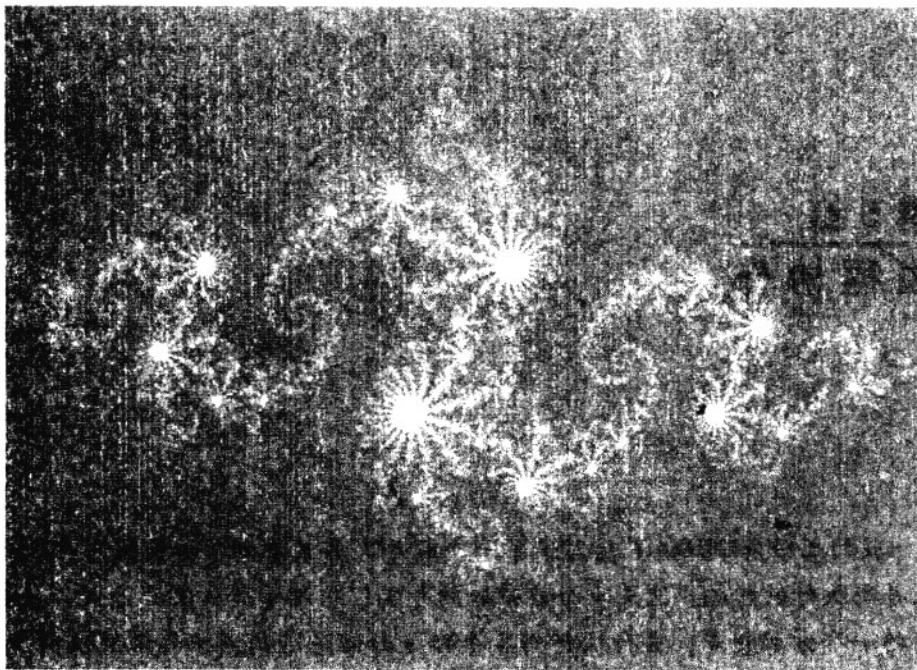


图4.2 基于GPU的Julia集应用程序的截屏

4.3 本章小结

恭喜你，你现在可以编写、编译并在图形处理器上运行大规模的并行代码了，你也可以向你的朋友吹嘘了。如果他们仍然认为GPU计算是非常复杂并且难以掌握的，那么他们肯定会被你的话打动。如果他们相信你的话，那么建议他们也购买这本书。

到目前为止，我们已经看到了如何告诉CUDA运行时在线程块上并行执行程序。我们把在GPU上启动的线程块集合称为一个线程格。从名字的含义可以看出，线程格既可以是一维的线程块集合，也可以是二维的线程块集合。核函数的每个副本都可以通过内置变量`blockIdx`来判断哪个线程块正在执行它。同样，它还可以通过内置变量`gridDim`来获得线程格的大小。这两个内置变量在核函数中都是非常有用的，可以用来计算每个线程块需要的数据索引。

第 5 章

线程协作

我们已经用CUDA C编写了第一个程序，并且看到了如何编写在GPU上并行执行的代码。这是一个非常好的开端！但对于并行编程来说，最重要的一个方面就是，并行执行的各个部分如何通过相互协作来解决问题。只有在极少数情况下，各个处理器才不需要了解其他处理器的执行状态而彼此独立地计算出结果。即使对于一些成熟的算法，也仍然需要在代码的各个并行副本之间进行通信和协作。到目前为止，我们还没有看到有任何机制可以在并行执行的CUDA C代码段之间实现这种通信。但是，在本章中我们将介绍这个问题的一种解决方案。

5.1 本章目标

通过本章的学习，你可以：

- 了解CUDA C中的线程。
- 了解不同线程之间的通信机制。
- 了解并行执行线程的同步机制。

5.2 并行线程块的分解

在第4章中，我们看到了如何在GPU上启动并行代码。具体的实现方法是，告诉CUDA运行时启动核函数的多个并行副本。我们将这些并行副本称为线程块（Block）。

CUDA运行时将把这些线程块分解为多个线程。当需要启动多个并行线程块时，只需将尖括号中的第一个参数由1改为想要启动的线程块数量。例如，在介绍矢量加法示例时，我们为矢量中的每个元素（共有N个元素）都启动了一个线程块，如下所示：

```
add<<<N,1>>>>( dev_a, dev_b, dev_c );
```

在尖括号中，第二个参数表示CUDA运行时在每个线程块中创建的线程数量。上面这行代码表示在每个线程块中只启动一个线程。总共启动的线程数量可按以下公式计算：

$$N \text{ 个线程块} \times 1 \text{ 个线程/线程块} = N \text{ 个并行线程}$$

事实上，我们也可以启动N/2个线程块，每个线程块包含2个线程，或者N/4个线程块，每个线程块包含4个线程，以此类推。接下来，我们将围绕着CUDA C的这种特性来重新回顾矢量相加示例。

5.2.1 矢量求和：重新回顾

在本节中将完成与第4章中相同的任务，即将两个输入矢量相加并将结果保存在第三个输出矢量中。然而，这一次我们将使用线程而不是线程块来实现。

你可能会奇怪，使用线程与使用线程块相比存在哪些优势？就目前而言，几乎没有任何优势。但线程块中的并行线程能够完成并行线程块无法完成的工作。因此，在分析接下来基于并行线程的矢量求和程序时，要保持耐心。

1. 使用线程实现GPU上的矢量求和

在将实现方式由并行线程块改为并行线程时，首先需要修改两个地方。第一个地方是，之前核函数的调用中是启动N个线程块，并且每个线程块对应一个线程，如下所示：

```
add<<<N,1>>>>( dev_a, dev_b, dev_c );
```

现在，我们将启动N个线程，并且所有线程都在一个线程块中：

```
add<<<1,N>>>>( dev_a, dev_b, dev_c );
```

第二个要修改的地方是对数据进行索引的方法。之前核函数中是通过线程块索引对输入数据和输出数据进行索引。

```
int tid = blockIdx.x;
```

你应该对上面这行代码很熟悉了。现在，由于只有一个线程块，因此需要通过线程索引来对数据进行索引。

```
int tid = threadIdx.x;
```

这就是将使用并行线程块修改为使用并行线程时需要进行的两处修改。下面给出了完整的源代码，其中有变化的代码行用粗体显示：

```
#include "../common/book.h"

#define N 10

__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    //在GPU上分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    //在CPU上为数组 'a' 和 'b' 赋值
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    //将数组 'a' 和 'b' 复制到GPU
    HANDLE_ERROR( cudaMemcpy( dev_a,
                                a,
```

```

        N * sizeof(int),
        cudaMemcpyHostToDevice );
HANDLE_ERROR( cudaMemcpy( dev_b,
        b,
        N * sizeof(int),
        cudaMemcpyHostToDevice ) );

add<<<1,N>>>( dev_a, dev_b, dev_c );

//将数组 'c' 从GPU复制到CPU
HANDLE_ERROR( cudaMemcpy( c,
        dev_c,
        N * sizeof(int),
        cudaMemcpyDeviceToHost ) );

//显示结果
for (int i=0; i<N; i++) {
    printf( "%d + %d = %d\n", a[i], b[i], c[i] );
}
//释放在GPU上分配的内存
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}

```

这段代码非常简单，对不对？在下一节中，我们将看到在这种只使用并行线程的方法中存在的局限。此外，我们还将看到为什么要将线程块分解为其他的并行部分。

2. 在GPU上对更长的矢量求和

在第4章中，我们注意到硬件将线程块的数量限制为不超过65 535。同样，对于启动核函数时每个线程块中的线程数量，硬件也进行了限制。具体来说，最大的线程数量不能超过设备属性结构中maxThreadsPerBlock域的值，我们在第3章中介绍过这个结构。对于当前的许多图形处理器而言，这个限制值是每个线程块512个线程。因此，如何通过并行线程对长度大于512的矢量进行相加？在这种情况下，我们必须将线程与线程块结合起来才能实现这个计算。

与前面一样，这里需要改动两个地方：核函数中的索引计算方法和核函数的调用方式。

现在，我们需要多个线程块并且每个线程块中包含了多个线程，计算索引的方法看上去非常类似于将二维索引空间转换为线性空间的标准算法。

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

在上面的赋值语句中使用了一个新的内置变量，`blockDim`。对于所有线程块来说，这个变量是一个常数，保存的是线程块中每一维的线程数量。由于使用的是一维线程块，因此只需用到`blockDim.x`。回顾第4章的内容，在`gridDim`中保存了一个类似的值，即在线程格中每一维的线程块数量。此外，`gridDim`是二维的，而`blockDim`实际上是三维的。也就是说，CUDA运行时允许启动一个二维线程格，并且线程格中的每个线程块都是一个三维的线程数组。没错，这是一种高维数组，虽然你很少会用到这种高维索引，但如果需要也可以支持。

通过前面的运算来对线性数组中的数据进行索引是非常直观的。它可以帮助你从空间上来思考线程集合，这类似于一个二维的像素数组。在图5.1中给出了这种空间组织形式。

线程块0	线程0	线程1	线程2	线程3
线程块1	线程0	线程1	线程2	线程3
线程块2	线程0	线程1	线程2	线程3
线程块3	线程0	线程1	线程2	线程3

图5.1 线程块集合与线程集合的二维组织形式

如果线程表示列，而线程块表示行，那么可以计算得到一个唯一的索引：将线程块索引与每个线程块中的线程数量相乘，然后加上线程在线程块中的索引。这与我们在Julia集中将二维图像索引线性化的方法是相同的。

```
int offset = x + y * DIM;
```

在这里，`DIM`表示线程块的大小（即线程的数量），`y`为线程块索引，并且`x`为线程块中的线程索引。因此可以计算得到以下索引：`tid = threadIdx.x + blockIdx.x * blockDim.x`。

另一处修改是核函数调用本身。虽然仍需要启动`N`个并行线程，但我们在多个线程块中启动它们，这样就不会超过512个线程的最大数量限制。其中一种解决方案是，将线程块的大小设置为某个固定数值，在这个示例中，我们每个线程块中包含的线程数量固定为128。然后，可以启动`N/128`个线程块，这样总共就启动了`N`个线程同时运行。

这里的问题在于，`N/128`是一个整数除法。例如，如果`N`为127，那么`N/128`等于0，因此将启动0个线程，而我们实际上也不会获得任何计算结果。事实上，当`N`不是128的整数倍时，启动的线程数量将少于预期数量，这种情况非常糟糕。事实上，我们希望这个除法能够向上取整。

我们可以通过一种常见的技术在整数除法中实现这个功能。不是调用`ceil()`，而是计算 $(N+127)/128$ 而不是 $N/128$ 。你也可以将这种算法理解为，计算大于或等于 N 的128的最小倍数。

我们选择了每个线程块拥有128个线程，因此使用以下的核函数调用：

```
add<<< (N+127)/128, 128 >>>( dev_a, dev_b, dev_c );
```

上面这行代码对除法进行了修改从而确保启动足够多的线程，因此当 N 不是128的整数倍时，将启动过多的线程。然而，在核函数中已经解决了这个问题。在访问输入数组和输出数组之前，必须检查线程的偏移是否位于0到 N 之间。

```
if (tid < N)
    c[tid] = a[tid] + b[tid];
```

因此，当索引越过数组的边界时，例如当启动的并行线程数量不是128的整数倍时就会出现这种情况，那么核函数将自动停止执行计算。更重要的是，核函数不会对越过数组边界的内存进行读取或者写入。

3. 在GPU上对任意长度的矢量求和

当第一次介绍在GPU上启动并行线程块时，我们并没有考虑所有可能的情况。除了在线程数量上存在限制外，在线程块的数量上同样存在着一个硬件限制（虽然这个限制值比线程数量的限制值更大）。在前面提到过，线程格每一维的大小都不能超过65 535。

因此，这就对矢量加法的实现带来了一个问题。如果启动 $N/128$ 个线程块将矢量相加，那么当矢量的长度超过 $65\,535 \times 128 = 8\,388\,480$ 时，核函数调用会失败。这看上去似乎是一个很大的数值，但在当前内存容量基本上在1GB到4GB的情况下，高端图形处理器能够容纳元素数量超过8百万的矢量。

幸运的是，这个问题的解决方案非常简单。我们首先对核函数进行一些修改。

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
        tid += blockDim.x * gridDim.x;
    }
}
```

这看上去非常像矢量加法的最初版本。我们将它与第4章中基于CPU的实现进行比较：

```
void add( int *a, int *b, int *c ) {
    int tid=;    //这是第0号CPU，因此索引从0开始
    while (tid < N) {
        c[tid] = a[tid] + b[tid];
```

```

        tid += 1;    //由于只有一个CPU，因此每次递增1
    }
}

```

在这里，我们使用了一个while()循环对数据进行迭代。之前提到过，在多CPU或者多核版本中，每次递增的数量不是1，而是CPU的数量。现在，我们在GPU版本中将使用同样的方法。

在GPU实现中，我们将并行线程的数量看成是处理器的数量。尽管GPU的处理单元数量可能小于（或者大于）这个值，但我们认为每个线程在逻辑上都可以并行执行，并且硬件可以调度这些线程以便实际执行。通过将并行化过程与硬件的实际执行过程解耦开来，这是CUDA C为软件开发人员减轻的负担之一。这种解耦起到了极大的帮助作用，特别是考虑到当前NVIDIA硬件在每个芯片上可能包含了从8个到480个的数学单元！

在理解了上述实现方式所包含的核心思想后，我们只需知道如何计算每个并行线程的初始索引，以及如何确定递增的量值。我们希望每个并行线程从不同的索引开始，因此就需要对线程索引和线程块索引进行线性化，正如在5.2.1节中看到的。每个线程的起始索引按照以下公式来计算：

```
int tid = threadIdx.x + blockIdx.x * blockDim.x;
```

在每个线程计算完当前索引上的任务后，接着就需要对索引进行递增，其中递增的步长为线程格中正在运行的线程数量。这个数值等于每个线程块中的线程数量乘以线程格中线程块的数量，即blockDim.x * gridDim.x。因此，递增语句如下所示：

```
tid += blockDim.x * gridDim.x;
```

我们几乎要完成了！剩下的事情就是修改核函数调用本身。前面提到过，之所以采用这种方式，是因为当 $(N+127)/128$ 大于65 535时，核函数调用`add<<<(N+127)/128,128>>>( dev_a, dev_b, dev_c )`会失败。为了确保不会启动过多的线程块，我们将线程块的数量固定为某个较小的值。由于我们非常喜欢直接复制和粘贴代码，因此在这里同样使用128个线程块，并且每个线程块包含128个线程。

```
add<<<128,128>>>( dev_a, dev_b, dev_c );
```

你也可以将这些值调整为你认为合适的任意值，只要你指定的值仍处于前面给出的限制范围之内。在本书的后面部分，我们将讨论不同的值对性能产生的影响，但就目前来说，选择128个线程块并且每个线程块包含128个线程就足够了。现在，我们将把任意长度的矢量相加起来，矢量长度的限制只取决于GPU上的内存容量。以下是完整的源代码清单：

```

#include "../common/book.h"

#define N (33 * 1024)
__global__ void add( int *a, int *b, int *c ) {

```

```

int tid = threadIdx.x + blockIdx.x * blockDim.x;
while (tid < N) {
    c[tid] = a[tid] + b[tid];
    tid += blockDim.x * gridDim.x;
}
}

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    //在GPU上分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    //在CPU上为数组 'a' 和 'b' 赋值
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }

    //将数组 'a' 和 'b' 复制到GPU
    HANDLE_ERROR( cudaMemcpy( dev_a,
                               a,
                               N * sizeof(int),
                               cudaMemcpyHostToDevice ) );
    HANDLE_ERROR( cudaMemcpy( dev_b,
                               b,
                               N * sizeof(int),
                               cudaMemcpyHostToDevice ) );

    add<<<128,128>>>>( dev_a, dev_b, dev_c );

    //将数组 'c' 从GPU复制到CPU
    HANDLE_ERROR( cudaMemcpy( c,
                               dev_c,
                               N * sizeof(int),
                               cudaMemcpyDeviceToHost ) );

    // 验证GPU确实完成了我们要求的工作
    bool success = true;
    for (int i=0; i<N; i++) {
        if ((a[i] + b[i]) != c[i]) {
            printf( "Error: %d + %d != %d\n", a[i], b[i], c[i] );
            success = false;
        }
    }
}

```

```

    }
}

if (success) printf( "We did it!\n" );

//释放在GPU上分配的内存
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

return 0;
}

```

5.2.2 在GPU上使用线程实现波纹效果

在第4章中，我们通过一个有趣的示例来说明已经学习到的各项技术。在本章中，我们同样将通过GPU的强大计算能力来生成一些图片。为了实现更有趣的演示效果，这一次我们将实现动画显示。当然，我们将把所有不相关的动画代码都封装到辅助函数中，这样你就无需预先掌握任何图形技术或者动画技术。

```

struct DataBlock {
    unsigned char    *dev_bitmap;
    CPUAnimBitmap    *bitmap;
};

//释放在GPU上分配的内存
void cleanup( DataBlock *d ) {
    cudaFree( d->dev_bitmap );
}

int main( void ) {
    DataBlock    data;
    CPUAnimBitmap    bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_bitmap,
                               bitmap.image_size() ) );

    bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
                          (void (*)(void*))cleanup );
}

```

main()函数的大部分复杂性都被隐藏在辅助类CPUAnimBitmap中。你将注意到，我们再次遵循了前面采用的模式，即首先调用cudaMalloc()，然后执行设备代码对内存内容进行计算，最后调用cudaFree()来释放这些内存。这些步骤对你来说应该已经很熟悉了。

在这个示例中，我们将把“执行设备代码对内存内容进行计算”这个步骤变得更复杂一些。我们将一个指向generate_frame()的函数指针传递给anim_and_exit()。每当要生成一帧新的动画时，都将调用函数generate_frame()。

```
void generate_frame( DataBlock *d, int ticks ) {
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<blocks,threads>>>( d->dev_bitmap, ticks );

    HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr(),
                               d->dev_bitmap,
                               d->bitmap->image_size(),
                               cudaMemcpyDeviceToHost ) );
}
```

尽管这个函数只包含了4行代码，但其中却包含CUDA C的所有重要概念。首先，代码声明了两个二维变量，blocks和threads。从名字就可以很清楚地看到，变量blocks表示在线程格中包含的并行线程块数量。变量threads表示在每个线程块中包含的线程数量。由于生成的是一幅图像，因此使用了二维索引，并且每个线程都有一个唯一的索引 (x,y)，这样可以很容易与输出图像中的像素一一对应起来。在我们选择的线程块中包含了一个大小为 16×16 的线程数组。如果图像有 $DIM \times DIM$ 个像素，那么就需要启动 $DIM/16 \times DIM/16$ 个线程块，从而使每个像素对应一个线程。在图5.2中给出了在一个（非常小的）48像素宽和32像素高的图像中的线程块与线程的配置情况。

如果你曾经编写过多线程的CPU程序，那么可能会奇怪为什么要启动这么多的线程。例如，如果绘制一个 1920×1080 的动画，那么这种方法将创建超过2百万个线程。虽然我们可以在GPU上创建和调度这么多的线程，但任何人都难以想象在CPU上创建这么多的线程。由于CPU的线程管理和调度必须在软件中完成，因此无法像GPU那样能支持数量如此多的线程。由于我们为每个需要处理的元素都创建了一个线程，因此在GPU上的并行编程就比在CPU上要简单得多。

在声明了表示线程块数量和线程数量的变量后，接下来就需要调用核函数来计算像素值。

```
kernel<<< blocks,threads>>>( d->dev_bitmap, ticks );
```

核函数需要包含两个参数。首先，它需要一个指针来指向保存输出像素值的设备内存。这是一个全局变量，它指向的内存是在main()中分配的。然而，这个变量只是对于主机代码来说是“全局的”，因此我们需要将其作为一个参数传递进去，从而确保设备代码能够访问这个变量。

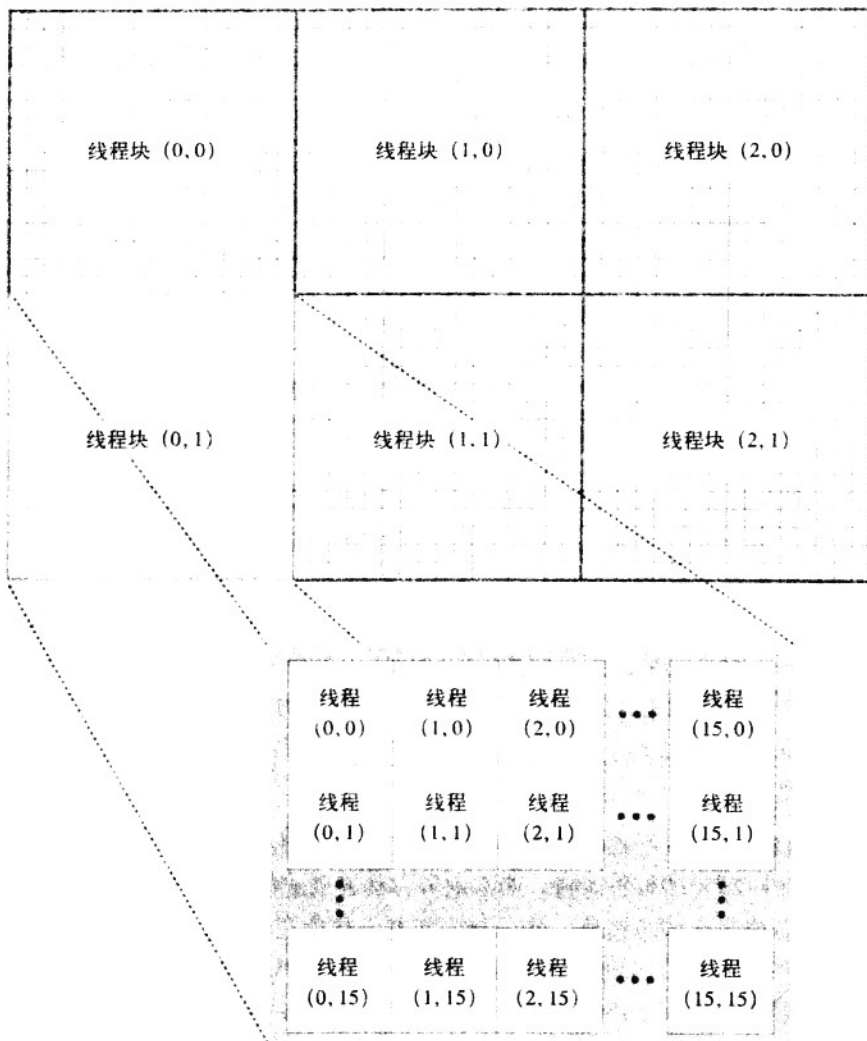


图5.2 线程块与线程的二维层次结构图，用于处理一个 48×32 像素的图像，其中每个像素对应一个线程

其次，核函数需要知道当前的动画时间以便生成正确的帧。在CPUAnimBitmap的代码中将当前时间ticks传递给generate_frame()，因此只需直接将这个值传递给核函数。

核函数的代码如下所示：

```
__global__ void kernel( unsigned char *ptr, int ticks ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
```

```

int offset = x + y * blockDim.x * gridDim.x;

// 现在计算这个位置上的值
float fx = x - DIM/2;
float fy = y - DIM/2;
float d = sqrtf( fx * fx + fy * fy );
unsigned char grey = (unsigned char)(128.0f + 127.0f *
                                   cos(d/10.0f - ticks/7.0f) /
                                   (d/10.0f + 1.0f));

ptr[offset*4 + 0] = grey;
ptr[offset*4 + 1] = grey;
ptr[offset*4 + 2] = grey;
ptr[offset*4 + 3] = 255;
}

```

前三行代码是核函数中最重要的代码。

```

int x = threadIdx.x + blockIdx.x * blockDim.x;
int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;

```

在这几行代码中，每个线程都得到了它在线程块中的索引以及这个线程块在线程格中的索引，并将这两个值转换为图形中的唯一索引 (x, y)。例如，当位于线程格中 (12, 8) 处的线程块中的 (3, 5) 处的线程开始执行时，它知道在其左边有12个线程块，而在它上边有8个线程块。在这个线程块内，(3, 5) 处线程的左边有3个线程，并且在它上边有5个线程。由于每个线程块都有16个线程，这就意味着这个线程有：

3线程+12线程块×16线程/线程块=195个线程在其左边

5线程+8线程块×16线程/线程块=133个线程在其上边

这个计算就是计算前两行中的x和y，也是将线程和线程块的索引映射到图像坐标的算法。然后，我们对x和y的值进行线性化从而得到输出缓冲区中的一个偏移。这与5.2.1节的第2个标题以及第3个标题中的工作是相同的。

```

int offset = x + y * blockDim.x * gridDim.x;

```

既然我们知道了线程要计算的像素值 (x, y)，以及在何时计算这个值，那么就可以计算 (x, y, t) 的任意函数，并将这个值保存在输出缓冲区中。在这里的情况中，函数将生成一个随时间变化的正弦曲线“波纹”。

```

float fx = x - DIM/2;
float fy = y - DIM/2;
float d = sqrtf( fx * fx + fy * fy );
unsigned char grey = (unsigned char)(128.0f + 127.0f *

```

```
cos(d/10.0f - ticks/7.0f) /  
(d/10.0f + 1.0f));
```

我们建议你不要花太多的时间去理解grey值的计算。这其实只是一个2维时间函数，当动画显示时，将会形成一个漂亮的波纹效果。其中一帧的截屏如图5.3所示。

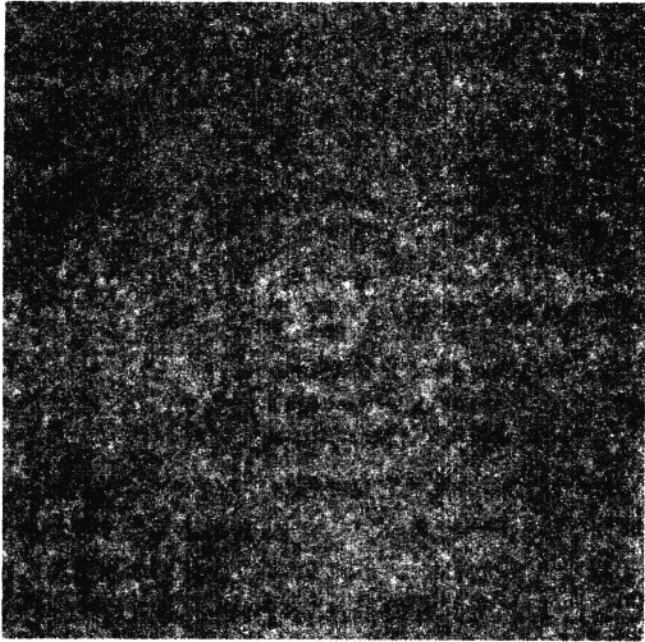


图5.3 GPU波纹示例的截图

5.3 共享内存和同步

到目前为止，将线程块分解为线程的目的只是为了解决线程块数量的硬件限制。这是一个很勉强的动机，因为这完全可以由CUDA运行时在幕后自动完成。显然，还有其他的原因需要将线程块分解为多个线程。

CUDA C支持共享内存。这块内存引出了对于C语言的另一个扩展，这个扩展类似于__device__和__global__。在编写代码时，你可以将CUDA C的关键字__share__添加到变量声明中，这将使这个变量驻留在共享内存中。那么，这么做的目的是什么？

很高兴你能问这个问题。CUDA C编译器对共享内存中的变量与普通变量将分别采取不同的处理方式。对于在GPU上启动的每个线程块，CUDA C编译器都将创建该变量的一个副本。线程块中的每个线程都共享这块内存，但线程却无法看到也不能修改其他线程块的变量副本。这就实现了一种非常好的方式，使得一个线程块中的多个线程能够在计算上进行通信和协作。

而且，共享内存缓冲区驻留在物理GPU上，而不是驻留在GPU之外的系统内存中。因此，在访问共享内存时的延迟要远远低于访问普通缓冲区的延迟，使得共享内存像每个线程块的高速缓存或者中间结果暂存器那样高效。

线程之间相互通信的功能或许已经令你很兴奋了，这个功能同样也使我们感到兴奋。但世上没有免费的东西，线程间的通信也不例外。如果想要在线程之间进行通信，那么还需要一种机制来实现线程之间的同步。例如，如果线程A将一个值写入到共享内存，并且我们希望线程B对这个值进行一些操作，那么只有当线程A的写入操作完成之后，线程B才能开始执行它的操作。如果没有同步，那么将发生竞态条件（Race Condition），在这种情况下，代码执行结果的正确性将取决于硬件的不确定性。

我们来看一个使用这些功能的示例。

5.3.1 点积运算

恭喜你！我们已经学习完了矢量的加法运算，接下来将介绍矢量的点积运算（Dot Product，也称为内积）。我们将快速介绍点积运算的过程，以免你不熟悉这种矢量运算（或者还是早在几年前学过）。这个计算包含两个步骤。首先，将两个输入矢量中相应的元素相乘。这非常类似于矢量的加法运算，只不过在这里使用的是乘法而不是加法。然而，在计算完这个步骤后，不是将值保存到第三个输出矢量中，而是将这些值相加起来以得到一个标量输出值。

例如，如果对两个包含4个元素的矢量进行点积运算，那么计算公示如5.1所示。

等式5.1

$$(x_1, x_2, x_3, x_4) \cdot (y_1, y_2, y_3, y_4) = x_1y_1 + x_2y_2 + x_3y_3 + x_4y_4$$

你或许已经理解了我们将要使用的算法。首先，可以像矢量加法那样，每个线程将两个相应的元素相乘，然后移动到下一个元素。由于最终的结果是所有乘积的总和，因此每个线程还要保存它所计算的乘积的加和。与矢量加法示例一样，线程每次对索引的增加值为线程的数量，这样能确保不会遗漏任何元素，而且也不会将任何元素相乘两次。下面的代码实现了点积函数的第一个步骤：

```
#include "../common/book.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;
const int threadsPerBlock = 256;

__global__ void dot( float *a, float *b, float *c ) {
```

```

__shared__ float cache[threadsPerBlock];
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;
float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}

// 设置cache中相应位置上的值
cache[cacheIndex] = temp;

```

在代码中声明了一个共享内存缓冲区，名字为cache。这个缓冲区将保存每个线程计算的加和值。我们稍后会看到为什么要这么做，但现在只分析在实现第一个步骤时采用的方法。要声明一个驻留在共享内存中的变量是很容易的，这相当于在标准C中声明一个static或volatile类型的变量。

```
__shared__ float cache[threadsPerBlock];
```

我们将数组的大小声明为threadsPerBlock，这样线程块中的每个线程都能将它计算的临时结果保存到某个位置上。之前在分配全局内存时，我们为每个执行核函数的线程都分配了足够的内存，即线程块的数量乘以threadPerBlock。但对于共享变量来说，由于编译器将为每个线程块生成共享变量的一个副本，因此只需根据线程块中线程的数量来分配内存。

在分配了共享内存后，像前面一样计算数据索引：

```

int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;

```

现在，你应该对变量tid的计算很熟悉了，通过线程块索引和线程索引计算出输入数组中的一个全局偏移。共享内存缓存中的偏移就等于线程索引。线程块索引与这个偏移无关，因为每个线程块都拥有该共享内存的私有副本。

最后，我们设置了共享内存缓冲区相应位置上的值，以便随后能将整个数组相加起来，并且无需担心某个特定的位置上是否包含有效的数据：

```

// 设置cache中相应位置上的值
cache[cacheIndex] = temp;

```

如果输入矢量的长度不是线程块中线程数量的整数倍，那么cache中的元素将不会被全部用到。在这种情况下，最后一个线程块中将有一些线程不做任何事情。

每个线程都计算数组a和b中相应元素乘积的总和，然后当到达数组末尾时，再将临时加和值保存到共享缓冲区中。

```
float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}

// 设置cache中相应位置上的值
cache[cacheIndex] = temp;
```

当算法执行到这一步时，我们需要将cache中所有的值相加起来。在执行这个运算时，需要通过一个线程来读取保存在cache中的值。然而，在前面已经提到过，这可能是一种危险的操作。我们需要某种方法来确保所有对共享数组cache[]的写入操作在读取cache之前完成。幸运的是，这种方法确实存在：

```
// 对线程块中的线程进行同步
__syncthreads();
```

这个函数调用将确保线程块中的每个线程都执行完__syncthreads()前面的语句后，才会执行下一条语句。这正是我们需要的功能。现在，我们知道，当一个线程执行__syncthreads()后面的第一条语句时，线程块中的其他线程肯定都已经执行完了__syncthreads()。

这时，我们可以确信cache已经填好了，因此可以将其中的值相加起来。这个相加过程可以抽象为更一般的形式：对一个输入数组执行某种计算，然后产生一个更小的结果数组，这种过程也称为归约（Reduction）。在并行计算中经常会遇到类似的运算，因此人们专门为这类算法起了一个名字。

实现归约运算的最简单方法是，由一个线程在共享内存上进行迭代并计算出总和值。计算的时间与数组的长度成正比。然而，在这里的示例中有数百个线程可用，因此我们可以以并行方式来执行归约运算，这样所花的时间将与数组长度的对数成正比。初看上去，下面的代码有些费解，我们将在随后对其进行分析。

代码的基本思想是，每个线程将cache[]中的两个值相加起来，然后将结果保存回cache[]。由于每个线程都将两个值合并为一个值，那么在完成这个步骤后，得到的结果数量就是计算开始时数值数量的一半。在下一个步骤中，我们对这一半数值执行相同的操作。在将这种操作执行 $\log_2(\text{threadsPerBlock})$ 个步骤后，就能得到cache[]中所有值的总和。对这里的示例来说，我们在每个线程块中使用了256个线程，因此需要8次迭代将cache[]中的256个值归约为1个值。

实现这个归约运算的代码如下所示：

```
// 对于归约运算来说，以下代码要求threadPerBlock必须是2的指数
int i = blockDim.x/2;
while (i != 0) {
```

```

    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

```

在第一个步骤中，我们取threadsPerBlock的一半作为i值，只有索引小于这个值的线程才会执行。只有当线程的索引小于i时，才可以把cache[]的两个数据项相加起来，因此我们将加法运算放在if(cacheIndex < i)的代码块中。执行加法运算的线程将cache[]中线程索引位置上的值和线程索引加上i得到的位置上的值相加起来，并将结果保存回cache[]中线程索引位置上。

假设在cache[]中有8个元素，因此i的值为4。归约运算的其中一个步骤如图5.4所示。

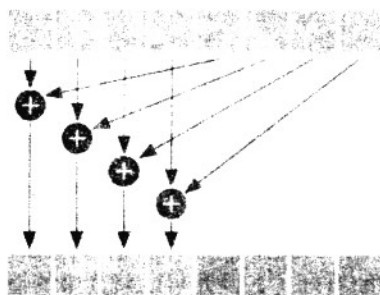


图5.4 求和归约运算中的一个步骤

在完成了一个步骤后，将出现在计算完两两元素乘积后相同的问题。在读取cache[]中的值之前，首先需要确保每个写入cache[]的线程都已经执行完毕。因此，在赋值运算后面增加了__syncthreads调用以确保满足这个条件。

在结束了while()循环后，每个线程块都得到了一个值。这个值位于cache[]的第一个元素中，并且就等于该线程块中两两元素乘积的加和。然后，我们将这个值保存到全局内存并结束核函数：

```

    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}

```

为什么只有cacheIndex==0的线程执行这个保存操作？这是因为只有一个值写入到全局内存，因此只需要一个线程来执行这个操作。当然，每个线程都可以执行这个写入操作，但这么做将使得在写入单个值时带来不必要的内存通信量。为了简单，我们选择了索引为0的线程，当然你也可以选择任何一个线程将cache[0]写入到全局内存。最后，由于每个线程块都只写入一个值到全局数据c[]中，因此可以通过blockIdx来索引这个值。

我们得到了一个数组c[]，其中该数组的每个元素中都包含了某个线程块计算得到的加和值。

点积运算的最后一个步骤就是计算c[]中所有元素的总和。此时，尽管点积运算还没有完全计算完毕，但我们却退出核函数并将执行控制返回到主机。那么，为什么要在尚未计算完成之前就返回到主机？

我们在前面将点积运算称为一种归约运算。大体来说，这是因为最终生成的输出数据数量要小于输入数据的数量。在点积运算中，无论输入的数量有多少，通常只会生成一个输出结果。事实证明，像GPU这种大规模的并行机器在执行最后的归约步骤时，通常会浪费计算资源，因为此时的数据集往往非常小。例如，当使用480个数学处理单元将32个数值相加时，将难以充分使用每一个数学处理单元。

因此，我们将执行控制返回给主机，并且由CPU来完成最后一个加法步骤，即将计算数组c[]中所有元素的总和。如果在一个更大的应用程序中，此时的GPU就可以开始执行其他的点积运算或者其他的大规模计算。然而，在这个示例中，对GPU的使用将到此为止。

在分析这个示例时，我们改变了之前的讲解方式，直接从核函数计算开始分析。接下来将给出main()函数及其对核函数的调用，这与我们在前面给出的代码非常相似。

```
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

int main( void ) {
    float  *a, *b, c, *partial_c;
    float  *dev_a, *dev_b, *dev_partial_c;
    // 在CPU上分配内存
    a = new float[N];
    b = new float[N];
    partial_c = new float[blocksPerGrid];

    // 在GPU上分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                               N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                               N*sizeof(float) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                               blocksPerGrid*sizeof(float) ) );

    // 填充主机内存
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i*2;
    }
}
```

```
// 将数组 'a' 和 'b' 复制到GPU上
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                          cudaMemcpyHostToDevice ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a,
                                          dev_b,
                                          dev_partial_c );
```

我们来简单总结一下这段代码：

- 1) 为输入数组和输出数组分配主机内存和设备内存。
- 2) 填充输入数组a[]和b[]，然后通过cudaMemcpy()将它们复制到设备上。
- 3) 在调用计算点积的核函数时指定线程格中线程块的数量以及每个线程块中的线程数量。

尽管大部分代码对你来说已经很熟悉了，但线程块数量的计算过程还值得进一步分析。我们介绍了点积运算是一种归约运算，以及每个线程块如何计算临时和值。对于CPU来说，虽然这段求和的代码非常短，但却足以生成足够多的线程块使GPU始终处于忙碌状态。我们选择了32个线程块，选择其他的值可能会产生更好或者更差的性能，这要取决于CPU和GPU的相对速度。

然而，对于一段非常短的代码选择32个线程块并且每个线程块包含256个线程，那么是否会造成线程过多的情况？如果有N个数据元素，那么通常只需要N个线程来计算点积。但在这样的情况中，线程数量应该为threadsPerBlock的最小整数倍，并且要大于或者等于N。在前面的矢量加法示例中遇到过这种情况。在这里采用的计算公式为 $(N + (\text{threadsPerBlock} - 1)) / \text{threadsPerBlock}$ 。你可能会说，这实际上是整数运算中一种很常见的方法，因此即使你的大部分时间都是放在与CUDA C相关的问题上，但仍然需要理解这种方法。

因此，我们启动的线程块的数量或者是32个，或者是 $(N + (\text{threadsPerBlock} - 1)) / \text{threadsPerBlock}$ ，选择二者中较小的。

```
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

现在，你应该很清楚main()中的代码了。在核函数执行完毕后，仍然需要通过求和操作计算出结果。但正如在启动核函数之前将输入数组复制到GPU一样，我们需要将输出数组从GPU复制回CPU，然后再对其进行计算。因此，在核函数执行完毕后，我们将保存临时和值的数组复制回来，并在CPU上完成最终的求和运算。

```
// 将数组 'c' 从GPU复制回CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
```

```

        blocksPerGrid*sizeof(float),
        cudaMemcpyDeviceToHost ) );

// 在CPU上完成最终的求和运算
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

```

最后，我们检查是否得到了正确的结果，并释放了在CPU和GPU上分配的内存。检查结果的过程很容易，因为我们有规律地将数据填充进输入数组，其中a[]中的值是从0到N-1的整数，而b[]则是2*a[]。

```

// 填充主机内存
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

点积计算结果应该是从0到N-1中每个数值的平方再乘以2。对于喜欢离散数学的读者来说，求解这个求和问题的闭合形式解（Closed-Form Solution）是一个有意思的消遣活动。对于那些缺乏耐心或者不感兴趣的读者，我们在下面直接给出了闭合形式解，以及main()函数的剩余部分。

```

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );

// 释放GPU上的内存
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_partial_c );

// 释放CPU上的内存
delete [] a;
delete [] b;
delete [] partial_c;
}

```

下面给出了完整的源代码清单：

```

#include "../common/book.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024;

```

```

const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;

    float temp = 0;
    while (tid < N) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }

    // 设置cache中相应位置上的值
    cache[cacheIndex] = temp;

    // 对线程块中的线程进行同步
    __syncthreads();

    // 对于归约运算来说，以下代码要求threadPerBlock必须是2的指数
    int i = blockDim.x/2;
    while (i != 0) {
        if (cacheIndex < i)
            cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
        i /= 2;
    }
    if (cacheIndex == 0)
        c[blockIdx.x] = cache[0];
}

int main( void ) {
    float *a, *b, c, *partial_c;
    float *dev_a, *dev_b, *dev_partial_c;

    // 在CPU上分配内存
    a = (float*)malloc( N*sizeof(float) );
    b = (float*)malloc( N*sizeof(float) );
    partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

    //在GPU上分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                               N*sizeof(float) ) );

```

```

HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                           blocksPerGrid*sizeof(float) ) );

```

// 填充主机内存

```

for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

//将数组 'a' 和 'b' 复制到GPU

```

HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
dot<<<blocksPerGrid,threadsPerBlock>>>( dev_a, dev_b,
                                           dev_partial_c );

```

//将数组 'c' 从GPU复制到CPU

```

HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                           blocksPerGrid*sizeof(float),
                           cudaMemcpyDeviceToHost ) );

```

// 在CPU上完成最终的求和运算

```

c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

```

```

#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );

```

// 释放GPU上的内存

```

cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_partial_c );

```

// 释放CPU上的内存

```

free( a );
free( b );
free( partial_c );

```

```

}

```

5.3.2 (不正确的) 点积运算优化

在前面曾简要地介绍了点积运算示例中的第二个__syncthreads()。现在，我们来进一步分析这个函数调用，并尝试对其进行改进。之所以需要第二个__syncthreads()，是因为在循环迭代中更新了共享内存变量cache[]，并且在循环的下次迭代开始之前，需要确保当前迭代中所有线程的更新操作都已经完成。

```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}
```

在上面的代码中，我们发现只有当cacheIndex小于i时才需要更新共享内存缓冲区cache[]。由于cacheIndex实际上就等于threadIdx.x，因而这意味着只有一部分的线程会更新共享内存缓存。由于调用__syncthreads的目的只是为了确保这些更新在下次迭代之前已经完成，因此如果将代码修改为只等待那些需要写入共享内存的线程，是不是就能获得性能提升？我们通过将同步调用移动到if()线程块中来实现这个想法：

```
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i) {
        cache[cacheIndex] += cache[cacheIndex + i];
        __syncthreads();
    }
    i /= 2;
}
```

虽然这种优化代码的初衷不错，但却不起作用。事实上，这种情况比优化之前的情况还要糟糕。在对核函数进行修改后，GPU将停止响应，而你也不得不强行终止程序。但为什么这个看似无害的修改会导致如此灾难性的错误。

要回答这个问题，我们可以这样想象：线程块中的每个线程依次通过代码，每次一行。每个线程都执行相同的指令，但对不同的数据进行运算。然而，当每个线程执行的指令位于一个条件语句中，例如if()，那么将出现什么情况。显然，并不是每个线程都会执行这个指令，对不对？例如，考虑一个包含以下代码段的核函数，这段代码试图让奇数索引的线程更新某个变量的值：

```
int myVar = 0;
if( threadIdx.x % 2 )
    myVar = threadIdx.x;
```

在前一个示例中，当线程到达粗体的代码行时，只有奇数索引的线程才会执行它，因为偶数索引的线程将无法通过条件`if(threadIdx.x % 2)`。当奇数索引的线程执行这条指令时，偶数索引的线程就不会执行任何操作。当某些线程需要执行一条指令，而其他线程不需要执行时，这种情况就称为线程发散（Thread Divergence）。在正常的环境中，发散的分支只会使得某些线程处于空闲状态，而其他线程将执行分支中的代码。

但在`__syncthreads()`情况中，线程发散造成的结果有些糟糕。CUDA架构将确保，除非线程块中的每个线程都执行了`__syncthreads()`，否则没有任何线程能执行`__syncthreads()`之后的指令。遗憾的是，如果`__syncthreads()`位于发散分支中，那么一些线程将永远都无法执行`__syncthreads()`。因此，由于要确保在每个线程执行完`__syncthreads()`后才能执行后面的语句，因此硬件将使这些线程保持等待。一直等，一直等，永久地等待下去。

因此，如果在点积运算示例中将`__syncthreads()`调用移入`if()`线程块中，那么任何`cacheIndex`大于或等于`i`的线程将永远都不能执行`__syncthreads()`。这将使处理器挂起，因为GPU在等待某个永远都不会发生的事件。

```
if (cacheIndex < i) {
    cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
}
```

上述内容是为了说明，虽然`__syncthreads()`是一种强大的机制，它能确保大规模并行应用程序计算出正确的结果，但由于可能会出现意想不到的结果，我们在使用它时仍然要小心谨慎。

5.3.3 基于共享内存的位图

我们已经看到了可以使用共享内存和`__syncthreads`来确保数据在继续进行之前就已经就绪。为了提高执行速度，你可能会冒险去掉对`__syncthreads()`的调用。现在我们就来看一个只有使用`__syncthreads()`才能保证正确性的图形示例。我们将给出预计的输出结果，以及在没有`__syncthreads()`时的输出结果。事实证明，错误的结果将是不漂亮的。

`main()`函数与基于GPU的Julia集示例类似，不过这一次我们在每个线程块中启动多个线程：

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 1024
#define PI 3.1415926535897932f

int main( void ) {
```

```

CPUBitmap bitmap( DIM, DIM );
unsigned char *dev_bitmap;

HANDLE_ERROR( cudaMalloc( (void*)&dev_bitmap,
                          bitmap.image_size() ) );

dim3    grids(DIM/16,DIM/16);
dim3    threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                          bitmap.image_size(),
                          cudaMemcpyDeviceToHost ) );

bitmap.display_and_exit();

cudaFree( dev_bitmap );
}

```

与Julia集示例一样，每个线程都将为一个输出位置计算像素值。每个线程要做的第一件事就是计算输出影像中相应的位置 x 和 y 。这种计算类似于矢量加法示例中对 tid 的计算，只是这次计算的是一个二维空间。

```

__global__ void kernel( unsigned char *ptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

```

由于将使用一个共享内存缓冲区来保存计算结果，我们声明了一个缓冲区，在 16×16 的线程块中的每个线程在该缓冲区中都有一个对应的位置。

```

__shared__ float    shared[16][16];

```

然后，每个线程都会计算出一个值，并将其保存到缓冲区中。

```

// 现在计算这个位置上的值
const float period = 128.0f;

shared[threadIdx.x][threadIdx.y] =
    255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
        (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;

```

最后，我们将把这些值保存回像素，保留 x 和 y 的次序：

```

ptr[offset*4 + 0] = 0;
ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];
ptr[offset*4 + 2] = 0;

```



```
ptr[offset*4 + 3] = 255;
}
```

当然，这些计算有些随意，我们只是绘制了一个由多个绿色球形构成的网格。在编译并运行这个核函数后，将输出图5.5中的图像。

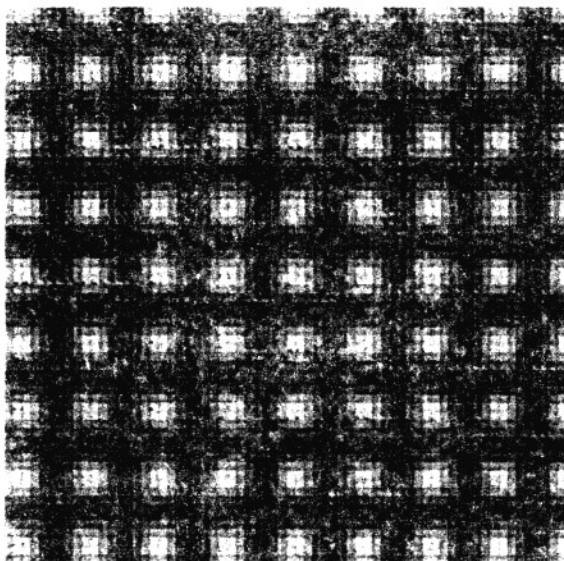


图5.5 在不正确同步情况下得到的截屏

这里发生了什么问题？从我们设计这个示例的初衷，你可能已经猜到了原因，在代码中忽略了一个重要的同步点。当线程将shared[][]中的计算结果保存到像素中时，负责写入到shared[][]的线程可能还没有完成写入操作。确保这种问题不会出现的方法就是使用__syncthreads()。没有使用__syncthreads()的结果就是得到一张被破坏的图片，充满了绿色的球形。

虽然在这个程序中并没有造成严重后果，但如果应用程序的计算非常重要，那么造成的破坏将是巨大的。

要解决这个问题，我们需要在写入共享内存与读取共享内存之间添加一个同步点。

```
shared[threadIdx.x][threadIdx.y] =
    255 * (sinf(x*2.0f*PI/ period) + 1.0f) *
    (sinf(y*2.0f*PI/ period) + 1.0f) / 4.0f;

__syncthreads();

ptr[offset*4 + 0] = 0;
```

```
ptr[offset*4 + 1] = shared[15-threadIdx.x][15-threadIdx.y];  
ptr[offset*4 + 2] = 0;  
ptr[offset*4 + 3] = 255;  
}
```

在添加__syncthreads()调用后，我们就可以获得一个预期的结果（并且从审美的角度来看也是更漂亮的），如图5.6所示。

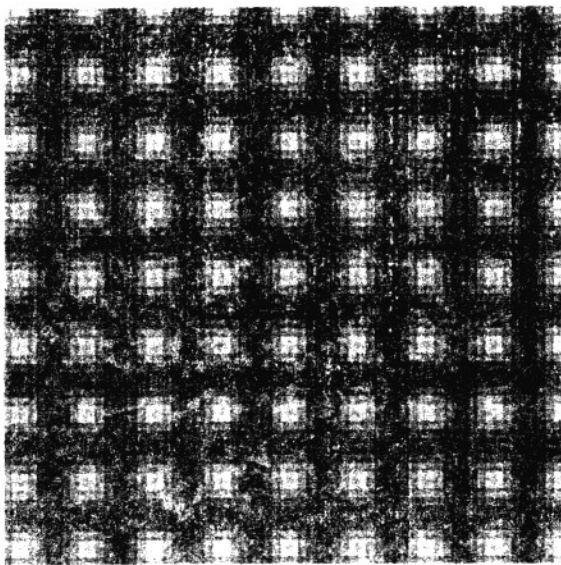


图5.6 在添加了正确同步之后的截屏

5.4 本章小结

我们知道了如何将线程块进一步分解为更小的并行执行单元，这种并行执行单元也称为线程。我们回顾了第4章中的矢量相加示例，并介绍了如何实现任意长度矢量的加法。我们还给出了一个归约运算的示例，以及如何通过共享内存和同步来实现这个运算。这个示例说明了如何对GPU与CPU进行协作从而完成计算。最后，我们还给出了当忽略同步时给应用程序造成的问题。

你已经学习了CUDA C的大部分基本概念，它与标准C在某些方面是相同的，但在大多数方面是不同的。此时，你可以考虑你曾经遇到的那些问题，以及哪些问题可能需要通过CUDA C的并行实现来解决。随着后面的进一步深入学习，我们将看到CUDA C的更多功能以及在GPU上实现各种任务，此外还将学习到CUDA提供的一些更高级API。

第 6 章

常量内存与事件

我们相信你已经学会了如何编写在GPU上执行的代码，也应该知道了如何生成并行线程块来执行核函数，并且还知道如何将这些线程块分解为并行线程。此外，你还看到了如何在这些线程之间进行通信与同步。但这本书的内容并未就此结束，你可能已经猜到了，CUDA C还有着更多有用的功能。

本章将介绍其中一些更高级的功能。具体来说，就是通过GPU上特殊的内存区域来加速应用程序的执行。我们将讨论其中一种内存区域：常量内存（Constant Memory）。此外，我们还将介绍一种增强CUDA C应用程序性能的方法，在这个过程中你将学习到如何通过事件来测量CUDA应用程序的性能。通过这些测量方法，你可以定量地分析对应用程序的某个修改是否会带来性能提升（或者性能下降）。

6.1 本章目标

通过本章的学习，你可以：

- 了解如何在CUDA C中使用常量内存。
- 了解常量内存的性能特性。
- 学习如何使用CUDA事件来测量应用程序的性能。

6.2 常量内存

之前，我们已经介绍了GPU中包含的强大数学处理能力。事实上，正是这种强大的计算优势激发了人们开始研究如何在图形处理器上执行通用计算。由于在GPU上包含有数百个数学计算单元，因此性能瓶颈通常并不在于芯片的数学计算吞吐量，而是在于芯片的内存带宽。由于在图形处理器上包含了非常多的数学逻辑单元（ALU），因此有时输入数据的速率甚至无法维持如此高的计算速率。因此，有必要研究一些手段来减少计算问题时的内存通信量。

到目前为止，我们已经看到了CUDA C程序中可以使用全局内存和共享内存。但是，CUDA C还支持另一种类型的内存，即常量内存。从常量内存的名字就可以看出，常量内存用于保存在核函数执行期间不会发生变化的数据。NVIDIA硬件提供了64KB的常量内存，并且对常量内存采取了不同于标准全局内存的处理方式。在某些情况中，用常量内存来替换全局内存能有效地减少内存带宽。

6.2.1 光线跟踪简介

我们将给出一个简单的光线跟踪（Ray Tracing）应用程序示例，并在这个示例中介绍如何使用常量内存。首先，我们将介绍光线跟踪的一些背景知识。如果你已经熟悉了光线跟踪的一些基本概念，那么可以直接跳到6.2.2节。

简单地说，光线跟踪是从三维对象场景中生成二维图像的一种方式。此时你会奇怪，这不是GPU的设计初衷么？当你玩游戏时，这与OpenGL和DirectX实现的功能有何不同？没错，GPU确实能解决相同的问题，但它们使用的是一种称之为光栅化（Rasterization）的技术。在许多参考书中都介绍了光栅化技术，因此在这里不会介绍二者的差异。但可以说，它们是解决相同问题的完全不同的方法。

那么，光线跟踪如何从三维场景中生成一张二维图像？原理很简单：在场景中选择一个位置放上一台假想的相机。这台数字相机包含一个光传感器来生成图像，因此我们需要判断哪些光将接触到这个传感器。图像中的每个像素与命中传感器的光线有着相同的颜色和强度。

由于在传感器中命中的光线可能来自场景中的任意位置，因此事实也证明了采用逆向计算或许是更容易实现的。也就是说，不是找出哪些光线将命中某个像素，而是想象从该像素发出一道射线进入场景中。按照这种思路，每个像素的行为都像一只“观察”场景的眼睛。图6.1说明了这些从每个像素投射光线并进入到场景的过程。

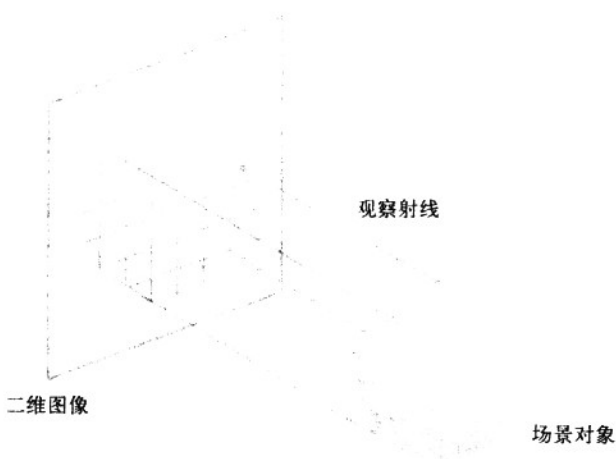


图6.1 一种简单的光线跟踪模式

我们将跟踪从像素中投射出的光线穿过场景，直到光线命中某个物体，然后计算这个像素的颜色。我们说像素都将“看到”这个物体，并根据它所看到物体的颜色来设置它的颜色。光线跟踪中的大部分计算都是光线与场景中物体的相交运算。

在更复杂的光线跟踪模型中，场景中的反光物体能够反射光线，而半透明的物体能够折射光线。这将生成二次射线（Secondary Ray）和三次射线（Tertiary Ray）等等。事实上，这正是光线跟踪最具吸引力的功能之一：实现基本的光线跟踪器是很容易的，如果需要的话，也可以在光线跟踪器中构建更为复杂的成像模型以生成更真实的图像。

6.2.2 在GPU上实现光线跟踪

由于OpenGL和DirectX等API都不是专门为了实现光线跟踪而设计的，因此我们必须使用CUDA C来实现基本的光线跟踪器。我们构造的光线跟踪器非常简单，这样可以将重点放在常量内存的使用上。因此，如果你希望基于这段代码来构建一个功能完备的渲染器，那么是不现实的。我们的光线跟踪器只支持一组包含球状物体的场景，并且相机被固定在Z轴，面向原点。此外，我们将不支持场景中的任何照明，从而避免二次光线带来的复杂性。我们也不计算照明效果，而只是为每个球面分配一个颜色值，然后如果它们是可见的，则通过某个预先计算的值得对其着色。

那么，光线跟踪器将实现哪些功能？它将从每个像素发射一道光线，并且跟踪这些光线会命中哪些球面。此外，它还将跟踪每道命中光线的深度。当一道光线穿过多个球面时，只有最接近相机的球面才会被看到。我们的“光线跟踪器”会把相机看不到的球面隐藏起来。

通过一个数据结构对球面建模，在数据结构中包含了球面的中心坐标 (x, y, z)，半径 radius，以及颜色值 (r, g, b)。

```
#define INF    2e10f

struct Sphere {
    float    r,b,g;
    float    radius;
    float    x,y,z;
    __device__ float hit( float ox, float oy, float *n ) {
        float dx = ox - x;
        float dy = oy - y;
        if (dx*dx + dy*dy < radius*radius) {
            float dz = sqrtf( radius*radius - dx*dx - dy*dy );
            *n = dz / sqrtf( radius * radius );
            return dz + z;
        }
        return -INF;
    }
};
```

我们还将注意到，在这个结构中定义了一个方法hit(float ox, float oy, float *n)。对于来自 (ox, oy) 处像素的光线，这个方法将计算光线是否与这个球面相交。如果光线与球面相交，那么这个方法将计算从相机到光线命中球面处的距离。我们需要这个信息，原因在前面已经提到了：当光线命中多个球面时，只有最接近相机的球面才会被看见。

main()函数遵循了与前面示例大致相同的代码结构。

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define rnd( x ) (x * rand() / RAND_MAX)
#define SPHERES 20

Sphere *s;

int main( void ) {
    // 记录起始时间
    cudaEvent_t    start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
```

```

HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );

CPUBitmap bitmap( DIM, DIM );
unsigned char *dev_bitmap;

// 在GPU上分配内存以计算输出位图
HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                          bitmap.image_size() ) );

// 为Sphere数据集分配内存
HANDLE_ERROR( cudaMalloc( (void**)&s,
                          sizeof(Sphere) * SPHERES ) );

```

我们为输入的数据分配了内存，这些数据是一个构成场景的Sphere数组。由于Sphere数组将在CPU上生成并在GPU上使用，因此我们必须分别调用cudaMalloc()和malloc()在GPU和CPU上分配内存。我们还需要分配一张位图图像，当在GPU上计算光线跟踪球面时，将用计算得到的像素值来填充这张图像。

在分配输入数据和输出数据的内存后，我们将随机地生成球面的中心坐标，颜色以及半径。

```

// 分配临时内存，对其初始化，并复制到
// GPU上的内存，然后释放临时内存
Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
for (int i=0; i<SPHERES; i++) {
    temp_s[i].r = rnd( 1.0f );
    temp_s[i].g = rnd( 1.0f );
    temp_s[i].b = rnd( 1.0f );
    temp_s[i].x = rnd( 1000.0f ) - 500;
    temp_s[i].y = rnd( 1000.0f ) - 500;
    temp_s[i].z = rnd( 1000.0f ) - 500;
    temp_s[i].radius = rnd( 100.0f ) + 20;
}

```

当前，程序将生成一个包含20个球面的随机数组，但这个数量值是通过一个#define宏指定的，因此可以相应地做出调整。

通过cudaMemcpy()将这个球面数组复制到GPU，然后释放临时缓冲区。

```

HANDLE_ERROR( cudaMemcpy( s, temp_s,
                          sizeof(Sphere) * SPHERES,
                          cudaMemcpyHostToDevice ) );

free( temp_s );

```

现在，输入数据位于GPU上，并且我们已经为输出数据分配好了空间，因此可以启动核函数。

```
// 从球面数据中生成一张位图
dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );
```

我们稍后将分析核函数本身，现在你只需知道这个函数将执行光线跟踪计算并且从输入的一组球面中为每个像素计算颜色数据。最后，我们将把输出图像从GPU中复制回来，并显示它。当然，我们还要释放所有已经分配但还未释放的内存。

```
// 将位图从GPU复制回到CPU以显示
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                           bitmap.image_size(),
                           cudaMemcpyDeviceToHost ) );

bitmap.display_and_exit();

// 释放内存
cudaFree( dev_bitmap );
cudaFree( s );
}
```

现在，所有这些操作对你来说应该很熟悉了。那么，如何实现光线跟踪算法？由于我们已经建立了一个非常简单的光线跟踪模型，因此核函数理解起来就很容易了。每个线程都会为输出影像中的一个像素计算颜色值，因此我们遵循一种惯用的方式，计算每个线程对应的x坐标和y坐标，并且根据这两个坐标来计算输出缓冲区中的偏移。此外，我们还将把图像坐标（x，y）偏移DIM/2，这样z轴将穿过图像的中心。

```
__global__ void kernel( unsigned char *ptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;
    float ox = (x - DIM/2);
    float oy = (y - DIM/2);
```

由于每条光线都需要判断与球面相交的情况，因此我们现在对球面数组进行迭代，并判断每个球面的命中情况。

```
float r=0, g=0, b=0;
float maxz = -INF;
for(int i=0; i<SPHERES; i++) {
    float n;
    float t = s[i].hit( ox, oy, &n );
    if ( t > maxz ) {
        float fscale = n;
```



```

        r = s[i].r * fscale;
        g = s[i].g * fscale;
        b = s[i].b * fscale;
    }
}

```

显然，判断相交计算的大部分代码都包含在for()循环中。对每个输入的球面进行迭代，并调用hit()方法来判断来自像素的光线能否“看到”球面。如果光线命中了当前的球面，那么接着判断命中的位置与相机之间的距离是否比上一次命中的距离更加接近。如果更加接近，那么我们将这个距离保存为新的最接近球面。此外，我们还将保存这个球面的颜色值，这样当循环结束时，线程就会知道与相机最接近的球面的颜色值。由于这就是从像素发出的光线“看到”的颜色值，也就是该像素的颜色值，因此这个值应该保存在输出图像的缓冲区中。

在判断了每个球面的相交情况后，可以将当前的颜色值保存到输出图像中，如下所示：

```

ptr[offset*4 + 0] = (int)(r * 255);
ptr[offset*4 + 1] = (int)(g * 255);
ptr[offset*4 + 2] = (int)(b * 255);
ptr[offset*4 + 3] = 255;
}

```

注意，如果没有命中任何球面，那么保存的颜色值将是变量r、b和g的初始值。在本示例中，r、b和g的初始值都设置为0，因此背景色是黑色。你可以修改这些值以便生成不同颜色的背景。在图6.2中给出了一个输出示例，其中绘制了20个球体，并且背景为黑色。

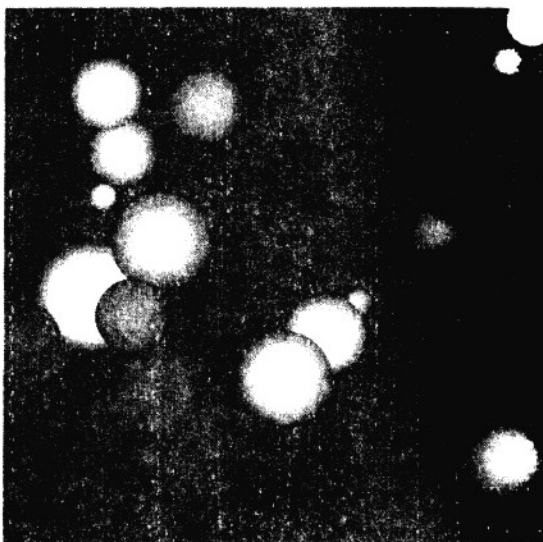


图6.2 光线跟踪示例的截图

由于我们随机设置了这些球面的位置、颜色和大小，因此，如果你得到的输出与这里的图像并不相同，那么也是正常的。

6.2.3 通过常量内存来实现光线跟踪

你已经注意到，在这个光线跟踪示例中并没有提到常量内存。现在，我们使用常量内存来改进这个示例。由于常量内存是不能修改的，因此显然无法用常量内存来保存输出图像的数据。在这个示例中只有一个输入数据，即球面数组，因此应该将这个数据保存到常量内存中。

常量内存的声明方法与共享内存是类似的。要使用常量内存，那么在代码中将不再像下面这样声明数组：

```
Sphere *s;
```

而是在变量前面加上__constant__修饰符：

```
__constant__ Sphere s[SPHERES];
```

注意，在最初的示例中，我们声明了一个指针，然后通过cudaMalloc()来为指针分配GPU内存。当我们将其修改为常量内存时，同样要将这个声明修改为在常量内存中静态地分配空间。我们不再需要对球面数组调用cudaMalloc()或者cudaFree()，而是在编译时为这个数组提交一个固定的大小。这对许多应用程序来说是可以接受的，因为常量内存能够带来性能的提升。我们稍后会看到常量内存的优势，但首先来看看如何将main()函数修改为使用常量内存：

```
int main( void ) {
    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    // 在GPU上分配内存以计算输出位图
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() ) );

    // 分配临时内存，对其初始化，并复制到
    // GPU上的内存，然后释放临时内存
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
        temp_s[i].b = rnd( 1.0f );
        temp_s[i].x = rnd( 1000.0f ) - 500;
        temp_s[i].y = rnd( 1000.0f ) - 500;
        temp_s[i].z = rnd( 1000.0f ) - 500;
        temp_s[i].radius = rnd( 100.0f ) + 20;
    }
}
```

```

HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                   sizeof(Sphere) * SPHERES ) );

free( temp_s );

// 从球面数据中生成一张位图
dim3    grids(DIM/16,DIM/16);
dim3    threads(16,16);
kernel<<<grids,threads>>>( dev_bitmap );

// 将位图从GPU复制回到CPU以显示
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                           bitmap.image_size(),
                           cudaMemcpyDeviceToHost ) );

bitmap.display_and_exit();

// 释放内存
cudaFree( dev_bitmap );
}

```

这段代码在很大程度上类似于之前main()的实现。正如在前面提到的，对main()函数的修改之一就是不再需要调用cudaMalloc()为球面数组分配空间。在下面给出了另一处修改：

```

HANDLE_ERROR( cudaMemcpyToSymbol( s, temp_s,
                                   sizeof(Sphere) * SPHERES ) );

```

当从主机内存复制到GPU上的常量内存时，我们需要使用这个特殊版本的cudaMemcpy()。cudaMemcpyToSymbol()与参数为cudaMemcpyHostToDevice()的cudaMemcpy()之间的唯一差异在于，cudaMemcpyToSymbol()会复制到常量内存，而cudaMemcpy()会复制到全局内存。

除了__constant__修饰符和对main()的两处修改之外，其他的代码都是相同的。

6.2.4 常量内存带来的性能提升

__constant__将把变量的访问限制为只读。在接受了这种限制后，我们希望获得某种回报。在前面曾提到，与从全局内存中读取数据相比，从常量内存中读取相同的数据可以节约内存带宽，主要有两个原因：

- 对常量内存的单次读操作可以广播到其他的“邻近 (Nearby)”线程，这将节约15次读取操作。
- 常量内存的数据将缓存起来，因此对相同地址的连续读操作将不会产生额外的内存通信量。

“邻近”这个词的含义是什么？要回答这个问题，我们需要解释线程束 (Warp) 的概念。

这里的“Warp”并不是《星际迷航》电影中的曲速引擎（Warp Drive），而是来自纺织（Weaving）领域的概念，这里的线程束与空间旅行速度没有任何关系。线程束可以看成是一组线程通过交织而形成的一个整体。在CUDA架构中，线程束是指一个包含32个线程的集合，这个线程集合被“编织在一起”并且以“步调一致（Lockstep）”的形式执行。在程序中的每一行，线程束中的每个线程都将在不同的数据上执行相同的指令。

当处理常量内存时，NVIDIA硬件将把单次内存读取操作广播到每个半线程束（Half-Warp）。在半线程束中包含了16个线程，即线程束中线程数量的一半。如果在半线程束中的每个线程都从常量内存的相同地址上读取数据，那么GPU只会产生一次读取请求并在随后将数据广播到每个线程。如果从常量内存中读取大量的数据，那么这种方式产生的内存流量只是使用全局内存时的1/16（大约6%）。

但在读取常量内存时，所节约的并不仅限于减少了94%的带宽。由于这块内存的内容是不会发生变化的，因此硬件将主动把这个常量数据缓存在GPU上。在第一次从常量内存的某个地址上读取后，当其他半线程束请求同一个地址时，那么将命中缓存，这同样减少了额外的内存流量。

在我们的光线跟踪器中，每个线程都要读取球面的相应数据从而计算它与光线的相交情况。在把应用程序修改为将球面数据保存在常量内存后，硬件只需要请求这个数据一次。在缓存数据后，其他每个线程将不会产生内存流量，原因有两个：

- 线程将在半线程束的广播中收到这个数据。
- 从常量内存缓存中收到数据。

然而，当使用常量内存时，也可能对性能产生负面影响。半线程束广播功能实际上是一把双刃剑。虽然当所有16个线程都读取相同地址时，这个功能可以极大地提升性能，但当所有16个线程分别读取不同的地址时，它实际上会降低性能。

只有当16个线程每次都只需要相同的读取请求时，才值得将这个读取操作广播到16个线程。然而，如果半线程束中的所有16个线程需要访问常量内存中不同的数据，那么这个16次不同的读取操作会被串行化，从而需要16倍的时间来发出请求。但如果从全局内存中读取，那么这些请求会同时发出。在这种情况下，从常量内存读取就慢于从全局内存中读取。

6.3 使用事件来测量性能

在充分理解了常量内存既可能带来正面影响，也可能带来负面影响后，你已经决定将光线跟踪器修改为使用常量内存。但是，如何判断常量内存对程序性能有着多大的影响？最简单的衡量标准之一就是回答这个问题：哪个版本的执行时间更短？我们可以使用CPU或者操作系统中的某个计时器，但这将带来各种延迟（包括操作系统线程调度，高精度CPU计时器可用性等）。而且，当GPU核函数运行时，我们还可以在主机上异步地执行计算。测量这些主机运

算时间的唯一方式是使用CPU或者操作系统的定时机制。为了测量GPU在某个任务上花费的时间，我们将使用CUDA的事件API。

CUDA中的事件本质上是一个GPU时间戳，这个时间戳是在用户指定的时间点上记录的。由于GPU本身支持记录时间戳，因此就避免了当使用CPU定时器来统计GPU执行的时间时可能遇到的诸多问题。这个API使用起来很容易，因为获得一个时间戳只需要两个步骤：首先创建一个事件，然后记录一个事件。例如，在某段代码的开头，我们告诉CUDA运行时记录当前时间。首先创建一个事件，然后记录这个事件：

```
cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord(start, 0);
```

你将注意到，当告诉运行时记录事件start时还指定了第二个参数。在前面的示例中，这个参数为0。这个参数当前并不重要，因此在这里不进行解释。如果你确实好奇，那么可以在我们讨论流（Stream）时再介绍这个参数。

要统计一段代码的执行时间，不仅要创建一个起始事件，还要创建一个结束事件。当在GPU上执行某个工作时，我们不仅要告诉CUDA运行时记录起始时间，还要记录结束时间：

```
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );
```

```
// 在GPU上执行一些工作
```

```
cudaEventRecord( stop, 0 );
```

然而，当按照这种方式来统计GPU代码的执行时间时，仍然存在一个问题。要修复这个问题，只需一行代码，但需要对这行代码进行一些解释。在使用事件时，最复杂的情况是当我们在CUDA C中执行的某些异步函数调用时。例如，当启动光线跟踪器的核函数时，GPU开始执行代码，但在GPU执行完之前，CPU会继续执行程序中的下一行代码。从性能的角度来看，这是非常好的，因为这意味着我们可以在GPU和CPU上同时进行计算，但从逻辑概念上来看，这将使计时工作变得更加复杂。

你应该将cudaEventRecord()视为一条记录当前时间的语句，并且把这条语句放入GPU的未完成工作队列中。因此，直到GPU执行完了在调用cudaEventRecord()之前的所有语句时，事件才会被记录下来。由stop事件来测量正确的时间正是我们所希望的，但仅当GPU完成了之前的工作并且记录了stop事件后，才能安全地读取stop时间值。幸运的是，我们有一种方式告诉CPU在某个事件上同步，这个事件API函数就是cudaEventSynchronize()：

```

cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

```

// 在GPU上执行一些工作

```

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop );

```

现在，我们已经告诉运行时阻塞后面的语句，直到GPU执行到达stop事件。当cudaEventSynchronize返回时，我们知道在stop事件之前的所有GPU工作已经完成了，因此可以安全地读取在stop中保存的时间戳。值得注意的是，由于CUDA事件是直接在GPU上实现的，因此它们不适用于对同时包含设备代码和主机代码的混合代码计时。也就是说，如果你试图通过CUDA事件对核函数和设备内存复制之外的代码进行计时，将得到不可靠的结果。

测量光线跟踪器的性能

为了对光线跟踪器计时，我们需要分别创建一个起始事件和一个结束事件。下面是一个支持计时的光线跟踪器，其中没有使用常量内存：

```

int main( void ) {
    // 记录起始时间
    cudaEvent_t      start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    CPUBitmap bitmap( DIM, DIM );
    unsigned char *dev_bitmap;

    // 在GPU上为输出位图分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&dev_bitmap,
                               bitmap.image_size() ) );

    // 为Sphere数据集分配内存
    HANDLE_ERROR( cudaMalloc( (void**)&s,
                               sizeof(Sphere) * SPHERES ) );

    // 分配临时内存，对其初始化，复制到
    // GPU上的内存，然后释放临时内存
    Sphere *temp_s = (Sphere*)malloc( sizeof(Sphere) * SPHERES );
    for (int i=0; i<SPHERES; i++) {
        temp_s[i].r = rnd( 1.0f );
        temp_s[i].g = rnd( 1.0f );
    }
}

```

```

    temp_s[i].b = rnd( 1.0f );
    temp_s[i].x = rnd( 1000.0f ) - 500;
    temp_s[i].y = rnd( 1000.0f ) - 500;
    temp_s[i].z = rnd( 1000.0f ) - 500;
    temp_s[i].radius = rnd( 100.0f ) + 20;
}
HANDLE_ERROR( cudaMemcpy( s, temp_s,
                          sizeof(Sphere) * SPHERES,
                          cudaMemcpyHostToDevice ) );

free( temp_s );

// 从球面数据中生成一张位图
dim3    grids(DIM/16,DIM/16);
dim3    threads(16,16);
kernel<<<grids,threads>>>( s, dev_bitmap );

// 将位图从GPU上复制回来并显示
HANDLE_ERROR( cudaMemcpy( bitmap.get_ptr(), dev_bitmap,
                          bitmap.image_size(),
                          cudaMemcpyDeviceToHost ) );

// 获得结束时间，并显示计时结果
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );

float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time to generate:   %3.1f ms\n", elapsedTime );

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );

// 显示位图
bitmap.display_and_exit();

// 释放内存
cudaFree( dev_bitmap );
cudaFree( s );
}

```

注意，我们调用了两个额外的函数，分别为cudaEventElapsedTime()和cudaEventDestroy()。cudaEventElapsedTime()是一个工具函数，用来计算两个事件之间经历的时间。第一个参数为某个浮点变量的地址，在这个参数中将包含两次事件之间经历的时间，单位为毫秒。


```

// 获得结束时间，并显示计时结果
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );

printf( "Time to generate:  %3.1f ms\n", elapsedTime );

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );

// 显示
bitmap.display_and_exit();

// 释放内存
cudaFree( dev_bitmap );
}

```

现在，当我们运行这两个版本的光线跟踪器时，就可以比较在GPU上完成相同工作时节约的时间。这将告诉我们，引入常量内存是提升应用程序的性能还是降低性能。在这里的情况中，使用常量内存将极大地提升性能。我们在GeForce GTX 280上进行了实验，实验结果表明，使用常量内存的光线跟踪器的性能比使用全局内存的性能提升了50%。在不同的GPU上，你得到的实验结果可能会有所不同，但使用常量内存的光线跟踪器应该比不使用常量内存的光线跟踪器要更快一些。

6.4 本章小结

除了在前面章节中介绍的全局内存和常量内存外，NVIDIA硬件还提供了其他类型的内存可供使用。与标准的全局常量内存相比，常量内存存在着一些限制，但在某些情况中，使用常量内存将提升应用程序的性能。特别是，当线程束中的所有线程都访问相同的只读数据时，将获得额外的性能提升。在这种数据访问模式中使用常量内存可以节约内存带宽，不仅是因为这种模式可以将读取操作在半线程束中广播，而且还因为在芯片上包含了常量内存缓存。在许多算法中，内存带宽都是一种瓶颈，因此采用一些机制来改善这种情况是非常有用的。

接着我们学习了如何通过CUDA事件在GPU执行过程的特定时刻上记录时间戳，看到了如何将CPU与GPU在某个事件上同步，以及如何计算在两个事件之间经历的时间。我们设计了一种方法来比较采用不同类型的内容来计算光线跟踪球面时的运行时间差异，并得出结论，对于本章的示例程序，使用常量内存将带来显著的性能提升。

第 7 章

纹理内存

在学习常量内存时，我们看到了在特定环境中使用这种特殊内存将极大地提升应用程序的性能。我们还学习了如何测量性能提升，以便确保在性能选择上做出合理的决策。在本章中，我们将学习如何分配和使用纹理内存 (Texture Memory)。和常量内存一样，纹理内存是另一种类型的只读内存，在特定的访问模式中，纹理内存同样能够提升性能并减少内存流量。虽然纹理内存最初是针对传统的图形处理应用程序而设计的，但在某些 GPU 计算应用程序中同样非常有用。

7.1 本章目标

通过本章的学习，你可以：

- 了解纹理内存的性能特性。
- 了解如何在CUDA C中使用一维纹理内存。
- 了解如何在CUDA C中使用二维纹理内存。

7.2 纹理内存简介

在读了本章开头的介绍后，你其实已经知道了纹理内存的秘密：它只不过是在CUDA C程序中可以使用的另一种只读内存。熟悉图形硬件工作原理的读者对纹理内存不会感到陌生，但本章要介绍的是纹理内存同样可以用于通用计算。虽然NVIDIA为OpenGL和DirectX等的渲染流水线都设计了纹理单元，但纹理内存具备的一些属性使其在计算中变得非常有用。

与常量内存类似的是，纹理内存同样缓存在芯片上，因此在某些情况中，它能够减少对内存的请求并提供更高效的内存带宽。纹理缓存是专门为那些在内存访问模式中存在大量空间局部性（Spatial Locality）的图形应用程序而设计的。在某个计算应用程序中，这意味着一个线程读取的位置可能与邻近线程读取的位置“非常接近”，如图7.1所示。

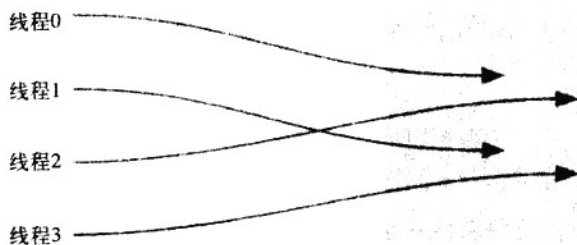


图7.1 将线程映射到二维内存区域

从数学角度来看，图中的四个地址并非连续的，在一般的CPU缓存模式中，这些地址将不会缓存。但由于GPU纹理缓存是专门为了加速这种访问模式而设计的，因此如果在这种情况下使用纹理内存而不是全局内存，那么将会获得性能提升。事实上，这种访问在通用计算中并非罕见，我们稍后将会看到。

7.3 热传导模拟

物理模拟问题或许是在计算上最具挑战性的问题之一。这类问题通常在计算精度与计算复杂性上存在着某种权衡。近年来,计算机模拟正变得越来越重要,这在很大程度上要归功于计算精度的不断提高,而这正是并行计算革命带来的好处。由于许多物理模拟计算都可以很容易地并行化,因此我们将在这个示例中看到一种非常简单的模拟模型。

7.3.1 简单的传热模型

为了说明一种可以有效使用纹理内存的情况,我们将构造一个简单的二维热传导模拟。首先假设有一个矩形房间,将其分成一个格网。在格网中随机散布一些“热源”,它们有着不同的固定温度。在图7.2中给出了这个房间的示意图。

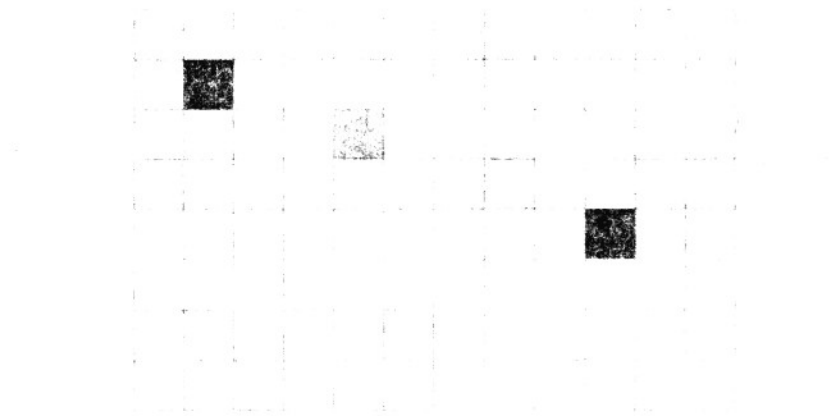


图7.2 一个带有不同温度“热源”的房间

在给定了矩形格网以及热源分布后,我们可以计算格网中每个单元的温度随时间的变化情况。为了简单,热源单元本身的温度将保持不变。在时间递进的每个步骤中,我们假设热量在某个单元及其邻接单元之间“流动”。如果某个单元的邻接单元的温度更高,那么热量将从邻接单元传导到该单元。相反地,如果某个单元的温度比邻接单元的温度高,那么它将变冷。图7.3给出了这种热量流动示意图。



图7.3 热量从更热的单元传导到更冷的单元

在热传导模型中，我们对单元中新温度的计算方法为，将单元与邻接单元的温差相加起来，然后加上原有温度，计算公式如等式7.1所示。

等式7.1

$$T_{NEW} = T_{OLD} + \sum_{NEIGHBORS} k(T_{NEIGHBOR} - T_{OLD})$$

在上面计算单元温度的等式中，常量 k 表示模拟过程中热量的流动速率。 k 值越大，表示系统会更快地达到稳定温度，而 k 值越小，则温度梯度将存在更长的时间。由于我们只考虑4个邻接单元（上，下，左，右）并且等式中的 k 和 T_{OLD} 都是常数，因此把等式7.1展开后将如等式7.2所示。

等式7.2

$$T_{NEW} = T_{OLD} + k(T_{TOP} + T_{BOTTOM} + T_{LEFT} + T_{RIGHT} - 4T_{OLD})$$

就像第6章的光线跟踪示例一样，在产品代码中不能使用这个模型（事实上，它与实际的物理热传导情况相差很远）。我们极大地简化了这个模型，从而将重点放到所使用的技术上。因此，让我们来看看如何在GPU上实现等式7.2中的更新。

7.3.2 温度更新的计算

我们稍后将介绍每个步骤的具体细节，首先给出更新流程的基本介绍：

1) 给定一个包含初始输入温度的格网，将其中作为热源的单位温度值复制到格网相应的单元中。这将覆盖这些单元之前计算出的温度，因此也就确保了“加热单元将保持恒温”这个条件。这个复制操作是在`copy_const_kernel()`中执行的。

2) 给定一个输入温度格网，根据等式7.2中的更新公式计算输出温度格网。这个更新操作是在`blend_kernel()`中执行的。

3) 将输入温度格网和输出温度格网交换，为下一个步骤的计算做好准备。当模拟下一个时间步时，在步骤2中计算得到的输出温度格网将成为步骤1中的输入温度格网。

在开始模拟之前，我们假设已经获得了一个格网。格网中大多数单元的温度值都是0，但有些单元包含了非0的温度值，这些单元就是拥有固定温度的热源。在模拟过程中，缓冲区中这些常量值不会发生变化，并且在每个时间步中读取。

根据我们对热传导的建模方式，首先获得前一个时间步的输出温度格网并将其作为当前时间步的输入温度格网。然后，根据步骤1，将作为热源的单元的温度值复制到输出格网中，从而覆盖该单元之前计算出的温度值。这么做是因为我们已经假设这些热源单元的温度将保持不变。通过以下核函数将格网中的热源单元复制到输入格网中：

```
__global__ void copy_const_kernel( float *iptr,
                                   const float *cptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    if (cptr[offset] != 0) iptr[offset] = cptr[offset];
}
```

前三行代码看上去非常熟悉。前两行代码将线程的threadIdx和blockIdx转换为x坐标和y坐标。第三行代码计算在输入缓冲区中的线性偏移。在加粗的一行中把cptr[]中的热源温度复制到iptr[]的输入单元中。注意，只有当格网中单元的温度值非0时，才会执行复制操作。我们这么做是为了维持非热源单元在上一个时间步中计算得到的温度值。热源单元在cptr[]中对应的元素为非零值，因此调用这个复制核函数后，热源单元的温度值在不同的时间步中将保持不变。

算法步骤2中包含的计算最多。为了执行这些更新操作，可以在模拟过程中让每个线程都负责计算一个单元。每个线程都将读取对应单元及其邻接单元的温度值，执行前面给出的更新运算，然后用计算得到的新值更新它的温度。这个核函数的大部分代码与之前使用的技术都是类似的。

[illegible]

注意，代码的开头与生成输出图像示例的开头是一样的，只是此时线程不是计算像素的颜色值，而是计算模拟单元中的温度值。首先，代码将threadIdx和blockIdx转换为x, y和offset。现在，你可以在睡觉的时候都能复诵这些代码行（但我们并不真的希望你在睡觉时还想到它们）。

接下来，代码会计算出上，下，左，右四个邻接单元的偏移并读取这些单元的温度。我们需要这些值来计算当前单元中的已更新温度。这里唯一需要注意的地方就是需要调整边界的索引，从而在边缘的单元上不会回卷。最后，在加粗的代码行中执行了等式7.2的更新运算，将原来的温度与该单元温度和邻接单元的温度的温差相加。

7.3.3 模拟过程动态演示

剩下的代码主要是设置好单元，然后显示热量的动画输出。下面给出了这段代码：

```
#include "cuda.h"
#include "../common/book.h"
#include "../common/cpu_anim.h"

#define DIM 1024
#define PI 3.1415926535897932f
#define MAX_TEMP 1.0f
#define MIN_TEMP 0.0001f
#define SPEED 0.25f

//更新函数中需要的全局变量
struct DataBlock {
    unsigned char    *output_bitmap;
    float            *dev_inSrc;
    float            *dev_outSrc;
    float            *dev_constSrc;
    CPUAnimBitmap    *bitmap;

    cudaEvent_t      start, stop;
    float            totalTime;
    float            frames;
};

void anim_gpu( DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);
    CPUAnimBitmap    *bitmap = d->bitmap;
```

```

for (int i=0; i<90; i++) {
    copy_const_kernel<<<blocks,threads>>>( d->dev_inSrc,
                                              d->dev_constSrc );

    blend_kernel<<<blocks,threads>>>( d->dev_outSrc,
                                       d->dev_inSrc );

    swap( d->dev_inSrc, d->dev_outSrc );
}
float_to_color<<<blocks,threads>>>( d->output_bitmap,
                                    d->dev_inSrc );

HANDLE_ERROR( cudaMemcpy( bitmap->get_ptr(),
                          d->output_bitmap,
                          bitmap->image_size(),
                          cudaMemcpyDeviceToHost ) );

HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
float   elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    d->start, d->stop ) );

d->totalTime += elapsedTime;
++d->frames;
printf( "Average Time per frame: %3.1f ms\n",
        d->totalTime/d->frames );
}

void anim_exit( DataBlock *d ) {
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}

```

在代码中加入了基于事件的计时功能，这与第6章中光线跟踪示例一样。示例中计时代码的作用与前面的示例相同：由于我们希望提高算法的执行速度，因此在代码中添加了测量性能的机制从而确保成功提升了性能。

动画框架的每一帧都将调用函数anim_gpu()。这个函数的参数是一个指向DataBlock的指针，以及动画已经经历的时间ticks。在动画示例中，每个线程块包含了256个线程，构成了一个 16×16 的二维单元。anim_gpu()中for()循环的每次迭代都将计算模拟过程的一个时间步，我们在7.3.2节的开头介绍了这个算法，共包含三个步骤。由于DataBlock包含了表示热源的常量缓冲区，以及在上一个时间步中计算得到的输出值，因此能够表示动画的完整状态，因而

anim_gpu()实际上并不需要用到ticks的值。

值得注意的是,在每帧中将执行90个时间步。这个数值并非有着特殊含义,而是通过实验得出的,这个数值既能避免在每个时间步中都需要复制一张位图图片,又可以避免在每一帧计算过多时间步(这将导致不稳定的动画)。如果你更关心每个模拟步骤的输出结果,而不是实时地播放动画结果,那么可以将这个值修改为在每帧中只计算一个时间步。

在计算了90个时间步后,anim_gpu()就已经准备好将当前动画的位图帧复制回CPU。由于for()循环会交换输入和输出,因此我们首先交换输入缓冲区和输出缓冲区,这样在输出缓冲区中实际上将包含第90个时间步的输出。我们通过核函数float_to_color()将温度转换为颜色,然后将结果图像通过cudaMemcpy()复制回CPU,并将复制的方向指定为cudaMemcpyDeviceToHost。最后,为了准备下一系列时间步,我们将输出缓冲区交换到输入缓冲区以便作为下一个时间步的输入。

```
int main( void ) {
    DataBlock  data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );

    HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
                               bitmap.image_size() ) );

    //假设float类型的大小为4个字符 (即rgba)
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                               bitmap.image_size() ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                               bitmap.image_size() ) );
    HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                               bitmap.image_size() ) );

    float *temp = (float*)malloc( bitmap.image_size() );
    for (int i=0; i<DIM*DIM; i++) {
        temp[i] = 0;
        int x = i % DIM;
        int y = i / DIM;
        if ((x>300) && (x<600) && (y>310) && (y<601))
            temp[i] = MAX_TEMP;
    }
    temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
```

```

temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                           bitmap.image_size(),
                           cudaMemcpyHostToDevice ) );

for (int y=800; y<DIM; y++) {
    for (int x=0; x<200; x++) {
        temp[x+y*DIM] = MAX_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                           bitmap.image_size(),
                           cudaMemcpyHostToDevice ) );

free( temp );

bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                     (void (*)(void*))anim_exit );
}

```

图7.4给出了输出图像的一个示例。注意在这张图像中，某些“热源”看上去像一些像素大小的小岛屿，它们打乱了温度分布的连续性。

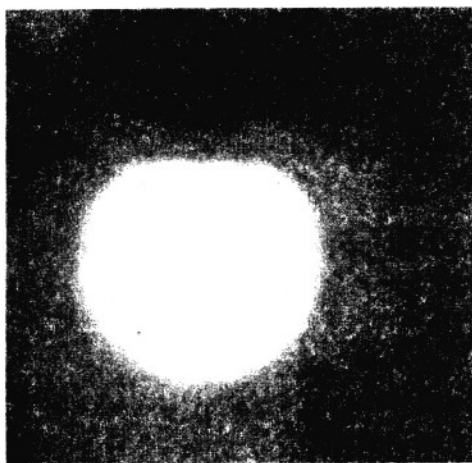


图7.4 热传导模拟动画的截屏

7.3.4 使用纹理内存

在温度更新计算的内存访问模式中存在着巨大的内存空间局部性。前面曾解释过，这种访问模式可以通过GPU纹理内存来加速。接下来，我们来学习如何使用纹理内存。

首先，需要将输入的数据声明为texture类型的引用。我们使用浮点类型纹理的引用，因为温度数值是浮点类型的。

```
// 这些变量将位于GPU上
texture<float> texConstSrc;
texture<float> texIn;
texture<float> texOut;
```

下一个需要注意的问题是，在为这三个缓冲区分配了GPU内存后，需要通过cudaBindTexture()将这些变量绑定到内存缓冲区。这相当于告诉CUDA运行时两件事情：

- 我们希望将指定的缓冲区作为纹理来使用。
- 我们希望将纹理引用作为纹理的“名字”。

在热传导模拟中分配了这三个内存后，需要将这三个内存绑定到之前声明的纹理引用(texConstSrc、texIn、texOut)。

```
HANDLE_ERROR( cudaMalloc( (void*)&data.dev_inSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void*)&data.dev_outSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void*)&data.dev_constSrc,
                           imageSize ) );

HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                               data.dev_constSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                               data.dev_inSrc,
                               imageSize ) );
HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                               data.dev_outSrc,
                               imageSize ) );
```

此时，纹理变量已经设置好了，现在可以启动核函数。然而，当读取核函数中的纹理时，需要通过特殊的函数来告诉GPU将读取请求转发到纹理内存而不是标准的全局内存。因此，当读取内存时不再使用方括号从缓冲区中读取，而是将blend_kernel()改为使用tex1Dfetch()。

此外，在全局内存和纹理内存的使用上还存在另一个差异。虽然tex1Dfetch()看上去像一个

函数，但它其实是一个编译器内置函数（Intrinsic）。由于纹理引用必须声明为文件作用域内的全局变量，因此我们不再将输入缓冲区和输出缓冲区作为参数传递给blend_kernel()，因为编译器需要在编译时知道tex1Dfetch()应该对哪些纹理采样。我们不是像前面那样传递指向输入缓冲区和输出缓冲区的指针，而是将一个布尔标志dstOut传递给blend_kernel()，这个标志会告诉我们使用哪个缓冲区作为输入，以及哪个缓冲区作为输出。下面是对blend_kernel()的修改：

```
__global__ void blend_kernel( float *dst,
                             bool dstOut ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    int left = offset - 1;
    int right = offset + 1;
    if (x == 0)    left++;
    if (x == DIM-1) right--;

    int top = offset - DIM;
    int bottom = offset + DIM;
    if (y == 0)    top += DIM;
    if (y == DIM-1) bottom -= DIM;

    float t, l, c, r, b;
    if (dstOut) {
        t = tex1Dfetch(texIn,top);
        l = tex1Dfetch(texIn,left);
        c = tex1Dfetch(texIn,offset);
        r = tex1Dfetch(texIn,right);
        b = tex1Dfetch(texIn,bottom);
    } else {
        t = tex1Dfetch(texOut,top);
        l = tex1Dfetch(texOut,left);
        c = tex1Dfetch(texOut,offset);
        r = tex1Dfetch(texOut,right);
        b = tex1Dfetch(texOut,bottom);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

由于核函数copy_const_kernel()将读取包含热源位置和温度的缓冲区，因此同样需要修改为从纹理内存而不是从全局内存中读取：


```

HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
float   elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     d->start, d->stop ) );

d->totalTime += elapsedTime;
++d->frames;
printf( "Average Time per frame: %3.1f ms\n",
        d->totalTime/d->frames );
}

```

对热传导函数的最后一个修改就是在应用程序运行结束后的清理工作。不仅要释放全局缓冲区，还需要清除与纹理的绑定：

```

//释放在GPU上分配的内存
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}

```

7.3.5 使用二维纹理内存

在本书的开头，我们提到了在某些二维问题中使用二维的线程块和线程格是非常有用的。对于纹理内存来说同样如此。在许多情况下，二维内存空间是非常有用的，它们对于熟悉标准C中多维数组的程序员来说都不会感到陌生。我们来看看如何将热传导应用程序修改为使用二维纹理。

首先，需要修改纹理引用的声明。如果没有具体说明，默认的纹理引用都是一维的，因此我们增加了代表维数的参数2，这表示声明的是一个二维纹理引用。

```

texture<float,2> texConstSrc;
texture<float,2> texIn;
texture<float,2> texOut;

```

二维纹理将简化blend_kernel()方法的实现。虽然我们需要将tex1Dfetch()调用修改为tex2D()调用，但却不再需要使用通过线性化offset变量以计算top、left、right和bottom等偏移。当使用二维纹理时，可以直接通过x和y来访问纹理。

而且,当使用tex2D()时,我们不再需要担心发生溢出问题。如果x或y小于0,那么tex2D()将返回0处的值。同理,如果某个值大于宽度,那么tex2D()将返回位于宽度处的值。注意,在我们的应用程序刚好需要这种行为,但在其他应用程序中可能需要其他的行为。

这些简化带来的好处之一就是,核函数的代码将变得更加简单。

```
__global__ void blend_kernel( float *dst,
                             bool dstOut ) {
    //将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float t, l, c, r, b;
    if (dstOut) {
        t = tex2D(texIn,x,y-1);
        l = tex2D(texIn,x-1,y);
        c = tex2D(texIn,x,y);
        r = tex2D(texIn,x+1,y);
        b = tex2D(texIn,x,y+1);
    } else {
        t = tex2D(texOut,x,y-1);
        l = tex2D(texOut,x-1,y);
        c = tex2D(texOut,x,y);
        r = tex2D(texOut,x+1,y);
        b = tex2D(texOut,x,y+1);
    }
    dst[offset] = c + SPEED * (t + b + r + l - 4 * c);
}
```

由于所有之前对tex1Dfetch()的调用都需要修改为对tex2D()的调用,因此我们需要在copy_const_kernel()中进行相应的修改。与核函数blend_kernel()类似的是,我们不再需要通过offset来访问纹理,而只需使用x和y来访问热源的常量数量:

```
__global__ void copy_const_kernel( float *iptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    int offset = x + y * blockDim.x * gridDim.x;

    float c = tex2D(texConstSrc,x,y);
    if (c != 0)
        iptr[offset] = c;
}
```

在使用一维纹理的热传导模拟版本中还剩下一些最后的修改，它们与之前的修改是类似的。在main()中需要对纹理绑定调用进行修改，并告诉运行时：缓冲区将被视为二维纹理而不是一维纹理。

```

HANDLE_ERROR( cudaMalloc( (void*)&data.dev_inSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void*)&data.dev_outSrc,
                           imageSize ) );
HANDLE_ERROR( cudaMalloc( (void*)&data.dev_constSrc,
                           imageSize ) );

cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
HANDLE_ERROR( cudaBindTexture2D( NULL, texConstSrc,
                                   data.dev_constSrc,
                                   desc, DIM, DIM,
                                   sizeof(float) * DIM ) );

HANDLE_ERROR( cudaBindTexture2D( NULL, texIn,
                                   data.dev_inSrc,
                                   desc, DIM, DIM,
                                   sizeof(float) * DIM ) );

HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                                   data.dev_outSrc,
                                   desc, DIM, DIM,
                                   sizeof(float) * DIM ) );

```

与不使用纹理内存和使用一维纹理内存相同的是，都需要为输入数组事先分配存储空间。接下来的与一维纹理示例有所不同，因为当绑定二维纹理时，CUDA运行时要求提供一个cudaChannelFormatDesc。在上面的代码中包含了一个对通道格式描述符（Channel Format Descriptor）的声明。在这里可以使用默认的参数，并且只要指定需要的是一个浮点描述符。然后，我们通过cudaBindTexture2D()，纹理的维数（DIM×DIM）以及通道格式描述符（desc）将这三个输入缓冲区绑定为二维纹理。main()函数的其他部分保持不变。

```

int main( void ) {
    DataBlock data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );

```



```

int imageSize = bitmap.image_size();

HANDLE_ERROR( cudaMalloc( (void**)&data.output_bitmap,
                          imageSize ) );
//假设float类型的大小为4个字符 (即rgba)
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_inSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_outSrc,
                          imageSize ) );
HANDLE_ERROR( cudaMalloc( (void**)&data.dev_constSrc,
                          imageSize ) );

cudaChannelFormatDesc desc = cudaCreateChannelDesc<float>();
HANDLE_ERROR( cudaBindTexture2D( NULL, texConstSrc,
                                data.dev_constSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );

HANDLE_ERROR( cudaBindTexture2D( NULL, texIn,
                                data.dev_inSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );

HANDLE_ERROR( cudaBindTexture2D( NULL, texOut,
                                data.dev_outSrc,
                                desc, DIM, DIM,
                                sizeof(float) * DIM ) );

// 初始化常量数据
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
    temp[i] = 0;
    int x = i % DIM;
    int y = i / DIM;
    if ((x>300) && (x<600) && (y>310) && (y<601))
        temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}

```

```

    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                          imageSize,
                          cudaMemcpyHostToDevice ) );

// 初始化输入数据
for (int y=800; y<DIM; y++) {
    for (int x=0; x<200; x++) {
        temp[x+y*DIM] = MAX_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                          imageSize,
                          cudaMemcpyHostToDevice ) );

free( temp );

bitmap.anim_and_exit( (void (*)(void*,int))anim_gpu,
                     (void (*)(void*))anim_exit );
}

```

虽然我们需要通过不同的函数来告诉运行时绑定一维纹理还是绑定二维纹理，但可以通过同一个函数来取消纹理的绑定，即cudaUnbindTexture()。正因为如此，执行释放操作的函数可以保持不变。

```

//释放在GPU上分配的内存
void anim_exit( DataBlock *d ) {
    cudaUnbindTexture( texIn );
    cudaUnbindTexture( texOut );
    cudaUnbindTexture( texConstSrc );
    cudaFree( d->dev_inSrc );
    cudaFree( d->dev_outSrc );
    cudaFree( d->dev_constSrc );

    HANDLE_ERROR( cudaEventDestroy( d->start ) );
    HANDLE_ERROR( cudaEventDestroy( d->stop ) );
}

```

无论使用二维纹理还是一维纹理，热传导模拟应用程序的性能基本相同。因此，基于性能来选择使用一维纹理还是二维纹理可能没有太大的意义。对于这个应用程序来说，当使用二维纹理时，代码会更简单一些，因为模拟的问题刚好是二维的。但通常来说，如果问题不是二维的，那么究竟选择一维纹理还是二维纹理要视具体情况而定。

7.4 本章小结

与第6章介绍的常量内存一样，纹理内存的优势之一在于芯片上的缓存。在像热传导模拟这样的应用程序中，这种优势尤其值得注意：在数据访问模式中存在某种空间局部性。我们看到了既可以使用一维纹理，也可以使用二维纹理，二者都有着类似的性能特性。正如我们有时候会选择是使用线程块还是线程格一样，当选择一维纹理还是二维纹理时，在很大程度上取决于能否带来便利。当使用二维纹理时，代码会更为整洁一些，并且能自动处理边界问题，因此在热传导应用程序中适合使用二维纹理。但是你也已经看到，无论采样何种纹理都能得到不错的效果。

如果使用纹理采样器（Texture Sampler）自动执行的某种转换，那么纹理内存还能带来额外的加速，例如将打包的数据释放到不同的变量，或者将8位或16位的整数转换为标准化的浮点数值。在热传导应用程序中并没有使用这些功能，但它们对你来说或许是有用的！

第 8 章

图形互操作性

由于本书重点介绍的是通用计算，因此我们在很大程度上忽略了一些包含特殊功能的GPU。GPU的成功要归功于它能实时计算复杂的渲染任务，同时系统的其他部分还可以执行其他的工作。这就带来了一个显而易见的问题：能否在同一个应用程序中GPU既执行渲染计算，又执行通用计算？如果要渲染的图像依赖通用计算的结果，那么该如何处理？或者，如果想在已经渲染的帧上执行某种图像处理或者统计，又该如何实现？

幸运的是，在通用计算与渲染模式之间确实存在这种互操作，而且还是非常容易实现的。CUDA C应用程序可以无缝地与OpenGL和DirectX这两种实时渲染API进行交互。本章将介绍如何使用这种互操作性功能。

本章的示例与前面章节中的示例将有所不同。本章将假设你已经具备了一些其他的技术背景知识，因为在这些示例中包含了大量的OpenGL和GLUT (OpenGL Utility Toolkit) 代码，但我们并不会对它们做出详细解释。在许多参考资源中都介绍了图形API，包括在线资源和书店里的书籍，但在本书中将不讨论这些主题。本章将重点介绍CUDA C及其与图形应用程序集成的功能。如果你不熟悉OpenGL或者DirectX，那么很可能无法充分理解本章内容，并且可能想要跳过本章。

8.1 本章目标

通过本章的学习，你可以：

- 了解图形互操作性是什么以及为什么需要使用它。
- 了解如何设置某个CUDA设备的图形互操作性。
- 了解如何在CUDA C核函数和OpenGL渲染函数之间共享数据。

8.2 图形互操作

为了说明在图形库与CUDA C之间的互操作机制，我们将编写一个包含两步骤的应用程序。第一个步骤是使用CUDA C核函数来生成图像数据。在第二个步骤中，应用程序将这个数据传递给OpenGL驱动程序并进行渲染。要实现这个功能，我们将使用在前面章节中介绍的大部分CUDA C，以及一些OpenGL或者GLUT函数调用。

首先，我们要包含GLUT和CUDA的头文件从而确保定义了正确的函数和枚举类型。我们还定义了应用程序渲染窗口的大小。窗口为 512×512 个像素，这是一个相对较小的绘制量。

```
#define GL_GLEXT_PROTOTYPES
#include "GL/glut.h"
#include "cuda.h"
#include "cuda_gl_interop.h"
#include "../common/book.h"
#include "../common/cpu_bitmap.h"

#define DIM 512
```

此外，我们声明了两个全局变量来保存句柄，这些句柄指向将要在OpenGL和CUDA C之间共享的数据。我们马上会看到如何使用这两个变量，它们将保存指向同一个缓冲区的不同句柄。之所以需要两个独立的变量，是因为OpenGL和CUDA对于这个缓冲区各自有着不同的“名字”。变量bufferObj是OpenGL对这个数据的命名，而变量resource则是CUDA C对这个变量的命名。

```
GLuint bufferObj;
cudaGraphicsResource *resource;
```

现在，让我们来看看实际的应用程序。要做的第一件事情就是选择运行应用程序的CUDA设备。在许多系统上，这并不是一个复杂的过程，因为这些系统通常只包含一个支持CUDA的GPU。然而，随着越来越多的系统包含了多个支持CUDA的GPU，就需要通过某种方法从中进行选择。幸运的是，CUDA运行时提供了这种功能。

```
int main( int argc, char **argv ) {
```

```

cudaDeviceProp prop;
int dev;

memset( &prop, 0, sizeof( cudaDeviceProp ) );
prop.major = 1;
prop.minor = 0;
HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );

```

你或许还记得在第3章中看到的cudaChooseDevice(), 我们现在再次使用它。基本上, 这段代码告诉运行时选择一个拥有1.0或者更高版本计算功能集的GPU。代码的原理是, 首先创建一个cudaDeviceProp结构并将其初始化为空, 然后将major版本设置为1, minor版本设置为0。接下来, 将这个结构传递给cudaChooseDevice(), 这个函数将告诉运行时选择系统中的某个满足cudaDeviceProp结构指定条件的GPU。在第9章中, 我们将看到GPU计算功能集的更多含义, 但就目前来说, 我们只需要知道它基本上表示GPU支持的各种功能。所有支持CUDA的GPU都至少包含1.0版本的计算功能集, 因此这个函数调用的结果就是运行时可以选择任何一个支持CUDA的设备, 并且在变量dev中返回这个设备的标识符。然而, 我们无法确保这个设备是最好的或是最快的GPU, 也不能确保不同版本的GPU运行时会选择同一个设备。

如果设备选择的结果看上去没有太多的作用, 那么为什么还要费力填充一个cudaDeviceProp结构, 并调用cudaChooseDevice()来获得一个有效的设备ID? 而且, 我们之前从来没有这么做过, 那么为什么现在需要这么做? 这些都是很好的问题。事实证明, 我们需要知道CUDA设备的ID, 这样才可以告诉CUDA运行时应该使用哪个设备来执行CUDA和OpenGL。我们通过调用cudaGLSetGLDevice()来实现这个功能, 并把在cudaChooseDevice()中获得的设备ID dev传递进去。

```
HANDLE_ERROR( cudaGLSetGLDevice( dev ) );
```

在CUDA运行时初始化之后, 就可以继续调用GL工具箱 (GL Utility Toolkit, GLUT) 的设置函数来初始化OpenGL驱动程序。如果你之前使用过GLUT, 那么下面这些函数调用看上去就很熟悉:

```

// 在执行其他的GL调用之前, 需要首先执行这些GLUT调用。
glutInit( &argc, argv );
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
glutInitWindowSize( DIM, DIM );
glutCreateWindow( "bitmap" );

```

在main()的这个位置上, 我们通过调用cudaGLSetGLDevice()为CUDA运行时使用OpenGL驱动程序做好准备。然后, 我们初始化GLUT并且创建一个名为“bitmap”的窗口, 并将在这个窗口中绘制结果。现在, 我们可以开始执行实际的OpenGL互操作!

共享数据缓冲区是在CUDA C核函数和OpenGL渲染操作之间实现互操作的关键部分。要在

OpenGL和CUDA之间传递数据，我们首先要创建一个缓冲区在这两组API之间使用。首先在OpenGL中创建一个像素缓冲区对象，并将句柄保存在全局变量GLuint bufferObj中：

```
glGenBuffers( 1, &bufferObj );
glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, DIM * DIM * 4,
              NULL, GL_DYNAMIC_DRAW_ARB );
```

如果你从来没有用过OpenGL中的像素缓冲区对象（Pixel Buffer Object, PBO），那么可以通过以下三个步骤来创建一个：首先，通过glGenBuffers()生成一个缓冲区句柄。然后，通过glBindBuffer()将句柄绑定到像素缓冲区。最后，通过glBufferData()请求OpenGL驱动程序来分配一个缓冲区。在这个示例中请求分配一个缓冲区来保存DIM×DIM个32位的值，并且使用枚举值GL_DYNAMIC_DRAW_ARB来表示这个缓冲区将被应用程序反复修改。由于没有任何数据预先加载到缓冲区，因此将glBufferData()的倒数第二个参数设置为NULL。

在设置图形互操作性中，剩下的工作就是通知CUDA运行时，缓冲区bufferObj将在CUDA与OpenGL之间共享。要实现这个操作，需要将bufferObj注册为一个图形资源（Graphics Resource）。

```
HANDLE_ERROR(
    cudaGraphicsGLRegisterBuffer( &resource,
                                  bufferObj,
                                  cudaGraphicsMapFlagsNone )
);
```

通过调用cudaGraphicsGLRegisterBuffer()，我们告诉CUDA运行时希望在OpenGL和CUDA中使用OpenGL PBO bufferObj。CUDA运行时将在变量resource中返回一个句柄指向缓冲区。在随后对CUDA运行时的调用中，将通过这个句柄来访问bufferObj。

标志cudaGraphicsMapFlagsNone表示不需要为缓冲区指定特定的行为，当然我们也可以通过标志cudaGraphicsMapFlagsReadOnly将缓冲区指定为只读的。我们还可以通过标志cudaGraphicsMapFlagsWriteDiscard来指定缓冲区中之前的内容应该抛弃，从而使缓冲区变成只写的。这些标志使得CUDA和OpenGL驱动程序根据缓冲区的访问模式对硬件配置进行优化，当然这些标志并不一定必须设置。

对glBufferData()的调用需要OpenGL驱动程序分配一个足够大的缓冲区来保存DIM×DIM个32位的值。在随后的OpenGL调用中，我们通过bufferObj来引用这个缓冲区，而在CUDA运行时调用中，则通过指针resource来引用这个缓冲区。由于我们将在CUDA C核函数中对这个缓冲区进行读写，因此需要多个指向该对象的句柄。我们需要设备内存中的一个实际地址并传递给核函数。首先告诉CUDA运行时映射这个共享资源，然后请求一个指向被映射资源的指针。

```

uchar4* devPtr;
size_t size;
HANDLE_ERROR( cudaGraphicsMapResources( 1, &resource, NULL ) );
HANDLE_ERROR(
    cudaGraphicsResourceGetMappedPointer( (void**)&devPtr,
                                          &size,
                                          resource )
);

```

然后，可以把devPtr作为设备指针来使用，此外这个数据还可以作为一个像素源由OpenGL使用。在完成这些设置步骤后，main()剩余工作的执行流程为：首先，启动核函数并将指向共享缓冲区的指针传递给它。尽管我们还没有看到这个核函数的代码，但可以提前告诉你该核函数的作用是生成将要显示的图像数据。接下来，取消对共享资源的映射。一定要在执行绘制任务之前执行取消映射的调用，这是为了确保在应用程序的CUDA部分和图形部分之间实现同步。特别是，取消映射的调用将使得在cudaGraphicsUnmapResources()之前的所有CUDA操作完成之后，才会开始执行图形调用。

最后，我们通过GLUT注册键盘回调函数和显示回调函数（key_func 和 draw_func），并通过glutMainLoop()将执行控制交给GLUT绘制循环。

```

dim3 grids(DIM/16,DIM/16);
dim3 threads(16,16);
kernel<<<grids,threads>>>( devPtr );

HANDLE_ERROR( cudaGraphicsUnmapResources( 1, &resource, NULL ) );

// 设置好GLUT并启动循环
glutKeyboardFunc( key_func );
glutDisplayFunc( draw_func );
glutMainLoop();
}

```

这个应用程序的剩余部分包括三个函数，kernel()、key_func()、draw_func()。现在我们就来看这些函数。

核函数的参数包括一个设备指针，函数的任务是生成图像数据。在下面的示例中，我们将使用从第5章波纹示例中修改而来的核函数：

```

// 根据波纹代码修改而来，其中使用了uchar4类型，
// 这是图形交互使用的数据类型
__global__ void kernel( uchar4 *ptr ) {
    // 将threadIdx/BlockIdx映射到像素位置
    int x = threadIdx.x + blockIdx.x * blockDim.x;

```



```

int y = threadIdx.y + blockIdx.y * blockDim.y;
int offset = x + y * blockDim.x * gridDim.x;

// 现在计算这个位置上的值
float fx = x/(float)DIM - 0.5f;
float fy = y/(float)DIM - 0.5f;
unsigned char green = 128 + 127 *
                    sin( abs(fx*100) - abs(fy*100) );

// 此时访问的uchar4类型而不是unsigned char*类型
ptr[offset].x = 0;
ptr[offset].y = green;
ptr[offset].z = 0;
ptr[offset].w = 255;
}

```

这里用到了许多熟悉的知识。例如，将线程索引和块索引转换为x坐标和y坐标的方法，以及线性化偏移的方法等，在前面已经分析过多次了。然后，我们执行一些计算来判断位于(x, y)位置上像素的颜色，并将这些值保存到内存中。我们再次使用CUDA C在GPU上生成一张图像。需要注意的是，这张图像随后将在不需要CPU介入的情况下直接交给OpenGL。另一方面，在第5章的波纹示例中，在GPU上生成图像的方式与这里的方式非常相似，但应用程序随后需要将缓冲区复制回CPU以便显示。

那么，如何通过OpenGL来绘制CUDA生成的缓冲区？好的，回顾在main()中执行的设置过程，你会发现以下函数：

```
glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
```

这个调用将共享缓冲区绑定为一个像素源，OpenGL驱动程序随后会在所有对glDrawPixels()的调用中使用这个像素源。这意味着，我们需要调用glDrawPixels()来绘制CUDA C核函数生成的图像数据。因此，下面就是draw_func()需要执行的工作：

```

static void draw_func( void ) {
    glDrawPixels( DIM, DIM, GL_RGBA, GL_UNSIGNED_BYTE, 0 );
    glutSwapBuffers();
}

```

你可能已经发现glDrawPixels()的最后参数为一个缓冲区指针。如果没有任何缓冲区绑定为GL_PIXEL_UNPACK_BUFFER_ARB源，那么OpenGL驱动程序将从这个缓冲区中进行复制。然而，由于数据已经位于GPU上，并且我们已经将共享缓冲区绑定为GL_PIXEL_UNPACK_BUFFER_ARB源，因此最后一个参数将变成绑定缓冲区内的一个偏移。由于我们要绘制整个缓冲区，因此这个偏移值就是0。

示例代码的最后一部分看上去有些奇怪，这是因为我们决定为用户提供一种方法来退出应用程序。回调函数key_func()将响应Esc键，并将这个键作为释放内存并退出的信号：

```
static void key_func( unsigned char key, int x, int y ) {
    switch (key) {
        case 27:
            // 释放OpenGL和CUDA
            HANDLE_ERROR( cudaGraphicsUnregisterResource( resource ) );
            glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, 0 );
            glDeleteBuffers( 1, &bufferObj );
            exit(0);
    }
}
```

当运行这个示例时，将用“NVIDIA绿色（NVIDIA Green）”和黑色绘制一个具有催眠效果的图片，如图8.1所示。你可以尝试用这张图片对你的朋友（或者敌人）进行催眠。

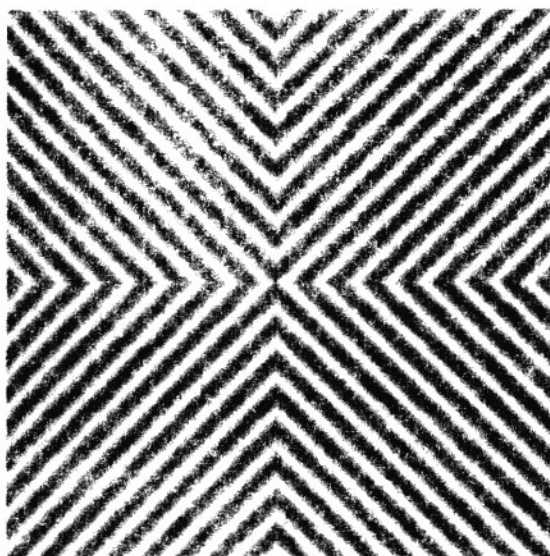


图8.1 基于图像互操作性的催眠图片示例

8.3 基于图形互操作性的GPU波纹示例

在8.2节中，我们多次提到了第5章的GPU波纹示例。在第5章的示例程序中创建了一个CPUAnimBitmap并将其传递给一个函数，每当需要生成一帧图像时都会调用这个函数。

```

int main( void ) {
    DataBlock    data;
    CPUAnimBitmap bitmap( DIM, DIM, &data );
    data.bitmap = &bitmap;

    HANDLE_ERROR( cudaMalloc( (void*)&data.dev_bitmap,
                               bitmap.image_size() ) );
    bitmap.anim_and_exit( (void (*)(void*,int))generate_frame,
                          (void (*)(void*))cleanup );
}

```

根据在前面章节中学到的技术，我们希望创建一个GPUAnimBitmap结构。这个结构的作用与CPUAnimBitmap相同，但在GPUAnimBitmap中，CUDA和OpenGL组件能够不需要CPU的介入而实现相互协作。当在应用程序中使用GPUAnimBitmap()时，main()函数将变得更加简单，如下所示：

```

int main( void ) {
    GPUAnimBitmap bitmap( DIM, DIM, NULL );

    bitmap.anim_and_exit(
        (void (*)(uchar4*,void*,int))generate_frame, NULL );
}

```

GPUAnimBitmap使用了在8.2一节中介绍的相同函数调用。然而，现在这些调用将被放入GPUAnimBitmap结构中，从而使示例（或者包括你自己的应用程序）变得更为简洁。

8.3.1 GPUAnimBitmap结构

在GPUAnimBitmap结构中有几个数据成员与8.2节中介绍的内容十分相似。

```

struct GPUAnimBitmap {
    GLuint    bufferObj;
    cudaGraphicsResource *resource;
    int       width, height;
    void      *dataBlock;
    void      (*fAnim)(uchar4*,void*,int);
    void      (*animExit)(void*);
    void      (*clickDrag)(void*,int,int,int,int);
    int       dragStartX, dragStartY;
}

```

我们知道OpenGL和CUDA运行时对于GPU缓冲区分别有着不同的名字，并且还知道在调用OpenGL或CUDA C时需要使用不同的名字。因此，在结构中同时保存了OpenGL的名字bufferObj和CUDA运行时名字的resource。此外，由于代码计算的是一张将要显示的位图图像，因此这张图像需要包含宽度和高度。

为了在使用GPUAnimBitmap时可以注册特定的回调事件，我们还保存了一个void*指针dataBlock指向用户的数据。然而，在GPUAnimBitmap中将不会访问这个数据，而只是将其传递给已注册的回调函数。用户注册的回调函数将保存在fAnim、animExit以及clickDrag中。在每次调用glutIdleFunc()时都将调用fAnim()，这个函数将负责生成在动画中绘制的图像数据。在动画退出时，将调用函数animExit()一次。用户应该在这个函数中实现当动画结束时需要执行的清理代码。最后还有一个可选函数clickDrag()，这个函数将响应用户的鼠标点击/拖曳等事件。如果用户注册了这个函数，那么每当按下、拖曳或者释放鼠标时，都将调用该函数。鼠标点击的初始位置保存在(dragStartX, dragStartY)中，当释放鼠标时，点击或拖曳等事件的起始位置和结束位置就会传递给用户。这可以用于实现令人印象深刻的交互式动画。

在初始化GPUAnimBitmap之后紧接着是在前面示例中相同的代码。在将参数保存到相应的结构成员中后，首先向CUDA运行时查询CUDA设备。

```
GPUAnimBitmap( int w, int h, void *d ) {
    width = w;
    height = h;
    dataBlock = d;
    clickDrag = NULL;
    // 首先找到一个CUDA设备并将其设置到图形互操作中
    cudaDeviceProp prop;
    int dev;
    memset( &prop, 0, sizeof( cudaDeviceProp ) );
    prop.major = 1;
    prop.minor = 0;
    HANDLE_ERROR( cudaChooseDevice( &dev, &prop ) );
```

在找到一个兼容的CUDA设备后，调用CUDA运行时中的cudaGLSetGLDevice()，并通知运行时将dev作为与OpenGL互操作的设备：

```
cudaGLSetGLDevice( dev );
```

由于接下来将使用GLUT来创建一个窗口绘制环境，因此需要初始化GLUT。然而，在这个过程中存在一个问题，因为glutInit()希望将命令行参数传递给窗口系统。由于我们不需要传递任何参数，因此可以指定0个命令行参数。然而，在某些版本的GLUT中存在一个错误，即当传递0个参数时应用程序会崩溃。因此，我们要欺骗GLUT，使其认为正在传递一个参数，这样就不会使程序崩溃。

```
int    c=1;
char    *foo = "name" ;
glutInit( &c, &foo );
```

接下来，继续像前面示例中一样初始化GLUT。我们创建了一个要绘制的窗口，并将窗口

的标题指定为字符串“bitmap”。当然，你也可以选择其他的名字作为窗口标题。

```
glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
glutInitWindowSize( width, height );
glutCreateWindow( "bitmap" );
```

接下来，我们请求OpenGL驱动程序分配一个缓冲区句柄，并将其绑定到GL_PIXEL_UNPACK_BUFFER_ARB，从而确保之后在调用glDrawPixels()时将绘制互操作缓冲区：

```
glGenBuffers( 1, &bufferObj );
glBindBuffer( GL_PIXEL_UNPACK_BUFFER_ARB, bufferObj );
```

最后，我们请求OpenGL驱动程序分配一块GPU内存。在完成分配操作后，将这个缓冲区通知给CUDA运行时，并通过`cudaGraphicsGLRegisterBuffer()`来注册`bufferObj`从而获得一个CUDA C名字。

```
glBufferData( GL_PIXEL_UNPACK_BUFFER_ARB, width * height * 4,
             NULL, GL_DYNAMIC_DRAW_ARB );
```

[illegible]

}

在设置好GPUAnimBitmap后，还剩下一个问题就是如何执行绘制操作。绘制操作的主要工作是在函数glutIdleFunction()中完成的。这个函数要做三件事情。首先，它将映射共享缓冲区并获得指向该缓冲区的GPU指针。

[illegible]

其次，它调用用户指定的函数fAnim()，这个函数可能启动一个CUDA C核函数并用图像数据来填充devPtr指向缓冲区。

```
bitmap->fAnim( devPtr, bitmap->dataBlock, ticks++ );
```

最后，它将取消GPU指针的映射，这会释放OpenGL驱动程序使用的缓冲区。绘制图像的操作将在调用`glutPostRedisplay()`时触发。

[illegible]

GPUAnimBitmap结构中还包括了一些重要但却有些不太相关的基础代码。如果你对这些代码感兴趣,可以对其进行分析。然而,即使没有时间或者兴趣来理解GPUAnimBitmap的剩余代码,也没关系,你仍然能够理解接下来的内容。

8.3.2 重新实现基于GPU的波纹动画示例

现在，我们已经实现了一个GPU版本的CPUAnimBitmap，接下来可以继续将基于GPU的波纹应用程序修改为完全在GPU上执行动画。首先要包含gpu_anim.h头文件，其中包含了GPUAnimBitmap的实现。我们还包含了与第5章中几乎相同的核函数。

[illegible]

```

(d/10.0f + 1.0f));

ptr[offset].x = grey;
ptr[offset].y = grey;
ptr[offset].z = grey;
ptr[offset].w = 255;
}

```

上面代码中的粗体部分是我们唯一进行的修改。这个修改的原因是因为OpenGL互操作要求共享内存满足图形操作的需求。由于在绘制过程中，每个元素通常包含4部分信息（红/绿/蓝/Alpha），因此目标缓冲区将不再像以前那样只是一个unsigned char数组。现在要求是一个uchar4类型的数组。在第5章中，我们将缓冲区的每四个字节作为一个数据，因此通过ptr[offset*4+k]对进行索引，其中k的取值范围是0到3。但现在，uchar4类型很好表达了在数据中包含四部分信息的语义。

由于CUDA函数kernel()将生成图像数据，因此剩下的工作就是编写一个主机函数，并将其作为回调函数保存在GPUAnimBitmap的成员idle_func()中。对于这里的应用程序，该函数执行的操作就是启动CUDA C核函数：

```

void generate_frame( uchar4 *pixels, void*, int ticks ) {
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( pixels, ticks );
}

```

由于所有的辅助工作都是在GPUAnimBitmap结构中完成的，因此这个函数的代码很简洁。然后，我们只需创建一个GPUAnimBitmap，并且注册动画回调函数generate_frame()。

```

int main( void ) {
    GPUAnimBitmap  bitmap( DIM, DIM, NULL );

    bitmap.anim_and_exit(
        (void (*)(uchar4*,void*,int))generate_frame, NULL );
}

```

8.4 基于图形互操作性的热传导

那么，前面介绍的这些内容的要点在哪里？如果观察在前面动画示例中使用的CPUAnimBitmap结构，我们会发现它的工作原理与8.2节中的绘制代码非常相似。

CPUAnimBitmap与前一个示例的关键差异在于对glDrawPixels()的调用。

```

glDrawPixels( bitmap->x,

```

```

    bitmap->y,
    GL_RGBA,
    GL_UNSIGNED_BYTE,
    bitmap->pixels );

```

在本章的第一个示例中，你可能已经看到了对glDrawPixels()的调用，其中最后的参数是一个缓冲区指针。如果你之前没有注意，那么现在就看到了。在调用CPUAnimBitmap的Draw()函数时，将把bitmap->pixels中的CPU缓存复制到GPU中以便进行绘制。为了实现这个操作，CPU需要停止正在执行的工作，并将每一帧都复制到GPU。这需要在CPU与GPU之间进行同步。此外，在PCI Express总线上启动和完成传输还会导致额外的延迟。由于在glDrawPixels()的调用中最后一个参数是主机指针，这就意味着，在通过CUDA C核函数生成一帧图像数据后，需要通过cudaMemcpy()将该帧从GPU复制到CPU。

```

void generate_frame( DataBlock *d, int ticks ) {
    dim3    grids(DIM/16,DIM/16);
    dim3    threads(16,16);
    kernel<<<grids,threads>>>( d->dev_bitmap, ticks );

    HANDLE_ERROR( cudaMemcpy( d->bitmap->get_ptr(),
                               d->dev_bitmap,
                               d->bitmap->image_size(),
                               cudaMemcpyDeviceToHost ) );
}

```

总的来说，在最初的GPU波纹应用程序中存在着许多可以改进的地方。这个程序使用CUDA C来计算在每一帧绘制的图像，但在计算完成后，我们将缓冲区复制到CPU，然后又将缓冲区复制回GPU以显示。可以看到，在主机与设备之间存在着不必要的数据迁移，使得程序无法实现最优性能。接下来，我们将重新回顾计算密集的动画应用程序，并通过图形互操作来实现绘制操作从而实现性能提升。

如果回顾第7章的热模拟应用程序，你会发现其中同样使用了CPUAnimBitmap来显示模拟计算的输出结果。我们将把这个应用程序修改为使用新实现的GPUAnimBitmap结构，并观察性能将发生怎样的变化。与波纹示例一样，GPUAnimBitmap可以很容易地替代CPUAnimBitmap，只需将unsigned char改为uchar4即可。因此，我们修改动画函数的原型从而适应这种在数据类型上的变化。

```

void anim_gpu( uchar4* outputBitmap, DataBlock *d, int ticks ) {
    HANDLE_ERROR( cudaEventRecord( d->start, 0 ) );
    dim3    blocks(DIM/16,DIM/16);
    dim3    threads(16,16);

```

```

// 由于tex是全局的并且有界的，我们必须使用一个标志来表示

```



```

// 在每次迭代中哪个是输入以及哪个是输出
volatile bool dstOut = true;
for (int i=0; i<90; i++) {
    float    *in, *out;
    if (dstOut) {
        in = d->dev_inSrc;
        out = d->dev_outSrc;
    } else {
        out = d->dev_inSrc;
        in = d->dev_outSrc;
    }
    copy_const_kernel<<<blocks,threads>>>( in );
    blend_kernel<<<blocks,threads>>>( out, dstOut );
    dstOut = !dstOut;
}
float_to_color<<<blocks,threads>>>( outputBitmap,
                                   d->dev_inSrc );
HANDLE_ERROR( cudaEventRecord( d->stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( d->stop ) );
float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                   d->start, d->stop ) );
d->totalTime += elapsedTime;
++d->frames;
printf( "Average Time per frame:  %3.1f ms\n",
        d->totalTime/d->frames );
}

```

由于核函数float_to_color()是唯一一个使用outputBitmap的函数，因此在将数据类型修改为uchar4后，只需对这个函数进行修改。在第7章中，只认为这个函数是一段辅助代码，在这里将仍然如此。但是，我们重载了这个函数，并在book.h中同时包含了unsigned char和uchar4的版本。你将注意到，这些函数间的差异与GPU波纹示例中不同版本kernel()之间的差异是类似的。为了简单，在这里省略了float_to_color()核函数的大部分代码，如果需要进一步观察，可以参考book.h。

```

__global__ void float_to_color( unsigned char *optr,
                               const float *outSrc ) {

    // 将浮点值转换为4个颜色值

    optr[offset*4 + 0] = value( m1, m2, h+120 );
    optr[offset*4 + 1] = value( m1, m2, h );
    optr[offset*4 + 2] = value( m1, m2, h -120 );

```

```

    optr[offset*4 + 3] = 255;
}
__global__ void float_to_color( uchar4 *optr,
                                const float *outSrc ) {

// 将浮点值转换为4个颜色值

    optr[offset].x = value( m1, m2, h+120 );
    optr[offset].y = value( m1, m2, h );
    optr[offset].z = value( m1, m2, h -120 );
    optr[offset].w = 255;
}

```

除了这些修改外，另一个主要的差异在于将CPUAnimBitmap修改为GPUAnimBitmap以便执行动画操作。

```

int main( void ) {
    DataBlock data;
    GPUAnimBitmap bitmap( DIM, DIM, &data );
    data.totalTime = 0;
    data.frames = 0;
    HANDLE_ERROR( cudaEventCreate( &data.start ) );
    HANDLE_ERROR( cudaEventCreate( &data.stop ) );

    int imageSize = bitmap.image_size();

    // 假设float类型的大小为4字符 (即rgba)
    HANDLE_ERROR( cudaMalloc( (void*)&data.dev_inSrc,
                               imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void*)&data.dev_outSrc,
                               imageSize ) );
    HANDLE_ERROR( cudaMalloc( (void*)&data.dev_constSrc,
                               imageSize ) );

    HANDLE_ERROR( cudaBindTexture( NULL, texConstSrc,
                                    data.dev_constSrc,
                                    imageSize ) );
    HANDLE_ERROR( cudaBindTexture( NULL, texIn,
                                    data.dev_inSrc,
                                    imageSize ) );

    HANDLE_ERROR( cudaBindTexture( NULL, texOut,
                                    data.dev_outSrc,
                                    imageSize ) );
}

```

```

// 初始化常量数据
float *temp = (float*)malloc( imageSize );
for (int i=0; i<DIM*DIM; i++) {
    temp[i] = 0;
    int x = i % DIM;
    int y = i / DIM;
    if ((x>300) && (x<600) && (y>310) && (y<601))
        temp[i] = MAX_TEMP;
}
temp[DIM*100+100] = (MAX_TEMP + MIN_TEMP)/2;
temp[DIM*700+100] = MIN_TEMP;
temp[DIM*300+300] = MIN_TEMP;
temp[DIM*200+700] = MIN_TEMP;
for (int y=800; y<900; y++) {
    for (int x=400; x<500; x++) {
        temp[x+y*DIM] = MIN_TEMP;
    }
}
HANDLE_ERROR( cudaMemcpy( data.dev_constSrc, temp,
                           imageSize,
                           cudaMemcpyHostToDevice ) );

// 初始化输入数据
for (int y=800; y<DIM; y++) {
    for (int x=0; x<200; x++) {
        temp[x+y*DIM] = MAX_TEMP;
    }
}

HANDLE_ERROR( cudaMemcpy( data.dev_inSrc, temp,
                           imageSize,
                           cudaMemcpyHostToDevice ) );

free( temp );

bitmap.anim_and_exit( (void (*)(uchar4*,void*,int))anim_gpu,
                     (void (*)(void*))anim_exit );
}

```

虽然在改进前后的热模拟程序中还存在一些差异，但它们并不重要，在这里也就不进一步讨论了。最重要的是回答这个问题：当我们把程序完全迁移到GPU上后，程序的性能将发生怎样的改变？现在，程序无需将每一帧都复制回主机进行显示，因此比之前的情况更为乐观。

那么，基于图形互操作行来实现图像绘制将获得哪种程度的性能提升？之前，在基于 GeForce GTX 285 的测试机器上，热传导示例大概每帧消耗25.3毫秒。在将应用程序修改为使用图形互操作性后，降低到了每帧21.6毫秒。结果就是绘制操作快了15%，并且在每次显示一

帧图像时不再需要主机的介入。这是个还不错的结果！

8.5 DirectX互操作性

虽然我们只讨论了与OpenGL之间的互操作性，但与DirectX之间的互操作性几乎是相同的。你仍然通过`cudaGraphicsResource`来表示在DirectX与CUDA之间共享的缓冲区，并且调用`cudaGraphicsMapResources()`和`cudaGraphicsResourceGetMappedPointer()`来获得这些共享资源的CUDA指针。

在很大程度上，OpenGL互操作性与DirectX互操作性之间的差异是很小的，因此要使用DirectX也是非常简单的。例如，在使用DirectX时，将调用`cudaD3D9SetDirect3DDevice()`而不是`cudaGLSetGLDevice()`来指定某个CUDA设备应该启用Direct3D 9.0互操作性。类似的是，`cudaD3D10SetDirect3DDevice()`启用设备支持Direct3D 10互操作性，而`udaD3D11SetDirect3DDevice()`启用设备支持Direct3D 11互操作性。

如果你学习并理解了本章的OpenGL示例，那么将不会对DirectX互操作性感到陌生。如果想通过DirectX互操作性来实现一个小项目，那么我建议你将本章的示例修改为使用DirectX。我们建议参考NVIDIA CUDA编程指南，了解这个API并阅读GPU Computing SDK中与DirectX互操作性相关的代码示例。

8.6 本章小结

虽然本书的大部分内容都是介绍如何使用GPU来实现并行的通用计算，但我们不能忘记GPU的本职工作是作为一个渲染引擎。在许多应用程序都需要使用标准的计算机图形渲染功能。GPU是图形渲染领域的主要设备，因此如果不了解CUDA运行时和图形驱动程序之间的协作，那么我们将无法充分利用这些资源的强大功能。现在，我们已经学习了这些内容，因此在显示计算得到的图形结果时将不再需要主机的介入。这不仅加速了应用程序的渲染循环，而且还使主机可以同时执行其他的计算。即使没有其他计算需要执行，那么我们的系统在响应其他时间或者应用程序时也将变得更加灵敏。

我们还可以通过许多其他方式来使用图形互操作性，但在这里没有介绍。我们主要介绍了如何使CUDA C核函数能够写入到窗口的显示缓冲区中。这里的图像数据还可以作为纹理应用于场景中的任意表面。除了可以修改像素缓冲区对象外，你还可以在CUDA和图形引擎之间共享顶点缓冲区对象。这样，在CUDA C核函数中可以执行对象间的碰撞检测，或者计算顶点位置映射，从而对与用户交互的对象或者表面进行渲染。如果你对计算机图形学感兴趣，那么CUDA C的图形互操作API能够为应用程序带来大量全新的功能！

第9章

原子性

在本书的前半部分，我们看到了许多任务在单线程应用程序中是难以计算的，但使用CUDA C却很容易实现。例如在动画演示或者热传导模拟等示例中，当使用CUDA运行时来执行幕后工作时，就不再需要for()循环对每个像素依次更新。类似的是，只需在主机代码中调用__global__函数，就可以创建数千个并行的线程块和线程。

然而，在某些情况中，对于单线程应用程序来说非常简单的任务，在大规模并行架构上实现时却会变成一个复杂的问题。在本章中，我们将看到其中一些情况，并在这些情况中使用特殊的原语从而确保安全地完成传统单线程应用程序中的简单任务。

9.1 本章目标

通过本章的学习，你可以：

- 了解不同NVIDIA GPU的计算功能集。
- 了解原子操作以及为什么需要使用它们。
- 了解如何在CUDA C核函数中执行带有原子操作的运算。

9.2 计算功能集

到目前为止，我们介绍的是所有支持CUDA的GPU的通用功能。例如，所有支持CUDA架构的GPU都可以启动核函数，访问全局内存，以及读取常量内存和纹理内存。但正如不同架构的CPU有着不同的功能和指令集（例如MMX、SSE、SSE2等），对于支持CUDA的不同图形处理器来说同样如此。NVIDIA将GPU支持的各种功能统称为计算功能集（Compute Capability）。

9.2.1 NVIDIA GPU的计算功能集

到本书截稿时间为止，NVIDIA GPU可以支持计算功能集包括1.0、1.1、1.2、1.3以及2.0。高版本计算功能集是低版本计算功能集的超集，类似于一种“洋葱式”或者“俄罗斯套娃玩具式”的嵌套结构（你可以选择自己喜欢的比喻）。例如，支持1.2版本计算功能集的GPU同样支持1.0版本和1.1版本的所有功能。在NVIDIA CUDA编程指南中包含了所有支持CUDA的GPU的最新列表，以及它们相应的计算功能集。表9.1列出了在截稿时所有可用的NVIDIA GPU，同时还列出了每个GPU支持的计算功能集。

表9.1 支持CUDA的GPU及其相应的计算功能集

GPU	计算功能集
GeForce GTX 480, GTX 470	2.0
GeForce GTX 295	1.3
GeForce GTX 285, GTX 280	1.3
GeForce GTX 260	1.3
GeForce 9800 GX2	1.1
GeForce GTS 250, GTS 150, 9800 GTX, 9800 GTX+, 8800 GTS 512	1.1
GeForce 8800 Ultra, 8800 GTX	1.0
GeForce 9800 GT, 8800 GT, GTX 280M, 9800M GTX	1.1
GeForce GT 130, 9600 GSO, 8800 GS, 8800M GTX, GTX 260M, 9800M GT	1.1
GeForce 8800 GTS	1.0
GeForce 9600 GT, 8800M GTS, 9800M GTS	1.1
GeForce 9700M GT	1.1

(续)

GPU	计算功能集
GeForce GT 120, 9500 GT, 8600 GTS, 8600 GT, 9700M GT, 9650M GS, 9600M GT, 9600M GS, 9500M GS, 8700M GT, 8600M GT, 8600M GS	1.1
GeForce G100, 8500 GT, 8400 GS, 8400M GT, 9500M G, 9300M G, 8400M GS, 9400 mGPU, 9300 mGPU, 8300 mGPU, 8200 mGPU, 8100 mGPU	1.1
GeForce 9300M GS, 9200M GS, 9100M G, 8400M G	1.1
Tesla S2070, S2050, C2070, C2050	2.0
Tesla S1070, C1060	1.3
Tesla S870, D870, C870	1.0
Quadro Plex 2200 D2	1.3
Quadro Plex 2100 D4	1.1
Quadro Plex 2100 Model S4	1.0
Quadro Plex 1000 Model IV	1.0
Quadro FX 5800	1.3
Quadro FX 4800	1.3
Quadro FX 4700 X2	1.1
Quadro FX 3700M	1.1
Quadro FX 5600	1.0
Quadro FX 3700	1.1
Quadro FX 3600M	1.1
Quadro FX 4600	1.0
Quadro FX 2700M	1.1
Quadro FX 1700, FX 570, NVS 320M, FX 1700M, FX 1600M, FX 770M, FX 570M	1.1
Quadro FX 370, NVS 290, NVS 140M, NVS 135M, FX 360M	1.1
Quadro FX 370M, NVS 130M	1.1

当然，由于NVIDIA一直在发布新的图形处理器，因此在本书出版时，这张列表肯定会过时。幸运的是，NVIDIA公布了一个网络站点，在这个站点的CUDA Zone中给出了支持CUDA的最新设备列表。如果你没有在表9.1中找到你使用的GPU，那么我建议你参考这个列表，或者也可以运行第3章的示例程序将系统中每个CUDA设备的计算功能集显示出来。

由于本章的内容是关于原子性的，因此与这部分内容相关的就是硬件在内存上执行原子操作的能力。在了解原子操作是什么以及为什么要使用原子操作之前，你需要知道，只有1.1或者更高版本的GPU计算功能集才能支持全局内存上的原子操作。此外，只有1.2或者更高版本的GPU计算功能集才能支持共享内存上的原子操作。由于高版本计算功能集是低版本计算功能集的超集，因此计算功能集为1.2的GPU既支持共享内存原子操作又支持全局内存原子操作。同样，计算功能集为1.3的GPU也支持这两种原子操作。

如果你使用的GPU的计算功能集为1.0版本，那么它不支持全局内存上的原子操作，因此需要首先升级硬件设备。如果你的图形处理器支持原子操作，那么可以继续阅读下面的内容。但

如果你发现无法运行这些示例，那么可以直接跳到下一章。

9.2.2 基于最小计算功能集的编译

假设在编写的代码中要求计算功能集的版本最低不能低于某个版本。例如，假设你阅读完本章，并开始编写一个需要使用全局内存原子操作的应用程序。你知道要支持全局内存原子操作，计算功能集的最低版本为1.1。当编译代码时，你需要告诉编译器，如果硬件支持的计算功能集的版本低于1.1，那么将无法运行这个核函数。而且，当告诉编译器这个要求时，还可以指定一些只有在1.1或者更高版本的计算功能集中才支持的编译优化。要将这个信息告诉编译器，只需在调用nvcc时增加一个命令行选项：

```
nvcc -arch=sm_11
```

同样的，在编译需要使用共享内存原子操作的核函数时，你要告诉编译器代码需要1.2版本或者更高的计算功能集。

```
nvcc -arch=sm_12
```

9.3 原子操作简介

在编写传统的单线程应用程序时，程序员通常不需要使用原子操作。即使需要使用原子操作也无需担心，我们接下来将详细解释原子操作是什么，以及为什么在多线程程序中需要使用它们。为了解释原子操作，我们首先来分析C或者C++的基础知识之一，即递增运算符：

```
x++;
```

这在标准C中的一个表达式，在执行完这条语句后，x的值应该比执行这条语句之前的值大1。这条语句中包含了哪些操作？要将x的值加1，首先要知道x的当前值。只有读取了x的值之后，才可以对x进行递增。最后，我们要将递增后的值写回x。

因此，在这个操作中包含了三个步骤，如下所示：

- 1) 读取x中的值。
- 2) 将步骤1中读到的值增加1。
- 3) 将递增后的结果写回到x。

有时候，这个过程也称为读取-修改-写入（Read-Modify-Write）操作，其中第2步的递增操作也可以换成其他修改x值的操作。

现在考虑一种情况：有两个线程都需要对x的值进行递增。我们将这两个线程分别称为A和B。A和B在递增x的值时都需要执行上面三个操作。假设x的初始值为7。在理想情况下，我们

希望线程A和线程B执行表9.2中的步骤。

表9.2 两个线程同时递增x的值

步 骤	示 例
1) 线程A读取x中的值	A从x中读到7
2) 线程A将读到的值增加1	A计算得到8
3) 线程A将结果写回到x	$x \leftarrow 8$
4) 线程B读取x中的值	B从x中读到8
5) 线程B将读到的值增加1	B计算得到9
6) 线程B将结果写回到x	$x \leftarrow 9$

由于x的起始值为7, 并且由两个线程进行递增, 因此在递增运算完成后, x的值变为9。根据前面的操作顺序, 这确实是我们得到的结果。遗憾的是, 除了这个操作顺序外, 还存在其他一些操作顺序可能导致错误的结果。例如, 考虑表9.3中的顺序, 其中线程A和线程B的操作彼此交叉进行。

表9.3 两个线程递增x中的值, 并且彼此的操作相互交叉

步 骤	示 例
线程A读取x中的值	A从x中读到7
线程B读取x中的值	B从x中读到7
线程A将其读到的值增加1	A计算得到8
线程B将其读到的值增加1	B计算得到8
线程A将结果写回到x	$x \leftarrow 8$
线程B将结果写回到x	$x \leftarrow 8$

因此, 如果线程的调度方式不正确, 那么最终将得到错误的结果。除了上面两种执行顺序外, 这6个步骤还有许多其他的排序方式, 其中有些方式能产生正确的结果, 而其他的方式则不能。当把程序从单线程改写为多线程时, 如果多个线程需要对共享值进行读取或者写入时, 那么很可能会遇到不可预测的结果。

在前面的示例中, 我们需要通过某种方式一次性地执行完读取-修改-写入这三个操作, 并且在执行过程中不会被其他线程中断。具体来说, 除非已经完成了这三个操作, 否则其他的线程都不能读取或者写入x的值。由于这些操作的执行过程不能分解为更小的部分, 因此我们将满足这种条件限制的操作称为原子操作。CUDA C支持多种原子操作, 当有数千个线程在内存访问上发生竞争时, 这些操作能够确保在内存上实现安全的操作。

现在, 我们已经看到了一个只有使用原子操作才能计算出正确结果的示例。

9.4 计算直方图

通常，有些算法需要计算某个数据集的直方图。如果之前没有计算或使用过直方图，也不用担心，直方图是个很简单的概念。给定一个包含一组元素的数据集，直方图表示每个元素的出现频率。例如，如果计算词组“Programming with CUDA C”中字符频率的直方图，那么将得到图9.1所示的结果。

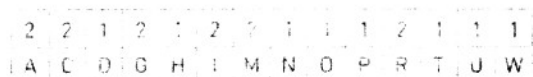


图9.1 字符串“Programming with CUDA C”中字符频率直方图

虽然直方图的定义很简单，但却在计算机科学领域得到了非常广的应用。在各种算法中都用了直方图，包括图像处理、数据压缩、计算机视觉、机器学习、音频编码等等。我们将把直方图计算作为接下来代码示例的算法。

9.4.1 在CPU上计算直方图

由于并非所有的读者都熟悉直方图的计算，因此我们首先将给出如何在CPU上计算直方图的示例。这个示例同时也说明了在单线程应用程序中计算直方图是非常简单的。这个应用程序将处理一个大型的数据流。在实际程序中，这个数据可以是像素的颜色值，或者音频采样数据，但在我们的示例程序中，这个数据将是随机生成的字节流。我们可以通过工具函数big_random_block()来生成这个随机的字节流。在应用程序中将生成100MB的随机数据。

```
#include "../common/book.h"

#define SIZE    (100*1024*1024)

int main( void ) {
    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
```

由于每个随机字节（8比特）都有256个不同的可能取值（从0x00到0xFF），因此在直方图中需要包含256个元素，每个元素记录相应的值在数据流中的出现次数。我们创建了一个包含256个元素的数组，并将所有元素的值初始化为0。

```
unsigned int    histo[256];
for (int i=0; i<256; i++)
    histo[i] = 0;
```

在创建了直方图并将数组元素初始化为0后，接下来需要计算每个值在buffer[]数据中的出现频率。算法的思想是，每当在数组buffer[]中出现某个值z时，就递增直方图数组中索引为z的元素。这样就能计算出值z的出现次数。

如果当前看到的值为`buffer[i]`，那么将递增数组中索引等于`buffer[i]`的元素。由于元素`buffer[i]`位于`histo[buffer[i]]`，我们只需一行代码就可以递增相应的计数器。

```
histo[buffer[i]]++;
```

我们在一个简单的`for()`循环中对`buffer[]`中的每个元素执行这个操作：

```
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]++;
```

此时，我们已经计算完了输入数据的直方图。在实际的应用程序中，这个直方图可能作为下一个计算步骤的输入数据。但在这里的简单示例中，这就是要执行的所有工作，因此接下来将验证直方图的所有元素相加起来是否等于正确的值，然后结束程序。

```
long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( "Histogram Sum: %ld\n", histoCount );
```

思考一下就会发现，无论输入数组的值是什么，这个和值总是相同的。每个元素都将统计相应数值的出现次数，因此所有这些元素值的总和就应该等于数组中元素的总数量。在示例中，这个值就等于`SIZE`。

无需多言（但我们总是会提醒），在执行完运算后需要释放内存并返回。

```
free( buffer );
return 0;
}
```

我们在一台Core 2 Duo机器上测试这个程序，数组大小为100MB，数组直方图的计算时间为0.416秒。这相当于一个基线性能，稍后我们将把这个性能与GPU版本程序的性能进行比较。

9.4.2 在GPU上计算直方图

我们把这个直方图计算示例改在GPU上运行。如果输入的数组足够大，那么通过由多个线程来处理缓冲区不同部分，将节约大量的计算时间。其中，由不同的线程来读取不同部分的输入数据是非常容易的。这与我们到目前为止看到的示例都非常相似。在计算输入数组的直方图时存在一个问题，即多个线程可能同时对输出直方图的同一个元素进行递增。在这种情况下，我们需要通过原子的递增操作来避免9.3节中提到的问题。

`main()`函数非常类似于CPU版本的函数，只是我们需要增加一些CUDA C的代码，包括将输入数据放入GPU以及从GPU得到结果。`main()`函数的开头与基于CPU的版本完全一样：

```
int main( void ) {
    unsigned char *buffer = (unsigned char*)big_random_block( SIZE );
```

由于要测量代码的执行性能，因此我们将初始化计时事件。

```
cudaEvent_t    start, stop;
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

在设置好输入数据和事件后，我们需要在GPU上为随机输入数据和输出直方图分配内存空间。在分配了输入缓冲区后，我们将big_random_block()生成的数组复制到GPU上。同样，在分配了直方图后，像CPU版本中那样将其初始化为0。

```
// 在GPU上为文件的数据分配内存
unsigned char *dev_buffer;
unsigned int *dev_histo;
HANDLE_ERROR( cudaMalloc( (void*)&dev_buffer, SIZE ) );
HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE,
                          cudaMemcpyHostToDevice ) );

HANDLE_ERROR( cudaMalloc( (void*)&dev_histo,
                          256 * sizeof( int ) ) );
HANDLE_ERROR( cudaMemset( dev_histo, 0,
                          256 * sizeof( int ) ) );
```

你可能已经注意到，在上面的代码中引入了一个新的CUDA 运行时函数，cudaMemset()。这个函数的原型与标准C函数memset()的原型是相似的，并且这两个函数的行为也基本相同。二者的差异在于，cudaMemset()将返回一个错误码，而C库函数memset()则不是。这个错误码将告诉调用者在设置GPU内存时发生的错误。除了返回错误码外，还有一个不同之处就是，cudaMemset()是在GPU内存上执行，而memset()是在主机内存上运行。

在初始化输入缓冲区和输出缓冲区后，就做好了计算直方图的准备。你马上就会看到如何准备并启动直方图核函数。我们暂时假设已经在GPU上计算好了直方图。在计算完成后，需要将直方图复制回CPU，因此我们分配了一个包含256个元素的数组，并且执行从设备到主机的复制。

```
unsigned int    histo[256];
HANDLE_ERROR( cudaMemcpy( histo, dev_histo,
                          256 * sizeof( int ),
                          cudaMemcpyDeviceToHost ) );
```

此时，我们完成了直方图的计算，因此可以停止计时器并显示经历的时间。这与在前面几章中使用的计时代码基本相同。

```
// 得到停止时间并显示计时结果
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );
printf( "Time to generate:  %3.1f ms\n", elapsedTime );
```

此时，我们可以将直方图作为输入数据传递给算法的下一个步骤。然而，在这个示例中不需要将直方图用于其他任何操作，而只是验证在GPU上计算得到的直方图与在CPU上计算得到的直方图是否相同。首先，我们验证直方图的总和等于正确的值。这与CPU版本中的代码是相同的，如下所示：

```
long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( "Histogram Sum:  %ld\n", histoCount );
```

为了验证在GPU上计算得到的直方图是正确的，我们将使用CPU来计算同一个直方图。一种简单的方式是分配一个新的直方图数组，使用9.4.1节中的代码来计算输入数据，并且最后验证GPU直方图中的各个元素与CPU中各个元素的值相等。然而，我们在这个示例中并没有分配一个新的直方图数组，而是首先在GPU上计算直方图，然后以“逆序”方式在CPU上计算直方图。

以“逆序”方式计算直方图意味着，首先计算出GPU直方图，并在遍历每个数值时，递减直方图中相应元素的值。因此，如果完成计算时直方图中每个元素的值都为0，那么CPU计算的直方图与GPU计算的直方图相等。从某种意义上来看，我们是在计算这两个直方图之间的差异。这段代码看上去非常像CPU直方图计算，但使用的是递减运算符而不是递增运算符。

```
// 验证与CPU得到的是相同的计数值
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]--;
for (int i=0; i<256; i++) {
    if (histo[i] != 0)
        printf( "Failure at %d!\n", i );
}
```

和通常一样，在程序结束时要释放已分配的CUDA事件，GPU内存和主机内存。

```
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
cudaFree( dev_histo );
```

```

    cudaFree( dev_buffer );
    free( buffer );
    return 0;
}

```

之前，我们假设已经通过一个核函数计算好了直方图。出于性能的考虑，这个示例中的核函数调用比通常的核函数调用要更复杂一些。由于直方图包含了256个元素，因此可以在每个线程块中包含256个线程，这种方式不仅方便而且高效。但是，在线程块的数量上还可以有更多选择。例如，在100MB数据中共有104 857 600个字节。我们可以启动一个线程块，并且让每个线程处理409 600个数据元素。同样，我们还可以启动409 600个线程块，并且让每个线程处理一个数据元素。

你可能已经猜到了，最优的接近方案是在这两种极端情况之间。通过一些性能实验，我们发现当线程块的数量为GPU中处理器数量的2倍时，将达到最优性能。例如，在GeForce GTX 280中包含了30个处理器，因此当启动60个并行线程块时，直方图核函数将运行得最快。

在第3章中，我们曾介绍了一种方法来查询GPU硬件设备的各种属性。如果要基于当前的硬件平台来动态调整线程块的数量，那么就要用到其中一个设备属性。我们通过以下代码片段来实现这个操作。虽然还没有看到这个核函数的实现，但你应该能理解它要执行的任务。

```
cudaDeviceProp prop;
HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
int blocks = prop.multiProcessorCount;
histo kernel<<<blocks*2,256>>>( dev buffer, SIZE, dev histo );
```

综上所述，以下给出了完整的main()函数：

```
int main( void ) {
    unsigned char *buffer =
        (unsigned char*)big_random_block( SIZE );

    cudaEvent_t      start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    // 在GPU上为文件的数据分配内存
    unsigned char *dev_buffer;
    unsigned int  *dev_histo;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_buffer, SIZE ) );
    HANDLE_ERROR( cudaMemcpy( dev_buffer, buffer, SIZE,
        cudaMemcpyHostToDevice ) );
}
```

```

HANDLE_ERROR( cudaMalloc( (void**)&dev_histo,
                          256 * sizeof( int ) ) );
HANDLE_ERROR( cudaMemset( dev_histo, 0,
                          256 * sizeof( int ) ) );
cudaDeviceProp prop;
HANDLE_ERROR( cudaGetDeviceProperties( &prop, 0 ) );
int blocks = prop.multiProcessorCount;
histo_kernel<<<blocks*2,256>>>( dev_buffer, SIZE, dev_histo );

unsigned int    histo[256];
HANDLE_ERROR( cudaMemcpy( histo, dev_histo,
                          256 * sizeof( int ),
                          cudaMemcpyDeviceToHost ) );

// 得到停止时间并显示计时结果
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                    start, stop ) );
printf( "Time to generate: %3.1f ms\n", elapsedTime );

long histoCount = 0;
for (int i=0; i<256; i++) {
    histoCount += histo[i];
}
printf( "Histogram Sum: %ld\n", histoCount );

// 验证与基于CPU计算得到的结果是相同的
for (int i=0; i<SIZE; i++)
    histo[buffer[i]]--;
for (int i=0; i<256; i++) {
    if (histo[i] != 0)
        printf( "Failure at %d!\n", i );
}

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
cudaFree( dev_histo );
cudaFree( dev_buffer );
free( buffer );
return 0;
}

```

1. 使用全局内存原子操作的直方图核函数

现在来分析有趣的部分：在GPU上计算直方图的代码！计算直方图的核函数需要的参数包括，一个指向输入数组的指针，输入数组的长度，以及一个指向输出直方图的指针。核函数执行的第一个计算就是计算输入数据数组中的偏移。每个线程的起始偏移都是0到线程数量减1之间的某个值。然后，对偏移的增量为已启动线程的总数。我们希望你记住这项技术。当你第一次学习线程时，我们使用了相同的方法将任意长度的矢量相加起来。

```
#include "../common/book.h"

#define SIZE      (100*1024*1024)

__global__ void histo_kernel( unsigned char *buffer,
                               long size,
                               unsigned int *histo ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

当每个线程知道它的起始偏移*i*以及递增的数量，这段代码将遍历输入数组，并递增直方图中相应元素的值。

```
    while ( i < size ) {
        atomicAdd( &(amp;histo[buffer[i]]), 1 );
        i += stride;
    }
}
```

粗体代码行表示在CUDA C中使用原子操作的方式。函数调用atomicAdd(addr, y)将生成一个原子的操作序列，这个操作序列包括读取地址addr处的值，将y增加到这个值，以及将结果保存回地址addr。底层硬件将确保当执行这些操作时，其他任何线程都不会读取或写入地址addr上的值，这样就能确保得到预计的结果。在这里的示例中，这个地址就是直方图中相应元素的位置。如果当前字节为buffer[i]，那么直方图中相应的元素就是histo[buffer[i]]。原子操作需要这个元素的地址，因此第一个参数为&(histo[buffer[i]])。由于我们只是想把这个元素中的值递增1，因此第二个参数就是1。

在分析完这些要点后，GPU版本的直方图计算就非常类似于CPU版本的直方图计算。

```
#include "../common/book.h"

#define SIZE      (100*1024*1024)

__global__ void histo_kernel( unsigned char *buffer,
                               long size,
```



```

        unsigned int *histo ) {
    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
    while (i < size) {
        atomicAdd( &(histo[buffer[i]]), 1 );
        i += stride;
    }
}

```

然而，我们现在还不要高兴得太早。在运行这个示例时，我们发现在GeForce GTX 285的测试机器上，从100MB的输入数据中构造直方图需要1.752秒。与基于CPU的直方图计算相比，你会发现这个性能居然更糟糕。事实上，GPU版本的性能比CPU版本的性能要慢四倍，这就是我们为什么要设置一个基线性能标准。如果只是因为程序能在GPU上运行而就对这种低性能的实现感到满足，那么我们应该感到惭愧。

由于在核函数中只包含了非常少的计算工作，因此很可能是全局内存上的原子操作导致了性能的降低。当数千个线程尝试访问少量的内存位置时，将发生大量的竞争。为了确保递增操作的原子性，对相同内存位置的操作都将被硬件串行化。这可能导致保存未完成操作的队列非常长，因此会抵消通过并行运行线程而获得的性能提升。我们需要改进算法本身从而重新获得这种性能。

2. 使用共享内存原子操作和全局内存原子操作的直方图核函数

尽管这些原子操作是导致这种性能降低的原因，但解决这个问题的方法却出乎意料地需要使用更多而非更少的原子操作。这里的主要问题并非在于使用了过多的原子操作，而是有数千个线程在少量的内存地址上发生竞争。要解决这个问题，我们将直方图计算分为两个阶段。

在第一个阶段中，每个并行线程块将计算它所处理数据的直方图。由于每个线程块在执行这个操作时都是相互独立的，因此可以在共享内存中计算这些直方图，这将避免每次将写入操作从芯片发送到DRAM。但是，这种方式仍然需要原子操作，因为在线程块中的多个线程之间仍然会处理相同值的数据元素。然而，现在只有256个线程在256个地址上发生竞争，这将极大地减少在使用全局内存时在数千个线程之间发生竞争的情况。

然后，在第一个阶段中分配一个共享内存缓冲区并进行初始化，用来保存每个线程块的临时直方图。回顾第5章，由于随后的步骤将包括读取和修改这个缓冲区，因此需要调用__syncthreads()来确保每个线程的写入操作在线程继续前进之前完成。

```

__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {

    __shared__ unsigned int temp[256];

```

```
temp[threadIdx.x] = 0;
__syncthreads();
```

在将直方图初始化为0后，下一个步骤与最初GPU版本的直方图计算非常类似。这里唯一的差异在于，我们使用了共享内存缓冲区temp[]而不是全局内存缓冲区histo[]，并且需要随后调用__syncthreads()来确保提交最后的写入操作。

```
int i = threadIdx.x + blockIdx.x * blockDim.x;
int offset = blockDim.x * gridDim.x;
while (i < size) {
    atomicAdd( &temp[buffer[i]], 1);
    i += offset;
}
__syncthreads();
```

在修改后的直方图示例程序中，最后一步要求将每个线程块的临时直方图合并到全局缓冲区histo[]中。假设将输入数据分为两半，这样就有两个线程查看不同部分的数据，并计算得到两个独立的直方图。如果线程A在输入数据中发现字节0xFC出现了20次，线程B发现字节0xFC出现了5次，那么字节0xFC在输入数据中共出现了25次。同样，最终直方图的每个元素只是线程A直方图中相应元素和线程B直方图中相应元素的加和。这个逻辑可以扩展到任意数量的线程，因此将每个线程块的直方图合并为单个最终的直方图就是，将线程块的直方图的每个元素都相加到最终直方图中相应位置的元素上。这个操作需要自动完成：

```
atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}
```

由于我们使用了256个线程，并且直方图中包含了256个元素，因此每个线程将自动把它计算得到的元素只增加到最终直方图的元素上。如果线程数量不等于元素数量，那么这个阶段将更为复杂。注意，我们并不保证线程块将按照何种顺序将各自的值相加到最终直方图中，但由于整数加法是可交换的，无论哪种顺序都会得到相同的结果。

这就是我们的两阶段直方图计算核函数，以下是完整的代码：

```
__global__ void histo_kernel( unsigned char *buffer,
                             long size,
                             unsigned int *histo ) {
    __shared__ unsigned int temp[256];
    temp[threadIdx.x] = 0;
    __syncthreads();

    int i = threadIdx.x + blockIdx.x * blockDim.x;
    int offset = blockDim.x * gridDim.x;
    while (i < size) {
```

```

        atomicAdd( &temp[buffer[i]], 1);
        i += offset;
    }

    __syncthreads();
    atomicAdd( &(histo[threadIdx.x]), temp[threadIdx.x] );
}

```

这个直方图示例比之前的GPU版本在性能上有了极大的提升。通过使用共享内存，程序在 GeForce GTX 285 上的执行时间为0.057秒。这不仅比单纯使用全局内存原子操作的版本要好很多，而且比最初CPU实现也要高一个数量级（从0.416秒提升到0.057秒），大约提升了7倍多。因此，尽管在最初将这个直方图修改为在GPU上实现反而使性能下降，但如果同时使用了共享原子操作和全局原子性，那么仍然能实现性能的提升。

9.5 本章小结

虽然我们经常提到通过CUDA C来编写并行程序是多么容易，但在很大程度上忽略了一些情况，而在这些情况中的大规模并行架构（例如GPU）将使得程序员的工作更为困难。例如，有成千上万个线程同时修改同一个内存地址，在这种情况下，大规模的并行机器反而会带来负担。幸运的是，在硬件中支持的原子操作可以帮助减轻这种痛苦。

然而，正如在直方图计算中看到的，有时候依赖原子操作会带来性能问题，并且这些问题只能通过对算法的某些部分进行重构来加以解决。但是在直方图示例中，我们使用了一种两阶段算法，该算法降低了在全局内存访问上竞争程度。通常，这种降低内存竞争程度的策略总能带来不错的效果，因此当你在自己的应用程序中使用原子操作时，要记住这种策略。

第10章

流

在本书中已经多次看到，对于大规模数据的并行计算，代码在GPU上执行的性能要远远高于在CPU上执行的性能。除此之外，NVIDIA图形处理器还支持另一种类型的并行性（Parallelism）。这种并行性类似于CPU多线程应用程序中的任务并行性（Task Parallelism）。任务并行性是指并行执行两个或多个不同的任务，而并不是在大量数据上执行同一个任务。

在并行性环境中，任务可以是任意操作。例如，应用程序可以执行两个任务：其中一个线程重绘程序的GUI，而另一个线程通过网络下载更新包。这些任务并行执行，彼此之间没有任何共同的地方。虽然GPU上的任务并行性并不像CPU上的任务并行性那样灵活，但仍然可以进一步提高程序在GPU上的运行速度。在本章中，我们将介绍CUDA流，以及如何通过流在GPU上同时执行多个任务。

10.1 本章目标

通过本章的学习，你可以：

- 了解如何分配页锁定（Page-Locked）类型的主机内存。
- 了解CUDA流的概念。
- 了解如何使用CUDA流来加速应用程序。

10.2 页锁定主机内存

在前9章的各个示例中，都是通过`cudaMalloc()`在GPU上分配内存，以及通过标准的C库函数`malloc()`在主机上分配内存。除此之外，CUDA运行时还提供了自己独有的机制来分配主机内存：`cudaHostAlloc()`。如果`malloc()`已经能很好地满足C程序员的需求，那么为什么还要使用这个函数？

事实上，`malloc()`分配的内存与`cudaHostAlloc()`分配的内存之间存在着一个重要差异。C库函数`malloc()`将分配标准的，可分页的（Pagable）主机内存，而`cudaHostAlloc()`将分配页锁定的主机内存。页锁定内存也称为固定内存（Pinned Memory）或者不可分页内存，它有一个重要的属性：操作系统将不会对这块内存分页并交换到磁盘上，从而确保了该内存始终驻留在物理内存中。因此，操作系统能够安全地使某个应用程序访问该内存的物理地址，因为这块内存将不会被破坏或者重新定位。

由于GPU知道内存的物理地址，因此可以通过“直接内存访问（Direct Memory Access, DMA）”技术来在GPU和主机之间复制数据。由于DMA在执行复制时无需CPU的介入，这也就同样意味着，CPU很可能在DMA的执行过程中将目标内存交换到磁盘上，或者通过更新操作系统的可分页表来重新定位目标内存的物理地址。CPU可能会移动可分页的数据，这就可能对DMA操作造成延迟。因此，在DMA复制过程中使用固定内存是非常重要的。事实上，当使用可分页内存进行复制时，CUDA驱动程序仍然会通过DAM把数据传输给GPU。因此，复制操作将执行两遍，第一遍从可分页内存复制到一块“临时的”页锁定内存，然后再从这个页锁定内存复制到GPU上。

因此，每当从可分页内存中执行复制操作时，复制速度将受限于PCIE传输速度和系统前端总线速度相对较低的一方。在某些系统中，这些总线在带宽上有着巨大的差异。因此当在GPU和主机间复制数据时，这种差异会使页锁定主机内存的性能比标准可分页内存的性能要高大约2倍。即使PCIE的速度与前端总线的速度相等，由于可分页内存需要更多一次由CPU参与的复制操作，因此会带来额外的开销。

然而，你也不能进入另一个极端：查找每个`malloc`调用并将其替换为`cudaHostAlloc()`调用。

固定内存是一把双刃剑。当使用固定内存时，你将失去虚拟内存的所有功能。特别是，在应用程序中使用每个页锁定内存时都需要分配物理内存，因为这些内存不能交换到磁盘上。这意味着，与使用标准的malloc()调用相比，系统将更快地耗尽内存。因此，应用程序在物理内存较少的机器上会运行失败，而且意味着应用程序将影响在系统上运行的其他应用程序的性能。

这些情况并不是说不使用cudaHostAlloc()，而是提醒你你应该清楚页锁定内存的隐含作用。我们建议，仅对cudaMemcpy()调用中的源内存或者目标内存，才使用页锁定内存，并且在不再需要使用它们时立即释放，而不是等到应用程序关闭时才释放。cudaHostAlloc()与到目前为止学习的其他内容一样简单，我们来看一个示例，这个示例很好地说明了如何分配固定内存，以及它相对于标准可分页内存的性能优势。

这个应用程序非常简单，主要是测试cudaMemcpy()在可分配内存和页锁定内存上的性能。我们要做的就是分配一个GPU缓冲区，以及一个大小相等的主机缓冲区，然后在这两个缓冲区之间执行一些复制操作。我们允许用户指定复制的方向，例如为“上”（从主机到设备）或者为“下”（从设备到主机）。你还将注意到，为了获得精确的时间统计，我们为复制操作的起始时刻和结束时刻分别设置了CUDA事件。你可能会记得之前性能测试示例中是如何实现这些操作的，但如果忘了也不要紧，下面这些代码会帮你回忆它们的用法：

```
float cuda_malloc_test( int size, bool up ) {
    cudaEvent_t    start, stop;
    int            *a, *dev_a;
    float          elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    a = (int*)malloc( size * sizeof( *a ) );
    HANDLE_NULL( a );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                              size * sizeof( *dev_a ) ) );
```

首先为size个整数分别分配主机缓冲区和GPU缓冲区，然后执行100次复制操作，并由参数up来指定复制方向，在完成复制操作后停止计时器。

```
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
for (int i=0; i<100; i++) {
    if (up)
        HANDLE_ERROR( cudaMemcpy( dev_a, a,
                                   size * sizeof( *dev_a ),
                                   cudaMemcpyHostToDevice ) );
    else
        HANDLE_ERROR( cudaMemcpy( a, dev_a,
```



```

    }
    HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
    HANDLE_ERROR( cudaEventSynchronize( stop ) );
    HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                         start, stop ) );

    HANDLE_ERROR( cudaFreeHost( a ) );
    HANDLE_ERROR( cudaFree( dev_a ) );
    HANDLE_ERROR( cudaEventDestroy( start ) );
    HANDLE_ERROR( cudaEventDestroy( stop ) );

    return elapsedTime;
}

```

你可以看到，`cudaHostAlloc()`分配的内存与`malloc()`分配的内存使用方式是相同的。与`malloc()`的不同之处在于最后一个参数`cudaHostAllocDefault`。最后一个参数的取值范围是一组标志，我们可以通过这些标志来修改`cudaHostAlloc()`的行为，并分配不同形式的固定主机内存。在第11章中，我们将看到其他的标志值，但就目前而言，只需使用默认值，因此将参数指定为`cudaHostAllocDefault`以获得默认的行为。要释放`cudaHostAlloc()`分配的内存，必须使用`cudaFreeHost()`。也就是说，正如每次调用`malloc()`后都需要调用一次`free()`一样，在每次调用`cudaHostAlloc()`后，也需要调用一次`cudaFreeHost()`。

`main()`函数的代码如下所示。

```

#include "../common/book.h"

#define SIZE    (10*1024*1024)

int main( void ) {
    float      elapsedTime;
    float      MB = (float)100*SIZE*sizeof(int)/1024/1024;
    elapsedTime = cuda_malloc_test( SIZE, true );
    printf( "Time using cudaMalloc:  %3.1f ms\n",
            elapsedTime );
    printf( "\tMB/s during copy up:  %3.1f\n",
            MB/(elapsedTime/1000) );
}

```

由于`cuda_malloc_test()`的参数`up`为`true`，因此前一次调用将测试从主机到设备（或者说“上升”到设备）的复制性能。要测试相反方向的性能，可以执行相同的调用，只需将第二个参数指定为`false`。

```

elapsedTime = cuda_malloc_test( SIZE, false );
printf( "Time using cudaMalloc:  %3.1f ms\n",

```



```

        elapsedTime );
printf( "\tMB/s during copy down:  %3.1f\n",
        MB/(elapsedTime/1000) );

```

我们执行了相同的步骤来测试cudaHostAlloc()的性能, 将cuda_host_alloc_test()调用两次, 一次将up指定为true, 另一次指定为false。

```

        elapsedTime = cuda_host_alloc_test( SIZE, true );
printf( "Time using cudaHostAlloc:  %3.1f ms\n",
        elapsedTime );
printf( "\tMB/s during copy up:  %3.1f\n",
        MB/(elapsedTime/1000) );

        elapsedTime = cuda_host_alloc_test( SIZE, false );
printf( "Time using cudaHostAlloc:  %3.1f ms\n",
        elapsedTime );
printf( "\tMB/s during copy down:  %3.1f\n",
        MB/(elapsedTime/1000) );
}

```

在GeForce GTX 285上, 当使用固定内存而不是可分页内存时, 我们观察到从主机复制到设备的性能从2.77GB/s提升到5.11GB/s。当从设备复制到主机时, 获得的性能提升是类似的, 即从2.43GB/s提升到5.46GB/s。因此, 对于大多数PCIE带宽有限的应用程序, 当使用固定内存而不是标准可分页内存时, 可以观察到显著的性能提升。但页锁定内存的作用并不仅限于性能提升。在后面的章节中会看到, 在一些特殊情况中也需要使用页锁定内存。

10.3 CUDA流

在第6章, 我们引入了CUDA事件的概念。当时并没有介绍cudaEventRecord()的第二个参数, 而只是简要地指出这个参数用于指定插入事件的流 (Stream)。

```

cudaEvent_t start;
cudaEventCreate(&start);
cudaEventRecord( start, 0 );

```

CUDA流在加速应用程序方面起着重要的作用。CUDA流表示一个GPU操作队列, 并且该队列中的操作将以指定的顺序执行。我们可以在流中添加一些操作, 例如核函数启动、内存复制, 以及事件的启动和结束等。将这些操作添加到流的顺序也就是它们的执行顺序。你可以将每个流视为GPU上的一个任务, 并且这些任务可以并行执行。我们将首先介绍如何使用流, 然后介绍如何使用流来加速应用程序。

10.4 使用单个CUDA流

稍后将会看到，仅当使用多个流时才能显现出流的真正威力，但我们首先通过在应用程序中使用单个流来说明流的使用。假设有一个CUDA C核函数，该函数带有两个输入数据缓冲区，a和b。核函数将对这些缓冲区中相应位置上的值执行某种计算，并将生成的结果保存到输出缓冲区c。矢量加法示例就采用了类似的计算模式，但在这个示例中，我们将计算a中三个值和b中三个值的平均值：

```
#include "../common/book.h"

#define N (1024*1024)
#define FULL_DATA_SIZE (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```

这个核函数不是很重要，因此不需要花太多时间来理解该函数执行的操作。可以把它看成是一个抽象函数，因为这个示例中重要的是函数main()中与流相关的代码。

```
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no "
               "speed up from streams\n" );

        return 0;
    }
}
```

我们做的第一件事情就是选择一个支持设备重叠（Device Overlap）功能的设备。支持设备重叠功能的GPU能够在执行一个CUDA C核函数的同时，还能在设备与主机之间执行复制操作。正如前面提到的，我们将使用多个流来实现这种计算与数据传输的重叠，但首先来看看如何创建和使用一个流。与其他需要测量性能提升（或者降低）的示例一样，首先创建和启动一

个事件计时器：

```
cudaEvent_t      start, stop;
float            elapsedTime;

// 启动计时器
HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

在启动计时器后，接下来创建在应用程序中使用的流：

```
// 初始化流
cudaStream_t      stream;
HANDLE_ERROR( cudaStreamCreate( &stream ) );
```

这就是创建流时需要的全部工作，并没有太多值得注意的地方，接下来是数据分配操作。

```
int *host_a, *host_b, *host_c;
int *dev_a, *dev_b, *dev_c;

// 在GPU上分配内存
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                           N * sizeof(int) ) );

// 分配由流使用的页锁定内存
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );

for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}
```

我们在GPU和主机上分别分配好了输入内存和输出内存。注意，由于程序将使用主机上的固定内存，因此调用cudaHostAlloc()来执行内存分配操作。使用固定内存的原因并不只在于使

复制操作执行得更快，还存在另外一个好处。稍后将更详细地进行分析，我们将使用一种新的 `cudaMemcpy()` 函数，并且在这个新函数中需要页锁定主机内存。在分配完输入内存后，调用C的库函数 `rand()` 并用随机整数填充主机内存。

在创建了流和计时事件，并且分配了设备内存和主机内存后，就准备好了执行一些计算。通常，我们会将这个阶段一带而过，只是将两个输入缓冲区复制到GPU，启动核函数，然后将输出缓冲区复制回主机。我们将再次沿用这种模式，只是进行了一些小修改。

首先，我们不将输入缓冲区整体都复制到GPU，而是将输入缓冲区划分为更小的块，并在每个块上执行一个包含三个步骤的过程。我们将一部分输入缓冲区复制到GPU，在这部分缓冲区内运行核函数，然后将输出缓冲区中的这部分结果复制回主机。想象一下需要使用这种方法的一种情形：GPU的内存远少于主机内存，由于整个缓冲区无法一次性填充到GPU，因此需要分块进行计算。执行“分块”计算的代码如下所示：

```
//在整体数据上循环，每个数据块的大小为N
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // 将锁定内存以异步方式复制到设备上
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );

    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i,
                                   N * sizeof(int),
                                   cudaMemcpyHostToDevice,
                                   stream ) );

    kernel<<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );

    // 将数据从设备复制到锁定内存
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c,
                                   N * sizeof(int),
                                   cudaMemcpyDeviceToHost,
                                   stream ) );
}
```

你将注意到，在前面的代码段中有两个不同之处。首先，代码没有使用熟悉的 `cudaMemcpy()`，而是通过一个新函数 `cudaMemcpyAsync()` 在GPU与主机之间复制数据。这些函数之间的差异虽然很小，但却很重要。`cudaMemcpy()` 的行为类似于C库函数 `memcpy()`。尤其是，这个函数将以同步方式执行，这意味着，当函数返回时，复制操作就已经完成，并且在输出缓冲区中包含了复制进去的内容。

异步函数的行为与同步函数相反，通过名字`cudaMemcpyAsync()`就可以知道。在调用`cudaMemcpyAsync()`时，只是放置一个请求，表示在流中执行一次内存复制操作，这个流是通过参数`stream`来指定的。当函数返回时，我们无法确保复制操作是否已经启动，更无法保证它是否已经结束。我们能够得到的保证是，复制操作肯定会当下一个被放入流中的操作之前执行。任何传递给`cudaMemcpyAsync()`的主机内存指针都必须已经通过`cudaHostAlloc()`分配好内存。也就是，你只能以异步方式对页锁定内存进行复制操作。

注意，在核函数调用的尖括号中还可以带有一个流参数。此时核函数调用将是异步的，就像之前与GPU之间的内存复制操作一样。从技术上来说，当循环迭代完一次时，有可能不会启动任何内存复制或核函数执行。前面提到过，我们能够确保的是，第一次放入流中的复制操作将在第二次复制操作之前执行。此外，第二个复制操作将在核函数启动之前完成，而核函数将在第三次复制操作开始之前完成。正如在本章前面提到的，流就像一个有序的工作队列，GPU从该队列中依次取出工作并执行。

当`for()`循环结束时，在队列中应该包含了许多等待GPU执行的工作。如果想要确保GPU执行完了计算和内存复制等操作，那么就需要将GPU与主机同步。也就是说，主机在继续执行之前，要首先等待GPU执行完成。可以调用`cudaStreamSynchronize()`并指定想要等待的流：

```
// 将计算结果从页锁定内存复制到主机内存
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
```

当程序执行到`stream`与主机同步之后的代码时，所有的计算和复制操作都已经完成，因此可以停止计时器，收集性能数据，并释放输入缓冲区和输出缓冲区。

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );

HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );
```

// 释放流和内存

```
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
```

最后，在退出应用程序之前，记得销毁对GPU操作进行排队的流。

```
HANDLE_ERROR( cudaStreamDestroy( stream ) );
```

```

    return 0;
}

```

坦白地说，这个示例并没有充分说明流的强大功能。当然，如果当主机正在执行一些工作时，GPU也正忙于处理填充到流的工作，那么即使使用单个流也有助于应用程序速度的提升。但即使不需要在主机上做太多的工作，我们仍然可以通过使用流来加速应用程序，在下一节中，我们将看到如何实现这个目的。

10.5 使用多个CUDA流

我们将10.4节中的示例改为使用两个不同的流。在前面示例的开始时，我们检查了这个设备确实支持重叠功能，并将计算分解为多个块。改进这个程序的思想很简单，包括两个方面：“分块”计算以及内存复制和核函数执行的重叠。我们将实现，在第0个流执行核函数的同时，第1个流将输入缓冲区复制到GPU。然后，在第0个流将计算结果复制回主机的同时，第1个流将执行核函数。接着，第1个流将计算结果复制回主机，同时第0个流开始在下一块数据上执行核函数。假设内存复制操作和核函数执行的时间大致相同，那么应用程序的执行时间将如图10.1所示。这张图假设，GPU可以同时执行一个内存复制操作和一个核函数，因此空的方框表示一个流正在等待执行某个操作的时刻，这个操作不能与其他流的操作相互重叠。还需要注意的是，在本章的剩余图片中，函数调用`cudaMemcpyAsync()`被简写为复制。

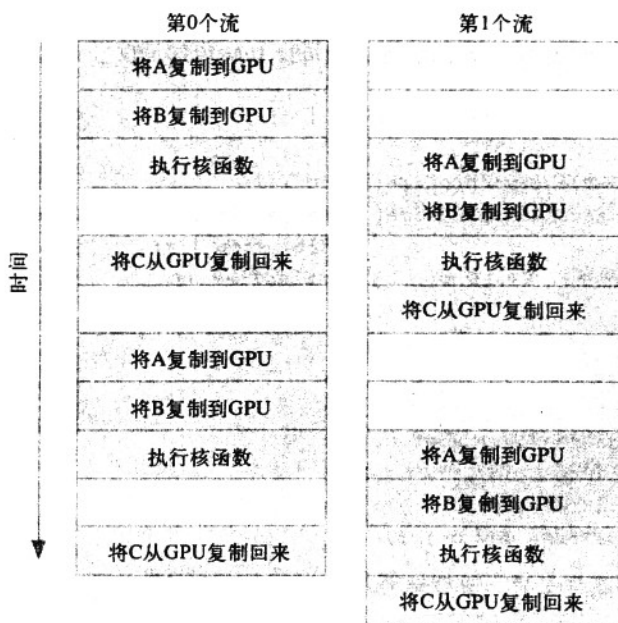


图10.1 应用程序在使用两个不同流时的执行时间线

事实上，实际的执行时间线可能比图中给出的更好看，在一些新的NVIDIA GPU中同时支持核函数和两次内存复制操作，一次是从主机到设备，另一次是从设备到主机。在任何支持内存复制和核函数的执行相互重叠的设备上，当使用多个流时，应用程序的整体性能都会提升。

尽管能够获得对应用程序的加速，但核函数将保持不变。

```
#include "../common/book.h"

#define N (1024*1024)
#define FULL_DATA_SIZE (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2
    }
}
```

与使用单个流的版本一样，我们将判断设备是否支持计算与内存复制操作的重叠。如果设备支持重叠，那么就像前面一样创建CUDA事件并对应用程序计时。

```
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no "
               "speed up from streams\n" );
        return 0;
    }

    cudaEvent_t start, stop;
    float elapsedTime;

    // 启动计时器
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );
```

接下来创建两个流，创建方式与在前面代码中创建单个流的方式完全一样。

```
// 初始化流
cudaStream_t    stream0, stream1;
HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
HANDLE_ERROR( cudaStreamCreate( &stream1 ) );
```

假设在主机上仍然是两个输入缓冲区和一个输出缓冲区。输入缓冲区中填充的是随机数据，与使用单个流的应用程序采用的方式一样。然而，现在我们将使用两个流来处理数据，分配两组相同的GPU缓冲区，这样每个流都可以独立地在输入数据块上执行工作。

```
int *host_a, *host_b, *host_c;
int *dev_a0, *dev_b0, *dev_c0; // 为第0个流分配的GPU内存
int *dev_a1, *dev_b1, *dev_c1; // 为第1个流分配的GPU内存
```

```
//在GPU上分配内存
HANDLE_ERROR( cudaMalloc( (void**)&dev_a0,
                          N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b0,
                          N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c0,
                          N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_a1,
                          N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b1,
                          N * sizeof(int) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c1,
                          N * sizeof(int) ) );
```

```
// 分配在流中使用的页锁定内存
HANDLE_ERROR( cudaHostAlloc( (void**)&host_a,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_b,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&host_c,
                             FULL_DATA_SIZE * sizeof(int),
                             cudaHostAllocDefault ) );
```

```
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}
```

然后，程序在输入数据块上循环。然而，由于现在使用了两个流，因此在for()循环的迭代

中需要处理的数据量也是原来的两倍。在stream()中,我们首先将a和b的异步复制操作放入GPU的队列,然后将一个核函数执行放入队列,接下来再将一个复制回c的操作放入队列:

```
//在整体数据上循环,每个数据块的大小为N
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // 将锁定内存以异步方式复制到设备上
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );

    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );

    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

    // 将数据从设备复制回锁定内存
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,
                                    N * sizeof(int),
                                    cudaMemcpyDeviceToHost,
                                    stream0 ) );
}
```

在将这些操作放入stream0的队列后,再把下一个数据块上的相同操作放入stream1的队列中。

```
// 将锁定内存以异步方式复制到设备上
HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,
                                N * sizeof(int),
                                cudaMemcpyHostToDevice,
                                stream1 ) );

HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,
                                N * sizeof(int),
                                cudaMemcpyHostToDevice,
                                stream1 ) );

kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

// 将数据从设备复制回到锁定内存
HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,
                                N * sizeof(int),
                                cudaMemcpyDeviceToHost,
                                stream1 ) );
}
```

这样,在for()循环的迭代过程中,将交替地把每个数据块放入这两个流的队列,直到所有

待处理的输入数据都被放入队列。在结束了for()循环后，在停止应用程序的计时器之前，首先将GPU与GPU进行同步。由于使用了两个流，因此需要对二者都进行同步。

```
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
```

我们采用了与单个流版本中相同的方式将main()包装起来。停止计时器，显示经历的时间，并且执行清理工作。当然，我们要记住，现在需要销毁两个流，并且需要释放两倍的GPU内存，除此之外，这段代码与之前看到的代码是相同的：

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );

HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );

// 释放流和内存
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );

return 0;
}
```

我们在GeForce GTX 285上分别测试了10.3节中的使用单个流的版本，以及改进后使用两个流的版本。在修改为使用两个流后，程序的执行时间为61ms。

非常棒！

这正是我们对程序进行计时的原因。有时候，我们自认为的性能“增强”实际上除了使代码变得更复杂外，不会起到任何作用。

这个程序能不能变得更快？确实，程序还能变得更快！因为我们实际上是通过第二个流来加速仅使用单个流的应用程序，但要想进一步加快程序的执行速度，我们需要首先理解CUDA驱动程序对流的处理方式，才能从设备重叠中获得好处。要理解流在幕后是如何工作的，我们

需要理解CUDA驱动程序和CUDA硬件架构的工作原理。

10.6 GPU的工作调度机制

虽然从逻辑上来看,不同的流之间是相互独立的,但事实上这种理解并不完全符合GPU的队列机制。程序员可以将流视为有序的操作序列,其中既包含内存复制操作,又包含核函数调用。然而,在硬件中并没有流的概念,而是包含一个或多个引擎来执行内存复制操作,以及一个引擎来执行和函数。这些引擎彼此独立地对操作进行排队,因此将导致如图10.2所示的任务调度情形。图中的箭头说明了硬件引擎如何调度流中队列的操作并实际执行。

因此,在某种程度上,用户与硬件关于GPU工作的排队方式有着完全不同的理解,而CUDA驱动程序则负责对用户和硬件进行协调。首先,在操作被添加到流的顺序中包含了重要的依赖性。例如,在图10.2中,第0个流对A的内存复制需要在对B的内存复制之前完成,而对B的复制又要在核函数A启动之前完成。然而,一旦这些操作放入到硬件的内存复制引擎和核函数执行引擎的队列中时,这些依赖性将丢失,因此CUDA驱动程序需要确保硬件的执行单元不破坏流内部的依赖性。

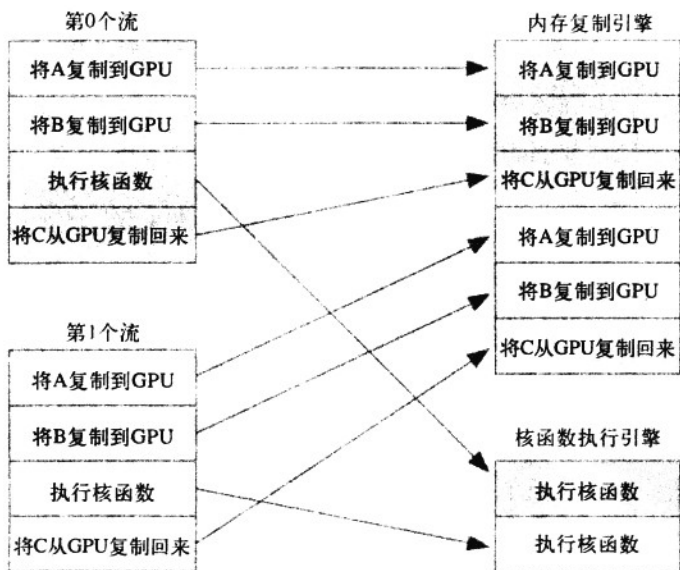


图10.2 将CUDA流映射到GPU引擎

这意味着什么?我们来看看10.4节中实际发生的情况。如果重新观察代码,将看到应用程序基本上是对a调用一次`cudaMemcpyAsync()`,对b调用一次`cudaMemcpyAsync()`,然后再是执行核函数以及调用`cudaMemcpyAsync()`将c复制回主机。应用程序首先将第0个流的所有操作放

入队列，然后是第1个流的所有操作。CUDA驱动程序负责按照这些操作的顺序把它们调度到硬件上执行，这就维持了流内部的依赖性。在图10.3中说明了这些依赖性，其中从复制操作到核函数的箭头表示，复制操作要等核函数执行完成之后才能开始。

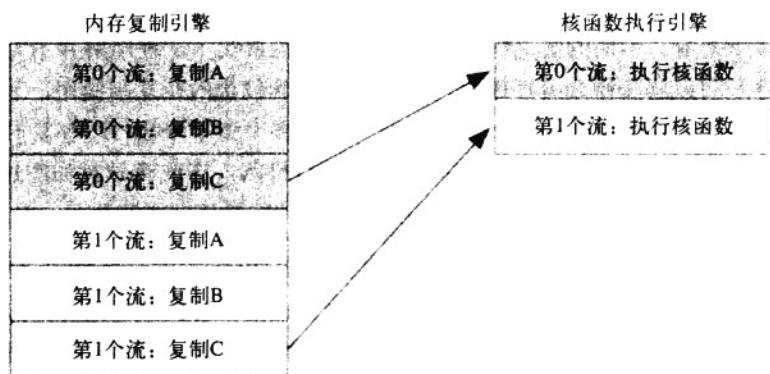


图10.3 箭头表示cudaMemcpyAsync()对核函数的依赖性

假定理解了GPU的工作调度原理后，我们可以得到关于这些操作在硬件上执行的时间线，如图10.4所示。

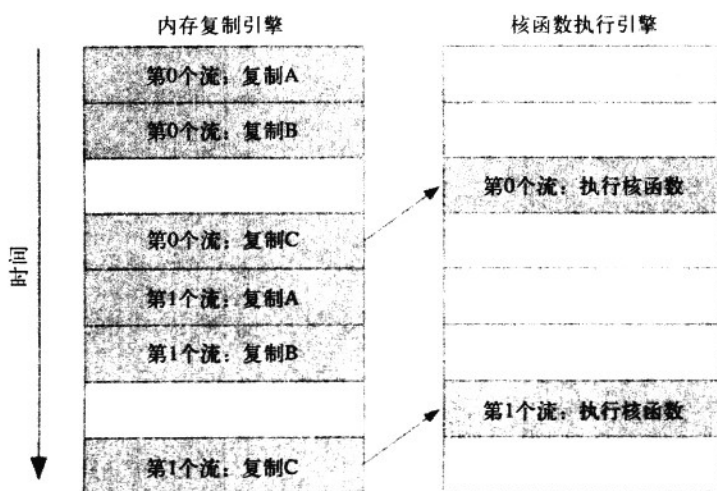


图10.4 在“使用多个CUDA流”中示例程序的执行时间线

由于第0个流中将c复制回主机的操作要等待核函数执行完成，因此第1个流中将a和b复制到GPU的操作虽然是完全独立的，但却被阻塞了，这是因为GPU引擎是按照指定的顺序来执行工作。这种情况很好地说明了为什么在程序中使用了两个流却无法获得加速的窘境。这个问题的直接原因是我们没有意识到硬件的工作方式与CUDA流编程模型的方式是不同的。

这里需要说明的情况是，当要确保相互独立的流能够真正地并行执行时，我们自己要起到一定的作用。记住，硬件在处理内存复制和核函数执行时分别采用了不同的引擎，因此我们需要知道，将操作放入流中队列中的顺序将影响着CUDA驱动程序调度这些操作以及执行的方式。在10.7节中，我们将看到如何帮助硬件实现内存复制操作与核函数执行的重叠。

10.7 高效地使用多个CUDA流

正如在10.6节中看到的，如果同时调度某个流的所有操作，那么很容易在无意中阻塞另一个流的复制操作或者核函数执行。要解决这个问题，在将操作放入流的队列时应采用宽度优先方式，而非深度优先方式。也就是说，不是首先添加第0个流的所有四个操作（即a的复制、b的复制、核函数以及c的复制），然后不再添加第1个流的所有四个操作，而是将这两个流之间的操作交叉添加。首先，将a的复制操作添加到第0个流，然后将a的复制操作添加到第1个流。接着，将b的复制操作添加到第0个流，再将b的复制操作添加到第1个流。接下来，将核函数调用添加到第0个流，再将相同的操作添加到第1个流中。最后，将c的复制操作添加到第0个流中，然后将相同的操作添加到第1个流中。

要更具体地理解这个操作，我们来看实际的代码。我们修改了这些操作被分配到两个流的顺序，这基本上只需要对代码进行复制/粘贴。应用程序中的其他代码保持不变，这意味着我们的修改只局限于for()循环。采用宽度优先方式将操作分配到两个流的代码如下所示：

```
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // 将复制a的操作放入stream0和stream1的队列
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );

    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream1 ) );

    // 将复制b的操作放入stream0和stream1的队列
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream0 ) );

    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N,
                                    N * sizeof(int),
                                    cudaMemcpyHostToDevice,
                                    stream1 ) );
}
```

```

// 将核函数的执行放入stream1和stream1的队列中
kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

// 将复制c的操作放入stream0和stream1的队列
HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,
                                N * sizeof(int),
                                cudaMemcpyDeviceToHost,
                                stream0 ) );

HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1,
                                N * sizeof(int),
                                cudaMemcpyDeviceToHost,
                                stream1 ) );
}

```

如果内存复制操作的时间与核函数执行的时间大致相当，那么新的执行时间线将如图10.5所示。引擎间的依赖性通过箭头高亮显示，可以看到在新的调度顺序中，这些依赖性仍然能得到满足。

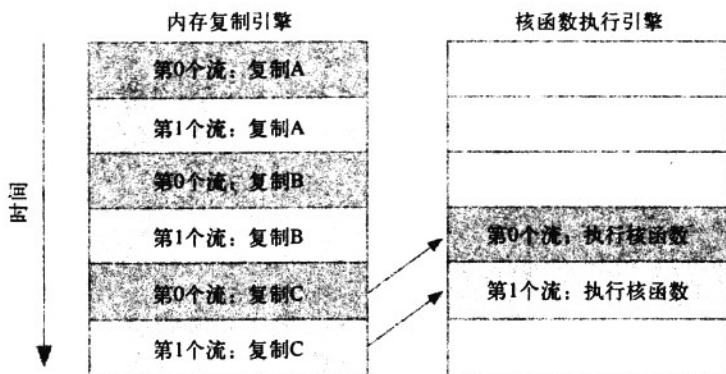


图10.5 改进后示例的执行时间线，其中箭头表示引擎间的依赖性

由于采用了宽度优先方式将操作放入各个流的队列中，因此第0个流对c的复制操作将不会阻塞第1个流对a和b的内存复制操作。这使得GPU能够并行地执行复制操作和核函数，从而使应用程序的运行速度显著加快。新代码的运行时间为48ms，比最初使用两个流的应用程序快了21%。如果应用程序的所有计算和内存复制操作都能重叠，那么可以在性能上获得近两倍的提升，因为执行复制操作的引擎和执行核函数的引擎将始终保持在运行。

10.8 本章小结

在本章中，我们介绍了如何在CUDA C应用程序中实现任务级的并行性。通过使用两个

(或者多个) CUDA流, 我们可以使GPU在执行核函数的同时, 还能在主机和GPU之间执行复制操作。然而, 当采用这种执行方式时, 需要注意两个因素。首先, 需要通过`cudaHostAlloc()`来分配主机内存, 因为接下来需要通过`cudaMemcpyAsync()`对内存复制操作进行排队, 而异步复制操作需要在固定缓冲区执行。其次, 我们要知道, 添加这些操作到流中的顺序将对内存复制操作和核函数执行的重叠情况产生影响。通常, 应该采用宽度优先或者轮询方式将工作分配到的流。如果没有理解硬件的排队工作原理, 那么初看上去会觉得这种方式有违直觉, 因此当你编写自己的应用程序时, 要记住这个情况。

第11章

多GPU系统上的CUDA C

有一句古老的谚语，当用于GPU时可以表述为：“比在一个GPU上计算要更好的，只有在两个GPU上计算。”近年来，在系统中包含多个图形处理器已变得越来越常见。当然，多GPU系统在某种程度上类似于多CPU系统，因为二者都还没有成为主流的系统配置。但是，在系统中包含多个GPU却是很容易的。例如，GeForce GTX 295在单块显卡上就集成了两个GPU，NVIDIA的Tesla S1070计算系统包含了4个支持CUDA的图形处理器，在基于最新NVIDIA芯片组构建的系统中，主板上就有一个支持CUDA的集成GPU，如果在某个PCI EXPRESS插槽中再添加一块独立的NVIDIA GPU，那么系统中就包含了多个GPU。这些情况都是很常见的，因此我们需要了解如何利用多GPU系统中的资源。

11.1 本章目标

通过本章的学习，你可以：

- 了解如何分配和使用零拷贝内存（Zero-Copy Memory）。
- 了解如何在同一个应用程序中使用多个GPU。
- 了解如何分配和使用可移动的固定内存（Portable Pinned Memory）。

11.2 零拷贝主机内存

在第10章中，我们介绍了固定内存（或者说页锁定内存），这种新型的主机内存能够确保不会交换出物理内存。我们通过调用`cudaHostAlloc()`来分配这种内存，并且传递参数`cudaHostAllocDefault`来获得默认的固定内存。在前面曾提到，本章会介绍在分配固定内存时可以使用的其他参数值。除了`cudaHostAllocDefault`外，还可以传递的标志之一是`cudaHostAllocMapped`。通过`cudaHostAllocMapped`分配的主机内存也是固定的，它与通过`cudaHostAllocDefault`分配的固定内存有着相同的属性，特别是当它不能从物理内存中交换出去或者重新定位时。但这种内存除了可以用于主机与GPU之间的内存复制外，还可以打破第3章中的主机内存规则之一：可以在CUDA C核函数中直接访问这种类型的主机内存。由于这种内存不需要复制到GPU，因此也称为零拷贝内存。

11.2.1 通过零拷贝内存实现点积运算

通常，GPU只能访问GPU内存，而CPU也只能访问主机内存。但在某些环境中，打破这种规则或许能带来更好的效果。为了说明由GPU访问主机内存将带来哪些好处，我们来重新回顾前面的归约运算：矢量点积运算。如果你已经完整地读了本书，那么肯定还记得第一个点积运算版本：将两个输入矢量复制到GPU，对相应的元素执行乘积计算，然后将中间结果复制回到主机，并在CPU上完成求和计算。

在这个版本中，我们不需要将输入矢量显式复制到GPU，而是使用零拷贝内存从GPU中直接访问数据。这个版本的点积运算非常类似于对固定内存的性能测试程序。我们将编写两个函数，其中一个函数是对标准主机内存的测试，另一个函数将在GPU上执行归约运算，并使用零拷贝内存作为输入缓冲区和输出缓冲区。首先，我们来看看点积运算的主机内存版本。按照惯例，首先创建计时事件，然后分配输入缓冲区和输出缓冲区，并用数据填充输入缓冲区。

```
float malloc_test( int size ) {
    cudaEvent_t      start, stop;
    float            *a, *b, c, *partial_c;
    float            *dev_a, *dev_b, *dev_partial_c;
```

```

float          elapsedTime;

HANDLE_ERROR( cudaEventCreate( &start ) );
HANDLE_ERROR( cudaEventCreate( &stop ) );

// 在CPU上分配内存
a = (float*)malloc( size*sizeof(float) );
b = (float*)malloc( size*sizeof(float) );
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

//在GPU上分配内存
HANDLE_ERROR( cudaMalloc( (void*)&dev_a,
                          size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void*)&dev_b,
                          size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void*)&dev_partial_c,
                          blocksPerGrid*sizeof(float) ) );

// 用数据填充主机内存
for (int i=0; i<size; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

在分配好内存并且创建完数据后，就可以开始计算。启动计时器，将输入数据复制到GPU，执行点积核函数，并将中间计算结果复制回主机。

```

HANDLE_ERROR( cudaEventRecord( start, 0 ) );
//将数组 'a' 和 'b' 复制到GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float),
                          cudaMemcpyHostToDevice ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                          dev_partial_c );

//将数组 'c' 从GPU复制到CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );

```

现在，我们需要像第5章中那样结束CPU上的计算。在执行这个操作前，首先要停止事件计时器，因为它只需测量在GPU上完成的工作：

```

HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );

```

最后，将中间计算结果相加起来，并释放输入缓冲区和输出缓冲区。

```

// 结束CPU上的计算
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );

// 释放CPU上的内存
free( a );
free( b );
free( partial_c );

// 释放事件
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );

printf( "Value calculated: %f\n", c );

return elapsedTime;
}

```

使用零拷贝内存的版本是非常类似的，只是在内存分配上有所不同。我们首先分配输入缓冲区和输出缓冲区，并同样用数据来填充输入内存：

```

float cuda_host_alloc_test( int size ) {
    cudaEvent_t      start, stop;
    float            *a, *b, c, *partial_c;
    float            *dev_a, *dev_b, *dev_partial_c;
    float            elapsedTime;

    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    //在CPU上分配内存
    HANDLE_ERROR( cudaHostAlloc( (void**)&a,
                                 size*sizeof(float),

```

```

        cudaHostAllocWriteCombined |
        cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&b,
                             size*sizeof(float),
                             cudaHostAllocWriteCombined |
                             cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&partial_c,
                             blocksPerGrid*sizeof(float),
                             cudaHostAllocMapped ) );

// 用数据填充主机内存
for (int i=0; i<size; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

与第10章一样，我们再次使用了cudaHostAlloc()，只是通过参数flags来指定内存的其他行为。cudaHostAllocMapped这个标志告诉运行时将从GPU中访问这块内存。换句话说，这个标志意味着分配零拷贝内存。对于两个输入缓冲区，我们还指定了标志cudaHostAllocWriteCombined。这个标志表示，运行时应该将内存分配为“合并式写入（Write-Combined）”内存。这个标志并不会改变应用程序的功能，但却可以显著地提升GPU读取内存时的性能。然而，当CPU也要读取这块内存时，“合并式写入”会显得很低效，因此在决定是否使用这个标志之前，必须首先考虑应用程序的可能访问模式。

在使用标志cudaHostAllocMapped来分配主机内存后，就可以从GPU中访问这块内存。然而，GPU的虚拟内存空间与CPU是不同的，因此在GPU上访问它们与在CPU上访问它们有着不同的地址。调用cudaHostAlloc()将返回这块内存存在CPU上的指针，因此需要调用cudaHostGetDevicePointer()来获得这块内存存在GPU上的有效指针。这些指针将被传递给核函数，并在随后由GPU对这块内存执行读取和写入等操作：

```

HANDLE_ERROR( cudaHostGetDevicePointer( &dev_a, a, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_b, b, 0 ) );
HANDLE_ERROR( cudaHostGetDevicePointer( &dev_partial_c,
                                         partial_c, 0 ) );

```

在获得了有效的设备指针后，就可以启动计时器以及核函数。

```

HANDLE_ERROR( cudaEventRecord( start, 0 ) );

dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                         dev_partial_c );
HANDLE_ERROR( cudaThreadSynchronize() );

```

即使指针dev_a、dev_b和dev_partial_c都位于主机上，但对于核函数来说，它们看起来与GPU内存一样，这正是由于调用了cudaHostGetDevicePointer()。由于部分计算结果已经位于主机上，因此就不再需要通过cudaMemcpy()将它们从设备上复制回来。然而，你可能注意到了在程序中调用了cudaThreadSynchronize()将CPU与GPU同步。如果在核函数中会修改零拷贝内存的内容，那么在核函数的执行期间，零拷贝内存的内容是未定义的。在同步完成后，就可以确信核函数已经完成，并且在零拷贝内存中包含了计算好的结果，因此就可以停止计时器并结束在CPU上的计算。

```
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );
```

// 结束CPU上的操作

```
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
```

在使用cudaHostAlloc()的点积运算代码中，唯一剩下的事情就是执行释放操作。

```
HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFreeHost( b ) );
HANDLE_ERROR( cudaFreeHost( partial_c ) );
```

// 释放事件

```
HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
```

```
printf( "Value calculated: %f\n", c );
```

```
return elapsedTime;
```

```
}
```

需要注意的是，无论在cudaHostAlloc()中使用什么标志，总是按照相同的方式来释放内存，即只需调用cudaFreeHost()。

基本内容就是这样！剩下的工作就是观察main()如何将代码片段组合在一起。在main()中，首先要判断设备是否支持映射主机内存。我们使用第10章中判断设备是否支持重叠的方法，即调用cudaGetDeviceProperties()。

```
int main( void ) {
    cudaDeviceProp prop;
```

```

int whichDevice;
HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
if (prop.canMapHostMemory != 1) {
    printf( "Device cannot map memory.\n" );
    return 0;
}

```

如果设备支持零拷贝内存，那么接下来就是将运行时置入能分配零拷贝内存的状态。通过调用cudaSetDeviceFlags()来实现这个操作，并且传递标志值cudaDeviceMapHost来表示我们希望设备映射主机内存：

```
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
```

这就是main()中的主要操作。我们运行两个测试，分别显示二者的执行时间，并退出应用程序：

```

float elapsedTime = malloc_test( N );
printf( "Time using cudaMalloc: %3.1f ms\n",
        elapsedTime );

elapsedTime = cuda_host_alloc_test( N );
printf( "Time using cudaHostAlloc: %3.1f ms\n",
        elapsedTime );
}

```

核函数本身与第5章中的核函数没有区别，但我们还是给出了完整的核函数：

```

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );

__global__ void dot( int size, float *a, float *b, float *c ) {
    __shared__ float cache[threadsPerBlock];
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int cacheIndex = threadIdx.x;
    float temp = 0;
    while (tid < size) {
        temp += a[tid] * b[tid];
        tid += blockDim.x * gridDim.x;
    }
}

```

```

// 设置cache中的值
cache[cacheIndex] = temp;

// 同步这个线程块中的线程
__syncthreads();

// 对于归约运算, threadsPerBlock 必须为2的幂

int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

if (cacheIndex == 0)
    c[blockIdx.x] = cache[0];
}

```

11.2.2 零拷贝内存的性能

零拷贝内存能够带来哪些好处？对于独立GPU和集成GPU，答案是不同的。独立GPU自己拥有专门的DRAM，通常位于CPU之外的电路板上。例如，如果在计算机中已经安装了一块图形卡，那么这个GPU就是一个独立GPU。集成GPU是系统芯片组中内置的图形处理器，通常与CPU共享系统内存。在许多基于NVIDIA nForce媒体与通信处理器（Media and Communications Processor, MCP）构建的现代系统中，都包含了支持CUDA的集成GPU。除了nForce MCP外，基于NVIDIA新推出的ION平台的上网本、笔记本以及桌面计算机都包含了集成的和支持CUDA的GPU。对于集成GPU，使用零拷贝内存通常都会带来性能提升，因为内存物理上与主机是共享的。将缓冲区声明为零拷贝内存的唯一作用就是避免不必要的复制。但要记住，天下没有免费的午餐，所有类型的固定内存都存在一定的局限性，零拷贝内存同样不例外：每个固定内存都会占用系统的可用物理内存，这最终将降低系统的性能。

当输入内存和输出内存都只能使用一次时，那么在独立GPU上使用零拷贝内存将带来性能提升。由于GPU在设计时考虑了隐藏内存访问带来的延迟，因此这种机制在某种程度上将减轻PCI-E总线上读取和写入等操作的延迟，从而会带来可观的性能提升。但由于GPU不会缓存零拷贝内存的内容，如果多次读取内存，那么最终将得不偿失，还不如一开始就将数据复制到GPU。

如何判断某个GPU是集成的还是独立的？当然，你可以打开计算机的机箱来观察，但这种方法对于CUDA C应用程序来说是不可行的。在代码中可以通过cudaGetDeviceProperties()返回

的结构来判断GPU的这个属性。在该结构中有一个域integrated，如果设备是集成GPU，那么这个域的值为true，否则为false。

由于点积运算应用程序满足“仅读取/写入一次”这个约束条件，因此在使用零拷贝内存时能获得性能提升。事实上，程序在性能上确实有一定程度的提升。在GeForce GTX 285上，当使用零拷贝内存时，程序的执行时间减少了45%，从98.1毫秒降到52.1毫秒。在GeForce GTX 280上同样能获得性能提升，执行时间减少34%，从143.9毫秒降为94.7毫秒。当然，由于计算量与带宽的比值不同，以及芯片组之间PCIE总线带宽的不同，不同的GPU将表现出不同的性能特性。

11.3 使用多个GPU

在11.2节中，我们指出设备要么是集成的GPU，要么是独立的GPU，其中前者是构建在系统芯片组中的，而后者通常是位于PCIE槽中的扩展卡。然而，在越来越多的系统中同时包含了集成GPU和独立GPU，这意味着它们拥有多个支持GUDA的处理器。NVIDIA同样也会不断地推出包含多个GPU的产品，例如GeForce GTX 295。虽然GeForce GTX 295在物理上占用一个扩展槽，但在CUDA应用程序看来则是两个独立的GPU。而且，用户还可以将多个GPU添加到独立的PCIE槽上，通过NVIDIA的SLI技术将它们桥接。这些趋势导致的结果之一就是，在运行CUDA应用程序的系统中，包含多个图形处理器逐渐成为一种常见情况。由于CUDA应用程序是可高度并行化的，如果可以使系统中的每个CUDA设备都实现最大的吞吐量，那么无疑是最好的情形。因此，我们来看看如何实现这个目标。

为了避免学习新的示例，我们将把点积应用程序修改为使用多个GPU。为了降低编码难度，我们将在一个结构中把计算点积所需的全部数据都相加起来。你马上将看到为什么这会降低难度。

```
struct DataStruct {
    int    deviceID;
    int    size;
    float  *a;
    float  *b;
    float  returnValue;
};
```

这个结构包含了在计算点积时使用的设备标识，以及输入缓冲区的大小和指向两个输入缓冲区的指针a和b。最后，它还包含了一个成员用于保存a和b的点积运算结果。

要使用N个GPU，我们首先需要准确地知道N值是多少。因此，在应用程序的开头调用cudaGetDeviceCount()，从而判断在系统中安装了多少个支持CUDA的处理器。


```

int main( void ) {
    int deviceCount;
    HANDLE_ERROR( cudaGetDeviceCount( &deviceCount ) );
    if (deviceCount < 2) {
        printf( "We need at least two compute 1.0 or greater "
               "devices, but only found %d\n", deviceCount );
        return 0;
    }
}

```

这个示例只是说明多个GPU的使用，因此如果系统只有一个CUDA设备，程序将退出（这并不是因为出现了什么错误）。显然，这种做法并不是最佳的。为了使问题尽可能简单，我们将为输入缓冲区分配标准的主机内存，并按照之前的方式来填充它们。

```

float  *a = (float*)malloc( sizeof(float) * N );
HANDLE_NULL( a );
float  *b = (float*)malloc( sizeof(float) * N );
HANDLE_NULL( b );

// 用数据填充主机内存
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

```

现在，我们就可以进一步深入研究多GPU代码。当通过CUDA运行时API来使用多个GPU时，要意识到每个GPU都需要由一个不同的CPU线程来控制。由于之前只使用了单个GPU，因此不需要担心这个问题。我们将多线程代码的大部分复杂性都移入到辅助代码文件book.h中。在精简了代码后，我们需要做的就是填充一个结构来执行计算。虽然在系统中可以有任意数量的GPU，但为了简单起见，在这里只使用两个：

```

DataStruct  data[2];

data[0].deviceID = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;

data[1].deviceID = 1;
data[1].size = N/2;
data[1].a = a + N/2;
data[1].b = b + N/2;

```

我们将其中一个DataStruct变量传递给辅助函数start_thread()。此外，还将一个函数指针传递给了start_thread()，新创建的线程将调用这个函数，在这个示例中的线程函数为routine()。函

数start_thread()将创建一个新线程，这个线程将调用routine()，并将DataStruct变量作为参数传递进去。在应用程序的默认线程中也将调用routine()（因此只多创建了一个线程）。

```
CUTThread  thread = start_thread( routine, &(amp;data[0]) );
routine( &(data[1]) );
```

通过调用end_thread()，主应用程序线程将等待另一个线程执行完成。

```
end_thread( thread );
```

由于这两个线程都在main()的这个位置上执行完成，因此可以安全地释放内存并显示结果。

```
free( a );
free( b );

printf( "Value calculated: %f\n" ,
        data[0].returnValue + data[1].returnValue );

return 0;
}
```

注意，我们要将每个线程的计算结果相加起来。这是点积归约运算的最后一步。在其他算法中，可能还需要其他的步骤将这些结果合并起来。事实上，在某些应用程序中，这两个GPU可能在不同的数据集上执行不同的代码。为了简单起见，在我们点积运算示例中不考虑这些情况。

由于这里的点积函数与之前版本中的点积函数是相同的，因此在本节中不介绍它。但是，我们要注意routine()函数的代码。在声明routine()时指定该函数带有一个void*参数，并返回void*，这样在start_thread()部分代码保持不变的情况下可以任意实现线程函数。虽然这个想法确实不错，但这却是C中回调函数的标准过程。

```
void* routine( void *pvoidData ) {
    DataStruct *data = (DataStruct*)pvoidData;
    HANDLE_ERROR( cudaSetDevice( data->deviceID ) );
```

每个线程都调用cudaSetDevice()，并且每个线程都传递一个不同的ID给这个函数。因此，我们知道每个线程都将使用一个不同的GPU。这些GPU可能有着相同的性能，例如双GPU的GeForce GTX 295，或者有着不同的性能，例如在系统中同时包含一个集成GPU和一个独立GPU时。对于应用程序来说，这些细节并不重要，但它们对你来说或许是需要关注的。特别是，如果在启动核函数时需要某个最低的计算功能集版本，或者如果希望在系统的多个GPU之间实现负载均衡，那么这些细节将非常有用。如果各个GPU的性能不同，那么你需要对计算进行划分，从而使每个GPU都执行基本相等的时间。然而，在这个示例中，我们不需要担心这些细节问题。

除了调用cudaSetDevice()来指定希望使用的CUDA设备外, routine()的实现非常类似于11.1.1节中的malloc_test()。我们为输入数据和临时计算结果分别分配了内存, 随后调用cudaMemcpy()将每个输入数组复制到GPU。

```
int      size = data->size;
float    *a, *b, c, *partial_c;
float    *dev_a, *dev_b, *dev_partial_c;

// 在CPU上分配内存
a = data->a;
b = data->b;
partial_c = (float*)malloc( blocksPerGrid*sizeof(float) );

// 在GPU上分配内存
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                          size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                          size*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_partial_c,
                          blocksPerGrid*sizeof(float) ) );

// 将数组 'a' 和 'b' 复制到GPU上
HANDLE_ERROR( cudaMemcpy( dev_a, a, size*sizeof(float),
                          cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, size*sizeof(float),
                          cudaMemcpyHostToDevice ) );

然后, 我们启动点积核函数, 复制回计算结果, 并且结束CPU上的操作。

dot<<<blocksPerGrid,threadsPerBlock>>>( size, dev_a, dev_b,
                                          dev_partial_c );

// 将数组 'c' 从GPU复制回CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );

// 结束CPU上的操作
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}
```

和通常一样, 接下来释放GPU内存, 并在DataStruct结构的returnValue成员中返回计算好的点积结果。

```

HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_partial_c ) );

// 释放CPU侧的内存
free( partial_c );

data->returnValue = c;
return 0;
}

```

因此，除了主机线程的管理问题外，使用多个GPU的程序并不比使用单个GPU的程序要复杂。如果使用我们的辅助代码来创建一个线程，并在线程上执行一个函数，那么将极大地降低编码的复杂性。如果你有自己的线程库，那么也可以在程序中使用它们。你只需要记住，每个GPU都有自己的线程，除此之外其他的使用方式都是类似的。

11.4 可移动的固定内存

在使用多个GPU的程序中，最后一个重要的部分就是固定内存的使用。在第10章中已经介绍了，固定内存实际上是主机内存，只是该内存页锁定在物理内存中，以便防止被换出或者重定位。然而，这些内存页仅对于单个CPU线程来说是“固定的”。也就是说，如果某个线程分配了固定内存，那么这些内存只是对于分配它们的线程来说是页锁定的。如果在线程之间共享指向这块内存的指针，那么其他的线程将把这块内存视为标准的、可分页的内存。

这种行为的副作用之一就是，当其他线程（不是分配固定内存的线程）试图在这块内存上执行cudaMemcpy()时，将按照标准的可分页内存速率来执行复制操作。在第10章中曾介绍过，这种速率大约为最高传输速度的50%。更糟糕的是，如果线程试图将一个cudaMemcpyAsync()调用放入CUDA流的队列中，那么将失败，因为cudaMemcpyAsync()需要使用固定内存。由于这块内存对于除了分配它的线程以外的其他线程来说似乎是可分页的，因此这个调用会失败，甚至导致任何后续操作都无法进行。

然而，对于这个问题有一种补救方案。我们可以将固定内存分配为可移动的，这意味着可以在主机线程之间移动这块内存，并且每个线程都将其视为固定内存。要达到这个目的，需要使用cudaHostAlloc()来分配内存，并且在调用时使用一个新的标志：cudaHostAllocPortable。这个标志可以与其他标志一起使用，例如cudaHostAllocWriteCombined和cudaHostAllocMapped。这意味着在分配主机内存时，可将其作为可移动、零拷贝以及合并式写入等的任意组合。

为了说明可移动固定内存的作用，我们将进一步修改使用多GPU的点积运算应用程序。我们将修改最初使用零拷贝内存的点积运算程序，因此在这个版本中将融合零拷贝版本的代码和

多GPU版本的代码。首先，我们需要验证至少有两个支持CUDA的GPU，并且二者都能处理零拷贝缓冲区。

```
int main( void ) {
    int deviceCount;
    HANDLE_ERROR( cudaGetDeviceCount( &deviceCount ) );
    if (deviceCount < 2) {
        printf( "We need at least two compute 1.0 or greater "
                "devices, but only found %d\n", deviceCount );
        return 0;
    }

    cudaDeviceProp prop;
    for (int i=0; i<2; i++) {
        HANDLE_ERROR( cudaGetDeviceProperties( &prop, i ) );
        if (prop.canMapHostMemory != 1) {
            printf( "Device %d cannot map memory.\n", i );
            return 0;
        }
    }
}
```

在前面的示例中，已经准备好了在主机上分配内存以便保存输入矢量。然而，为了分配可移动的固定内存，首先需要设置将在哪个CUDA设备上运行。由于我们还希望在这个设备上分配零拷贝内存，因此在调用cudaSetDevice()之后，接着调用了cudaSetDeviceFlags()，和11.1.1节中一样。

```
float *a, *b;
HANDLE_ERROR( cudaSetDevice( 0 ) );
HANDLE_ERROR( cudaSetDeviceFlags( cudaDeviceMapHost ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&a, N*sizeof(float),
                             cudaHostAllocWriteCombined |
                             cudaHostAllocPortable |
                             cudaHostAllocMapped ) );
HANDLE_ERROR( cudaHostAlloc( (void**)&b, N*sizeof(float),
                             cudaHostAllocWriteCombined |
                             cudaHostAllocPortable |
                             cudaHostAllocMapped ) );
```

在本章的前面，我们等到已经分配了内存并且创建了线程后才调用cudaSetDevice()。然而，在使用cudaHostAlloc()分配页锁定内存时，首先要通过调用cudaSetDevice()来初始化设备。你还将注意到，我们将新介绍的标志cudaHostAllocPortable传递给这两个内存分配操作。由于这些内存是在调用了cudaSetDevice(0)之后才分配的，因此，如果没有将这些内存指定为可移动的内存，那么只有第0个CUDA设备会把这些内存视为固定内存。

继续之前的应用程序，为输入矢量生成数据，并采用11.2节中多GPU示例中的方式来准备DataStruct结构。

```
// 用数据填充主机内存
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}
```

```
// 为使用多线程做好准备
DataStruct data[2];
data[0].deviceID = 0;
data[0].offset = 0;
data[0].size = N/2;
data[0].a = a;
data[0].b = b;
```

```
data[1].deviceID = 1;
data[1].offset = N/2;
data[1].size = N/2;
data[1].a = a;
data[1].b = b;
```

然后，我们创建第二个线程，并调用routine()开始在每个设备上执行计算。

```
CUTThread thread = start_thread( routine, &(data[1]) );
routine( &(data[0]) );
end_thread( thread );
```

由于主机内存是由CUDA运行时分配的，因此需要用cudaFreeHost()而不是free()来释放它。

```
// 释放CPU上的内存
HANDLE_ERROR( cudaFreeHost( a ) );
HANDLE_ERROR( cudaFreeHost( b ) );

printf( "Value calculated: %f\n",
        data[0].returnValue + data[1].returnValue );

return 0;
}
```

为了在多GPU应用程序中支持可移动的固定内存和零拷贝内存，我们需要对routine()的代码进行两处修改。第一处修改有些微妙。

```
void* routine( void *pvoidData ) {
```



```
// 将数组 'c' 从GPU复制回CPU
HANDLE_ERROR( cudaMemcpy( partial_c, dev_partial_c,
                          blocksPerGrid*sizeof(float),
                          cudaMemcpyDeviceToHost ) );
```

最后，像之前的点积示例一样，在CPU上将临时和值相加起来，释放内存，并返回到main()。

```
// 结束CPU上的操作
c = 0;
for (int i=0; i<blocksPerGrid; i++) {
    c += partial_c[i];
}

HANDLE_ERROR( cudaFree( dev_partial_c ) );

// 释放CPU上的内存
free( partial_c );

data->returnValue = c;
return 0;
}
```

11.5 本章小结

我们在本章中看到了一些新的主机内存分配方式，所有这些内存都是通过cudaHostAlloc()来分配的。通过使用一组不同的标志，我们可以分配具备零拷贝、可移动或者合并式写入等属性的内存。通过使用零拷贝内存，可以避免CPU与GPU之间的显式复制操作，这对许多应用程序来说都可能会带来加速。通过使用支持线程的库，我们可以在同一个应用程序中对多个GPU进行操作，使点积运算能够跨越多个设备执行。最后，我们看到了通过可移动的固定内存使多个GPU共享固定内存。在最后一个示例中使用了可移动的固定内存，多个GPU，以及零拷贝内存，以演示第5章中点积运算的增强版。随着多GPU系统的日益普及，这些技术能帮助你充分发挥这些系统的总体计算能力。

第12章

后 记

祝贺你！我们希望你能享受学习CUDA C的过程，并在GPU计算中使用它。从第1章到现在，我们经历了一段漫长的学习历程，接下来将简要地回顾一下。首先，我们学习了如何通过CUDA运行时的尖括号语法在多处处理器上启动多个核函数副本。我们将这些概念进一步扩展为使用线程集和线程块集，实现对任意长度的输入数据都可以执行算术操作。这些复杂的核函数通过GPU上的共享内存来实现线程间通信，并采用了专门的同步原语，从而确保在支持数千个并行线程的环境中实现正确的操作。

在了解了如何在NVIDIA CUDA架构上实现并行编程的基本概念后，我们接下来介绍了NVIDIA提供的一些更高级的概念和API。事实证明，GPU中的专门图形硬件对于GPU计算来说是非常有用的，因此我们学习了如何利用纹理内存来加速一些常见的内存访问模式。由于许多用户都在他们的交互式图形应用程序中添加了GPU计算支持，因此我们介绍了CUDA C核函数与工业标准的图形API（例如OpenGL与DirectX）之间的交互。全局内存和共享内存上的原子操作则实现了对共享内存的安全多线程访问。然后，我们还介绍了一些高级主题，流可以使系统尽可能地保持忙碌，在执行核函数的同时，还可以在主机和GPU之间执行内存复制操作。最后，我们学习了如何通过零拷贝内存存在集成GPU上加速应用程序。此外，我们学习了如何初始化多个GPU设备以及如何分配可移动的固定内存，这使得CUDA C程序能够进一步发挥多GPU平台的强大功能。

12.1 本章目标

通过本章的学习，你可以：

- 了解一些辅助开发CUDA C程序的工具。
- 了解一些可以帮助你提升CUDA C开发能力的代码资源。

12.2 CUDA工具

在本书中，我们用到了CUDA C软件系统的数个部件。我们编写的应用程序需要通过CUDA C编译器将核函数编译为可在NVIDIA GPU上执行的代码。CUDA运行时负责完成在核函数调用以及与GPU通信幕后的大部分繁琐工作，它将调用CUDA驱动程序直接与系统中的硬件进行会话。除了这些已经使用的部件外，NVIDIA还提供了其他一组软件来帮助简化CUDA C应用程序的开发工作。本节并不会对这些产品做详细介绍，而只是给出这些软件的用途。

12.2.1 CUDA工具箱

在你编写代码的机器上肯定已经安装了CUDA工具箱（CUDA Toolkit），因为在这个软件包中包含了CUDA C编译器工具集。如果在机器上没有CUDA工具箱，那么肯定就没有编写过或者编译过任何CUDA C代码。当然，没有安装CUDA工具箱也不是什么大问题（但我们怀疑你为什么要读这本书）。相反，如果亲手实现了书中的所有示例，那么就已经拥有了接下来将要讨论的一些库。

12.2.2 CUFFT

CUDA工具箱包含了两个重要的工具库，它们可以帮助你在应用程序中实现GPU编程。第一个库是NVIDIA提供的一个优化后的快速傅里叶变换（Fast Fourier Transform）库，也称为CUFFT。从3.0版本开始，CUFFT库支持一组非常有用的功能，包括以下：

- 在实数值与复数值之间进行一维变换、二维变换和三维变换。
- 以批处理的方式并行执行多个一维变换。
- 二维变换和三维变换，其中每一维的大小范围为2到16 384。
- 一维变换，其中元素的数量最高可达8百万。
- 实数值数据和复数值数据的就地（In-Place）变换和非就地（Out-of-Place）变换。

NVIDIA提供免费的CUFFT库和一个许可，这个许可使得我们可以在任意应用程序中免费使用CUFFT库，而不论程序是属于个人兴趣，学术研究还是商业开发。

12.2.3 CUBLAS

除了快速傅里叶变换库外，NVIDIA还提供了—个线性代数函数（Linear Algebra Routine）库，其中包含了著名的基本线程代数子程序（Basic Linear Algebra Subprograms, BLAS）。这个库被命名为CUBLAS，同样是免费的，它包含了BLAS的一个子功能集。这个库中的每个函数都有多个版本，函数参数可以是单精度类型或者双精度类型，既可以是实数值，也可以是复数值。由于BLAS最初是一个由FORTRAN编写的线性代数函数库，NVIDIA尝试最大程度地兼容这些已有的实现。特别是，在CUBLAS库中，数组采用了列主（Column-Major）形式的存储布局，而不是像C和C++中才用行主（Row-Major）形式的布局。在实际情况中通常不需要考虑这种差异，从而使得使用BLAS的用户只需很小的代价就可以将应用程序修改为使用CUBLAS。NVIDIA还发布了对CUBLAS的FORTRAN绑定，用来说明如何将现有的FORTRAN应用程序链接到CUDA库。

12.2.4 NVIDIA GPU Computing SDK

除了NVIDIA驱动程序和CUDA工具箱外，在GPU Computing SDK中还包含了许多GPU计算示例程序。我们在本书前面提到过这个SDK，它包含的示例能很好地补充本书前11章的内容。这些示例代码涵盖了CUDA C的各个不同应用领域，分布在不同的主题文档中。这些示例大致可分为以下类别：

CUDA基本主题

CUDA高级主题

CUDA系统集成

数据并行算法

图形互操作

纹理

性能策略

线性代数

图像/视频处理

计算金融

数据压缩

物理模拟

这些示例可以在任何支持CUDA C的平台上运行，也可以作为你自己编写应用程序的基础。对于在这些领域有着丰富经验的读者，需要提醒的是，在GPU computing SDK中通常不会包含算法的最新/最优实现。这些代码不应作为产品级别的库代码，而只能作为CUDA C程序的学习材料，它们与本书的示例没有差别。

12.2.5 NVIDIA性能原语

除了在CUFFT和CUBLAS等库中提供的函数外，NVIDIA还提供了一个函数库来执行基于CUDA加速的数据处理操作，称为NVIDIA性能原语（NVIDIA Performance Primitives, NPP）。当前，NPP的基本功能集合主要侧重于图像处理和视频处理，这些功能广泛适用于这些领域的开发人员。NVIDIA计划在将来进一步增强NPP的功能以便解决更广泛领域中的计算任务。如果你对高性能的图像处理或者视频应用程序感兴趣，那么应该优先考虑是否能使用NPP，可以从网址www.nvidia.com/object/npp.html上免费下载（或者从网页搜索引擎中访问）。

12.2.6 调试CUDA C

很少有计算机软件能够在第一次执行时就完全无误地按照最初的设计执行。通常，有些代码计算得到不正确的值，有些代码无法结束，还有些代码甚至使计算机死机，只有按下电源开关才能重新恢复。虽然我个人从来没有写过这样的代码，但还是意识到，有些软件工程师可能希望获得某些工具对他们的CUDA C核函数进行调试。NVIDIA提供了一些工具来降低调试过程的复杂性。

1. CUDA-GDB

CUDA-GDB是其中最有用的CUDA工具之一，Linux系统上的CUDA C程序员在开发代码时可以下载这个工具。NVIDIA对开源的GUN调试器（gdb）进行了扩展，使其支持设备代码的实时调试，同时保持gdb的常规接口不变。在CUDA-GDB之前，除了在CPU上模拟设备代码的运行外，程序员没有其他更好的方法来调试设备代码。这种方法使调试工作的进展非常缓慢，而且也只能对核函数在GPU上的实际执行做非常粗略的估计。而NVIDIA的CUDA-GDB使程序员能够直接在GPU上调试核函数，为程序员提供了他们熟悉的CPU调试器的控制方式。CUDA-GDB包括以下功能：

- 观察CUDA状态，例如已安装的GPU及其支持的功能等信息。
- 在CUDA C源代码中设置断点。
- 分析GPU内存，包括全局内存和共享内存。
- 分析当前驻留在GPU上的线程块和线程。
- 单步调试线程束。
- 中断并进入到当前正在运行的应用程序，包括挂起的或者发生死锁的应用程序。

除了调试器外，NVIDIA还提供了CUDA内存检查器（CUDA Memory Checker），我们可以通过CUDA-GDB或者独立的工具Cuda-Memcheck来使用CUDA内存检查器的功能。由于在CUDA架构中包含基于硬件构建的完备内存管理单元，因此所有的非法内存访问都可以通过硬件来检测和阻止。内存访问违例造成的结果是，程序将无法执行预期的功能，因此你肯定希望找出这种类型的错误。当启用CUDA内存检查器时，它将检测出核函数的全局内存访问违例或者未对齐的全局内存访问等错误，并以非常详细的方式来报告这些错误。

2. NVIDIA Parallel Nsight

对于在硬件上实时调试CUDA C核函数来说，CUDA-GDB无疑是一种成熟的和功能强大的工具，但NVIDIA意识到，并不是每个开发人员都喜欢使用Linux。Windows用户同样需要某种方式来调试CUDA应用程序。在2009年，NVIDIA推出了NVIDIA Parallel Nsight（最初称为Nexus），这是第一个集成在Microsoft Visual Studio中的GPU/CPU调试器。Parallel Nsight类似于CUDA-GDB，它能够调试包含数千个线程的CUDA应用程序。用户可以在CUDA C源代码中设置任意数量的断点，包括在写入内存位置时触发的断点。用户可以直接在Visual Studio Memroy窗口中直接查看内存，以及检查越界的内存访问。在本书出版时已经发布了这个工具的Beta版本，并且马上会发布最终版本。

12.2.7 CUDA Visual Profiler

我们经常宣称CUDA架构是开发高性能计算应用程序的重要基础组件。然而，事实却是，即使在找出应用程序中所有的错误后，那些所谓的“高性能计算”应用程序，更准确地说只能叫做“计算”应用程序，它们表现出的行为与“高性能”不符。我们经常会遇到一些类似的疑问：为什么代码在Sam Hill中运行的性能如此糟糕？在这些情况中，如果能通过某个可视化的分析工具来运行核函数，那么将是非常有帮助的。NVIDIA就提供了一个这样的工具Visual Profiler，可以从CUDA Zone网址上下载。在图12.1中给出了在Visual Profiler中比较矩阵转置运算两种不同实现的性能。虽然看不到任何一行代码，但仍然很容易判断核函数transpose()在内存吞吐量和指令吞吐量上都要优于核函数transpose_naive()。（通常，如果在函数名中带有“native”，那么这个函数的性能通常不是最优的。）

CUDA Visual Profiler将执行应用程序，同时分析GPU中内置的特殊性能计数器。在执行完成后，性能分析器可以根据这些计数器来统计数据并给出报告。Visual Profiler可以统计应用程序在执行每个核函数时花了多少时间，并判断线程块的数量、核函数的内存访问操作是否被合并了、在代码执行中的分支数量等。如果你需要解决某些复杂的性能问题，那么我们建议通过CUDA Visual Profiler来分析。

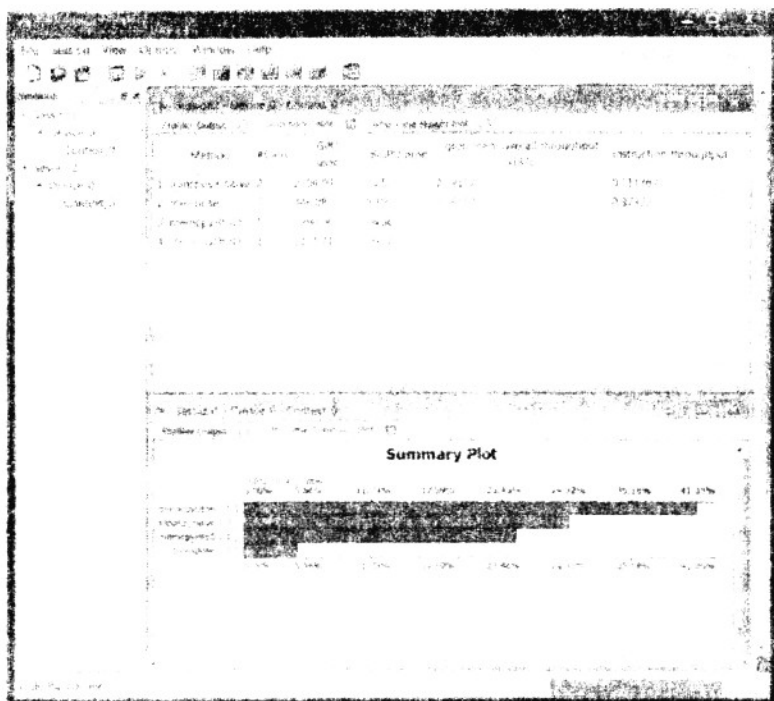


图12.1 在CUDA Visual Profiler中对矩阵转置应用程序进行分析

12.3 参考资料

如果还没有厌倦本书给出的所有内容，那么你可能还希望阅读更多的学习资料。我们知道，有些读者希望通过练习代码来继续学习，而对于其他的读者，则希望通过阅读其他一些参考资料来成为一名CUDA C编码人员。

12.3.1 《Programming Massively Parallel Processors:a Hands-On Approach》

在第1章中，我们已经指出本书并不是一本讲解并行架构的书。虽然我们提到了一些专业术语，例如多处理器和线程束，但本书重点介绍的是如何使用CUDA C及其API来编程。我们在《NVIDIA CUDA Programming Guide》给出的编程模型中学习了CUDA C语言，而并没有详细介绍NVIDIA硬件在实际完成这些任务时采用的方式。

然而，如果你想成为一名优秀的CUDA C程序员，那么就需要进一步熟悉CUDA架构以及NVIDIA GPU的底层工作原理。要了解这些内容，我们建议你阅读《Programming Massively Parallel Processors; A Hands-on Approach》。本书是由NVIDIA前首席科学家David Kirk与伊利诺斯大学电子与计算机工程系教授Wen-mei W. Hwu一起合作完成的。在这本书中，你不仅能

看到许多熟悉的术语和概念，还将学习到NVIDIA CUDA架构的许多细节，包括线程调度机制和延迟容忍（Latency Tolerance）、内存带宽的使用和效率、浮点计算规范等。该书还介绍了更广泛意义上的并行编程，因此你可以更好地了解如何为大规模的复杂问题设计并行解决方案。

12.3.2 CUDA U

在我们当中，有些读者在开始学习GPU编程之前没有上过大学。对于那些幸运地上过大学或在不久将要上大学的读者，需要知道的是，在全世界大约有300所大学开设了CUDA课程。然而，在开始大学生活之前，还有另一种方法来学习CUDA！在CUDA Zone网址上可以找到介绍CUDA U的链接，这基本上是一个CUDA在线教育大学。或者，你可以直接跳转到网址www.nvidia.com/object/cuda_education。如果你参加了一些CUDA U在线课程，那么将学到不少关于GPU计算的知识，但在本书出版时，还没有出现一些在线协会可以在课后参与。

1. 大学课程资料

在众多的CUDA教学资料中，伊利诺斯大学的CUDA C编程课程是最好的课程之一。NVIDIA和伊利诺斯大学将这些课程制作成M4V视频免费下载，可以在iPod，iPhone或者兼容的视频播放器上播放。我们知道你肯定在想：“终于有一种方法让我在车辆管理局排队时还可以学习CUDA！”你可能会奇怪，为什么要等到本书的结尾时才告诉你本书有相应的视频教学版本。抱歉这么晚才告诉你，但无论如何，视频教学的效果总是不如课本的教学效果，对不对？除了伊利诺斯大学和加州大学戴维斯分校的课程资料外，你还可以找到CUDA教学音频以及到第三方培训和咨询服务的链接。

2. Dr. Dobb's

在创刊以来30多年的时间里，《Dr. Dobb's》期刊涵盖了计算技术领域中的所有主要发展方向，NVIDIA的CUDA也不例外。《Dr. Dobb's》已经发表了连续若干期的文章，广泛地介绍了CUDA。这些文章的系列标题为《CUDA, Supercomputing for the Masses》，首先介绍了GPU计算，然后介绍了第一个核函数以及CUDA编程模型的其他方面。《Dr. Dobb's》中的这系列文章涵盖的主题包括：错误处理、全局内存性能、共享内存、CUDA Visual Profiler、纹理内存、CUDA-GDB、包含数据并行CUDA原语的CUDPP库，以及许多其他主题。这系列的文章很好地补充了我们在本书中介绍的内容。而且，对于书中只是简要介绍的一些工具，例如性能分析和调试等，在这些文章中同样给出了更详细的信息。在CUDA Zone网页上包含了这些文章的链接，也可以通过检索“Dr Dobbs CUDA”来访问这些文章。

12.3.3 NVIDIA论坛

有时候，即使你仔细研读了所有的NVIDIA文档，但还是会发现有些问题无法解答。或许你还想知道，是否有人曾经有过与你类似的经历。或者，你正在筹办一个CUDA庆祝会，因此

想邀请一些有着相同兴趣的人。对于你感兴趣的任何问题，我们建议将它们发布到NVIDIA网站的论坛上。论坛的地址是<http://forums.nvidia.com>，在这里你可以向其他CUDA使用者提出各种问题。事实上，在读完本书后，你已经可以帮助其他人并回答他们的问题了！NVIDIA雇了专人来跟踪论坛上的问题，因此即使最困难的问题，也会得到及时且权威的解答。我们还希望获得用户对新推出功能的反馈和建议，无论是赞扬还是批评，都将欣然接受。

12.4 代码资源

尽管NVIDIA GPU Computing SDK中包含了大量的示例代码，但这些代码通常只适合用于教学。如果你希望获得一些产品级别的源代码或者库，那么还需要更进一步。幸运的是，在CUDA开发人员社区中提供了各种非常棒的解决方案。在本书中介绍了其中一些工具和库，但你也可以在网页上搜索需要的解决方案。当然，在将来某天你也可以为CUDA C社区做出自己的贡献！

12.4.1 CUDA数据并行原语库

在加州大学戴维斯分校的研究人员的帮助下，NVIDIA公布了CUDA数据并行原语库（CUDA Data Parallel Primitives Library, CUDPP）。从名字可以看出，CUDPP包含的是一组数据并行算法原语。例如，并行前缀求和（扫描），并行排序以及并行归约等。这些原语为许多数据并行算法提供了重要的基础，包括排序、流压缩、构建数据结构以及其他并行算法等。如果你正在编写某个复杂的算法，那么很可能CUDPP已经提供了这个算法，或者提供了算法的大部分功能。CUDPP的下载地址为<http://code.google.com/p/cudpp>。

12.4.2 CULAtools

在12.2.3节中曾提到，在CUDA工具箱中包含了对BLAS的实现。如果读者需要更多的线性代数函数库，那么可以看看EM Photonics公司基于CUDA实现的线性代数（LAPACK）库。这个库也称为CLUAtools，它包含了许多更为复杂的线性代数函数，这些函数都是基于NVIDIA的CUBLAS技术构建。在免费的基本版本中包含了LU矩阵分解、QR因数分解、线性系统求解、奇异值分解，以及最小二乘法求解和约束最小二乘法。你可以从网址www.culatools.com/versions/basic上下载这个基本版本。你将注意到，EM Photonics公司提供了软件使用许可，其中包含了大部分的LAPACK函数，以及当发布基于CULAtools的商业软件时需要包含的许可条款。

12.4.3 语言封装器

本书使用的编程语言为C和C++，然而有许多项目并不是采用这两种语言。幸运的是，一些第三方机构编写了各种语言封装器，用来在非NVIDIA官方支持的其他语言中访问CUDA。

NVIDIA本身为CUBLAS库提供了FROTRAN绑定，此外在www.jcuda.org上还包含了对数个CUDA库的Java绑定。同样，在Python应用程序中可以通过Python封装器来访问CUDA C核函数，下载地址为<http://mathematician.de/software/pycuda>。最后，在CUDA.NET项目中推出了在Microsoft .NET环境下使用的绑定，下载地址为www.hoopoe-cloud.com/Solutions/CUDA.NET。

虽然这些项目都不是由NVIDIA正式支持的，但它们在CUDA的数个版本中都存在，并且都拥有大量的用户群。这里需要指出的是，如果你选择的（或者老板规定的）语言不是C或C++，那么你首先要做的不是排除GPU计算，而是看看能否找到相应的语言绑定。

12.5 本章小结

这就是本书的所有内容。即使在读完了前11章的内容后，还有许多的资源可以下载、阅读、思考和编译。随着异构计算平台的日趋成熟，学习GPU计算也显得非常重要。我们希望你能学习并掌握当前最主流的并行编程环境之一CUDA。而且，我们希望你能够进一步研究出一些新方法来与计算机交互，以及处理软件面临的日益增加的信息量。你的这些研究思想和技术将把GPU计算推向更高境地。

附 录

高级原子操作

在第9章中介绍了原子操作的一些使用方法，以及如何通过原子操作使数百个线程对共享数据进行安全地并发修改。在本附录中，我们将介绍原子操作的一种高级用法，即实现锁定数据结构。表面上来看，这个主题似乎与之前介绍的内容一样，事实也确实如此。在本书中你已经学习了许多复杂的主题，而锁定数据结构也不会比这些主题更难。然而，为什么要把这部分内容放到附录中？我们暂时先不透露任何原因，如果你感兴趣，那么可以继续读下去，我们将在附录中给出解释。

A.1 回顾点积运算

在第5章中，我们看到了如何通过CUDA C来实现矢量点积运算。点积算法是归约算法的一种，这个算法通过以下步骤来计算两个输入矢量的点积：

- 1) 线程块中的每个线程将输入矢量中的两个对应元素相乘，并将结果保存在共享内存中。
- 2) 在一个线程块中将计算得到多个乘积，然后通过线程将这些乘积结果两两相加，并将结果保存回共享内存。每执行完一次相加后，得到结果的数量将是计算之前结果数量的一半（这正是“归约”这个名字的含义）。
- 3) 最终，每个线程块都将得到一个临时和值，它会把这个值保存到全局内存中并退出。
- 4) 如果共有N个并行线程块运行核函数，那么CPU将把N个值相加起来从而得到最终的点积。

如果你对这里给出的点积算法步感到有些陌生，那么可以花点时间重新回顾第5章的内容。如果你已经熟悉了点积运算的代码，那么可以将主要注意力放到算法的第4步。虽然该算法并不需要大量数据复制到主机，也不需要CPU上执行任何的计算，但将最终结果的计算步骤移到CPU完成却似乎是一种低效方式。

然而，这里的问题并不只是低效。考虑一种情况，其中点积运算只是某个操作序列中的一步。如果CPU正在执行其他的任务或者计算，并需要将每个操作都放在GPU上执行，那么将出现问题。此时，程序将不得不停止GPU上的计算，将中间结果复制回到主机，在CPU上完成计算，并最终将结果再传回GPU，然后继续计算下一个核函数。

由于本附录的内容主要是关于原子操作的使用，并且我们已经指出了在最初点积算法中的问题，因此你应该能预计到接下来将要介绍的内容。我们尝试使用原子操作来解决点积运算中的问题，使得整个运算都可以在GPU上执行，而CPU则可以不受干扰地执行其他任务。在理想情况下，不是在步骤3中退出核函数并在步骤4中返回到CPU，而是每个线程块都将自己的临时结果相加到全局内存中的最终总和。如果每个临时结果都以原子方式相加起来，那么就不用担心可能发生冲突或者得到不确定的结果。由于我们已经在直方图运算中使用了一个`atomicAdd()`操作，因此看上去似乎可以选择这个操作来实现我们的设想。

然而，在计算功能集2.0之前，`atomicAdd()`只能在整数上运算。如果只计算整数矢量的点积，那么是没有问题的，但浮点矢量的计算却更为常见。然而，大部分的NVIDIA硬件并不支持在浮点数值上进行原子运算！但这种做法是有原因的，因此暂时还不要将GPU弃之不用。

原子操作只能确保，每个线程在完成读取-修改-写入的序列之前，将不会有其他的线程读取或者写入目标内存。然而，原子操作并不能确保这些线程将按照何种顺序执行，例如当有三个线程都执行加法运算时，加法运行的执行顺序可以为 $(A+B)+C$ ，也可以为 $A+(B+C)$ 。这对于

整数来说是可以接受的，因为整数数学运算是可结合的，因此 $(A+B)+C = A+(B+C)$ 。然而，浮点算法并非可结合的，因为中间结果可能被截断，因此 $(A+B)+C$ 通常并不等于 $A+(B+C)$ 。因此，浮点数值上的原子数学运算是否有用就值得怀疑，因为在像GPU这种多线程的环境中，这种运算将带来不确定的结果。许多应用程序都不允许在运行两次时得到不同的结果，因此在早期的硬件中，浮点数学运算并不是优先支持的功能。

然而，如果可以容忍在计算结果中存在某种程度的不确定性，那么仍然可以完全在GPU上实现归约运算。但是，我们首先需要通过某种方式来绕开原子浮点数学运算。在这个解决方案中仍将使用原子操作，但不是用于算术本身。

A.1.1 原子锁

在构建GPU直方图中，我们使用了atomicAdd()函数，这个函数在执行读取-修改-写入操作时不会被其他线程中断。你可以想象底层硬件是如何执行这个运算的：硬件锁定目标内存位置，并且当锁定时，其他任何线程都不能读取或者写入这个位置上的值。如果可以通过某种方式在CUDA C核函数中模拟这种锁定行为，那么就可以在相应的内存位置或者数据结构上执行任意运算。锁定机制的行为与CPU互斥体非常相似。如果你不熟悉互斥行为，也不要苦恼，互斥体并不比你到目前为止学到的内容复杂。

基本思想是：分配一小块内存作为互斥体。互斥体的行为就像控制对某个资源访问的交通信号灯。这个资源可以是一个数据结构，一个缓冲区，或者是一个需要以原子方式修改的内存位置。当某个线程从这个互斥体中读到0时，它会把这个值解释为“绿灯”，表示没有其他线程正在使用这块内存。因此，该线程可以锁定这块内存，并执行想要的修改，而不会受到其他线程的干扰。要锁定这个内存位置，线程将1写入到互斥体。1就表示“红灯”，这将防止其他竞争的线程锁定这个内存。然后，其他竞争线程必须等待直到互斥体的所有者线程将0写入到互斥体后才能尝试修改被锁定的内存。

实现锁定过程的代码可以像下面这样：

```
void lock( void ) {
    if( *mutex == 0 ) {
        *mutex = 1; //将1保存到锁
    }
}
```

遗憾的是，在这段代码中存在着一个严重问题，这也是我们熟悉的一个问题：如果在线程读取到0并且还没有修改这个值之前，另一个线程将1写入到互斥体，那么会发生什么情况？也就是说，这两个线程都将检查mutex上的值，并判断其是否为0。然后，它们都将1写入到这个位置，从而告诉其他的线程，该结构已经被锁定了并不能修改。之后，这两个线程都认为它们拥有相应内存或者数据结构的所有，然后开始执行并不安全的修改操作。这肯定会

造成严重后果!

我们想要完成的操作很简单: 将mutex的值与0相比较, 如果mutex等于0时, 则将1写入到这个位置。要正确地实现这个操作, 整个运算都需要以原子方式执行, 这样就可以确保当线程检查和更新mutex的值时, 不会有其他的线程进行干扰。在CUDA C中, 这个操作可以通过函数atomicCAS()来实现, 这是一个原子的比较-交换操作 (Compare-and-Swap)。函数atomicCAS()的参数包括一个指向目标内存的指针, 一个与内存中的值进行比较的值, 以及一个当比较相等时保存到目标内存上的值。通过这个操作, 我们可以实现一个GPU锁定函数, 如下所示:

```
__device__ void lock( void ) {
    while( atomicCAS( mutex, 0, 1 ) != 0 );
}
```

调用atomicCAS()将返回位于mutex地址上的值。因此, while()循环会不断运行, 直到atomicCAS发现mutex的值为0。当发现为0时, 比较操作成功, 并线程将把1写入到mutex。本质上来看, 这个线程将在while()循环中自旋, 直到它成功地锁定这个数据结构。我们将使用这种锁定机制来实现GPU散列表。然而, 我们首先需要把代码封装在一个结构中, 这样在点积应用程序中使用起来就更为整洁:

```
struct Lock {
    int *mutex;
    Lock( void ) {
        int state = 0;
        HANDLE_ERROR( cudaMalloc( (void**)& mutex,
                                   sizeof(int) ) );
        HANDLE_ERROR( cudaMemcpy( mutex, &state, sizeof(int),
                                   cudaMemcpyHostToDevice ) );
    }

    ~Lock( void ) {
        cudaFree( mutex );
    }

    __device__ void lock( void ) {
        while( atomicCAS( mutex, 0, 1 ) != 0 );
    }

    __device__ void unlock( void ) {
        atomicExch( mutex, 0 );
    }
};
```

注意，在代码中通过`atomicExch( mutex, 1 )`来重置`mutex`的值。函数`atomicExch()`将读取`mutex`的值，将其与第二个参数（在这里是1）进行交换，并返回它读到的值。为什么要使用一个原子函数来实现这个操作，而不是使用某种更简单的方法来重置`mutex`的值？例如：

```
*mutex = 0;
```

如果你认为不采用这种简单方法是因为某个复杂的原因，那我们将告诉你这种想法是错误的，因为这种简单的方法同样也是有效的。那么为什么不使用这种更简单的方法？原子事务和常规的全局内存操作将采用不同的执行路径。如果同时使用原子操作和标准的全局内存操作，那将使得`unlock()`与后面的`lock()`调用看上去似乎没有被同步。虽然这种混合使用的方式仍可以实现正确的功能，但为了增加应用程序的可读性，对于所有对互斥体的访问都应该使用相同的方式。因此，在使用原子语句来锁定资源后，同样应用使用原子语句来解锁资源。

A.1.2 修改点积运算：原子锁

在最早的点积运算示例中，唯一要修改的部分就是最后在CPU上执行的归约部分。在前一节中，我们介绍了如何在GPU上实现互斥体。Lock结构位于`lock.h`中，在修改后的点积示例中将包含这个头文件：

```
#include "../common/book.h"
#include "lock.h"

#define imin(a,b) (a<b?a:b)

const int N = 33 * 1024 * 1024;
const int threadsPerBlock = 256;
const int blocksPerGrid =
    imin( 32, (N+threadsPerBlock-1) / threadsPerBlock );
```

除了两处不同的地方外，点积核函数的开头部分与第5章中使用的核函数完全相同。其中一个不同之处是核函数的原型：

```
__global__ void dot( Lock lock, float *a, float *b, float *c )
```

在修改后的点积函数中，将Lock类型的变量，以及输入矢量和输出缓冲区传递给核函数。Lock将被用于在最后的累加步骤中控制对输出缓冲区的访问。另一处修改从函数原型中看得出来，但却与原型有关。之前，参数`float *c`是一个包含N个浮点数的缓冲区，其中每个线程块都将其计算得到的临时结果保存到相应的元素中。这个缓冲区将被复制到CPU以便计算最终的点积值。然而，现在的参数`c`将不再指向一个缓冲区，而是指向一个浮点数值，这个值表示`a`和`b`中矢量的点积。除了这些修改，核函数的开头与第5章完全一样：

```
__global__ void dot( Lock lock, float *a,
```

```

        float *b, float *c ) {
__shared__ float cache[threadPerBlock];
int tid = threadIdx.x + blockIdx.x * blockDim.x;
int cacheIndex = threadIdx.x;

float temp = 0;
while (tid < N) {
    temp += a[tid] * b[tid];
    tid += blockDim.x * gridDim.x;
}

//设置cache中相应位置上的值
cache[cacheIndex] = temp;

//对线程块中的线程进行同步
__syncthreads();

// 对于归约运算来说，以下代码要求threadPerBlock必须是2的幂
int i = blockDim.x/2;
while (i != 0) {
    if (cacheIndex < i)
        cache[cacheIndex] += cache[cacheIndex + i];
    __syncthreads();
    i /= 2;
}

```

在执行的这个位置时，每个线程块中的256个线程都已经把各自计算的乘积相加起来，并且保存到cache[0]中。现在，每个线程块都需要将其临时结果相加到c执行的内存位置上。为了安全地执行这个操作，我们将使用锁来控制对该内存位置的访问，因此每个线程在更新*c的值之前，要先获取这个锁。在将线程块的临时结果与c处的值相加后，将解锁互斥体，这样其他的线程可以继续累加它们的值。在将临时值与最终结果相加后，这个线程块将不再需要任何计算，因此从核函数中返回。

```

    if (cacheIndex == 0) {
        lock.lock();
        *c += cache[0];
        lock.unlock();
    }
}

```

函数main()非常类似于最初的实现，只是存在两个不同。首先，不需要像第5章那样为临时结果分配一个缓冲区，而只需分配一个浮点大小的空间：

```

int main( void ) {

```

```
float    *a, *b, c = 0;
float    *dev_a, *dev_b, *dev_c;

// 在CPU上分配内存
a = (float*)malloc( N*sizeof(float) );
b = (float*)malloc( N*sizeof(float) );

// 在GPU上分配内存
HANDLE_ERROR( cudaMalloc( (void**)&dev_a,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_b,
                           N*sizeof(float) ) );
HANDLE_ERROR( cudaMalloc( (void**)&dev_c,
                           sizeof(float) ) );
```

与第5章一样，初始化输入数组并将它们复制到GPU。但需要注意的是，在这个示例中多了一次复制操作：将0复制到dev_c，我们将在这个位置上累加最终的点积值。由于每个线程块都将读取这个值，然后将这个值与线程块的临时加和相加，并将结果保存回去，因此我们需要将初始值置为0以便得到正确的结果。

```
// 用数据填充主机内存
for (int i=0; i<N; i++) {
    a[i] = i;
    b[i] = i*2;
}

//将数组 'a' 和 'b' 复制到GPU
HANDLE_ERROR( cudaMemcpy( dev_a, a, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_b, b, N*sizeof(float),
                           cudaMemcpyHostToDevice ) );
HANDLE_ERROR( cudaMemcpy( dev_c, &c, sizeof(float),
                           cudaMemcpyHostToDevice ) );
```

剩下的工作就是声明Lock，调用核函数，并将结果复制回CPU。

```

Lock    lock;
dot<<<blocksPerGrid,threadsPerBlock>>>( lock, dev_a,
                                           dev_b, dev_c );

//将数组 'c' 从GPU复制到CPU
HANDLE_ERROR( cudaMemcpy( &c, dev_c,
                          sizeof(float),
                          cudaMemcpyDeviceToHost ) );

```


在第5章示例程序的这个位置上，我们将执行for()循环把所有临时和值相加起来。然而，这个计算已经在GPU上通过原子锁完成了，因此可以直接跳到结果校验和内存释放的代码：

```
#define sum_squares(x) (x*(x+1)*(2*x+1)/6)
printf( "Does GPU value %.6g = %.6g?\n", c,
        2 * sum_squares( (float)(N - 1) ) );

// 释放GPU上的内存
cudaFree( dev_a );
cudaFree( dev_b );
cudaFree( dev_c );

// 释放CPU上的内存
free( a );
free( b );
}
```

由于我们无法通过某种方式来预测每个线程块按照何种顺序将其临时和值与最终结果相加，因此很可能（几乎是肯定的）最终结果的相加顺序与CPU上的相加顺序不同。由于浮点加法的非结合性，因此GPU上的最终结果与CPU上的最终结果之间很可能存在细微的差异。对于这个问题没有太多的解决方法，除非增加大段的代码来确保各个线程块以某种确定的顺序来获取锁，并且这个顺序与CPU上的求和顺序完全相同。如果你觉得有必要这么做，那么不妨一试。接下来，我们将继续介绍如何使用原子锁来实现另一种多线程的数据结构。

A.2 实现散列表

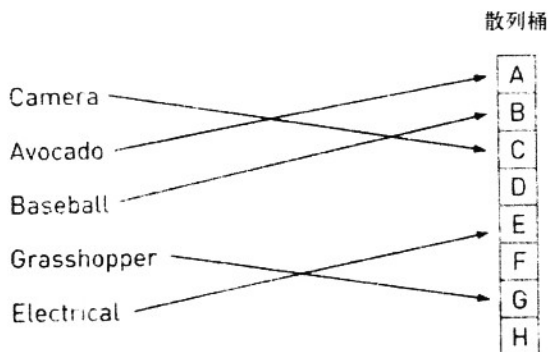
散列表是计算机科学中最重要也是最常用的数据结构之一，它在大量应用程序中都起着重要的作用。对于还不熟悉散列表的读者来说，我们在这里会简要介绍一下。数据结构的研究远比这里给出的内容要深入，但我们只是为了引出接下来的内容，因此将只做简单介绍。如果你已经熟悉了散列表的相关概念，那么可以直接跳到A.2.2节。

A.2.1 散列表简介

散列表是一种保存键-值二元组的数据结构。例如，词典就可以被视为一个散列表。词典中的每个单词都是一个键，并且每个单词都有一个相关的定义。这个定义就是与单词相关的值，这样，词典中的每个单词及其定义就构成了一个键-值二元组。然而，在使用散列表时，重要的是要将在查找与某个键的对应值时需要的时间降至最低。通常，这应该是一个常量时间。也就是说，给定某个键，无论在散列表中保存了多少个键/值二元组，找出与这个键相对应值的时间应该是不变的。

散列表根据与值相应的键，把值放入“桶 (Bucket)”中。这种将键映射到值的方法通常被称为散列函数。好的散列函数可以把键均匀地映射到所有桶中，因为这种方式能够满足定长查找时间的要求，而不论有多少个值被添加到散列表中。

例如，考虑词典散列表。一个很容易想到的散列函数就是使用26个桶，并让每个桶对应于字母表中的一个字母。这个散列函数将查看键的第一个字母，并根据这个字母将放入26个桶中的某一个。在图A.1中给出了这个散列函数如何分配将一些单词放入桶中。



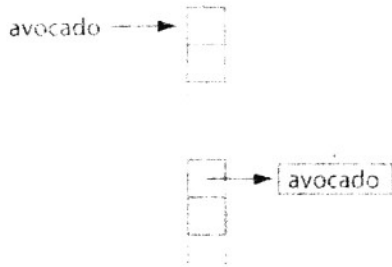
图A.1 通过散列函数将单词分配到各个桶中

我们都知道关于英语中单词的分布情况，这个散列函数远不能令人满意，因为它并不能把单词均匀地映射到26个桶中。在某些桶中包含了很少的键/值二元组，而在另一些桶中又包含大量的二元组。因此，如果某个单词的起始字母是一个常见字母，例如S，那么查找这个单词所花的时间将超过查找以X为起始字母的单词的时间。由于我们希望散列函数对于任意单词都实现定长的查找时间，因此这个结果是不能接受的。国内外有大量关于散列函数的研究，但本书并不会介绍所有这些技术。

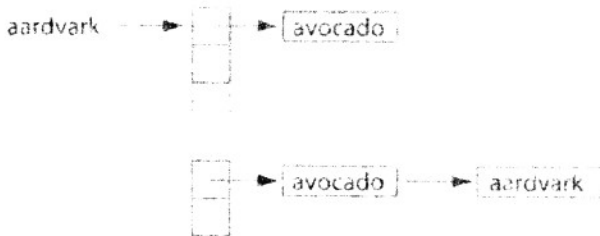
散列表数据结构的最后一个问题是关于桶。如果有一个理想的散列函数，那么每个键都会被映射到一个不同的桶。在这种情况下，我们可以只将键/值二元组保存到一个数组中，并且数组中的每个元素都是一个桶。然而，即使这种理想的散列函数存在，在大多数情况下，我们仍需要处理各种冲突。当有多个键被映射到同一个桶时，就会发生冲突，例如将单词avocado和aardvark同时添加到词典散列表中。在一个桶中保存多个值的最简单方式是，将桶中的所有值保存到一个链表。当遇到冲突时，例如将aardvark添加到已经包含avocadl的词典中时，只需将aardvark放到桶A的链表末尾，如图A.2所示。

在图A.2中添加了单词avocado后，第一个桶的链表中包含了一个键/值二元组。接下来又添加了aardvark，这个单词与avocado发生了冲突，因为这两个单词的起始字母都是A。你将注意

到，图A.3中的aardvark被放在第一个桶的链表末尾。



图A.2 将单词avocado插入到散列表



图A.3 解决当添加单词aardvark时发生的冲突

在了解了散列函数的一些概念和解决冲突的方式之后，我们接下来就来看如何实现散列表。

A.2.2 CPU散列表

在A.2.1节中介绍过，散列表主要包含两个部分：一个散列函数和一个表示桶的数据结构。桶的实现与之前完全一样：我们将分配一个长度为N的数组，并且数组中的每个元素都表示一个键/值二元组链表。在介绍散列函数之前，首先来看看数据结构：

```
#include "../common/book.h"

struct Entry {
    unsigned int    key;
    void*          value;
    Entry          *next;
};

struct Table {
    size_t count;
```

```

Entry    **entries;
Entry    *pool;
Entry    *firstFree;
};

```

正如在A.2.1节中描述的，Entry结构包含了一个键和一个值。在应用程序中，键是无符号整数。与键相关的值可以是任意数据类型，因此我们将value声明为一个void*变量。由于这个程序重点在于说明如何创建散列表数据结构，因此在value域中实际上不保存任何内容。为了保证完整性，我们将它包含在结构中，从而使你可以在自己的应用程序中使用这段代码。结构Entry的最后一个成员是指向下一个Entry节点的指针。在遇到冲突时，在同一个桶中将包含多个Entry节点，因此我们决定将这些对象保存为一个链表。因此，每个对象都将指向桶中的下一个节点，因而就形成了一个节点链表。最后一个Entry节点的next指针为NULL。

本质上，Table结构本身是一个“桶”数组。这个桶数组是一个长度为count的数组，其中entries中的每个桶都是一个指向某个Entry的指针。如果每添加一个Entry节点时都分配新的内存，那么将对程序的性能产生负面影响，为了避免这种情况，散列表将在成员pool中维持一个可用Entry节点的数组。firstFree这个域指向下一个可用的Entry节点，因此当需要将一个节点添加到散列表时，只需使用由firstFree指向的Entry，然后递增这个指针，就避免了新分配内存。注意，这种方式还将简化释放节点内存的代码，因为只需调用free()一次就可以释放所有这些节点。如果在每添加一个节点时都分配新的内存，那么就必须遍历散列表并依次释放每个节点。

在理解了数据结构后，我们来看一些其他的支持代码：

```

void initialize_table( Table &table, int entries,
                      int elements ) {
    table.count = entries;
    table.entries = (Entry**)calloc( entries, sizeof(Entry*) );
    table.pool = (Entry*)malloc( elements * sizeof( Entry ) );
    table.firstFree = table.pool;
}

```

在散列表的初始化过程中，主要操作包括为桶数组entries分配内存。我们还为节点池分配了内存，并将指针firstFree初始化为指向节点池数组中的第一个节点。

在程序的末尾，我们将释放已分配的内存，这将释放桶数组和空闲节点池：

```

void free_table( Table &table ) {
    free( table.entries );
    free( table.pool );
}

```

在对散列表的简介中，我们提到了关于散列函数的不少内容。具体来说，我们讨论了一个好的散列函数在优秀的散列表和糟糕的散列表中将表现出不同的性能。在这个示例中，我们使

用了无符号整数作为键，并且需要将这些键映射到桶数组的索引。实现这个操作的最简单方式就是，将键作为索引。也就是说，将节点e保存在table.entries[e.key]中。然而，我们无法确保键的取值范围将小于桶数组的长度。幸运的是，这个问题的解决方法相对简单：

```
size_t hash( unsigned int key, size_t count ) {
    return key % count;
}
```

既然散列函数非常重要，那么这个简单的函数能否实现我们的目标？理想情况是，我们希望将键均匀地映射到表中所有的桶，并且在这里采取的方式是将键对数组长度取模。在实际情况中，散列函数并不会这么简单，但由于这只是一个示例程序，因此将随机地生成我们的键。如果我们假设随机数值生成器生成的值大致是均匀的，那么这个散列函数应该将这些键均匀地映射到散列表的所有桶中。在你自己的散列表中，则可能需要一个更为复杂的散列函数。

在介绍了这个散列表结构以及散列函数后，接下来将介绍如何将键/值二元组添加到散列中。这个过程包括三个基本的步骤：

- 1) 将键放入散列函数中计算出新节点所属的桶。
- 2) 从节点池中取出一个预先分配的Entry节点，将初始化节点的key和value等域。
- 3) 将这个节点插入到计算得到的桶的链表首部。

我们以一种直观的方式将这些步骤转换为代码。

```
void add_to_table( Table &table, unsigned int key, void* value )
{
    // 步骤1
    size_t hashValue = hash( key, table.count );

    // 步骤2
    Entry *location = table.firstFree++;
    location->key = key;
    location->value = value;

    // 步骤3
    location->next = table.entries[hashValue];
    table.entries[hashValue] = location;
}
```

如果之前没有见过链表（或者有段时间没见过了），那么在理解步骤3时可能会有些困难。链表的第一个节点被保存在table.entries[hashValue]中。在理解了这点后，就可以通过以下两个步骤在链表的头节点中插入一个新节点：首先，将新节点的next指针设置为指向链表的第一个节点。然后，将这个新节点保存到桶数组中，这样它将成为新链表的第一个节点。

为了判断这段代码能否工作，我们实现了一个函数对散列表执行完好性检查。在检查过程中将首先遍历这张表，并查看每个节点。将节点的键放入散列函数计算，并确认这个节点被保存到了正确的桶中。在检查了每个节点后，还要验证散列表中的节点数量确实等于添加到散列表的元素数量。如果这些数值并不相等，那么要么是无意中将一个节点添加到多个桶，要么没有正确地插入节点。

```
#define SIZE      (100*1024*1024)
#define ELEMENTS   (SIZE / sizeof(unsigned int))

void verify_table( const Table &table ) {
    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d hashed to %ld, but was located "
                        "at %ld\n", current->value,
                        hash( current->value, table.count ), i );
            current = current->next;
        }
    }
    if (count != ELEMENTS)
        printf( "%d elements found in hash table. Should be %ld\n",
                count, ELEMENTS );
    else
        printf( "All %d elements found in hash table.\n", count);
}
```

暂时不考虑所有基础代码，我们先来看看main()。与书中的许多示例中一样，大部分复杂的功能都被放入辅助函数中，因此这个main()就变得相对简单：

```
#define HASH_ENTRIES    1024

int main( void ) {
    unsigned int *buffer =
        (unsigned int*)big_random_block( SIZE );

    clock_t start, stop;
    start = clock();

    Table table;
    initialize_table( table, HASH_ENTRIES, ELEMENTS );
```

```

for (int i=0; i<ELEMENTS; i++) {
    add_to_table( table, buffer[i], (void*)NULL );
}

stop = clock();
float   elapsedTime = (float)(stop - start) /
                    (float)CLOCKS_PER_SEC * 1000.0f;
printf( "Time to hash:  %3.1f ms\n", elapsedTime );

verify_table( table );

free_table( table );
free( buffer );
return 0;
}

```

你可以看到，首先分配了一大块内存来保存随机数值。这些随机生成的无符号整数将被作为插入到散列表中的键。在生成了这些数值后，接下来将读取系统时间以便统计程序的性能。我们对散列表进行初始化，然后通过for()循环将每个随机键插入到散列表。在添加了所有的键后，再次读取系统时间，通过之前读取的系统时间与这次系统时间就可以计算出在初始化和添加键上花费的时间。最后，我们通过完好性检查函数来验证散列表，并且释放了已分配的内存。

你可能已经注意到，每个键/值二元组中的值都是NULL。在应用程序中，键/值二元组中的值通常是某个有用的数据，但由于我们在这里主要关注散列表的实现本身，因此将使用一个无意义的值。

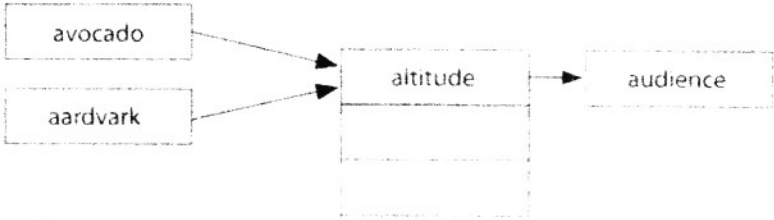
A.2.3 多线程环境下的散列表

在CPU散列表中包含的某些假设在GPU环境下将不再成立。首先，我们假设每次只有一个节点被添加到散列表中，这使得节点的插入操作更加简单。如果有多个线程试图同时将一个节点添加到散列表，那么我们最终将遇到与第9章多线程加法问题中类似的问题。

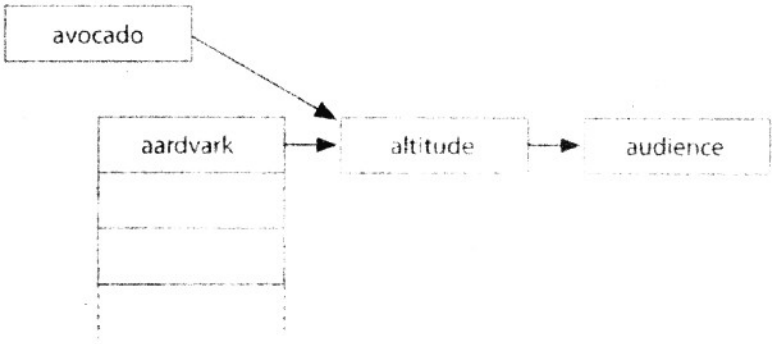
例如，重新回顾“avocado与aardvark”示例，并且想象一下，有两个线程A和B同时尝试将这些节点添加到散列表中。线程A在avocado上计算一个散列函数，线程B在aardvark计算散列函数。它们都计算得到同一个桶。在将新节点添加到链表时，线程A和B首先将它们新节点的next指针设置为指向线程链表中的第一个节点，如图A.4所示。

然后，这两个线程都会尝试用它们各自的新节点替换桶中的节点。然而，只有第二个完成的线程才能将它的添加结果保存下来，因为它将覆盖前一个线程的工作。例如，线程A首先用avocado节点替换altitude节点。在添加完成后，线程B则用它的aardvark节点进行替换，并认为

被替换的节点仍然是altitude。然而，它现在替换的是avocado而不是altitude，结果就导致了图A.5中所示的情况。



图A.4 多个线程都尝试将一个节点添加到同一个桶



图A.5 在两个线程不正确地并发修改后得到的散列表

线程A的节点“漂浮在”散列表之外。幸运的是，在完好性检查函数中能发现并提示这个问题，因为它统计的节点小于我们预期的节点数量。然而，我们仍然需要回答这个问题：如何在GPU上构建一个散列表？我们观察到，每次只有一个线程可以安全地修改桶。这类似于前面的点积运算示例，在点积示例中，每次只有一个线程可以安全地将它的值与最终结果相加。如果每个桶都有一个相应的原子锁，那么我们可以确保每次只有一个线程对指定的桶进行修改。

A.2.4 GPU散列表

在有了某种方法来确保对散列表实现安全的多线程访问，我们可以继续实现在2.2节中GPU版本的散列表应用程序。我们需要包含头文件lock.h，其中包含了A.1.1节中对Lock结构的实现，我们需要把散列函数_声明为一个_device__函数。除了这些修改外，其他的重要数据结构和散列函数与CPU实现中的都相同。

```
#include "../common/book.h"
```



```

#include "lock.h"

struct Entry {
    unsigned int    key;
    void*          value;
    Entry          *next;
};

struct Table {
    size_t count;
    Entry **entries;
    Entry *pool;
};

__device__ __host__ size_t hash( unsigned int value,
                                size_t count ) {
    return value % count;
}

```

在上面的代码中使用了关键字`__host__`，当这个关键字与`__device__`一起使用时，将告诉NVIDIA编译器同时生成函数在设备上和主机上的版本。设备版本的函数将在设备上运行，并且只能从设备代码中调用。同样，主机版本的函数将在主机上运行，并且只能从主机代码中调用。如果在编写函数时，希望这个函数既可以在设备上使用，又可以在主机上使用，那么使用关键字`__host__`将是一种很方便的方式。

在初始化和释放散列表的过程中包含了与CPU版本中相同的步骤，但与之前的示例一样，我们使用CUDA运行时函数来实现这些步骤。我们使用`cudaMalloc()`来分配一个桶数组和一个节点池，并使用`cudaMemset()`将桶数组节点初始化为0。为了在应用程序完成时释放这些内存，我们使用`cudaFree()`。

```

void initialize_table( Table &table, int entries,
                     int elements ) {
    table.count = entries;
    HANDLE_ERROR( cudaMalloc( (void**)&table.entries,
                              entries * sizeof(Entry*) ) );
    HANDLE_ERROR( cudaMemset( table.entries, 0,
                              entries * sizeof(Entry*) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&table.pool,
                              elements * sizeof(Entry) ) );
}

void free_table( Table &table ) {
    cudaFree( table.pool );
    cudaFree( table.entries );
}

```

在CPU实现中，我们通过一个函数来检查散列表的正确性。在GPU版本中需要一个类似的函数。我们可以编写一个在GPU上执行的verify_table()，或者可以使用在CPU版本中相同的代码，同时增加一个函数将散列表从GPU复制到CPU。虽然这两种方式都可以实现我们需要的功能，但第二种方式看上去更好，原因有两个：首先，它可以重用CPU版本的verify_table()。与其他的代码重用情况一样，这将节约开发时间，并且确保将来对代码进行修改时，对于两个版本的散列表只需修改一个地方。其次，在实现复制函数时将遇到一个有趣的问题，这个问题的解决方案在将来或许对你非常有用。

这里的verify_table()与CPU版本中的实现完全相同：

```
#define SIZE      (100*1024*1024)
#define ELEMENTS   (SIZE / sizeof(unsigned int))
#define HASH_ENTRIES    1024

void verify_table( const Table &dev_table ) {
    Table table;
    copy_table_to_host( dev_table, table );

    int count = 0;
    for (size_t i=0; i<table.count; i++) {
        Entry *current = table.entries[i];
        while (current != NULL) {
            ++count;
            if (hash( current->value, table.count ) != i)
                printf( "%d hashed to %ld, but was located "
                        "at %ld\n", current->value,
                        hash(current->value, table.count), i );
            current = current->next;
        }
    }
    if (count != ELEMENTS)
        printf( "%d elements found in hash table. Should be %ld\n",
                count, ELEMENTS );
    else
        printf( "All %d elements found in hash table.\n", count );

    free( table.pool );
    free( table.entries );
}
```

由于我们选择重用CPU版本中的verify_table()，因此需要一个函数将散列表从GPU内存复制到主机内存。这个函数包括三个步骤，其中前两个步骤相对简单，而第三个步骤则相对复杂一些。前两个步骤包括为散列表数据分配主机内存，并通过cudaMemcpy()将GPU上的数据复制

到这块内存。我们已经多次编写了这段代码，因此并不困难。

```
void copy_table_to_host( const Table &table, Table &hostTable) {
    hostTable.count = table.count;
    hostTable.entries = (Entry**)calloc( table.count,
                                         sizeof(Entry*) );
    hostTable.pool = (Entry*)malloc( ELEMENTS *
                                     sizeof( Entry ) );

    HANDLE_ERROR( cudaMemcpy( hostTable.entries, table.entries,
                              table.count * sizeof(Entry*),
                              cudaMemcpyDeviceToHost ) );
    HANDLE_ERROR( cudaMemcpy( hostTable.pool, table.pool,
                              ELEMENTS * sizeof( Entry ),
                              cudaMemcpyDeviceToHost ) );
}
```

这个函数的复杂之处在于，在复制的数据中，有一部分数据是指针。我们不能简单地将这些指针复制到主机，因为这些指针指向的地址是在GPU上，它们在主机上并不是有效的指针。然而，这些指针的相对偏移仍然是有效的。每个指向Entry节点的GPU指针都指向数组table.pool[]中的某个位置，但为了在主机上使用散列表，我们需要它们指向数组hostTable.pool[]中相同的Entry。

给定一个GPU指针X，需要将这个指针相对于table.pool的偏移与hostTable.pool相加，从而获得一个有效的主机指针。也就是说，新指针应该按照以下公式计算：

$$(X - \text{table.pool}) + \text{hostTable.pool}$$

对于每个被复制的Entry指针，都要执行这个更新操作：包括hostTable.entries中的Entry指针，以及散列表的节点池中每个Entry的next指针：

```
for (int i=0; i<table.count; i++) {
    if (hostTable.entries[i] != NULL)
        hostTable.entries[i] =
            (Entry*)((size_t)hostTable.entries[i] -
                    (size_t)table.pool + (size_t)hostTable.pool);
}
for (int i=0; i<ELEMENTS; i++) {
    if (hostTable.pool[i].next != NULL)
        hostTable.pool[i].next =
            (Entry*)((size_t)hostTable.pool[i].next -
                    (size_t)table.pool + (size_t)hostTable.pool);
}
}
```

在介绍完了数据结构、散列函数、初始化过程、内存释放过程以及验证代码后，还剩下的重要部分就是CUDA C原子语句的使用。核函数add_to_table()的参数包括一个键数组、一个值数组、散列表本身以及一个锁数组，这个数组将被用于锁定散列表中的每个桶。由于输入的数据是两个数组，并且在线程中需要对这两个数组进行索引，因此还需要将索引线性化：

```
__global__ void add_to_table( unsigned int *keys, void **values,
                             Table table, Lock *lock ) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    int stride = blockDim.x * gridDim.x;
```

线程会像点积示例那样遍历输入数组。对于数组key[]中的每个键，线程都将通过散列函数计算出这个键/值二元组属于哪个桶。在计算出目标桶之后，线程会锁定这个桶，添加它的键/值二元组，然后解锁这个桶。

```
while (tid < ELEMENTS) {
    unsigned int key = keys[tid];
    size_t hashValue = hash( key, table.count );
    for (int i=0; i<32; i++) {
        if ((tid % 32) == i) {
            Entry *location = &(table.pool[tid]);
            location->key = key;
            location->value = values[tid];
            lock[hashValue].lock();
            location->next = table.entries[hashValue];
            table.entries[hashValue] = location;
            lock[hashValue].unlock();
        }
    }
    tid += stride;
}
```

然而，在这段代码中存在一个非常特别的地方，for()循环和后面的if()语句看上去显然是不必要的。在第6章中引入了线程束的概念。线程束是一个包含32个线程的集合，并且这些线程以步调一致的方式执行。在本书中并不讨论如何在GPU上实现这种步调一致的执行方式，但每次在线程束中只有一个线程可以获取这个锁，并且如果让线程束中的所有32个线程都同时竞争这个锁，那么将发生严重的问题。在这种情况下，最好的方式是在软件中执行一部分工作，遍历线程束中的线程，并给每个线程一次机会来获取数据结构的锁，执行它的工作，然后释放锁。

main()的执行流程看上去与CPU版本的相同。我们首先分配一大块随机数据作为散列表的键。然后，创建起始CUDA事件和停止CUDA事件，并且记录起始事件从而测试程序的性能。

接下来，继续为随机键数组分配GPU内存，将数组复制到GPU设备上并初始化散列表：

```
int main( void ) {
    unsigned int *buffer =
        (unsigned int*)big_random_block( SIZE );

    cudaEvent_t      start, stop;
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );
    HANDLE_ERROR( cudaEventRecord( start, 0 ) );

    unsigned int *dev_keys;
    void          **dev_values;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_keys, SIZE ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_values, SIZE ) );
    HANDLE_ERROR( cudaMemcpy( dev_keys, buffer, SIZE,
                              cudaMemcpyHostToDevice ) );

    // 在这里将值复制到dev_values

    Table table;
    initialize_table( table, HASH_ENTRIES, ELEMENTS );
```

在构建散列表的过程中，最后一个步骤就是为散列表的桶准备好锁。我们为散列表中每个桶都分配一个锁。显然，如果对于整个散列表都只使用一个锁，那么将节约大量的内存。但这么做将严重降低性能，因为当一组线程同时将节点添加到散列表时，每个线程都必须竞争散列表的锁。因此，我们声明了一个锁数组，数组中的每个锁对应于中桶数组中的每个桶。然后，我们为这些锁分配了一个GPU数组，并将它们复制到GPU设备上：

```
Lock    lock[HASH_ENTRIES];
Lock    *dev_lock;
HANDLE_ERROR( cudaMalloc( (void**)&dev_lock,
                          HASH_ENTRIES * sizeof( Lock ) ) );
HANDLE_ERROR( cudaMemcpy( dev_lock, lock,
                          HASH_ENTRIES * sizeof( Lock ),
                          cudaMemcpyHostToDevice ) );
```

main()的剩余部分类似于CPU版本：将键添加到散列表，停止性能计数器，验证散列表的正确性，并且执行内存释放工作。

```
add_to_table<<<60,256>>>>( dev_keys, dev_values,
                             table, dev_lock );

HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
```

```

HANDLE_ERROR( cudaEventSynchronize( stop ) );
float    elapsedTime;
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime,
                                     start, stop ) );
printf( "Time to hash: %3.1f ms\n", elapsedTime );

verify_table( table );

HANDLE_ERROR( cudaEventDestroy( start ) );
HANDLE_ERROR( cudaEventDestroy( stop ) );
free_table( table );
cudaFree( dev_lock );
cudaFree( dev_keys );
cudaFree( dev_values );
free( buffer );
return 0;
}

```

A.2.5 散列表的性能

在Intel Core 2 Duo上, A.2.2节中在100MB的数据中构建散列表时需要360毫秒。在编译代码时使用了编译选项-O3, 以确保最大限度地优化CPU代码。A.2.4节中的多线程GPU散列表在完成任务时需要375毫秒。二者的差异小于百分之五, 执行时间基本上大致相当, 这就引出了一个问题: 为什么像GPU这种大规模并行机器的性能还不如在CPU上执行的单线程应用程序? 坦白地说, 这是因为GPU并不是专门为了提升多线程对复杂数据结构(例如散列表)的访问性能而设计的。基于这个原因, 在GPU上构建散列表这种数据结构很少是为了提升性能。因此, 如果应用程序需要构建一个散列表或者类似的数据结构, 那么最好在CPU上进行实现。

另一方面, 你有时候会发现, 在某个计算流水线中包括1到2个步骤, 这些步骤在GPU上的执行性能并不比在CPU上的执行性能高。在这些情况中, 你可以有三种选择:

- 在GPU上执行流水线的每个步骤。
- 在CPU上执行流水线的每个步骤。
- 在GPU上执行一些步骤, 同时在GPU上执行另一些步骤。

最后一种方式似乎是最好的方式。然而, 这就意味着, 每当应用程序需要将计算从GPU移动到CPU, 或者从CPU移动到GPU时, 都需要对CPU和GPU进行同步。这种同步以及在主机和GPU之间的数据迁移通常会抵消采用混合方法时带来的性能优势。

在这种情况下, 更好的方式或许是在GPU上执行每个步骤, 即便算法的某些步骤并不适合使用GPU。GPU散列表可以减少一次CPU/GPU同步, 这将把主机与GPU之间的数据传输量降至最低, 并且使CPU能执行其他的计算。在这种情况下, 在GPU的执行性能可能会好于在

CPU/GPU上混合执行的性能，尽管在某些步骤上GPU并不会比GPU更快（或者在某些情况中可能要慢于CPU）。

A.3 小结

我们在附录中介绍了如何使用原子的比较-交换操作来实现GPU互斥体。通过使用基于该互斥体构建的锁，我们介绍了如何将最初的点积应用程序修改为完全在GPU上运行。然后，我们基于这个思想进一步实现了一个多线程的散列表，并使用了一个锁数组来防止多个线程同时进行不安全的修改。事实上，我们实现的互斥体可以在任意并行数据结构中使用，并且通过自己的实践你会发现这个互斥体确实有用。当然，如果在应用程序中使用GPU来实现基于互斥体的数据结构，那么需要对应用程序的性能进行分析。GPU散列表的性能比CPU散列表的性能要更糟糕，因此只有在某些特定情况下，在这种类型的应用程序中使用GPU才是有意义的。在判断是只使用GPU，还是只使用CPU或者使用混合方法才能实现最优性能时，并没有固定的规则可供遵循，但如果知道如何使用原子语句，那么你就能根据具体的情况做出决策。

GPU高性能编程 CUDA实战

CUDA By Example

an Introduction to General-Purpose
GPU Programming

“对于开发基于GPU加速的并行计算
系统的读者来说，本书绝对值得一读。”

—— Jack Dongarra

田纳西大学杰出教授
美国橡树岭国家实验室杰出研究人员

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，与综合性软件平台相结合，CUDA架构能充分发挥GPU的强大计算功能。很长时间以来，GPU一直用于图形和游戏应用程序中。但是现在，使用CUDA可将GPU用于科学计算、工程以及金融等其他应用领域。由于在CUDA中使用的编程语言只是一种对标准C语言进行简单扩展的语言，所以开发人员不需要具备任何计算机图形学的背景知识就可以掌握。

本书由CUDA软件平台小组的两位高级工程师撰写，向广大程序员介绍了如何使用这项新技术。作者通过多个示例详细介绍了CUDA开发中的方方面面。本书首先简要介绍了CUDA平台和架构，并快速介绍了CUDA C，随后详细介绍了CUDA每个功能中的关键技术与权衡因素。通过学习这些内容，你可以很清楚地了解CUDA C中每个功能的适用场合，并编写出高性能的CUDA软件。

本书主要内容：

- 并行编程
- 线程协作
- 常量内存与事件
- 纹理内存
- 图形互操作
- 原子操作
- 流
- 多GPU上的CUDA C
- 高级原子操作
- 其他CUDA资源

所需的CUDA软件工具均可以从NVIDIA公司的网站免费下载。

作者简介

Jason Sanders 是NVIDIA公司CUDA平台小组的高级软件工程师，他参与了CUDA系统软件早期版本的开发，并参与了OpenCL 1.0规范的制定，该规范是一个定义异构计算的行业标准。他曾经在ATI 技术公司、Apple公司以及Novell公司工作过。

Edward Kandrot 是NVIDIA公司CUDA平台小组的高级软件工程师。他在代码性能优化方面拥有20多年的工作经验，他曾经在Adobe公司、Microsoft公司以及Autodesk公司等工作过。

PEARSON

www.pearsonhighered.com

客服热线: (010) 88378991, 88361066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzjsj@hzbook.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计: 陈静

上架指导: 计算机 / 程序设计

ISBN 978-7-111-32679-3



9 787111 326793

定价: 39.00元

[General Information]

书名=GPU高性能编程CUDA实战

作者=（美）桑德斯著；聂雪军等译

页数=201

SS号=12804258

出版日期=2011.01

出版社=机械工业出版社

原书定价=39.00

参考文献格式=（美）桑德斯著.GPU高性能编程CUDA实战.北京市：机械工业出版社,2011.01.

内容提要=CUDA是用来促进并行程序开发的一种计算架构。它与各种广泛的软件平台一起使用，使得程序员在构建高性能的应用程序的时候，可以借助图形处理单元(GPU)的强大力量。尽管GPU在图形和游戏编程领域应用多年，现在，CUDA使得开发其他领域的应用程序的程序员，也能够使用GPU的宝贵资源。本书由CUDA软件平台团队的两位高级成员撰写，介绍程序员如何利用这一新的技术。作者介绍了CUDA的各个方面，及其高级功能，还介绍了如何使用CUDA C扩展，以及如何编写真正表现出优秀性能的CUDA软件。