



PEARSON

英伟达公司CUDA首席架构师Nicholas Wilt亲笔撰写，英伟达中国首批CUDA官方认证
工程师翻译

全面而系统地讲解CUDA编程的各方面知识，深度解析CUDA各种优化技术，包含大量
实用代码示例，是深入掌握主流异构并行计算技术的权威指南

高性能计算系列丛书

The CUDA Handbook
A Comprehensive Guide to GPU Programming

CUDA专家手册

GPU编程权威指南

(美) Nicholas Wilt 著

苏统华 马培军 刘曙 吕家明〇译 英伟达中国〇技术审校



机械工业出版社
China Machine Press

CUDA专家手册

GPU编程权威指南

The CUDA Handbook
A Comprehensive Guide to GPU Programming

本书详细讨论CUDA的硬件和软件，包括CUDA 5.0和开普勒架构的最新特性。每个CUDA开发人员，不论新手还是高手，都可以在这里找到感兴趣的内容并即时上手。新晋的CUDA开发者将理解硬件如何处理命令以及驱动程序如何检查状态；更有经验者，将会在驱动程序API、上下文迁移以及如何让CPU/GPU最有效地进行数据交换和同步等骨灰级的主题上得到指导。

本书所附的开源代码有25000多行，欢迎开发者自由重用。

本书不仅是权威手册，也是实用代码大全。全书分为以下三个部分：

- 第一部分是基础知识概述，对支持CUDA的硬件和软件进行高屋建瓴的描述。
- 第二部分是CUDA编程细节，对CUDA进行全方位的描述，包括内存，流和事件，执行模型（包括动态并行特性以及CUDA 5.0和SM 3.5的新特性），流处理器簇（包括SM 3.5的所有功能介绍），多GPU编程，纹理操作。这部分附带的源代码作为可重用的验证型代码和演示型代码，旨在展示特殊的硬件特性或强调特定的应用方法。
- 第三部分是案例剖析，深入分析精选的CUDA应用场景以及关键的并行算法，包括流式负载、归约、扫描（并行前缀求和）、N-体问题和图像处理，这些算法全方位涵盖各种CUDA应用场景。



上架指导：计算机/并行计算

ISBN 978-7-111-47265-0



9 787111 472650 >

定价：85.00元

PEARSON

www.pearson.com

投稿热线：(010) 88379604

客服热线：(010) 88378991 88361066

购书热线：(010) 68326294 88379649 68995259



华章网站：www.hzbook.com

网上购书：www.china-pub.com

数字阅读：www.hzmedia.com.cn

The CUDA Handbook
A Comprehensive Guide to GPU Programming

CUDA专家手册

GPU编程权威指南

(美) Nicholas Wilt 著
苏统华 马培军 刘曙 吕家明◎译 英伟达中国◎技术审校



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

CUDA 专家手册：GPU 编程权威指南 / (美) 威尔特 (Wilt. N.) 著；苏统华等译。—北京：
机械工业出版社，2014.8

(高性能计算系列丛书)

书名原文：The CUDA Handbook: A Comprehensive Guide to GPU Programming

ISBN 978-7-111-47265-0

I. C… II. ①威… ②苏… III. 图象处理－程序设计－手册 IV. TP391.41-62

中国版本图书馆 CIP 数据核字 (2014) 第 161732 号

本书版权登记号：图字：01-2013-5981

Authorized translation from the English language edition, entitled *The CUDA Handbook: A Comprehensive Guide to GPU Programming*, 9780321809469 by Nicholas Wilt, published by Pearson Education, Inc., Copyright © 2013.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2014.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内（不包括中国台湾地区和中国香港、澳门特别行政区）独家出版发行。未经出版者书面许可，不得以任何方式抄袭、复制或节录本书中的任何部分。

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

CUDA 专家手册：GPU 编程权威指南

[美] Nicholas Wilt 著

出版发行：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：关 敏

责任校对：董纪丽

印 刷：三河市宏图印务有限公司

版 次：2014 年 8 月第 1 版第 1 次印刷

开 本：186mm×240mm 1/16

印 张：23.25

书 号：ISBN 978-7-111-47265-0

定 价：85.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：(010) 88378991 88361066

投稿热线：(010) 88379604

购书热线：(010) 68326294 88379649 68995259

读者信箱：hzjsj@hzbook.com

版权所有 • 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Forward 中文版序

得知本书正在被翻译成瑰丽的汉字，我难抑内心的激动。众核计算对中国读者来说并不陌生。中国已经建造了多个最快的超级计算机，其中很多采用了 CUDA 技术。我们耳熟能详的天河 -1A，建成于 2010 年 10 月，配备了 7168 片特斯拉（Tesla）M2050 型号的 GPU。本书第 4 章介绍的亚马逊 cg1.4xlarge 实例，也采用了同种型号的 GPU。天河 -1A 曾雄霸世界超级计算机 500 强榜首逾半年之久，即使现在，它仍排在世界最快计算机的前 12 位。它惊人的高速度源于中国自主研发的互联技术把来自英特尔和英伟达的硬件高效地集成起来了。

当然，CUDA 并非只能应用于超级计算机。GPU 的出货量巨大，相较其他众核计算技术，它们具有价格优势。因此，支持 CUDA 的 GPU 很容易用来搭建经济的计算平台，这对于并行程序设计的教育和教学意义重大。

CUDA 技术仍在迅猛发展着。我敢断定，它将继续在并行计算的大潮中扮演重要角色。本书向大家展示了 CUDA 的内部工作机制，此中译本将推动 CUDA 在中国的普及和发展。

最后，向承担本书翻译工作的苏博士及其团队，致以诚挚谢意！感谢他们孜孜不倦地致力于 GPU 计算的推广，特别是把本书带给广大中国读者。

Nicholas Wilt

推荐序 *Forward*

自 CPU 单核性能在 2004 年左右停止提升后，功耗的限制及大量并行应用本身的特点决定了多核（CPU）加众核（GPU）的异构架构成为从最快的超级计算机到手机等移动设备上计算芯片的主流架构。所以如果想在现在和未来的任何计算设备上写出性能具有竞争力的程序，所有的开发者都需要学习异构架构下的并行计算技术。CUDA 是目前最成熟、使用者最多的异构并行计算技术。本书对开发者更深入地掌握主流异构并行计算技术会有很大帮助。

程序优化的目的是提高整体性能，所以优化时需要做的第一件事就是找出程序的哪个部分需要提高整体性能，即瓶颈分析。这是至关重要而又经常被刚接触优化的开发者忽略的一件事。如果不先进行定量的瓶颈分析，而是立刻优化自己认为是瓶颈的地方，经常会事倍功半。瓶颈分析的第一步是热点分析，即找出哪些部分占程序的大部分运行时间。热点分析可以通过在源代码里加时间测量代码或使用性能分析工具（profiler）进行。当确认整体瓶颈是 GPU 的函数或数据传输后，读者就可参阅本书对应的 GPU 优化知识进行优化。

本书第一部分共 4 章，介绍 GPU 的架构、系统软件及编程环境。性能优化本质上就是通过理解架构及相应系统软件的特点，并在程序中利用这些特点更高效地完成计算。所以优化的第一步就应该是理解目标平台的架构及系统软件基础。这部分讨论的很多内容是第一次在公开的 CUDA 书籍中出现。

第二部分共 6 章，介绍 CUDA 编程各个方面的具体知识，特别是如何利用 GPU 特点的很多优化技术。

如果性能分析告诉我们 CPU-GPU 数据传输是瓶颈，那么可以参考第 6 章使用流来计算隐藏数据传输；另一方面，如果 GPU 函数的执行是瓶颈，那么下一步就是分析内核函数的架构瓶颈。此时，定量分析也是至关重要的。如果不先做定量分析找出瓶颈在哪里，而是立刻随机地尝试各种优化技术，往往难以收到成效。这样随机尝试，可能有些优化恰好有效

果，有些优化则没有效果，会浪费大量时间在尝试各种技术上。另外，我们也无法明白为什么一种优化有效果而其他的却没有效果。除此之外，如果不做分析，我们也无从得知一个内核函数到底优化到何种程度就可以结束，从而有可能花费大量时间在一个已没有太大优化空间的内核函数上。这些都是优化技术的初学者很容易觉得优化是一件复杂而低效的工作的原因。但其实如果能采用先做定量分析再采用相应的优化技术的方法，优化完全可以变成一种高效、有据可依的技术。

对内核函数做分析，本质就是找出内核函数的性能瓶颈在架构的什么地方。一般情况下，性能瓶颈有三种可能类型。

1. 内存密集型。这种情况是内核函数的大部分时间都花在数据的读写指令上。从硬件的角度看，在这个内核函数运行期间，大部分时间都花在数据在内存总线的传输上并且内存总线的带宽已充分利用。对这种情况，如何优化可以参考第 5 章及第 10 章的内容。

2. 指令密集型。这种情况是内核函数的大部分时间都花在各种指令的执行上。从硬件的角度看，在这个内核函数运行期间，大部分时间都花在处理其指令单元的执行上。对这种情况，如何优化可以参考第 8 章的内容。

3. 延迟密集型。这种情况是内核函数的大部分时间都花在等待高延迟的指令（如内存读写）上。从硬件的角度看，在这个内核函数运行期间，大部分时间都花在数据在内存总线的传输上但内存总线的带宽没有被充分利用。对这种情况，如何优化可以参考第 7 章占有率的内容。

本书的第三部分共 5 章，给出了多个领域的具体例子，介绍如何应用各种优化技术。本部分的内容有助于读者学以致用。

虽然本书具体讨论 CUDA，但并行优化的技术及想法对几乎任何众核架构都是适用的。所以读者如果在阅读本书了解具体 CUDA 优化技术之外，也能有意识地体会蕴含其中的并行优化的基本方法，会有更大收获。

王鹏

英伟达高性能计算开发技术经理

译者序 *Words of the Translator's*

CUDA 入门相对比较容易。也许只需寥寥几个小时的学习，你就可以开始编写支持 CUDA 的程序，初窥 GPU 加速的魅力。但是要修炼成为榨取 GPU 处理能力的高级 CUDA 工程师，却并非易事。你需要了解并行计算的基本理论，需要研究 CPU/GPU 的硬件架构，需要熟悉 CUDA 的软件抽象架构和工具，需要掌握必要的领域知识，等等。目前，可以助你入门的 CUDA 书籍和教程为数不少，但在利用 CUDA 来深度榨取 GPU 潜力方面，仍几乎空白。

本书为深度优化 CUDA 程序性能提供了全方位的指导。乍看起来，本书只是一本参考手册。不错，它确实基于英伟达官方 CUDA 参考手册的很多内容。然而它又不单单是一本传统意义上的手册，它比较全面地介绍了 CUDA 的高级编程特性，是该领域难得的权威参考书。全书高屋建瓴而又简明扼要。作者从繁芜的文献资源中提炼并予以升华形成独到的总结。对很多复杂的主题，能够举重若轻；在很多关键的问题上，给大家一种豁然开朗之感。很多精辟论述是市面上任何一本 GPU 书籍所缺失的。译者在阅读本书时，亦为作者在技术和语言上的双重驾驭能力所折服。

本书作者 Wilt 从事 GPU 底层开发逾 20 年，功力深厚。在 CUDA 诞生之前，从事了 8 年 Direct3D 的开发；在英伟达供职的 8 年间，参与了 CUDA 的开发，实现了多数 CUDA 的抽象机制；目前转战亚马逊，从事 GPU 云产品的开发。作者根据自身多年的经验积淀，对 CUDA 编程模型进行了深入浅出的介绍，并结合四个典型应用场合，完美诠释了它们的具体用法和优化过程。得益于本书的启发，我们研究组基于 CUDA 的文字识别算法得到 194X 的加速，使我们能够抢在其他竞争者之前，取得 2013 年手写汉字识别竞赛的两项冠军。很多时候，速度意味着胜利！

本书的技术深度和欲覆盖的读者层面，决定了本书翻译工作的难度。本书的翻译持续

了一年之久，期间得到过很多人的帮助。除了本书列出的四位译者外，还有 5 位译者参与了本书的初稿翻译：韦振良参与了第 2 章和第 7 章，李硕参与了第 4 章和第 6 章，李松泽参与了第 10 章，孙黎参与了第 14 章，胡光龙参与了第 15 章。其中，与光龙是在参加第一届英伟达认证工程师的考试途中相遇，我们一起迷了路、一起考试过关、一起分享 CUDA 技术点滴，又一起在本书的翻译期间进行合作，谢谢光龙的友谊和无私帮助。在本书译文统稿之后，我们努力想找到合适的 CUDA 专家为我们把关。如果没有英伟达公司的侯宇涛经理，这可能会变成无法完成的任务。感谢侯经理为我们推荐了最优秀的技术审校专家，帮我们邀请了高性能计算开发技术中国区经理王鹏博士为本译作写序并选定本译作在全国推广。感谢英伟达高性能计算部门的专家王泽寰和赖俊杰，即使在春节期间，他们也在为改进本书的译稿质量而绞尽脑汁。如果没有泽寰和赖经理的审校，原书的内容恐难以尽可能原汁原味地展现。王鹏经理特意为本书作了推荐序，其中浓缩了他多年的 CUDA 性能优化经验；非常感谢，您真的做到了“读者最需要什么，您就写点什么”的宗旨。最后，还要特别感谢机械工业出版社的编辑团队，他们做了大量卓有成效的工作，是本书的幕后英雄。

本书的翻译，还得到了多项项目的资助，在此一并致谢。国家自然科学基金（资助号：61203260）、黑龙江省科研启动基金（资助号：LBH-Q13066）、哈尔滨工业大学科研创新基金（资助号：HITNSRIF2015083）对本书的翻译提供了部分资助。另外，哈尔滨工业大学创新实验课《CUDA 高性能并行程序设计》、黑龙江省教育厅高等教育教学改革项目（资助号：JG2013010224）、哈尔滨工业大学研究生教育教学改革研究项目（资助号：JCJS-201309）也对本书的翻译提供了大力支持。

很多时候，翻译技术书籍是件吃力不讨好的事情。即使是最有资格的译者，也一定无法免受被读者指责的境遇。对读者负责，对 CUDA 教学推广负责，是我和我的团队始终铭记的准则。我们很享受翻译本书的过程，也很愿意承担因我们的水平和疏忽所招致的批评。文中可能存在的任何翻译问题，都是我们的责任。真诚地希望读者朋友不吝赐教，欢迎大家的任何意见：cudabook@gmail.com。

并行计算是计算的未来。希望本书的翻译可以帮助你让并行计算与大数据成功联姻，并享用由此带来的前所未有的新鲜体验。愿你的 CUDA 探索之旅愉快！

苏统华
哈尔滨工业大学软件学院

前　　言 *Preface*

如果你正在读本书，意味着我无须再向你兜售 CUDA。你应该已经对 CUDA 有所了解，可能使用过英伟达的 SDK 和文档，抑或参加过并行程序设计的课程，再或者阅读过类似《CUDA by Example》(Jason Sanders 和 Edward Kandrot 著，2011 年由 Addison-Wesley 出版) 这样的入门书籍。

我在审校《CUDA by Example》时，惊诧于该书内容的浅显。它假设读者是零基础，试图描述从内存类型及其实际应用到图形互操作性，甚至到原子操作等方方面面的内容。它是很优秀的 CUDA 入门书籍，但也只能做到泛泛而谈。对于更深层次的知识，例如，平台的工作机理、GPU 的硬件结构、编译器驱动程序 nvcc 以及以前缀求和（“扫描”）为代表的基本并行算法，则鲜有涉及。

本书考虑到不同基础的读者，旨在帮助 CUDA 新手进阶中级水平，同时帮助中级程序员继续提升他们的水平到一个新高度。对于入门性质的书籍，最好从头到尾阅读，而对本书则可以根据需要选读。如果你正准备建立支持 CUDA 的新平台并在上面进行编程，建议你精读第 2 章。如果你正纳闷你的应用程序是否将受益于 CUDA 流带来的额外并发性，你应该查看第 6 章。其他的章节分别提供了软件架构、GPU 的纹理操作和流处理器簇等 GPU 子系统的详细描述，还有依据不同数据存取模式和在并行算法领域的重要程度而精心挑选的应用案例。尽管不同章之间难免存在交互引用，但每一章的内容都是相对独立的。

本书将披露包括 CUDA 5.0 在内的最新技术。最近几年，CUDA 和它的目标平台得到了长足发展。在《CUDA by Example》出版之际，GeForce GTX 280 (GT200) 才刚刚面世。迄今，CUDA 硬件已历经两代演变。因此，本书除了在如映射锁页内存 (mapped pinned memory) 的现有特性上不吝笔墨外，还特别关注 CUDA 支持的新特性，像费米架构的 ballot、开普勒架构的 shuffle、64 位和统一虚拟寻址特性以及动态并行 (dynamic parallelism)

等。此外，本书还将讨论最近的平台进展，例如英特尔沙桥（Sandy Bridge）CPU 上集成的 PCIe 总线控制器。

尊敬的读者，你可以全篇通读本书，也可以把它放于电脑边随时查阅。但不管怎样，我真诚地希冀你能从中得到乐趣，如同我执笔分享时一样快意。

致谢

借此机会，感谢英伟达公司的朋友们。他们耐心地为我释疑、检视我的作品并反馈建设性意见。特别的谢意送给 Mark Harris、Norbert Juffa 和 Lars Nyland。

本书的审校者在成稿之前审阅了本书，他们付出了大量时间，提供的宝贵意见提高了本书的质量和清晰性，保证了技术的正确性。在此，特别感谢 Andre Brodtkorb、Scott Le Grand、Allan MacKinnon、Romelia Salomon-Ferrer 和 Patrik Tennberg 的反馈意见。

本书的写作过程历经了重重困难，如果没有编辑 Peter Gordon 的超凡耐心和支持，很难最终呈现给大家。Peter 的助手 Kim Boedigheimer 为本项目的顺利完成做了大量工作，帮助我设定了项目的专业标准。对她在征求、协调评审意见以及在本书成稿之后把本书上传到 Safari 网站的整个过程中付出的努力，表示特别感谢。

在本书的写作过程中，我的妻子 Robin 和我的儿子 Benjamin、Samuel 和 Gregory 一直对我支持有加，谢谢他们。

目 录 *Contents*

中文版序
推荐序
译者序
前 言

第一部分 基础知识

第1章 简介	2
1.1 方法	4
1.2 代码	4
1.2.1 验证型代码	5
1.2.2 演示型代码	5
1.2.3 探究型代码	5
1.3 资源	5
1.3.1 开源代码	5
1.3.2 CUDA 专家手册库 (chLib)	6
1.3.3 编码风格	6
1.3.4 CUDA SDK	6
1.4 结构	6
第2章 硬件架构	8
2.1 CPU 配置	8

2.1.1 前端总线	9
2.1.2 对称处理器簇	9
2.1.3 非一致内存访问 (NUMA)	10
2.1.4 集成的 PCIe	12
2.2 集成 GPU	13
2.3 多 GPU	14
2.4 CUDA 中的地址空间	17
2.4.1 虚拟寻址简史	17
2.4.2 不相交的地址空间	20
2.4.3 映射锁页内存	21
2.4.4 可分享锁页内存	21
2.4.5 统一寻址	23
2.4.6 点对点映射	24
2.5 CPU/GPU 交互	24
2.5.1 锁页主机内存和命令缓冲区	25
2.5.2 CPU/GPU 并发	26
2.5.3 主机接口和内部 GPU 同步	29
2.5.4 GPU 间同步	31
2.6 GPU 架构	31
2.6.1 综述	31
2.6.2 流处理器簇	34
2.7 延伸阅读	37
第3章 软件架构	39
3.1 软件层	39
3.1.1 CUDA 运行时和驱动程序	40
3.1.2 驱动程序模型	41
3.1.3 nvcc、PTX 和微码	43
3.2 设备与初始化	45
3.2.1 设备数量	46
3.2.2 设备属性	46

3.2.3 无 CUDA 支持情况	48
3.3 上下文	50
3.3.1 生命周期与作用域	51
3.3.2 资源预分配	51
3.3.3 地址空间	52
3.3.4 当前上下文栈	52
3.3.5 上下文状态	53
3.4 模块与函数	53
3.5 内核 (函数)	55
3.6 设备内存	56
3.7 流与事件	57
3.7.1 软件流水线	57
3.7.2 流回调	57
3.7.3 NULL 流	57
3.7.4 事件	58
3.8 主机内存	59
3.8.1 锁页主机内存	60
3.8.2 可分享的锁页内存	60
3.8.3 映射锁页内存	60
3.8.4 主机内存注册	60
3.9 CUDA 数组与纹理操作	61
3.9.1 纹理引用	61
3.9.2 表面引用	63
3.10 图形互操作性	63
3.11 CUDA 运行时与 CUDA 驱动程序 API	65
第 4 章 软件环境	69
4.1 nvcc——CUDA 编译器驱动程序	69
4.2 ptxas——PTX 汇编工具	73
4.3 cuobjdump	76
4.4 nvidia-smi	77

4.5 亚马逊 Web 服务	79
4.5.1 命令行工具	79
4.5.2 EC2 和虚拟化	79
4.5.3 密钥对	80
4.5.4 可用区域 (AZ) 和地理区域	81
4.5.5 S3	81
4.5.6 EBS	81
4.5.7 AMI	82
4.5.8 EC2 上的 Linux	82
4.5.9 EC2 上的 Windows	83

第二部分 CUDA 编程

第 5 章 内存	88
5.1 主机内存	89
5.1.1 分配锁页内存	89
5.1.2 可共享锁页内存	90
5.1.3 映射锁页内存	90
5.1.4 写结合锁页内存	91
5.1.5 注册锁页内存	91
5.1.6 锁页内存与统一虚拟寻址	92
5.1.7 映射锁页内存用法	92
5.1.8 NUMA、线程亲和性与锁页内存	93
5.2 全局内存	95
5.2.1 指针	96
5.2.2 动态内存分配	97
5.2.3 查询全局内存数量	100
5.2.4 静态内存分配	101
5.2.5 内存初始化 API	102
5.2.6 指针查询	103

5.2.7 点对点内存访问.....	104
5.2.8 读写全局内存.....	105
5.2.9 合并限制.....	105
5.2.10 验证实验：内存峰值带宽.....	107
5.2.11 原子操作.....	111
5.2.12 全局内存的纹理操作.....	113
5.2.13 ECC (纠错码).....	113
5.3 常量内存.....	114
5.3.1 主机与设备常量内存.....	114
5.3.2 访问常量内存.....	114
5.4 本地内存.....	115
5.5 纹理内存.....	118
5.6 共享内存.....	118
5.6.1 不定大小共享内存声明.....	119
5.6.2 束同步编码.....	119
5.6.3 共享内存指针.....	119
5.7 内存复制.....	119
5.7.1 同步内存复制与异步内存复制.....	120
5.7.2 统一虚拟寻址.....	121
5.7.3 CUDA 运行时.....	121
5.7.4 驱动程序 API.....	123
第6章 流与事件.....	125
6.1 CPU/GPU 的并发：隐藏驱动程序开销.....	126
6.2 异步的内存复制.....	129
6.2.1 异步的内存复制：主机端到设备端.....	130
6.2.2 异步内存复制：设备端到主机端.....	130
6.2.3 NULL 流和并发中断.....	131
6.3 CUDA 事件：CPU/GPU 同步.....	133
6.3.1 阻塞事件.....	135
6.3.2 查询.....	135

6.4 CUDA 事件：计时	135
6.5 并发复制和内核处理	136
6.5.1 concurrencyMemcpyKernel.cu	137
6.5.2 性能结果	141
6.5.3 中断引擎间的并发性	142
6.6 映射锁页内存	143
6.7 并发内核处理	145
6.8 GPU/GPU 同步：cudaStreamWaitEvent()	146
6.9 源代码参考	147
第 7 章 内核执行	148
7.1 概况	148
7.2 语法	149
7.2.1 局限性	150
7.2.2 高速缓存和一致性	151
7.2.3 异步与错误处理	151
7.2.4 超时	152
7.2.5 本地内存	152
7.2.6 共享内存	153
7.3 线程块、线程、线程束、束内线程	153
7.3.1 线程块网格	153
7.3.2 执行保证	156
7.3.3 线程块与线程 ID	156
7.4 占用率	159
7.5 动态并行	160
7.5.1 作用域和同步	161
7.5.2 内存模型	162
7.5.3 流与事件	163
7.5.4 错误处理	163
7.5.5 编译和链接	164
7.5.6 资源管理	164

7.5.7 小结	165
第 8 章 流处理器簇	167
8.1 内存	168
8.1.1 寄存器	168
8.1.2 本地内存	169
8.1.3 全局内存	170
8.1.4 常量内存	171
8.1.5 共享内存	171
8.1.6 栅栏和一致性	173
8.2 整型支持	174
8.2.1 乘法	174
8.2.2 其他操作 (位操作)	175
8.2.3 漏斗移位 (SM 3.5)	175
8.3 浮点支持	176
8.3.1 格式	176
8.3.2 单精度 (32 位)	180
8.3.3 双精度 (64 位)	181
8.3.4 半精度 (16 位)	181
8.3.5 案例分析: float 到 half 的转换	182
8.3.6 数学函数库	185
8.3.7 延伸阅读	190
8.4 条件代码	191
8.4.1 断定	191
8.4.2 分支与汇聚	191
8.4.3 特殊情况: 最小值、最大值和绝对值	192
8.5 纹理与表面操作	193
8.6 其他指令	193
8.6.1 线程束级原语	193
8.6.2 线程块级原语	194
8.6.3 性能计数器	195

8.6.4 视频指令	195
8.6.5 特殊寄存器	196
8.7 指令集	196
第 9 章 多 GPU	203
9.1 概述	203
9.2 点对点机制	204
9.2.1 点对点内存复制	204
9.2.2 点对点寻址	205
9.3 UVA：从地址推断设备	206
9.4 多 GPU 间同步	207
9.5 单线程多 GPU 方案	208
9.5.1 当前上下文栈	208
9.5.2 N- 体问题	210
9.6 多线程多 GPU 方案	212
第 10 章 纹理操作	216
10.1 简介	216
10.2 纹理内存	217
10.2.1 设备内存	217
10.2.2 CUDA 数组与块的线性寻址	218
10.2.3 设备内存与 CUDA 数组对比	222
10.3 一维纹理操作	223
10.4 纹理作为数据读取方式	226
10.4.1 增加有效地址范围	226
10.4.2 主机内存纹理操作	228
10.5 使用非归一化坐标的纹理操作	230
10.6 使用归一化坐标的纹理操作	237
10.7 一维表面内存的读写	238
10.8 二维纹理操作	240
10.9 二维纹理操作：避免复制	242

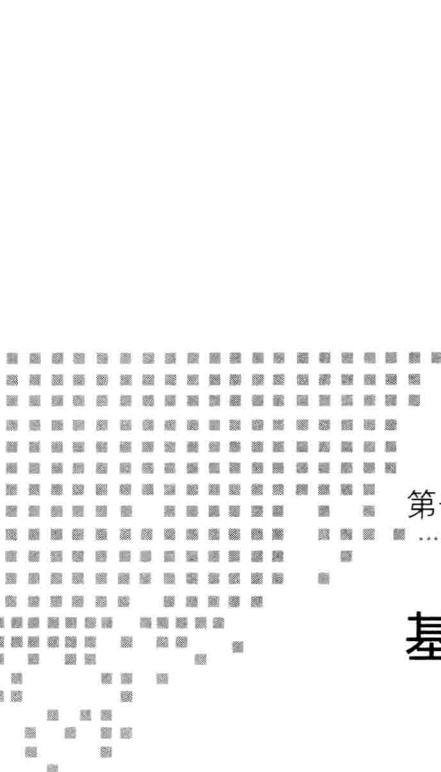
10.9.1	设备内存上的二维纹理操作	242
10.9.2	二维表面内存的读写	243
10.10	三维纹理操作	244
10.11	分层纹理	245
10.11.1	一维分层纹理	246
10.11.2	二维分层纹理	246
10.12	最优线程块大小选择以及性能	246
10.13	纹理操作快速参考	248
10.13.1	硬件能力	248
10.13.2	CUDA 运行时	249
10.13.3	驱动 API	250

第三部分 实例

第 11 章 流式负载	254	
11.1	设备内存	255
11.2	异步内存复制	258
11.3	流	259
11.4	映射锁页内存	260
11.5	性能评价与本章小结	261
第 12 章 归约算法	263	
12.1	概述	263
12.2	两遍归约	265
12.3	单遍归约	269
12.4	使用原子操作的归约	271
12.5	任意线程块大小的归约	272
12.6	适应任意数据类型的归约	273
12.7	基于断定的归约	276
12.8	基于洗牌指令的线程束归约	277

第 13 章 扫描算法	278
13.1 定义与变形	278
13.2 概述	279
13.3 扫描和电路设计	281
13.4 CUDA 实现	284
13.4.1 先扫描再扇出	284
13.4.2 先归约再扫描 (递归)	288
13.4.3 先归约再扫描 (两阶段)	291
13.5 线程束扫描	294
13.5.1 零填充	295
13.5.2 带模板的版本	296
13.5.3 线程束洗牌	297
13.5.4 指令数对比	298
13.6 流压缩	300
13.7 参考文献 (并行扫描算法)	302
13.8 延伸阅读 (并行前缀求和电路)	303
第 14 章 N- 体问题	304
14.1 概述	305
14.2 简单实现	309
14.3 基于共享内存实现	312
14.4 基于常量内存实现	313
14.5 基于线程束洗牌实现	315
14.6 多 GPU 及其扩展性	316
14.7 CPU 的优化	317
14.8 小结	321
14.9 参考文献与延伸阅读	323
第 15 章 图像处理的归一化相关系数计算	324
15.1 概述	324
15.2 简单的纹理实现	326

15.3 常量内存中的模板	329
15.4 共享内存中的图像	331
15.5 进一步优化	334
15.5.1 基于流处理器簇的实现代码	334
15.5.2 循环展开	335
15.6 源代码	336
15.7 性能评价	337
15.8 延伸阅读	339
附录 A CUDA 专家手册库	340
术语表	347



第一部分 *Part 1*

基础知识

- 第1章 简介
 - 第2章 硬件架构
 - 第3章 软件架构
 - 第4章 软件环境
- *****

简介

GPU 带给计算领域以巨大变革，这已在历史上留下了浓墨重彩的一笔。我喜欢饶有兴趣地阅读这类文字，因为我很早就参与其中了。在 20 世纪 90 年代中期，我任职于微软。当时英特尔和 AMD 正在引入第一批用以加速浮点计算的多媒体指令集，我领导一个开发小组开发 Direct3D 产品。英特尔为了阻止移植他们的 CPU 计算能力到三维栅格化任务，他们尝试跟微软合作出售使用他们多媒体扩展（MMX）指令集的光栅处理器，但最终失败了。当 MMX 光栅处理器运行在尚未发布的奔腾 2 处理器上时，我们发现其速度仅是正在出售的不起眼的 S3 Virge GX 光栅处理器的一半。那时我就知道这种努力注定是要失败的。

就 Direct3D 6.0 而言，我们与 CPU 厂商一起将他们的代码集成到我们的几何流水线（geometry pipeline），使开发人员可以以透明的方式使用来自英特尔和 AMD 的新指令集，并受益于厂商优化的代码执行路径。游戏开发人员接受了新的几何流水线，但由于新的指令集被用来生成 GPU 硬件的几何流水线所需要的顶点数据，因此，它没能阻止将 CPU 计算能力输送给 GPU 的持续迁移。

在那个时期，GPU 上的晶体管数量超过了 CPU 上的数量。转折点在 1997 ~ 1998 年间，当时奔腾 2 和英伟达 RIVA TNT 的晶体管数量都约为 800 万个。随后，GeForce 256（1500 万个晶体管）、GeForce 2（2800 万个晶体管）和 GeForce 3（6300 万个晶体管）的晶体管数量均超过了当时的 CPU。此外，两种设备之间的差异越来越清楚：大部分的 CPU 芯片面积是为了支持缓存，而大部分的 GPU 芯片面积是为了逻辑。英特尔能够添加大量新的扩展指令集（MMX、SSE、SSE2 等），而增加的面积代价几乎可以忽略不计。GPU 是专为并行吞吐处理而设计的，它们的小规模缓存是为了带来更多的带宽聚合（bandwidth aggregation）而不是为了减少指令延迟。

虽然 ATI 和英伟达等公司一直在生产越来越快速和越来越强大的 GPU，CPU 厂商仍然在摩尔定律 (Moore's Law)[⊖] 的指引下追求更高的时钟速率。第一款奔腾处理器 (1993 年) 的时钟速率为 60MHz，而启用 MMX 的奔腾处理器 (1997 年) 的时钟速率为 200MHz。到 20 世纪末，时钟速率已超过 1000MHz。但此后不久，计算史上发生了一桩重大事件：摩尔定律碰了钉子。晶体管将继续变小，但时钟速率可能不会再继续增加。

这一事件并非意料之外。英特尔的 Pat Gelsinger 在 2001 年美国电气和电子工程师协会 (IEEE) 举办的固态电路会议上发表主题演讲，曾表示：如果继续目前的设计思路，芯片会在 2010 年跟核反应堆一样热，而到 2015 年将达到太阳表面的热度。在未来，性能将来自“同步多线程” (simultaneous multithreading, SMT) 技术，可能要通过把多个 CPU 核心集成到一个芯片上来达到目的。事实上，CPU 厂商已经这么做了。今天，很难找到一个只有单个 CPU 核心的桌面 PC。几十年来，摩尔定律带给软件开发人员搭便车 (free ride) 的机会：提高 CPU 时钟频率，几乎根本不用软件开发人员操心，就允许程序运行更快。但这种好事将成为过去。多核心 CPU 需要多线程应用程序。只有当应用程序可以并行化时，众多 CPU 核心才能带来预期的高性能。

GPU 的定位很好地利用了摩尔定律的新趋势。虽然那些未曾考虑并行化的 CPU 应用程序将需要大量的重构 (如果它们有并行化的可能)，然而在图形应用程序上已经采用了可以利用各独立像素之间固有的并行模式的思路。对于 GPU 而言，通过增加执行核心的数量来提高性能是一个很自然的手段。事实上，GPU 设计者往往倾向于添加更多的核心而不是更强大的核心。他们摒弃了以最大限度地提高时钟频率 (GPU 的发展路线从来不是、现在依然不是靠制造接近晶体管电路极限的时钟频率)、预测执行 (speculative execution)、分支预测 (branch prediction) 和存储转发 (store forwarding) 等为代表的 CPU 厂商认为理所当然的策略。为了防止更加强大的处理器受限于 I/O 速度 (I/O bound)，GPU 设计者结合内存控制器，并与内存厂商一起研制可以远远超过 CPU 可用带宽很多的 GPU 内存带宽。

但是，GPU 的强劲计算能力对于非图形开发者来说很难利用。一些充满探索精神的程序员借助如 Direct3D 和 OpenGL 的图形 API，通过迷惑图形硬件来执行非图形的计算任务。这一方法被冠以专门的术语：通用 GPU 编程 (general-purpose GPU programming, GPGPU)。但 GPU 的大部分计算潜力仍一直处于未发掘状态，直到 CUDA 的出现改变了这一切。Ian Buck 当时在斯坦福大学开展一项名为 Brook 的项目，意在简化 GPGPU 应用程序的开发过程。Buck 后来进入英伟达并领导开发了一套新的开发工具，这使得 GPU 上非图形应用程序的开发更容易。这个英伟达的专用工具套装就是 CUDA。CUDA 允许 C 语言程序员使用一些简单易用的语言扩展编写 GPU 并行代码。

自 2007 年推出以来，CUDA 一直深受好评。数以万计使用该技术的研究论文已被发表。

[⊖] 摩尔定律主要有三种不同说法：集成电路芯片上所集成的电路的数目，每隔 18 个月就翻一倍；微处理器的性能每隔 18 个月提高一倍，或价格下降一半；用一美元所能买到的计算机性能，每隔 18 个月翻一倍。——译者注

它已被用于多个商业软件包，跨度从 Adobe 的 CS5 到 Manifold 的地理信息系统（geographic information system, GIS）。对于适合的工作任务，运行在支持 CUDA 的 GPU 上，相比同年代 CPU，可以获得 5 ~ 400 倍不等的加速。这些加速的来源各不相同。有的加速是因为 GPU 有更多的核心；有的是因为有更高的内存带宽；还有一些是因为应用程序可以利用在 CPU 中不曾有的专门 GPU 硬件，例如可以更快计算超越函数的纹理硬件或特殊函数单元（SFU）。但并非所有的应用程序都可以采用 CUDA 实现。更确切地说，并非所有的并行应用程序都可以采用 CUDA 实现。但 CUDA 已使用在各种应用中，比任何其他的 GPU 计算技术的应用更广泛。我希望本书可以帮助有一定 CUDA 基础的开发者更有效地使用 CUDA。

1.1 方法

写一本以 CUDA 为主题的书籍是件困难的事情。并行程序设计需要涉及的因素很多，即使暂不考虑操作系统因素（Windows、Linux 和 MacOS）、平台因素（特斯拉与费米，集成显卡与独立显卡，多 GPU）、CPU/GPU 并发因素以及 CUDA 的特定因素（例如，决定是使用 CUDA 运行时还是驱动程序 API 编写代码），也是相当复杂的。当你还需要考虑如何最好地组织 CUDA 内核程序时，其复杂程度可能让人窒息。

本书为了应对这一复杂性，在介绍多数主题时从不同角度不止一次地进行解释。“纹理映射硬件做哪些事情？”这一问题跟“怎样写一个进行纹理映射操作的内核程序？”是不同的。本书分别在不同章节回答这两个问题。异步内存复制操作可以在几处不同的主题下来解释：软件的抽象（例如，参与的主机内存必须分配为锁页内存）、不同的硬件实现、支持该功能的 API 以及优化策略之间的相互作用。读者有时不妨查阅索引，然后选取相应主题下的多处文本来阅读。

性能优化指南通常组织得像咨询栏目。很多时候，它们给出的指导缺乏足够的实际应用场景，而且往往前后矛盾。上述观点不是为了斥责它们，它们只是问题复杂性的一个症状而已。优化 CPU 的金律，是经历了至少 20 年的持续探索，而 GPU 是更难进行编写程序的，所以期望 CUDA 的优化建议很简单就有点不切实际了。

此外，GPU 计算是新生事物，即使是 GPU 架构师仍在学习最佳的编程方式，更不用说开发人员了。对 CUDA 开发人员而言，终极决定因素是性能，而性能通常采用系统时间来衡量！网格和线程块大小、何时以及如何使用共享内存、每个线程应该计算多少次运算以及占用率的数值，都可能影响性能。只有通过实现多种方法并测量每个方法的实际表现，才能给出切实的推荐。

1.2 代码

开发人员希望 CUDA 代码能说明问题而不仅仅是玩具，有实用性而无须具备冷门主题

上的技术储备以及有高性能且不能掩盖实现从初始移植到做出最终版本之间的尝试过程。为此，本书介绍了3种类型的代码示例，旨在分别迎合上述需求：验证型（microbenchmark）、演示型（microdemo）和探究型（optimization journey）。

1.2.1 验证型代码

验证型代码旨在说明一个非常具体的CUDA问题对性能的影响，例如，非合并的内存事务如何降低设备的内存带宽或执行内核转换（kernel thunk）会需要WDDM驱动程序多长时间才能完成。它们被设计成可以独立编译的程序并跟许多CUDA程序员自己实现的验证代码很相似。从某种意义上说，我编写了这一组验证型代码，是为了节省大家的时间。

1.2.2 演示型代码

演示型代码是小型应用程序，旨在揭示具体硬件或软件的行为问题。跟验证型代码一样，它们都很小、自成一体，但以强调功能性为主而不是强调性能。例如，纹理操作一章包括的演示型代码分别用以说明如何从1D设备内存得到纹理、如何进行浮点型到整型的转换、多种纹理寻址模式是如何工作的以及纹理的线性插值是如何被9位的权重所影响的。

跟验证型代码一样，这些演示型代码是考虑到开发者可能会编写它们或者需要它们。本书提供之后，大家就不必再去编写了。

1.2.3 探究型代码

基于CUDA的许多论文仅呈现所用方法的实验结果。也许会附带解释一句：文中方法是在调查多种方法并进行权衡后才采用的。作者往往受到篇幅和截止日期的限制，无法给出他们工作的更多细节。

基于CUDA进行数据并行编程比较核心的主题，本书给出探究型代码。逐步加大所考察问题的复杂性，不断优化程序性能。这是Mark Harris在《基于CUDA优化并行归约》^①一文中的处理思路。本书精选的几个案例均采用探究式分析，包括归约、并行前缀和（即“扫描”）和N-体问题。

1.3 资源

1.3.1 开源代码

本书的源代码可以在www.cudahandbook.com网站上下载。它们是开源的，著作权遵照2句版BSD许可证^②。

① <http://bit.ly/Z2q37x>。

② www.opensource.org/licenses/bsd-license.php。

1.3.2 CUDA 专家手册库 (chLib)

CUDA 专家手册库，可以在源代码的 chLib/ 目录下找到。它包含一个可以移植的库文件，支持计时、线程化操作、驱动程序 API 辅助工具等内容。在附录 A 中有针对它的更详细描述。

1.3.3 编码风格

不考虑括号里的参数，本书代码的主要特征，是基于 goto 方式 的错误处理机制。这可能会招致大家的批评。执行多个资源分配（或者其他可能出现失败的操作，并且失败会传递给调用者）的函数采用类似 Linux 内核代码的常用方式，按照初始化 / 错误检查 / 资源清理三步法模式来组织。

如果出现失败，所有的资源清理工作在函数末尾执行同一代码来完成。很重要的一点，记得在函数的开始部分初始化资源为特定的无效值，这样清理代码知道哪些资源必须被释放。如果资源分配或其他功能失败，代码执行一次 goto（清理代码）。A.6 节的 chError.h 文件中描述了使用 CUDA 运行时和驱动程序 API 实现此代码模式的错误处理宏。

1.3.4 CUDA SDK

SDK 为所有 CUDA 开发者提供一个共享的经验，所以我们假设你已经安装 CUDA SDK，并且可以使用它来生成 CUDA 程序。该 SDK 还包括 GLUT (GL 实用库)，便于从同一批程序代码中得到面向不同操作系统的 OpenGL 应用程序。GLUT 旨在生成演示级别的而不是产品级别的应用程序，但它符合我们的需求。

1.4 结构

本书按照逻辑结构分成三部分。第一部分包括第 1 ~ 4 章，概述 CUDA 硬件和软件的架构。

第 2 章详细介绍 CUDA 的硬件平台和 GPU 本身。

第 3 章介绍 CUDA 的软件架构。

第 4 章介绍 CUDA 软件环境，包括 CUDA 软件工具描述和亚马逊的 ECI 环境。

第二部分包括第 5 ~ 10 章，深入介绍 CUDA 编程模型的方方面面。

第 5 章涵盖内存，包括设备内存、常量内存、共享内存和纹理内存。

第 6 章介绍流和事件，这一机制用于 CPU 和 GPU 之间、GPU 不同硬件单元（例如，复制引擎和流处理器簇）之间或者多个独立 GPU 之间的“粗 - 细粒度”结合的并行处理。

第 7 章涵盖内核执行，包括动态并行这一 CUDA 5.0 和 SM 3.5 中的新特性。

第 8 章涵盖流处理器簇的各个方面。

第 9 章涵盖多 GPU 应用程序，包括以 N- 体为例的点对点操作和高难度的并行操作。

第 10 章涵盖 CUDA 纹理的各个方面。

最后，在 11 ~ 15 章的第三部分讨论几个有针对性的 CUDA 应用程序。

第 11 章介绍带宽限制、以向量乘法为例的流式负载等内容。

第 12 和 13 章描述归约和并行前缀求和（也称为“扫描”）算法，两者都是并行编程的重要基石。

第 14 章介绍 N- 体应用，这是一族可以大大受益于 GPU 计算的重要的高计算密集型应用。

第 15 章深入考查称为归一化相关系数特征提取的图像处理操作。第 15 章提供的代码是本书中唯一同时使用纹理和共享内存以提供最佳性能的地方。

硬件架构

本章分别从 GPU 的系统层面到功能单元对 CUDA 平台进行更为具体的描述。前三节讨论多种构建 CUDA 系统的不同方法；第四节主要讨论了地址空间以及 CUDA 的内存模型是如何在软硬件上实现的；第五节主要讨论 CPU/GPU 交互，并且特别关注指令如何提交到 GPU 以及 CPU/GPU 是如何进行同步的；最后，针对复制引擎和流处理器簇等功能单元进行了宏观描述，并特意给出了支持 CUDA 的三代硬件上流处理器簇的框图。

2.1 CPU 配置

本节及随后的两节介绍各种 CPU/GPU 架构，并对 CUDA 开发者如何编程方面给出了一些意见和建议。我们研究了 CPU 配置、集成 GPU 和多 GPU 配置。现在，让我们从图 2-1 开始谈起。

图 2-1 中省略了一个重要元素，那就是连接 CPU 与外界的“芯片组”（“核心逻辑”）。系统每一比特的输入与输出，包括磁盘、网络控制器、键盘、鼠标、USB 设备以及 GPU 的输入输出，都要通过芯片组。直到最近，芯片组被一分为二：一个是连接大多外围设备和系统的“南桥”[⊖]，另一个是包含图形总线（加速图形

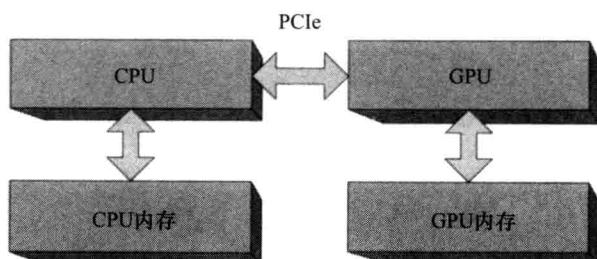


图 2-1 CPU/GPU 架构简略图

[⊖] 为简单起见，本节的所有图中均略去南桥。

端口，后被 PCIe 接口取代) 和内存控制器(通过前端总线与内存相连)的“北桥”。

每一个 PCIe (peripheral communications interconnect express) 2.0 的“通道”(lane)理论上可以提供 500MB/s 的带宽。对于一个给定的外设，其通道数可以是 1、4、8 或 16 个。GPU 需要平台上所有外设的最大带宽，所以它们一般被插入由 16 通道构成的 PCIe 插槽。考虑到数据包的附加消耗，该连接的 8G/s 带宽实际上能达到 6G/s。^①

2.1.1 前端总线

图 2-2 是将北桥和内存控制器添加到图 2-1 后的简化图。为了完整起见，图 2-2 还显示了 GPU 的集成内存控制器，它是在一个与 CPU 内存控制器完全不同的约束集下设计的。GPU 必须调解所谓的同步客户端，例如视频显示，其带宽要求是固定的。GPU 内存控制器在设计时还考虑了 GPU 对延迟的容忍度以及大量内存带宽方面的内在需求。在写作本书时，高端 GPU 通常所提供的本地显存带宽大大超过 100G/s。GPU 内存控制器总是和 GPU 集成，因此本章其余部分的图示中省略了它们。

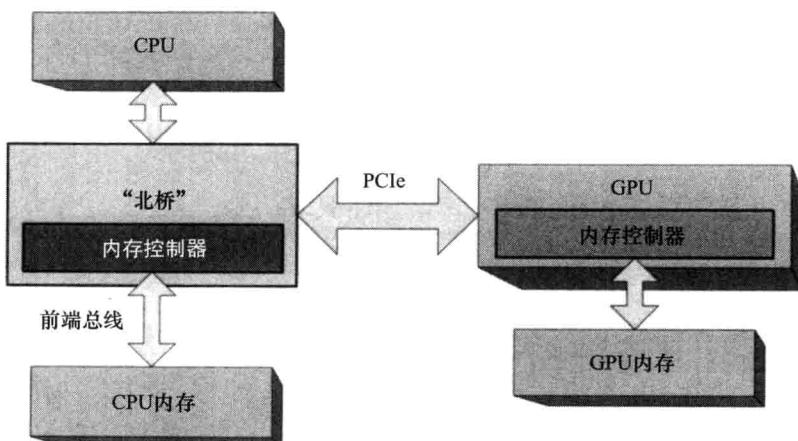


图 2-2 CPU/GPU 架构 (带北桥)

2.1.2 对称处理器簇

图 2-3 显示的是一个传统北桥配置上的多 CPU 系统。^②多核处理器之前，应用程序必须使用多线程以便充分利用多 CPU 的额外运算能力。即使每个 CPU 和北桥本身都包含有缓存，北桥也必须确保每个 CPU 看到相同的、一致的内存视图。由于这些所谓的“对称处理器簇”(SMP) 系统共享同一个通往 CPU 内存的路径，不同 CPU 的内存访问性能

^① PCI 3.0 所能提供的带宽相当于 PCIe 2.0 的两倍。

^② 我们之所以提供图 2-3，不只是因为该配置是支持 CUDA 的计算机，更是作为历史资料参照。

相对一致。

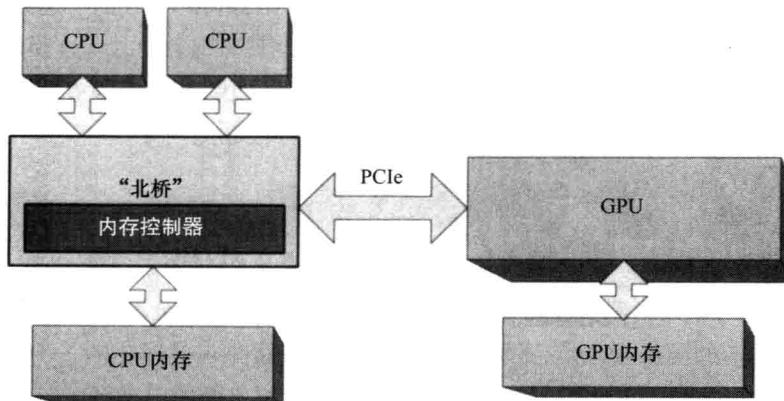


图 2-3 多 CPU (SMP 配置下)

2.1.3 非一致内存访问 (NUMA)

从 AMD 的 Opteron 处理器和 Intel 的 Nehalem (酷睿 i7) 处理器谈起，北桥的内存控制器直接集成到 CPU 中，如图 2-4 所示。这种结构的变化，提高了 CPU 的内存性能。

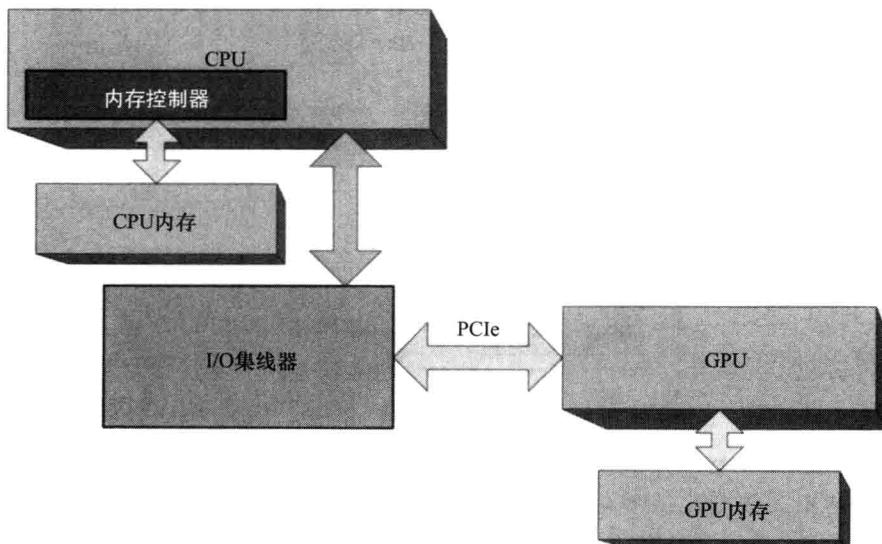


图 2-4 带有集成内存控制器的 CPU

对于开发人员，图 2-4 中的系统和我们讨论过的只是略有不同。而对于包含多个 CPU 的系统而言，如图 2-5 所示，那就变得很不一样了。

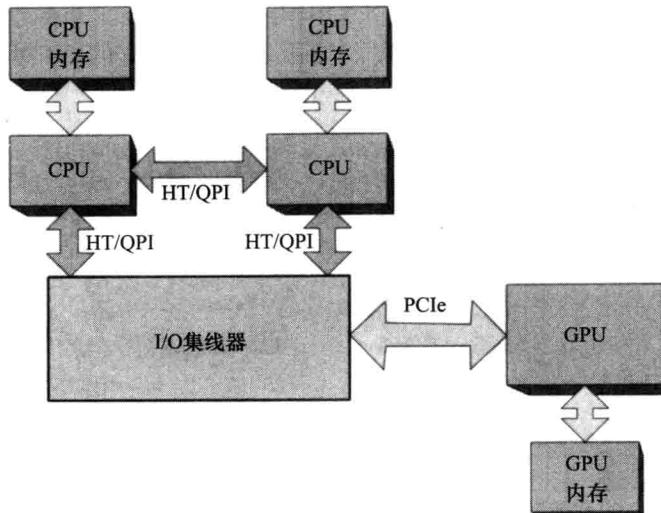


图 2-5 多 GPU (NUMA)

对于配置有多个 CPU 的机器[⊖]，这种架构意味着，每个 CPU 都有自己的内存带宽池。与此同时，由于多线程操作系统和应用程序依赖于前文中 CPU 和北桥配置下缓存的一致性，Opteron 和 Nehalem 架构分别推出了超传输 (HT) 和快速通道互联 (Quick Path Interconnect, QPI)。

HT 和 QPI 是 CPU 之间、CPU 与 I/O 集线器的点对点连接。在采用 HT/QPI 的系统中，每个 CPU 都可以访问任何内存存储单元。但是对内存的物理地址直接附属于 CPU 的本地内存单元的访问会更快。“非本地访问”的执行有赖于 HT/QPI 检查其他 CPU 的缓存，清除所请求数据的缓存副本，然后传递数据到发出请求的 CPU。通常，这些 CPU 的大型片上缓存降低了非本地内存访问的成本。而且，发出请求的 CPU 可以在自身缓存层次上保留该数据，直到另一个 CPU 向内存提出申请为止。

为了帮助开发者解决这些性能陷阱，Windows 和 Linux 系统引入 API。这样应用程序能够使用特定 CPU 以及设置“线程亲和的”(thread affinity) CPU。因此，当操作系统把线程调度到 CPU 时，可以保证大部分甚至是全部的内存都是本地访问的。

一个爱钻研的程序员可以使用这些 API 来精心设计代码以暴露 NUMA 的性能缺陷。但更常见（和隐性）的问题是，由于运行在不同 CPU 上的两个线程的“伪共享”，导致过多的 HT/QPI 事务在同一缓存行上访问内存存储单元。因此，NUMA API 必须慎用。它们虽然给程序员提供了提高性能的工具，但也容易因开发者误用而带来性能问题。

[⊖] 在这样的系统中，CPU 也可以被称为“节点”或“CPU 插槽”。

减轻访问非本地内存对性能影响的方法之一，是以缓存行边界为准，在 CPU 间均匀划分物理内存，也就是所谓的内存重叠[⊖]。对于 CUDA 而言，这种方法在如图 2-5 所示的系统中非常有效（其为一个 NUMA 模式配置，多 CPU 和 GPU 通过一个共享的 I/O 集线器连接）。由于 PCIe 带宽往往是整体应用性能的瓶颈，许多系统将使用独立的 I/O 集线器去服务更多的 PCIe 总线，如图 2-6 所示。

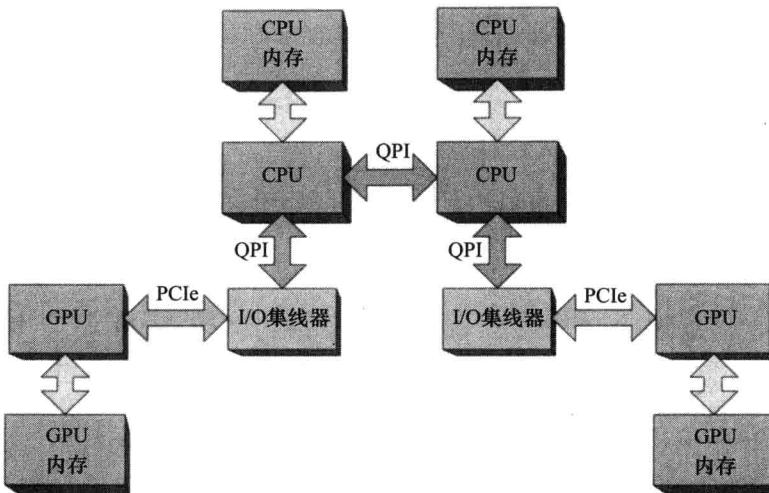


图 2-6 带多个总线的多 CPU (NUMA 配置)

为了很好地运行在这样一个“亲和”系统上，CUDA 应用程序必须注意 NUMA API 的使用，以便为连接到给定 GPU 的 PCIe 总线执行本地的内存分配和线程亲和的调度。否则，由 GPU 发起的内存复制将是非本地的，内存事务将在 HT/QPI 互连结构中需要额外的“跳跃”。由于 GPU 需要巨大的带宽，这些 DMA 操作会降低 HT/QPI 为其主要对象服务的能力。和伪共享相比，对于 CUDA 应用程序而言，GPU 的非本地内存复制操作对性能的影响有可能更为要命。

2.1.4 集成的 PCIe

如图 2-7 所示，通过将 I/O 集线器集成到 CPU 中，英特尔的沙桥 (Sandy Bridge) 系列处理器是迈向全面的系统集成的又一步。单个沙桥 CPU 的 PCIe 接口有多达 40 个通道（请记住，一个 GPU 最多可使用 16 个通道，所以 40 个通道足够支持两个完整的 GPU）。

对于 CUDA 开发者，集成的 PCIe 有利有弊。弊端是，PCIe 通路始终是亲和的。设计师不能建立单一 I/O 集线器服务于多个 CPU 的系统（即图 2-5 所示的系统）；所有的多 CPU 系统都类似于图 2-6。其导致的结果是，不同 CPU 上的 GPU 之间无法执行点对点操作。而优

[⊖] 也有相反的说法，说这使得所有的内存访问“同样糟糕”。

点是，CPU 缓存可以直接参与 PCIe 总线通信：DMA 读请求可以直接读取缓存，并且 GPU 写入的数据会放入缓存中。

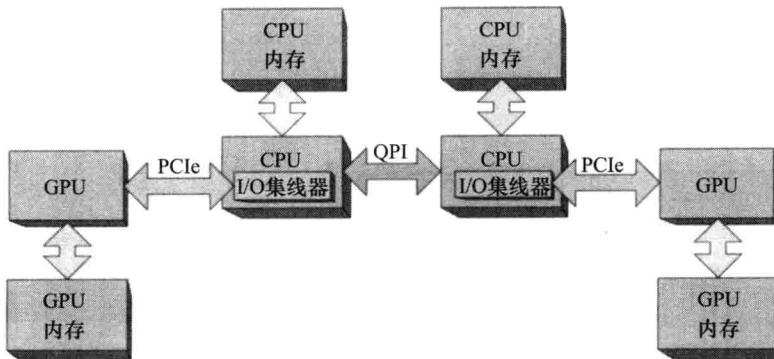


图 2-7 带集成 PCIe 的多 CPU

2.2 集成 GPU

在这里，术语“集成”的意思是“集成到芯片组”。如图 2-8 所示，先前只属于 CPU 的内存池现在可以被集成到芯片组的 CPU 和 GPU 所共享。例如，英伟达芯片组中支持 CUDA 的 GPU，包括 MCP79（适用于笔记本电脑和上网本）和 MCP89 等。MCP89 将是制造的最新、最好的可支持 CUDA 的 x86 芯片组，除了集成了三级缓存，它拥有 3 倍数量于 MCP7x 芯片组的流处理器簇。

CUDA 中用于映射锁页内存（mapped pinned memory）的 API，在集成 GPU 上具有特殊的意义。这些 API 把分配的主机内存映射到 CUDA 内核的地址空间，使它们能够直接被访问。这也被称为“零复制”，因为内存是共享的，复制操作不需要通过总线。事实上，在传输受限型工作量上，一个集成的 GPU 可以超过一个更大的独立 GPU。

“写结合”（write-combined, WC）内存的分配在集成 GPU 上也具有重要意义。CPU 的缓存侦测行为在访问这种内存时是被禁止的，这样可以提高访问内存过程中 GPU 的性能。当然，如果 CPU 从“写结合”内存中读取数据，通常的 WC 内存应用的性能会有损失。

集成 GPU 与独立 GPU 不是相互排斥的。MCP7x 和 MCP89 芯片组提供了 PCIe 连接接

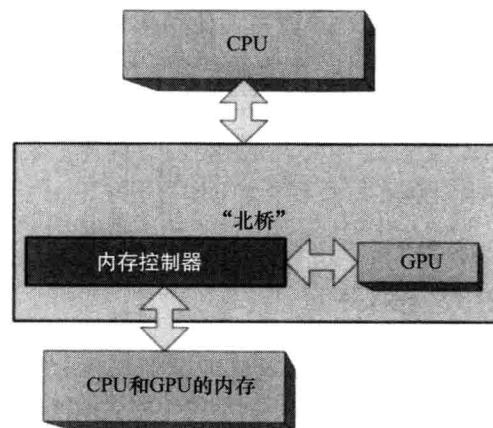


图 2-8 集成 GPU

口(图 2-9)。在这样的系统中, CUDA 更倾向于在独立 GPU 上运行, 因为大多数的 CUDA 应用程序将在独立 GPU 上部署。例如, 在单个 GPU 上运行时, 一个 CUDA 应用程序会自动选择在独立 GPU 上运行。

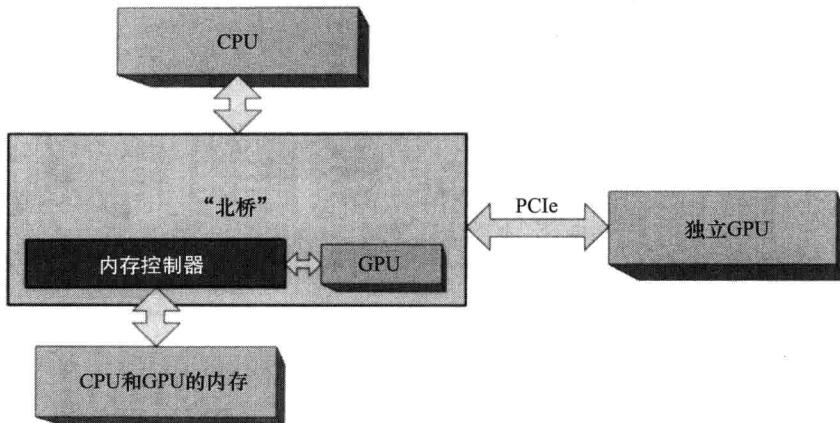


图 2-9 带独立 GPU 的集成 GPU

CUDA 应用程序可以通过检查 `cudaDeviceProp.integrated` 的值或者通过传递 `CU_DEVICE_ATTRIBUTE_INTEGRATED` 到 `cuDeviceGetAttribute()` 函数来查询一个 GPU 是否是集成 GPU。

对于 CUDA 而言, 使用集成 GPU 并不稀奇, 上百万的计算机在主板上集成了支持 CUDA 的 GPU。但它们会引起大家的好奇心。并且在短短几年内, 它们将会变得落伍, 因为英伟达已经退出了 x86 芯片组市场。据传言, 英伟达已经宣布他们的出货重点放在集成了支持 CUDA 的 GPU 与 ARM CPU 的片上系统 (*systems on a chip, SOC*) 上。可以判定, 零复制优化将在那些系统上工作得很好。

2.3 多 GPU

本节将探讨把多 GPU 安装在一个系统中的不同方法及其对 CUDA 开发者的影响。为了讨论方便, 我们会从下面的图中略去 GPU 内存。图中, 每个 GPU 默认连接到相应专用内存。

大约在 2004 年, 英伟达推出了“速力”(Scalable Link Interface, SLI) 技术, 可以让多 GPU 并行工作, 以提供更高的图形性能。使用可以容纳多个 GPU 的主板, 用户可以通过在他们的系统上安装两个 GPU 来提升近一倍的图形性能(图 2-10)。默认情况下, 英伟达驱动程序配置了这些 GPU 卡, 使它们表现得好像是单个速度更快的卡, 用以加速如 Direct3D 和 OpenGL GPU 的图形 API。打算使用 CUDA 的最终用户, 必须明确在 Windows 的显示控制面板上启用它。

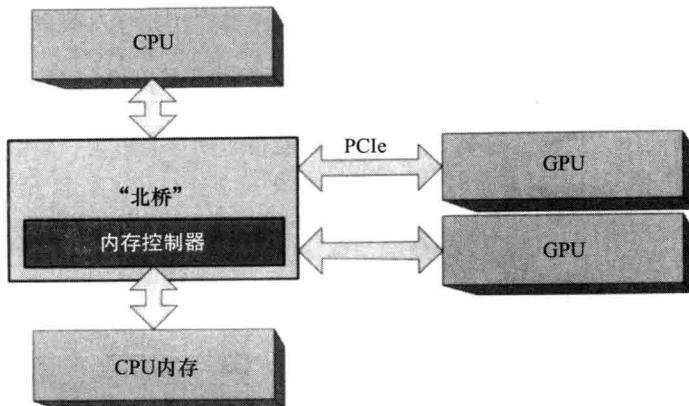


图 2-10 多插槽 GPU

还可以构造有多个 GPU 的 GPU 板（图 2-11）。这种板的例子有 GeForce 9800GX2（双 G92）、GeForce GTX295（双 GT200）、GeForce GTX590（双 GF110）和 GeForce GTX690（双 GK104）。这类 GPU 板上的 GPU 之间仅有桥接芯片是共享的，该芯片使一对 GPU 芯片可以通过 PCIe 交换信息。但它们并不共享内存资源，每个 GPU 都拥有一个集成内存控制器，使连接到 GPU 的内存具有全带宽性能。板上的 GPU 可以通过点对点的内存复制来交换信息，只需使用桥接芯片成功绕过主要的 PCIe 通道。此外，如果它们是费米架构或更高级别的 GPU，那么每个 GPU 都可以映射属于其他 GPU 的内存到其全局地址空间里。

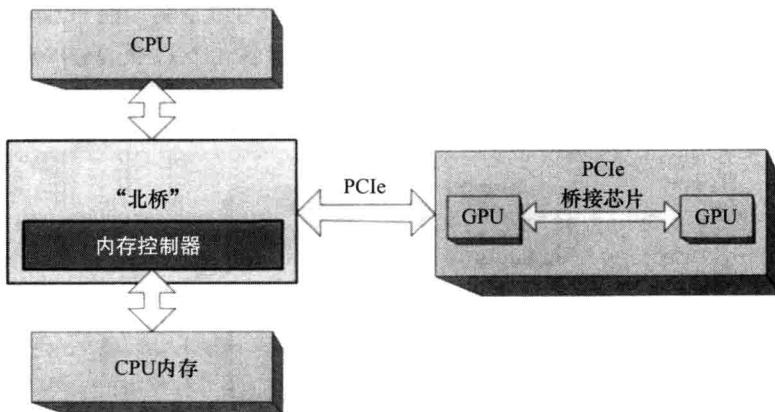


图 2-11 多 GPU 板

SLI 可以使用户能够像使用单个 GPU 一样使用多个 GPU，只不过性能更快（通常多个 GPU 在同一块电路板上，如图 2-11 所示）。当图形应用程序下载纹理或者其他数据时，英伟达的显卡驱动程序会将数据和大部分渲染命令以广播的方式传送到每个 GPU。而渲染命令可能会有一些小的变化，以使每个 GPU 控制的那部分输出缓冲区得以渲染。由于 SLI 导致多

GPU 表现为单个 GPU，并且 CUDA 应用程序不能像图形应用程序那样透明地得到加速，所以 CUDA 开发人员一般不使用 SLI。

这种双 GPU 板设计超支了供 GPU 使用的 PCIe 带宽。由于只一个 PCIe 插槽对应的带宽供板上的一对 GPU 使用，传输受限型工作任务的性能将遭受负面影响。如果有多个 PCIe 插槽可供选择，最终用户就可以安装多个双 GPU 板。如图 2-12 就显示了装载了 4 个 GPU 的机器。

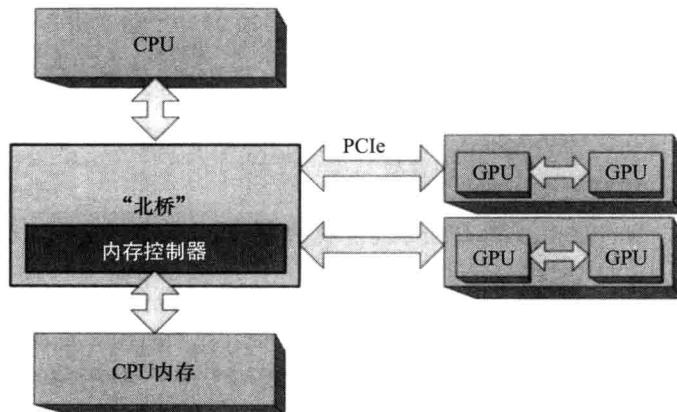


图 2-12 多插槽式多 GPU 板

如果有多个如图 2-6 中系统的 PCIe I/O 集线器，NUMA 系统考虑的布局和线程亲和将适用于插入该配置的单 GPU 板。

如果芯片组、主板、操作系统和驱动程序支持，甚至可以把更多的 GPU 装载进系统中。安特卫普大学（University of Antwerp）的研究人员，通过将 4 个 GeForce 9800GX2 整合到一个单一桌面电脑里，构建了一个叫做 FASTRA 的 8-GPU 系统，引起一时轰动。一个基于双 PCIe 芯片组构建的类似系统看起来如图 2-13 所示。

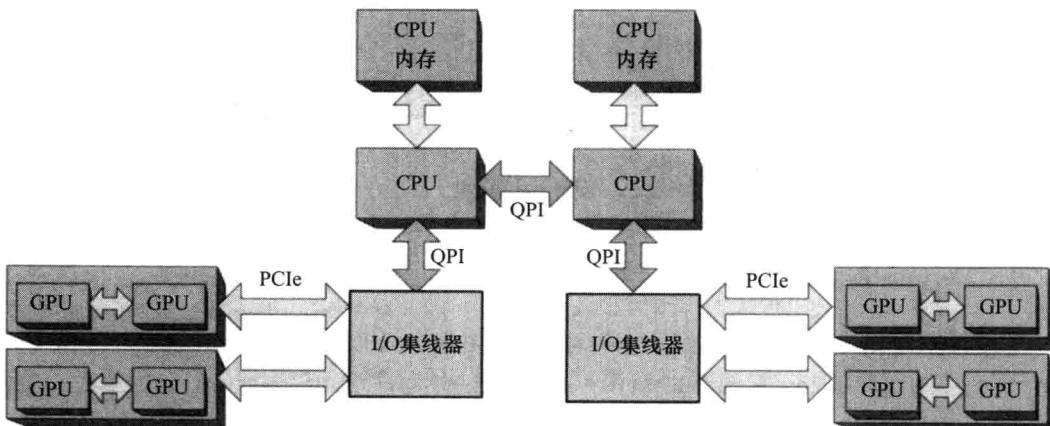


图 2-13 多 GPU 板与多 I/O 集线器连接图

作为边注，点对点内存访问（其他 GPU 设备内存的映射，而不是内存复制）不能跨越 I/O 集线器工作，也不适用于集成了 PCIe 插槽的沙桥 CPU。

2.4 CUDA 中的地址空间

每个 CUDA 初学者都知道，CPU 和 GPU 的地址空间是分开的。CPU 不能读取或写入 GPU 的设备内存，反过来，GPU 也无法读取或写入 CPU 的内存。其结果是，应用程序必须明确地从 GPU 的内存中传入 / 传出数据，以对其进行相应处理。

实际情况比较复杂，CUDA 已经增加了许多新的功能，如映射锁页内存和点对点访问。本节将以详细说明地址空间在 CUDA 中如何工作为第一要义。

2.4.1 虚拟寻址简史

虚拟地址空间是如此普遍和成功的抽象物，大多数程序员每天都使用它们，并从中受益，却不知道它们的存在。早期的人们发现，给计算机中的内存位置分配连续编号是非常有用的。虚拟地址空间是这一发现的延伸。其标准度量单位是字节，例如，64KB 内存中的计算机的内存位置由 0 到 65535。指定内存位置的 16 位值称为地址，地址的计算和相应内存位置上的操作过程则统称为寻址 (addressing)。

早期的计算机进行物理寻址。它们会计算内存位置，然后读取或写入相应存储单元，如图 2-14 所示。随着软件越来越复杂，以及计算机能承载多个用户或越来越普遍地运行多个作业，每个程序却可以读取或写入任何物理内存位置。很明显，这是不会被接受的！否则，机器上运行的软件很有可能会因为写入错误的内存位置而对其他软件造成致命破坏。除了健壮性问题，其还有安全方面的瑕疵：软件可通过读取属于其他人的内存位置来窥探其他软件。

因此，现代的计算机推行虚拟地址空间。每个程序（操作系统设计者称其为一个进程）都会得到类似于图 2-14 中的内存视图，但每个进程都有自己的地址空间。未经操作系统特别许可，它们不能读取或写入属于其他进程的内存。机器指令使用虚拟地址而不是物理地址，随后由操作系统执行一系列查找表的操作，将其翻译成一个物理地址。

在大多数系统中，虚拟地址空间被划分成许多页，它们是寻址的单位，其大小至少 4096

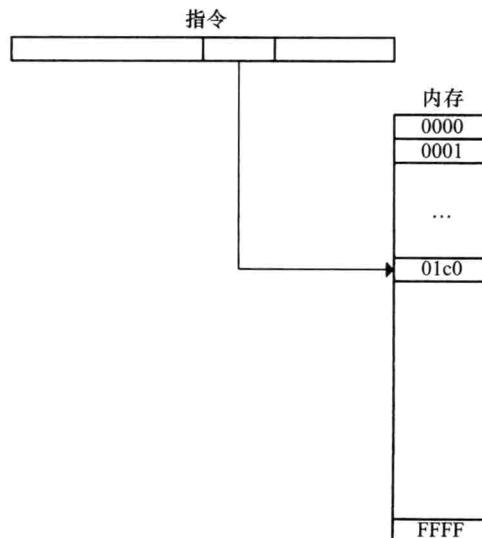


图 2-14 16 位地址空间简图

字节。通常，硬件查找指定页内存的页表项（PTE）来得到所在物理地址，而不直接引用物理内存的地址。

我们可以很清楚地从图 2-15 看出，虚拟寻址能使一个连续的虚拟地址空间映射到物理内存并不连续的一些页。此外，当应用程序试图读取或写入页面未被映射到物理内存的内存位置时，硬件会发送必须由操作系统处理的错误信号。

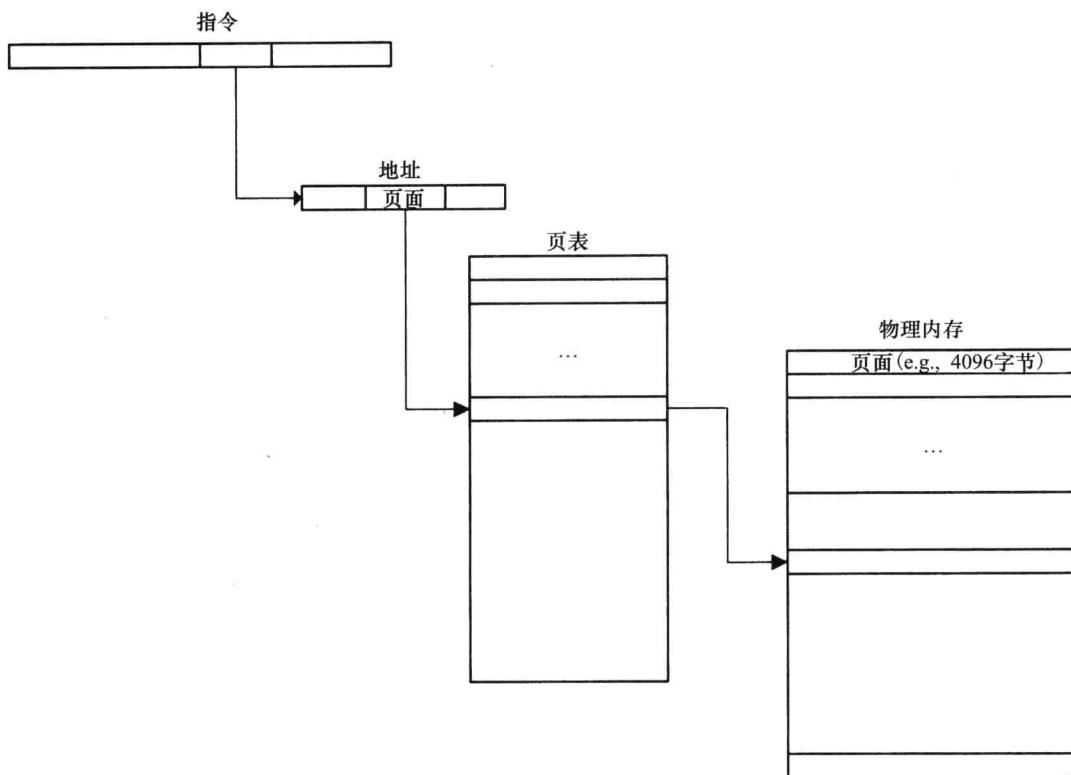


图 2-15 虚拟地址空间

另外做个边注：实际上，所有硬件都不会实现如图 2-15 所示的单级页表。最起码，地址会被分成至少两个索引：第一个索引指向页表的“页目录”，第二个索引代表第一个索引约束下选择的页表。这种分层设计减少了页表所需的内存量，把不活跃的页表标记为非常驻状态并交换到磁盘上，就像无效页面一样。

除了物理内存位置，PTE 还包含了在做地址转换时硬件可以验证的权限位。例如，操作系统可以使内存页只读。此时，如果应用程序试图写入该页面，硬件将会产生一个错误信号。

操作系统使用虚拟内存硬件来实现很多功能：

- 缓式分配 (lazy allocation)：通过设置 PTE 允许分配无物理内存支持的页，从而可以分配大容量的内存。如果请求内存的应用程序碰巧访问这些页面，操作系统会立刻找

到一个有物理内存的页面并解决这个故障。

- 请求式调页 (demand paging): 内存可以被复制到磁盘中并且页面被标记为“非常驻”。如果这样的内存再次被引用，硬件会产生“页面故障”信号，而且操作系统会将数据复制到一个物理页中并修正 PTE 指向该页，然后继续执行。如此，故障得以解决。
- 写时复制 (copy-on-write): 通过创建第二组映射到相同的物理页面的 PTE，并将两组 PTE 标记为只读，虚拟内存得以“复制”。如果硬件捕获到一个试图写入这些页面的操作，操作系统将会复制该组页面到另一组物理页面，并再次标志这两组 PTE 为可写，然后恢复执行。如果应用程序只写入一个很小比例的“复制页面”，写时复制也就在性能上具有明显优势。
- 映射文件 I/O (mapped file I/O): 文件可以被映射到地址空间，并且通过访问文件可以解决页面故障。对进行随机访问相关文件的应用程序，通过委托操作系统中高度优化的 VMM 代码进行内存管理是非常有用的，特别是因为它是紧密耦合的大容量存储驱动程序。

CPU 所执行的每一次内存访问都会有地址转换，明白这点是很重要的。为了使这个操作快捷，CPU 包含了一些特殊的硬件：被称为转换旁置缓冲器 (translation lookaside buffer, TLB) 的缓存以及“页面查询器”(page walker)。前者的作用是保留最近转换的地址区间；后者的作用是通过读取页表来处理在 TLB 中未命中的缓存。^① 现代的 CPU 还包括了支持“统一的地址空间”的硬件，可以让多 CPU 通过 AMD 的 HT (超传输) 和英特尔的快速通道互连 (QPI)，有效地访问另一个内存。由于这些硬件设施可以使 CPU 使用统一的地址空间来访问系统中的任何内存位置，所以本节前文在提到“CPU”和“CPU 地址空间”时未提及系统中 CPU 的数量。

附注：内核模式和用户模式

关于 CPU 的内存管理的最后一点是操作系统的代码必须使用内存保护，以防止应用程序破坏操作系统自身的数据结构（例如，控制地址转换的页表）。为了协助内存保护，操作系统拥有一个执行任务的“特权”模式，可以在实现关键的系统功能时使用。为了管理页表等低层次硬件资源，抑或是对磁盘、网络控制器、CUDA GPU 等外设上的硬件寄存器进行编程，CPU 必须在内核模式下运行。应用程序代码所使用的非特权执行模式称为用户模式。^② 除了操作系统提供商编写的代码，控制硬件外设的低级别驱动代码也在内核模式下运行。由于内核模式代码的错误可能会导致系统稳定性或安全性问题，内核模式代码会秉承一个更高的质量标准。此外，许多操作系统服务，如映射文件 I/O 或上面列出的设施外的其他内存管理，都不能在内核模式中使用。

为了保证系统的稳定性和安全性，用户模式和内核模式之间的接口是经过精心设计的。

^① 编写程序（对于 CPU 和 CUDA）来显示 TLB 的规模和结构以及页面查询器的内存开销，这也是有可能的（如果页面查询器在足够短的时间内通过足够的内存）。

^② 内核模式和用户模式分别对应 x86 下的“Ring 0”和“Ring 3”术语。

用户模式代码必须在内存中设立一个数据结构，并执行一个特殊的系统调用来验证内存和发出的请求的有效性。这种从用户模式到内核模式的过渡称为一个内核转换。内核转换代价不小，会增加 CUDA 开发者的成本。

每次用户模式驱动程序下实现的 CUDA 硬件交互都要由内核模式代码进行仲裁。这通常意味着它代其分配内存资源，以及硬件寄存器等硬件资源。例如，用户模式驱动程序下的硬件寄存器可以把任务提交给硬件。

大部分 CUDA 的驱动程序是在用户模式下运行。例如，为了分配系统锁页内存（例如，使用 `cudaHostAlloc()` 函数），CUDA 的应用程序会调用用户模式下的 CUDA 驱动程序，然后组成一个内核模式下的 CUDA 驱动程序请求，并执行内核转换。内核模式下的 CUDA 驱动程序，混合使用低级别的操作系统服务（例如，它可能调用系统服务来映射 GPU 硬件寄存器）和具有硬件特性的代码（例如，对 GPU 内存管理硬件进行编程）来满足上述请求。

2.4.2 不相交的地址空间

虽然 GPU 硬件不如 CPU 支持那么丰富的功能集，CUDA 也使用虚拟地址空间。GPU 确实执行内存保护，所以 CUDA 程序不可以随机地读取或破坏其他 CUDA 程序的内存，也不可以访问还没有被内核模式驱动程序映射的内存。但是，GPU 并不支持请求式调页，所以被 CUDA 分配的每一字节虚拟内存都必须对应一个字节的物理内存。此外，请求式调页功能是操作系统为实现前述特性而使用的基础硬件机制。

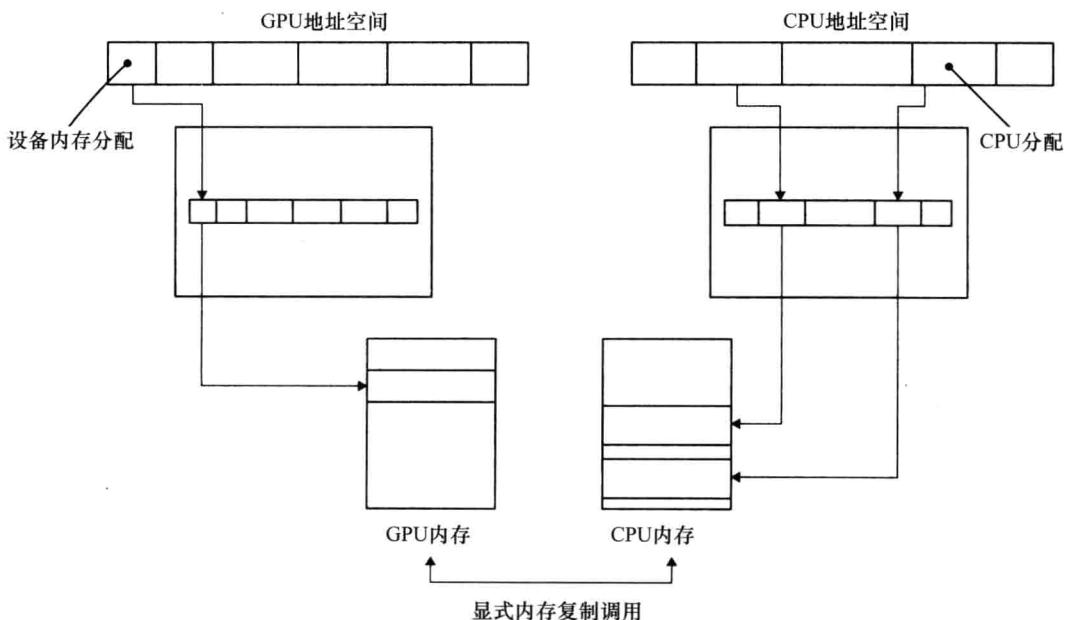


图 2-16 不相交的地址空间

由于每个 GPU 有它自己的内存和地址转换硬件，GPU 的地址空间和 CUDA 应用程序中的 CPU 地址空间是相互分开的。图 2-16 显示的是在使用映射锁页内存之前 CUDA 1.0 版的地址空间架构。CPU 和 GPU 都有自己的地址空间，用来映射各自设备自身的页表。两者的设备都要通过显式的内存复制命令来交换数据。GPU 可以分配锁页内存，该内存是 GPU 为 DMA 映射的页面锁定的内存。它只能使 DMA 的速度更快，却不能让 CUDA 内核程序访问主机内存。^②

CUDA 驱动程序跟踪锁页内存的起止范围，并自动加速引用它们的内存复制操作。异步内存复制调用需要锁页内存起止信息，以确保在内存复制完成之前操作系统不会取消映射或移动物理内存。

并非所有的 CUDA 应用程序都可以分配到让 CUDA 使用的主机内存。例如，一个大型应用程序中的 CUDA 插件可能会在非 CUDA 代码分配的主机内存上操作。为了适应这种情况，CUDA 4.0 增加了注册现有的主机内存地址起止范围（它可以锁定一个虚拟地址区间，并将其映射到 GPU）以及跟踪数据结构的地址范围的能力，以便于让 CUDA 明白它是锁页内存。然后，内存就可以被传递给异步内存复制调用，否则，它将被看作是由 CUDA 分配的。

2.4.3 映射锁页内存

如图 2-17 所示，CUDA 2.2 增加了一个叫做映射锁页内存的特性。映射锁页内存是被映射到 CUDA 地址空间的锁页主存，在 CUDA 内核程序里可以直接对其读取或写入。CPU 和 GPU 的页表更新了，以便 CPU 和 GPU 中拥有指向相同主机内存缓冲区的地址区间。由于地址空间不同，GPU 指向该缓冲区的指针必须使用 cuMemHostGetDevicePointer() 或 cudaHostGetDevicePointer() 函数来查询。^②

2.4.4 可分享锁页内存

如图 2-18 所示，CUDA 2.2 还增加了一个叫做可分享锁页内存的特性。设置锁页内存“可分享”，会导致 CUDA 驱动程序把该内存映射给系统中的所有 GPU，而不仅仅是当前上下文的 GPU。此特性为系统中的 CPU 和所有 GPU 创建一组独立的页表条目，使相应的设备将虚拟地址转换成实际物理内存。主机内存的起止范围也被添加到了每个活跃的 CUDA 上下文的跟踪机制中，因此每个 GPU 都能识别出作为可分享方式分配的锁页内存。

图 2-18 可能表示了开发者对多地址空间的容忍限度。在这里的每个分配，双 GPU 系统有 3 个地址，而 4GPU 系统则有 5 个地址。虽然 CUDA 通过 API 能够快速查找一个给定 CPU 的地址空间范围，并传回相应的 GPU 地址空间范围，但在 N 个 GPU 的系统中将拥有 $N+1$ 个地址且全部指向同一物理分配，这样是不方便的。

-
- ② 在 32 位操作系统上，为了执行内存复制，支持 CUDA 的 GPU 可以映射锁页内存到一个 40 位地址空间里，而该空间已经超过了 CUDA 内核程序的控制范围。
 - ② 对于多 GPU 配置，CUDA 2.2 还增加了一个叫做“可分享锁页内存”的特性，导致上述分配会被映射到每个 GPU 的地址空间。但这并不能保证 cu(da)HostGetDevicePointer() 函数在不同 GPU 上会返回相同的值。

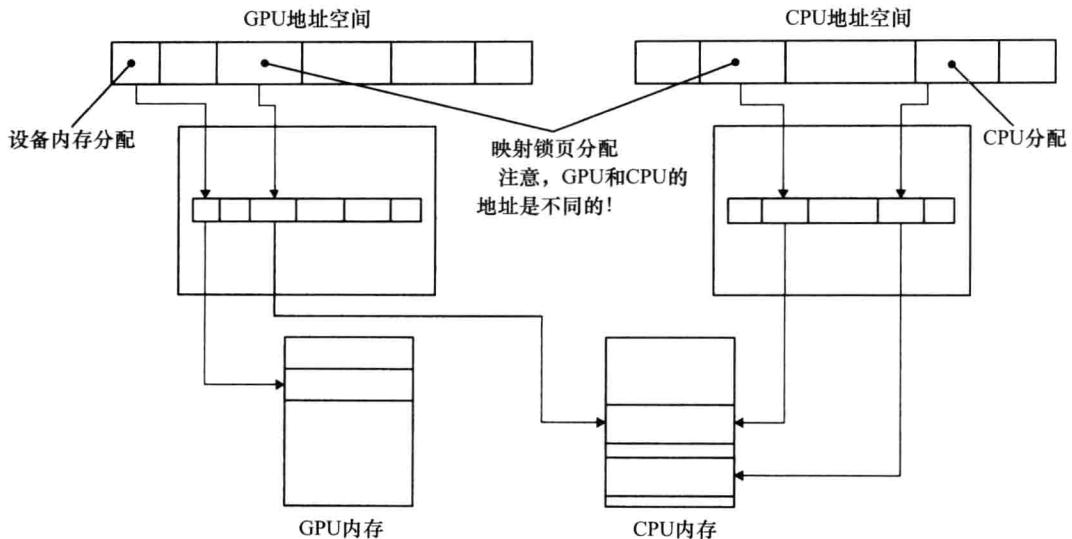


图 2-17 映射锁页内存

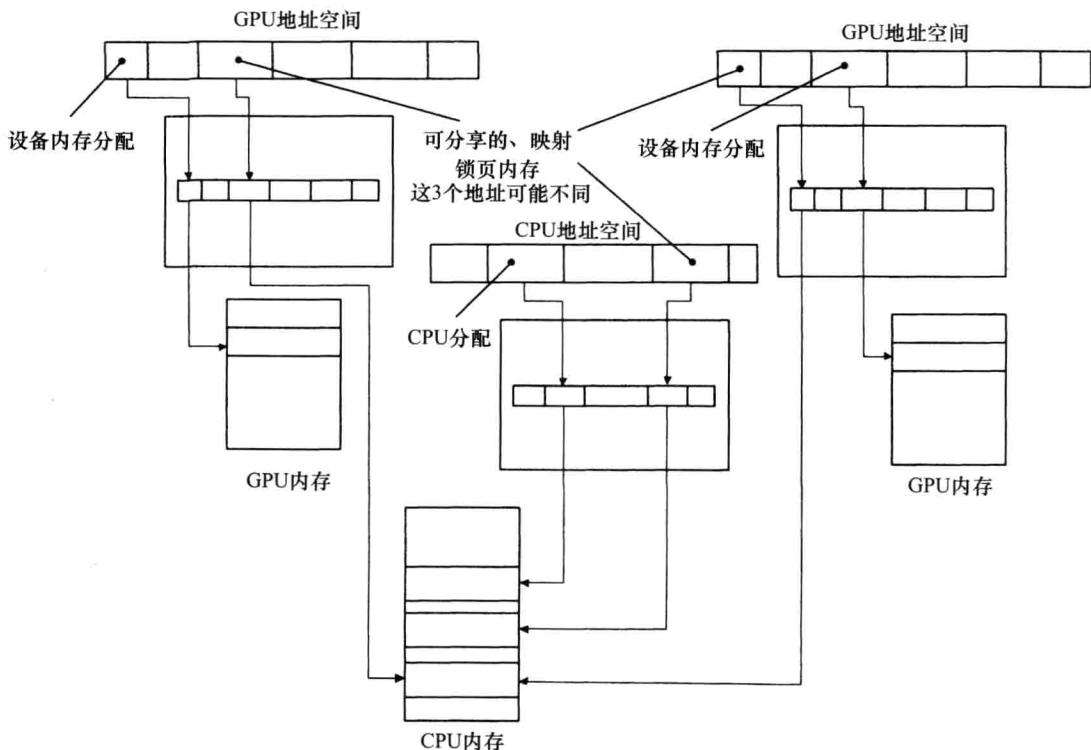


图 2-18 可分享、映射锁页内存

2.4.5 统一寻址

32位CUDA GPU需要多个地址空间，它最多只能映射 $2^{32}=4\text{GB}$ 的地址空间。由于一些高端的GPU有多达4GB的设备内存，所以它们很难在寻找所有设备内存地址的同时映射锁页内存，更不用说像CPU那样使用统一的地址空间了。但是，对于在费米架构或更高级GPU下的64位操作系统而言，如下的一个更简单抽象是可行的。

如图2-19所示，CUDA 4.0增加了一个叫做统一虚拟寻址(unified virtual addressing, UVA)的特性。当UVA生效时，CUDA会从相同的虚拟地址空间为CPU和GPU分配内存。CUDA驱动程序通过以下两步来完成上述任务：第一，初始化程序执行基于CPU地址空间的大型虚拟分配，该分配过程中可能会碰到无物理内存支持的情况；第二，将GPU分配的内存映射到上述地址空间。由于64位CPU支持48位虚拟地址空间^①，而CUDA GPU只支持40位，应用程序使用UVA时应确保CUDA被提前初始化，以保证CUDA所需要的虚拟地址先于CPU代码中的分配请求而被满足。

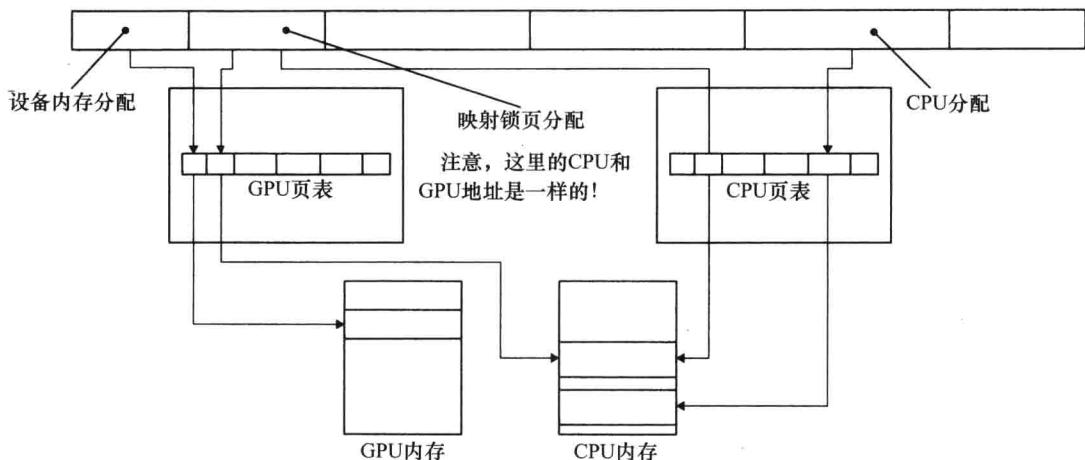


图2-19 统一虚拟寻址(UVA)

对于映射锁页分配，GPU和CPU的指针是相同的。对于其他类型的分配，CUDA可以通过地址推断出既定分配存在于哪个设备。其结果是，线性内存复制函数族（需要指定方向的`cudaMemcpy()`和`cuMemcpyHtoD()`、`cuMemcpyDtoH()`等）被更简单的`cuMemcpy()`和`cudaMemcpy()`函数所替代，而后两个函数并不需要指定内存方向。

支持UVA的系统会自动启用UVA。在本书撰写过程中，使用TCC驱动程序的64位Linux、64位MacOS和64位Windows都支持UVA。然而，WDDM驱动程序暂时还不支持UVA。当UVA生效时，CUDA所执行的所有锁页分配都是被映射的和可分享的。对已被使用`cuMemRegisterHost()`函数分配为锁页的系统内存，需要注意设备指针仍然需要使用`cu(da)`

^① 48位虚拟地址空间 = 256TB 。未来的64位CPU将支持更大的地址空间。

HostGetDevicePointer() 函数查询。即使在 UVA 生效时，CPU 也不能访问设备内存。另外，默认情况下，GPU 不能互相访问彼此的内存。

2.4.6 点对点映射

在介绍 CUDA 的虚拟内存抽象机制旅程的最后阶段，我们讨论一下点对点的设备内存映射如图 2-20 所示。点对点可以使费米架构 GPU 读写另一个费米架构 GPU 的内存。点对点映射仅支持启用 UVA 的平台，并且只对连接到相同 I/O 集线器上的 GPU 有效。由于使用点对点映射时 UVA 始终是有效的，不同设备的地址空间范围不重叠，并且驱动程序（和运行时）可以从指针值推断出所驻留的设备。

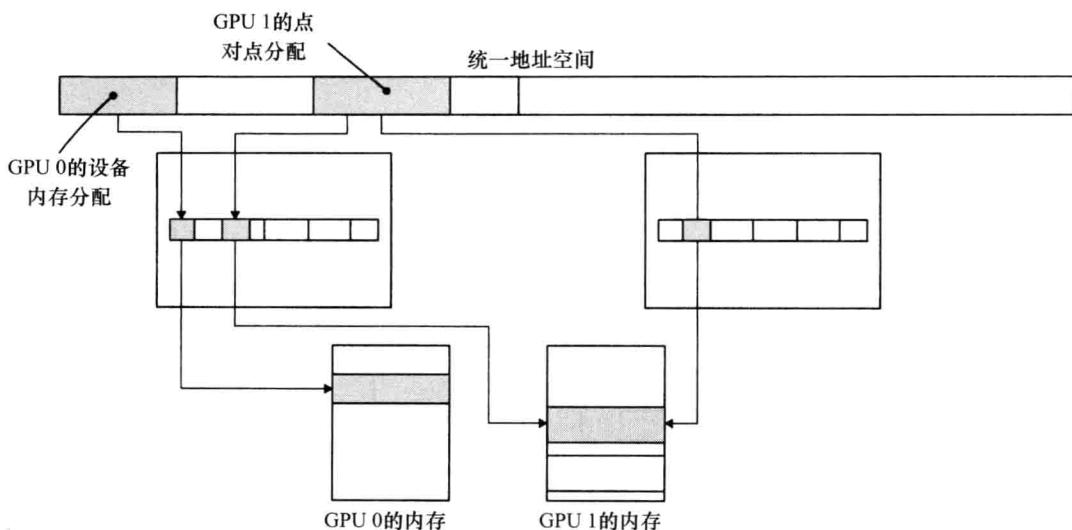


图 2-20 点对点映射

点对点内存寻址是非对称的。例如，图 2-20 为非对称的映射，在该图中，对于 1 号 GPU 的内存分配，0 号 GPU 是可见的，反之则不行。为了让 GPU 之间能够看到对方的内存，每个 GPU 必须显式地映射其他 GPU 的内存。而对于用来管理点对点映射的 API 的函数，我们将在 9.2 节进行相应讨论。

2.5 CPU/GPU 交互

本节描述 CPU/GPU 交互的关键知识点如下。

- 锁页主机内存：GPU 可以直接访问的 CPU 内存；
- 命令缓冲区：由 CUDA 驱动程序写入命令，GPU 从此缓冲区读取命令并控制其执行；
- CPU/GPU 同步：指的是 CPU 如何跟踪 GPU 的进度。

本节在硬件层次上介绍这些设施，仅为帮助读者了解它们如何适用到 CUDA 开发的需要时才使用 API。为了简单起见，本节使用图 2-1 中的 CPU/GPU 模型，省略了多 CPU 或者多 GPU 编程的复杂性。

2.5.1 锁页主机内存和命令缓冲区

出于某些显而易见的原因，CPU 和 GPU 最擅长访问它们自己的内存，但是 GPU 可以通过直接内存访问（direct memory access, DMA）方式来访问 CPU 中的锁页内存。锁页是操作系统常用的操作，可以使硬件外设直接访问 CPU 内存，从而避免过多的复制操作。“被锁定”的页面已被操作系统标记为不可被操作系统换出的，所以设备驱动程序给这些外设编程时，可以使用页面的物理地址直接访问内存。而 CPU 仍然可以访问上述锁页内存，但是此内存是不能移动或换页到磁盘上的。

由于 GPU 是不同于 CPU 的设备，DMA 还可以使 GPU 读取和写入 CPU 内存的操作与 CPU 的执行操作相互独立且并行执行。我们必须注意 CPU 和 GPU 之间的同步，以避免竞争。而对于可以在 GPU 活动时有效利用 CPU 时钟周期的应用程序而言，并发执行具有显著的性能优势。

图 2-21 描绘了一个为直接访问而被 GPU 映射的“锁页”缓冲区^①。CUDA 程序员都很熟悉锁页缓冲区，因为 CUDA 一直允许他们通过 `cudaMallocHost()` 等 API 来分配锁页内存。在表面下的内在机制中，对应这种缓冲区的主要应用程序之一是将命令提交到 GPU。CPU 将命令写入一个供 GPU 消耗的“命令缓冲区”，与此同时，GPU 读取并执行先前写入的命令。图 2-22 显示的是 CPU 和 GPU 如何共享这个缓冲区。此图是被简化了的，因为这类命令可能是几百字节大小，而该缓冲区大到足够容纳几千个这样的命令。缓冲区的“前缘”正由 CPU 构建且尚未准备好供 GPU 读取，“后缘”正在被 GPU 读取。介于中间的命令全部准备就绪，只要 GPU 进程准备好了就可以进行传输。

通常情况下，CUDA 驱动程序可以重用命令缓冲区的内存，因为一旦 GPU 处理完一条

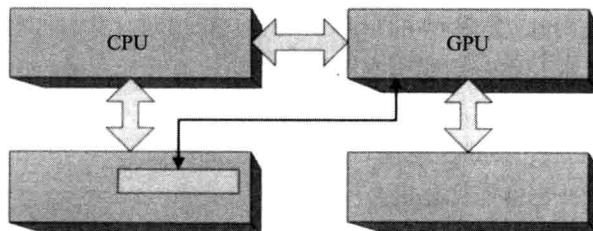


图 2-21 锁页缓冲区

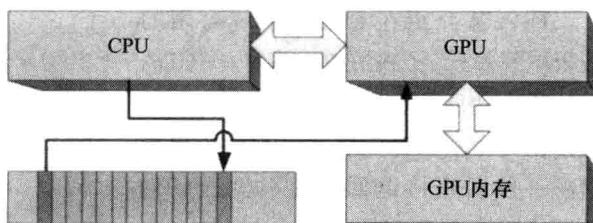


图 2-22 CPU/GPU 命令缓冲区

^① 重要提示：在此背景下，GPU 的“映射操作”涉及设置指向 CPU 内存的物理地址的硬件表。该内存也可能无法被映射到能被 CUDA 内核程序访问的地址空间中。

命令后，该命令占用的内存可以被 CPU 重新写入。图 2-23 显示了 CPU 是如何“循环处理”命令缓冲区的。

考虑到启动一个 CUDA 内核程序需要几千个 CPU 时钟周期，此时，一个 CPU/GPU 并发的重要方法是在 GPU 进行处理时准备更多的 GPU 命令。应用程序并不能均衡地保持 CPU 和 GPU 忙碌，有可能会出现“CPU 受限”或“GPU 受限”的情况，分别如图 2-24 和图 2-25 所示。在一个 CPU 受限型应用程序里，GPU 随时就绪，只要下一个命令准备好就马上进行处理；而在一个 GPU 受限型应用程序里，CPU 的命令缓冲区已经完全被填满，在写入下一个 GPU 命令之前必须等待 GPU 处理完上一命令。有些应用程序本质上是 CPU 或 GPU 受限型的，所以 CPU 和 GPU 受限并不一定说明应用程序的结构有什么本质的问题。然而，知道应用程序是 CPU 受限型的还是 GPU 受限型的有助于找出性能优化机会。

2.5.2 CPU/GPU 并发

前一节介绍在 CUDA 系统可用的最粗粒度并行：CPU/GPU 并发。CUDA 内核程序的所有启动都是异步的：CPU 通过将命令写入命令缓冲区来请求启动内核，然后直接返回，而不检查 GPU 进度。内存复制也可以选择异步方式，这使 CPU/GPU 并发以及可能使内存复制与内核处理并发执行。

1. 阿姆达尔法则 (Amdahl's Law)

当 CUDA 程序编写正确时，CPU 和 GPU 可以完全地进行并行操作，这有可能使性能翻倍。然而 CPU 或 GPU 受限型程序不会从 CPU/GPU 并发得到太大好处，因为即使其他装

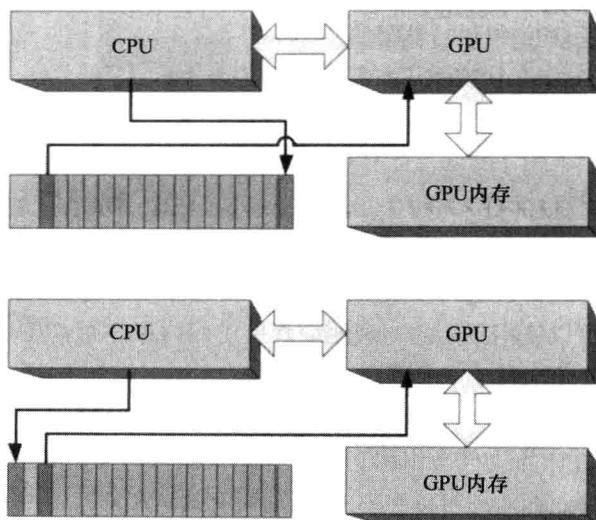


图 2-23 命令缓冲区的循环处理

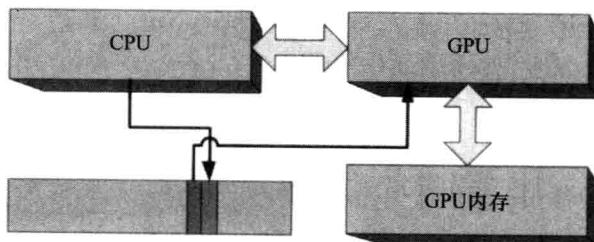


图 2-24 GPU 受限型应用程序

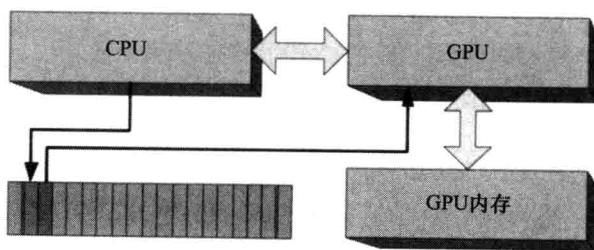


图 2-25 CPU 受限型应用程序

置也是并行运行的，CPU 或 GPU 的某一方也会限制整体性能。这种含糊不清的观察可以使阿姆达尔法则正确描述，该定律于 1967 年在一篇论文中被吉恩·阿姆达尔（Gene Amdahl）首次阐明。[⊖]阿姆达尔法则通常如下表示：

$$\text{加速比} = 1/(r_s + r_p / N)$$

其中， $r_s+r_p=1$ 且 r_s 代表的是串行部分比率。在研究 CPU/GPU 并发等小规模的性能场合时，这一公式形式似乎不太方便。所以，将其变形成如下公式：

$$\text{加速比} = N/(N(1-r_p)+r_p)$$

这清楚地表明，如果 $r_p=1$ ，则加速为 N 倍。如果 CPU 和 GPU 各有一个 ($N=2$)，那来自全并发的最大加速为 2 倍。对平衡工作负载这基本是可实现的，例如视频转码中，CPU 可以与 GPU 同时执行，其中 GPU 执行并行操作（如像素处理），而 CPU 执行串行操作（如可变长解码）。但对于更多的 CPU 或 GPU 受限型应用程序，这种类型的并发只能提供有限的优势。

对那些认为并行是解决一切性能问题的万能药的人来说，阿姆达尔的论文无异于一个警钟。我们将其应用于这本书中讨论内部 GPU 并发、多 GPU 并发，以及移植 CUDA 内核所得加速比等其他地方。它可以用来让我们知道哪些并发形式不能赋予一个给定的应用程序任何优势，因此开发人员可以把时间花在探索其他提高性能的途径上。

2. 错误处理

CPU/GPU 并发也会对错误处理造成影响。当 CPU 开始启动一批内核而其中一个导致内存故障时，CPU 并不能发现这一故障，直到它执行了 CPU/GPU 同步才行（在下一节描述）。开发人员可以手动执行 CPU/GPU 同步作为辅助，具体通过调用 `cudaThreadSynchronize()`、`cuCtxSynchronize()` 等函数来完成。在诸如 `cudaFree()` 或 `cuMemFree()` 的函数调用中也会导致 CPU/GPU 的同步。《CUDA C 语言编程指南》中通过使用可能会导致 CPU/GPU 同步的函数，提及了这一行为：“注意，这个函数也可能会从之前的异步启动返回错误代码。”

由于目前 CUDA 是并发执行的，如果真的发生了故障，我们并没有办法知道是哪个内核程序造成的故障。至于调试代码，如果难以通过同步操作来隔离故障，开发人员可以设置 `CUDA_LAUNCH_BLOCKING` 的环境变量，以迫使所有启动的内核同步。

3. CPU/GPU 同步

CUDA 的大多数 GPU 命令涉及内存复制或内核程序启动的执行，而也有一个重要的命令子集帮助 CUDA 驱动程序跟踪 GPU 处理命令缓冲区时的进展。由于应用程序不知道一个给定的 CUDA 内核程序会运行多久，所以 GPU 本身必须给 CPU 汇报工作进度。图 2-26 显示的是命令缓冲区和“同步位置”（也位于锁页主机内存上），它们被 CUDA 驱动程序和 GPU 使用，以跟踪相应进程。一个单调递增的整数值（“进度值”）是由驱动程序维护，每一个主

[⊖] <http://bit.ly/13UqBm0>。

要的 GPU 操作后都跟着一个将新进度值写入共享同步位置的命令。如图 2-26 中的例子，进度值将一直为 3，直到 GPU 完成当前命令的执行并将 4 写入同步位置中为止。

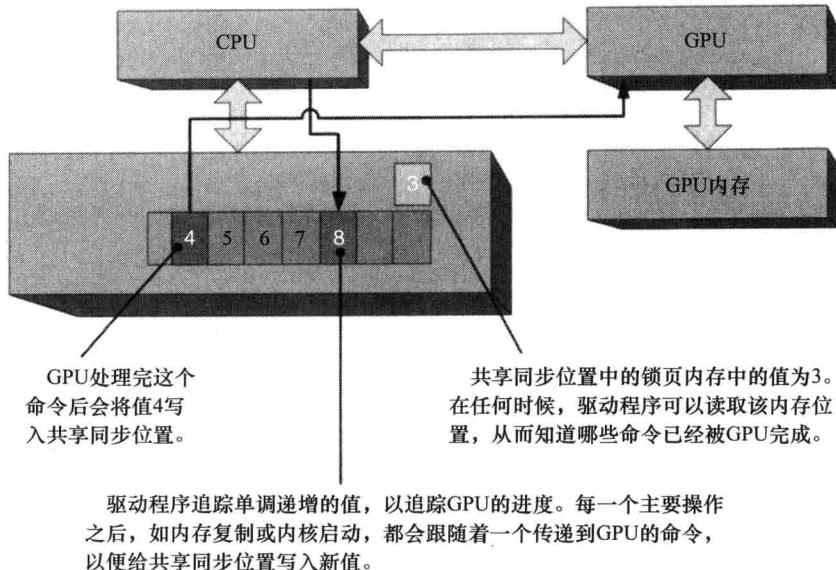


图 2-26 共享同步值（条件满足前）

CUDA 既隐式又显式地暴露了这些硬件功能。上下文范围的同步通过简单调用 `cuCtxSynchronize()`、`cudaThreadSynchronize()` 等函数来检查 GPU 请求的最近同步值，并且一直等待，直到同步位置获得该值。例如，在图 2-27 中，如果由 CPU 写入的命令 8 之后紧接着 `cuCtxSynchronize()` 或 `cudaThreadSynchronize()` 函数，则驱动程序将一直等待，直到共享同步值大于或等于 8 为止。

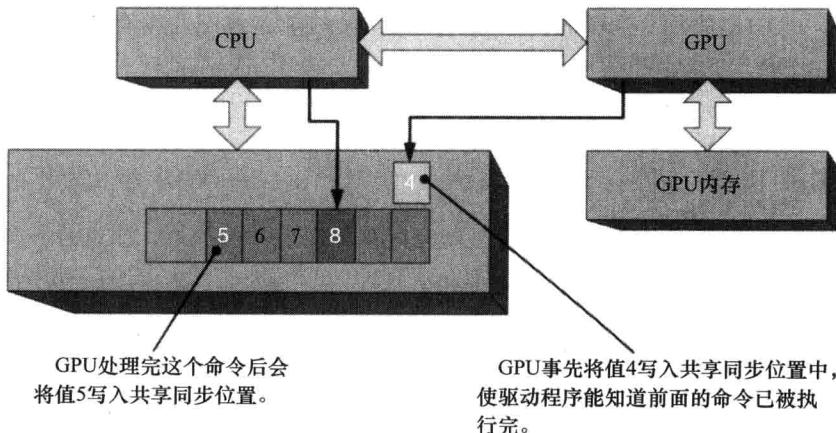


图 2-27 共享同步值（条件满足后）

CUDA 事件则更明确地暴露了这些硬件能力。cuEventRecord() 函数的作用是将一个命令加入队列使得一个新的同步值写入共享同步位置中，cuEventQuery() 和 cuEventSynchronize() 则分别用于检查和等待这个事件的同步值。

早期版本的 CUDA 只是简单地轮询共享的同步位置，反复地读内存，直到等待准则满足为止。但是这种方法代价很大，且只有当应用程序不必等待太久时才有用（即同步位置不一定要被读取很多遍，就可以因等待标准已经得到满足而退出）。对大多数应用程序来说，基于中断的方案（CUDA 公开称为“阻塞同步”）更好，因为它们使 CPU 等待线程挂起，直到 GPU 发出中断信号为止。驱动程序将 GPU 中断映射到一个特定的线程同步原语平台，如 Win32 事件或 Linux 的信号。当应用程序开始等待时，若等待条件不成立，这样做可以挂起 CPU 的线程。

通过指定 CU_CTX_BLOCKING_SYNC 到 cuCtxCreate() 或指定 cudaDeviceBlockingSync 到 cudaSetDeviceFlags()，应用程序可以强制使用上下文范围的同步进入阻塞状态。然而，使用阻塞的 CUDA 事件（指定 CU_EVENT_BLOCKING_SYNC 到 cuEventCreate() 或指定 cudaEventBlockingSync 到 cudaEventCreate()）更可取，因为它们粒度更细且可以与任何类型的 CUDA 上下文进行无缝互操作。

敏锐的读者可能会关注 CPU 和 GPU 在不使用原子操作或其他同步原语的情况下读取和写入这种共享的内存位置。但由于 CPU 只读取共享位置，竞争条件并不是关注点。这样，最坏的情况是 CPU 读取了一个“过时的”值，从而导致其等待时间要比实际的长。

4. 事件和时间戳

主机接口有一个机载高分辨率计时器，它可以在写入一个 32 位的同步值时同时写一个时间戳。CUDA 使用这个硬件设施实现 CUDA 事件中的异步计时功能。

2.5.3 主机接口和内部 GPU 同步

GPU 可能包含多个引擎，以使内核执行和内存复制并发进行。在这种情况下，GPU 的驱动程序将写入一些命令，这些命令被分发到同时运行的不同引擎中。每个引擎都有自己的命令缓冲区和共享同步值，引擎进度的跟踪如图 2-26 和图 2-27 所述。图 2-28 显示的是两个复制引擎和一个计算引擎并行工作时的情形。其中，主机接口负责读取命令并将其调度到相应引擎。在图 2-28 中，一个主机到设备的内存复制操作和两个相关操作（一个内核启动和一个设备到主机的内存复制）已被提交给硬件。依据 CUDA 编程抽象模型，这些操作都是在同一个流上进行的。这个流就像是一个 CPU 线程，在内存复制之后接受内核启动的提交。因此，CUDA 驱动程序为了实现内部 GPU 同步必须将命令插入主机接口的命令流中。

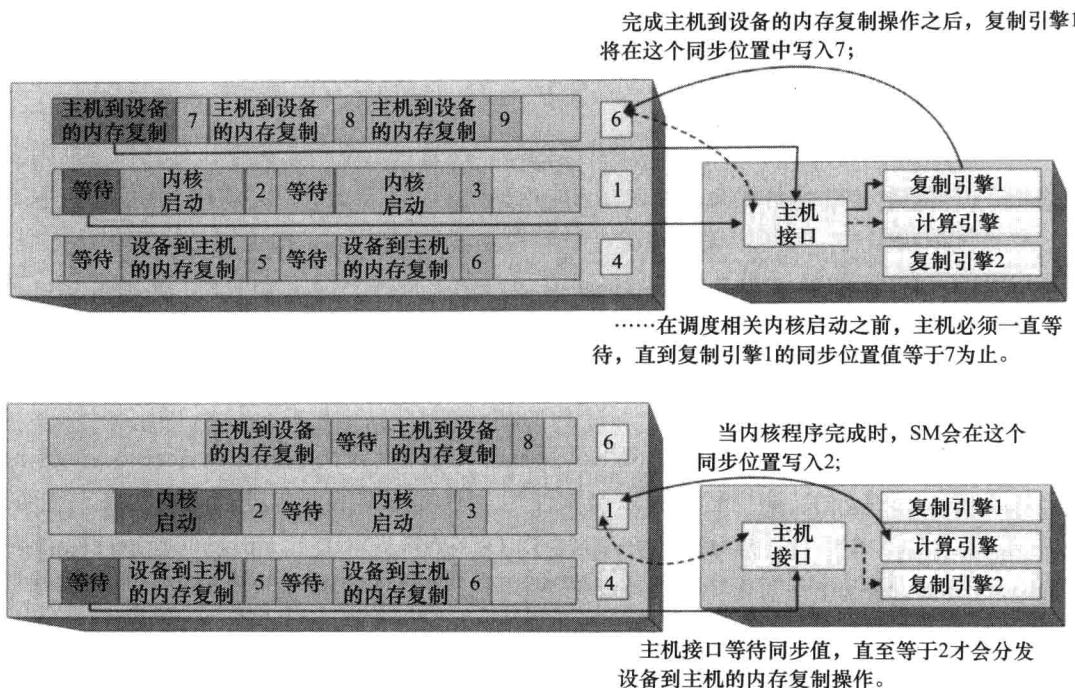


图 2-28 内部 GPU 同步

如图 2-28 所示，主机接口在协调流同步需求上起着核心作用。例如，在完成所需的内存复制之前，内核是启动不了的。此时，DMA 单元可以停止给既定引擎传递命令，直到共享的同步位置获得一个特定值为止。此操作与 CPU/GPU 同步类似，但 GPU 是对它内部的不同引擎进行同步的。

在此硬件机制之上的软件抽象模型是一个 CUDA 流。它在该操作中就像 CPU 线程一样，每个流都是串行排队，为了并发执行需要多个流。由于引擎间共享命令缓冲区，应用程序必须在不同的流里以软件的方式流水线化它们的请求。因此，它们必须完成

```
foreach stream
    Memcpy device←host
    Launch kernel
    Memcpy host←device
```

而不是

```
foreach stream
    Memcpy device←host
foreach stream
    Launch kernel
foreach stream
    Memcpy host←device
```

没有软件方式的流水线，DMA 引擎将会通过引擎的同步以维护每个流的串行执行模式，

这将打破并发性。

开普勒架构上的多 DMA 引擎

英伟达最新开普勒架构的硬件实现了一个引擎对应一个 DMA 单元，从而避免了应用程序需要以软件的方式对它们的流操作进行流水线化。

2.5.4 GPU 间同步

由于图 2-26 ~ 图 2-28 中的同步位置都是在主机内存上，所以它们可以被系统中的任何一个 GPU 访问。其结果是，在 CUDA 4.0 中，英伟达能够在 `cudaStreamWaitEvent()` 和 `cuStreamWaitEvent()` 函数的形式中添加 GPU 之间的同步。这些 API 调用导致驱动程序为主机接口将等待命令插入当前 GPU 的命令缓冲区中，使得 GPU 一直等待，直到事件的给定同步值被写入为止。从 CUDA 4.0 开始，事件不一定会被等待中的同一个 GPU 用信号唤醒。流原先只能在单个 GPU 的硬件单元之间同步执行，现在已提升到可以在 GPU 之间同步执行了。

2.6 GPU 架构

以下三种不同的 GPU 架构可以运行 CUDA：

- 特斯拉架构 (Tesla) 硬件，如在 2006 年推出的 GeForce 8800 GTX (G80) 中。
- 费米架构 (Fermi) 硬件，如在 2010 年推出的 GeForce GTX 480 (GF100) 中。
- 开普勒架构 (Kepler) 硬件，如在 2012 年推出的 GeForce GTX 680 (GK104) 中。

GF100/GK104 命名来源于实现 GPU 的 ASIC 集成电路。它们之中的“F”和“K”分别代表费米架构和开普勒架构。

特斯拉和费米系列继承了英伟达的传统，他们会首先出货大型高端旗舰级芯片来赢取基准测试。这些芯片是昂贵的，因为英伟达的制造成本与制造 ASIC 时所需要的晶体管数量（进而影响芯片面积大小）密切相关。通过大型芯片的基准测试之后，会出货更小的芯片：中端的尺寸为大型的 $\frac{1}{2}$ ，低端的为 $\frac{1}{4}$ ，诸如此类。

与这一传统不同的是，英伟达的第一个开普勒级芯片定位于中端层次。上述大型芯片出货几个月后，第一个开普勒级芯片成功完成。GK104 拥有 35 亿个晶体管，而 GK110 则拥有 71 亿晶体管。

2.6.1 综述

从 CUDA 看来，GPU 可以作如下简化：

- 一个连接 GPU 与 PCIe 总线的主机接口；
- 0 ~ 2 个复制引擎；
- 一个连接 GPU 与其设备内存的 DRAM 接口；
- 一定数目的 TPC 或 GPC (纹理处理集群或图形处理集群)，每个包含一定的缓存和一些流处理器簇 (SMs)。

在本章结尾引用的有关架构的论文将对支持 CUDA 的 GPU 的功能进行更为完整的叙述，其中包括支持抗锯齿渲染等特定图形功能。

1. 主机接口

主机接口实现了上一节中已经描述的功能。读取 GPU 命令（如内存复制和内核启动命令），并将其分派给相应的硬件单元，而且还具备了 CPU 和 GPU 之间、GPU 上不同引擎之间和不同 GPU 之间同步的设施。在 CUDA 中，通过流和事件 API（见第 6 章），主机接口的基本功能得以体现。

2. 复制引擎

复制引擎可以在流处理器簇做计算时执行主机与设备之间的内存传输。最早的 CUDA 硬件并没有任何复制引擎，后来版本的硬件则包括了一个复制引擎，可以传输线性设备内存（CUDA 数组除外），而最新的 CUDA 硬件则包括了两个复制引擎，可以使 PCIe 总线饱和并可以在 CUDA 数组和线性内存之间转换。[⊖]

3. DRAM 接口

GPU 的 DRAM 接口包含了用于合并内存请求的硬件，可以支持超过 100GB/s 的带宽。越是新的 CUDA 硬件所含的 DRAM 接口越高级。最早的硬件（SM 1.x）有复杂的合并要求，其要求地址要连续且 64、128 或 256 字节对齐（根据操作数大小而定）。从 SM 1.2（GT200 或 GeForce GTX 280）开始，无地址对齐要求，地址可以依据数据局部性被合并起来。费米架构硬件（SM 2.0 和更高版本）有一个直写模式的二级缓存，可以提供和 SM 1.2 合并硬件一样的好处，并在数据重用时带来额外性能提升。

4. TPC 和 GPC

TPC 和 GPC 是介于整个 GPU 和流处理器簇之间的硬件单元，用于执行 CUDA 计算。特斯拉架构硬件将 SM 组合成 TPC (纹理处理集群)，其中，TPC 包含有纹理硬件支持（特别包含一个纹理缓存）和 2 个或 3 个 SM，后面会有详细描述。费米架构硬件组则将 SM 组合为 GPC (图形处理器集群)，其中，每个 GPU 包含有一个光栅单元和 4 个 SM。

[⊖] 两个以上的复制引擎并没有多大的意义，因为每个引擎都能使 PCIe 的两个方向之一饱和。

在大多数情况下，CUDA 开发人员并不需要关心 TPC 或 GPC，因为对于计算硬件来说，SM 才是计算硬件的最重要的抽象单元。

5. 特斯拉架构和费米架构对比

第一代支持 CUDA 的 GPU 的代号为特斯拉，而第二代为费米。在开发过程中，它们属于机密代码，但英伟达决定将它们作为外部产品的名称来描述前两代支持 CUDA 的 GPU。为了制造悬念，英伟达选择使用“特斯拉”这一名称来形容使用 CUDA 机器构建计算集群的服务器级别的 GPU 板。^①为了区分昂贵的服务器级别特斯拉板和架构族，本书指的架构族是“特斯拉架构硬件”、“费米架构硬件”和“开普勒架构硬件。”



注意 特斯拉架构硬件与费米架构硬件之间的差异也适用于开普勒架构。

在执行非合并的内存事务的代码时，早期的特斯拉架构硬件存在严重的性能损失（高达 6 倍）。后来随着特斯拉架构硬件的实现，从 GeForce GTX 280 开始，将非合并事务的损失减少了约 2 倍。特斯拉架构硬件也有性能计数器，使开发人员能够测量有多少内存事务是非合并的。

特斯拉架构硬件只包括了一个 24 位整数乘法器，因此开发人员必须使用 `_mul24()` 等内建函数，以获得最佳性能。完整的 32 位乘法（即 CUDA 中原生的操作符 *）通过小型指令序列进行模拟。

特斯拉架构硬件对共享内存初始化为零，而费米架构硬件并不进行初始化。对于使用驱动程序 API 的应用程序而言，这种变化的一个细微副作用是：在费米架构下使用 `cuParamSeti()` 来传递指针参数到 64 位平台上的应用程序，并不能正常工作。由于参数是在特斯拉架构硬件的共享内存中传递的，参数未初始化的上半部分将成为 64 位指针中最重要的 32 位。

SM 1.3 中的 GT200 引入双精度支持。而 GT200 属于特斯拉架构第二代“赢取基准测试的”芯片。^② 在当时，该特性被认为是有风险的，因此它以节省面积的方式实现，可以按照英伟达期望的双精与单精比率来增删硬件（如对于 GT200，这一比率为 1 : 8）。费米架构更彻底的集成双精度支持且性能更好。^③ 最后，对于图形应用程序，特斯拉架构硬件是第一款支持 DirectX 10 的硬件。

费米架构硬件的功能比特斯拉架构硬件更强大。它支持 64 位寻址，增加了一级和二

① 当然，特斯拉品牌刚出现时，费米架构硬件还没出现。营销部门的工程师告诉我们，体系架构代号和品牌名称都是“特斯拉”只是一个巧合。

② 事实上，SM 1.2 与 SM 1.3 唯一的不同是 SM 1.3 支持双精度。

③ 在 SM 3.x 中，英伟达不再把双精度浮点性能当做影响 SM 版本的因素，因此，GK104 的双精度浮点运算性能较差，而 GK110 的较好。

级缓存，加入了一个完整的 32 位整数乘法指令和一些专门用于扫描算法的新指令，它增加了表面加载 / 存储操作，以便 CUDA 内核可以在不使用纹理硬件的情况下就能读取和写入 CUDA 数组，它是第一个以多复制引擎为特色的 GPU 系列，改善了虚函数等 C++ 代码支持。

费米架构硬件并不包括跟踪非合并内存传输所需的性能计数器。此外，因为它不包括有 24 位乘法器，费米架构硬件在运行要使用 24 位内建乘法函数时可能会出现一些小的性能损失。在费米架构中，使用乘法运算符 * 是最快速的方式。

对于图形应用程序，费米架构硬件可以运行 DirectX 11。表 2-1 总结了特斯拉架构和费米架构硬件之间的差异。

表 2-1 特斯拉架构和费米架构硬件之间的差异

特 性	特斯拉架构	费米架构
非合并内存事务造成的性能损失	高达 8 倍 ^①	高达 10% ^②
24 位乘法器	√	
32 位乘法器		√
共享内存 RAM	√	
一级缓存		√
二级缓存		√
并发内核		√
表面加载 / 存储		√

①：对于 Tesla 硬件，高达 2 倍。

②：启用 ECC 时，高达 2 倍。

6. 纹理操作的细微差别

特斯拉架构和费米架构硬件之间的一个细微差异是这样的：在特斯拉架构的硬件中，执行纹理操作的指令，其输出会覆盖输入寄存器向量。在费米架构硬件中，输入和输出寄存器向量可以是不同的。其结果是，特斯拉架构硬件可能要使用额外的指令来移动纹理坐标到输入寄存器，以恢复原始输入向量。

特斯拉架构和费米架构硬件的另一个微小的区别是，当对一维 CUDA 数组进行纹理操作时，费米架构硬件通过使用二维纹理并设置 0.0 为第二个坐标来模拟相应功能。由于这个模拟的成本只需一个额外的寄存器和少数额外指令，所以应用程序很少会注意到这一差异。

2.6.2 流处理器簇

GPU 的力量之源是流处理器簇（也称为 SM）。就像上一节所提到的，每个 SM 1.X 硬件中的 TPC 包含 2 个或 3 个 SM，而每个 SM 2.X 硬件中的 GPC 则包含 4 个 SM。最先支持 CUDA 的 GPU——G80 或 GeForce 8800 GTX，包含 8 个 TPC，而每一个 TPC 中有 2 个 SM，所以一共有 16 个 SM。接下来的一个支持 CUDA 的大 GPU 是 GT200 或 GeForce GTX 280，它增加了 TPC 中的 SM 数量，包含 10 个 TPC 并且每个 TPC 含有 3 个 SM，总共 30 个 SM。

每一个支持 CUDA 的 GPU 中 SM 的数量可能从二到几十个，并且每一个 SM 包含：

- 执行单位，用以执行 32 位整数和单、双精度浮点数运算；
- 特殊函数单元 (SFU)，用以计算 \log/exp , \sin/\cos , $rcp/rsqrt$ 的单精度近似值；
- 一个线程束调度器，用以协调把指令分发到执行单元；
- 一个常量缓存，用于广播式传送数据给 SM；
- 共享内存，用于线程之间的数据交换；
- 纹理映射的专用硬件。

图 2-29 展现了特斯拉架构流处理器簇 (SM 1.X)。它包含了 8 个流处理器，支持 32 位整数和单精度浮点数运算。第一个 CUDA 硬件根本不能支持双精度浮点数运算，但是自从 GT200 开始，SM 就包含了双精度浮点数运算的单元。^①

图 2-30 展示了费米架构流处理器簇 (SM 2.0)。不同于特斯拉架构的分离方式实现的双精度浮点数支持，费米架构的每一个 SM 2.0 拥有完整的双精度支持。虽然双精度执行速度比单精度慢，但由于 SM 2.0 的单 - 双精比率比特斯拉架构的 8 : 1 小得多，所以整体双精度性能要高很多。^②

图 2-31 显示的是更新过后的费米架构流处理器簇 (SM 2.1)，可以在诸如 GF104 芯片上找到。为了获得更高的性能，英伟达选择增加流处理器的数量到 48 个。SFU 同 SM 的比率从 1 : 8 增加到 1 : 6。

图 2-32 显示了最近的（截止到本文写作时）流处理器簇的设计，它们为我们展现了英伟达最新开普勒架构硬件的特色。这种设计与前几代的设计完全不同，被英伟达称为 SMX (下一代 SM)。其中，核心的数量增加了 6 倍，到了 192 个，并且每个 SMX 都远大于以往 GPU 中类似的 SM。最大的费米架构 GPU——GF110 有大约 30 亿晶体管，共 16 个 SM；而 GK104 有 35 亿个晶体管和更高的性能，却只有 8 个 SMX。为了节省面积和提高能耗效率，英伟达为每一个 SM 增加了许多共享内存 / 一级缓存资源。就像费米架构的 SM 一样，每个开普勒架构 SMX 拥有 64KB 缓存，它可以分割成 48KB 一级缓存与 16KB 共享内

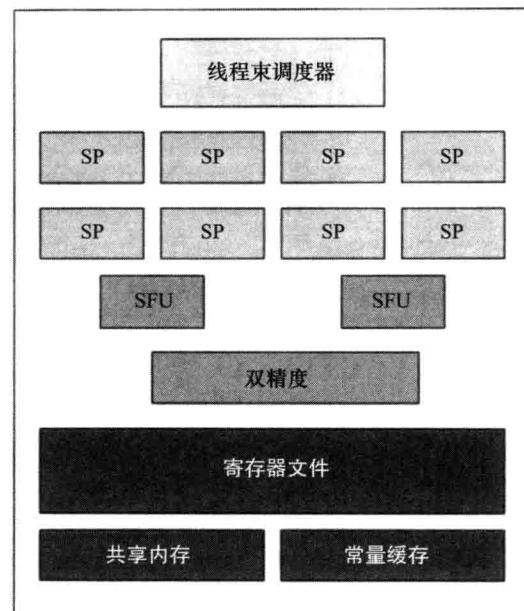


图 2-29 SM 1.x

^① GT200 添加了一些指令集和双精度支持（如共享内存的原子操作），因此 GT200 的指令集中没有双精度的属于 SM 1.2，有双精度的属于 SM 1.3。
^② 对于开普勒架构的硬件，英伟达可以根据 GPU 的目标市场不同将浮点性能进行调整。

存，或者是 48KB 共享内存与 16KB 一级缓存。对于 CUDA 开发者来说，开普勒架构相比之前的架构，更鼓励将数据存储在寄存器中（而不是一级缓存，或者共享内存）。

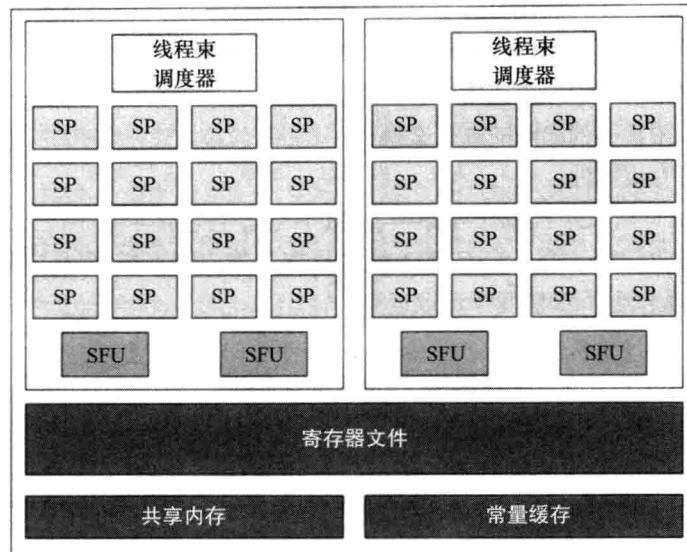


图 2-30 SM 2.0 (费米架构)

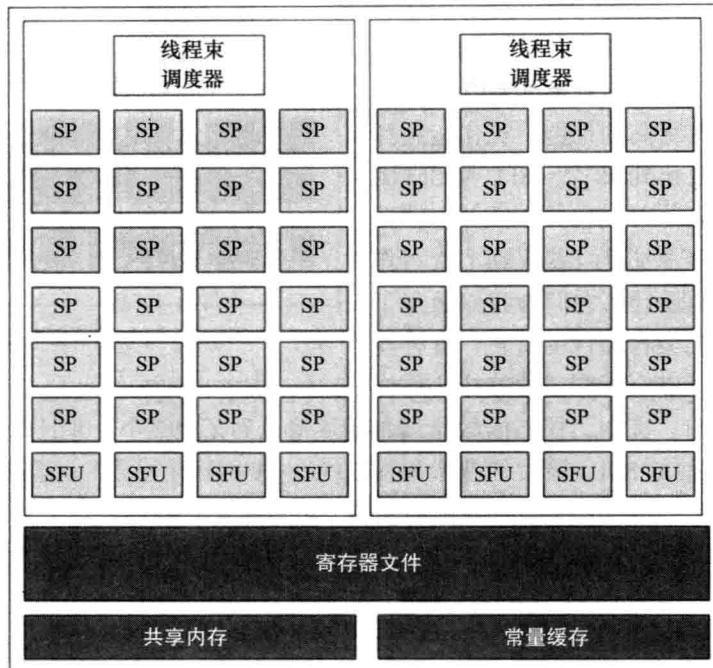


图 2-31 SM 2.1 (费米架构)

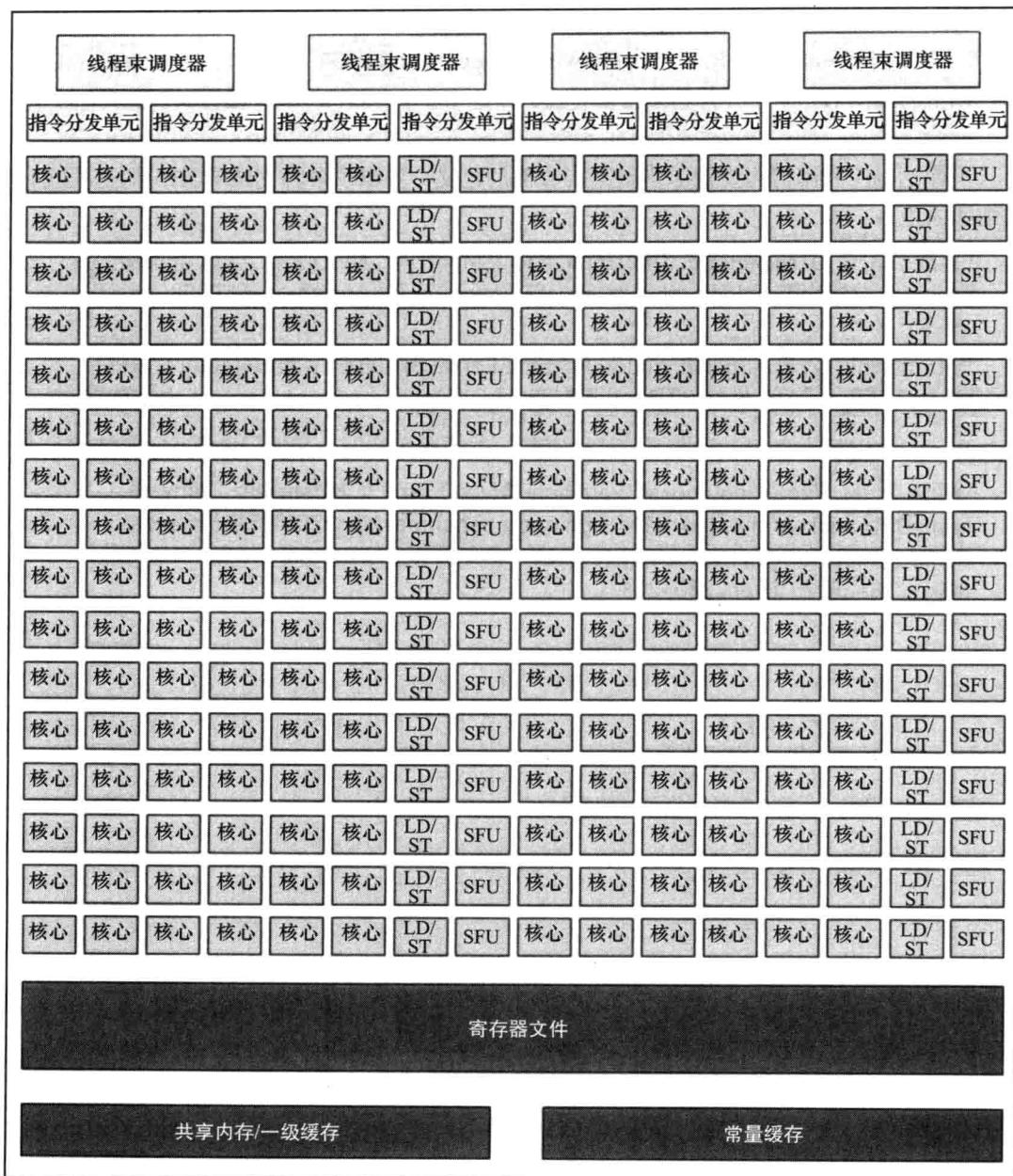


图 2-32 SM 3.0 (SMX)

2.7 延伸阅读

英伟达已经在其网站上的白皮书中详细描述了费米架构和开普勒架构。下面的白皮书介

绍了费米架构。

NVIDIA GeForce GPU 的下一代：www.nvidia.com/object/GTX_400_architecture.html。

以下的白皮书介绍了开普勒架构及其在英伟达 GeForce GTX 680 (GK104) 的具体实现：www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf。

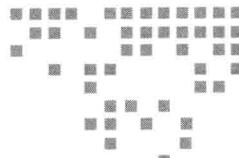
英伟达的工程师们也发表了多篇架构方面的论文，提供了各种支持 CUDA 的 GPU 的更详细的描述：

Lindholt, E., J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28 (2), March–April 2008, pp. 39–55.

Wittenbrink, C., E. Kilgariff, and A. Prabhu. Fermi GF100 GPU architecture. *IEEE Micro* 31 (2), March–April 2011, pp. 50–59.

Wong 等人使用 CUDA 开发了验证性实验，并阐明了特斯拉架构硬件方面的知识点：

Wong, H., M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. 2010 IEEE International Symposium on Performance Analysis of Systems and Software (IPSASS), March 28–30, 2010, pp. 235–246.



软件架构

本章将会介绍 CUDA 的软件架构。在第 2 章中，我们介绍了 CUDA 的硬件平台和同 CUDA 交互的方法，我们会在本章的开始部分向大家介绍 CUDA 的软件平台和 CUDA 所支持的操作环境。然后，会对 CUDA 中的每个软件抽象从设备与上下文（context）到模块（module）与内核到内存作简要的介绍。这一部分在描述特定的软件抽象如何被硬件所支持时会回顾第 2 章中的内容。最后，我们会花一些时间对比 CUDA 运行时（CUDA runtime）和驱动程序 API（driver API），并考察 CUDA 源代码是怎样编译为在 GPU 上执行的微码（microcode）的。当然本章仅仅是对 CUDA 软件架构的一个概览，更多主题以及更详细的内容会在后面的章节中展开。

3.1 软件层

图 3-1 展示了 CUDA 应用程序中的不同层次，从应用程序本身到 CUDA 硬件上执行的 CUDA 驱动程序。除了内核模式驱动程序以外的所有软件都执行在目标操作系统的非特权用户模式下。在现代多任务操作系统的安全模型中，用户模式是“不可信任的”，硬件与操作系统软件必须采取措施严格分割一个与另一个应用程序。对 CUDA 来说，这意味着一个 CUDA 应用程序使用的主机和设备内存无法被另一个 CUDA 程序使用，唯一的例外是这些程序申请内存共享，这只有在内核模式下才可做到。

CUDA 库，例如 cuBLAS，建立在 CUDA 运行时或驱动程序接口层之上。CUDA 运行时是支持 CUDA 的集成 C++/GPU 工具的库。当 nvcc 编译器分割 .cu 文件分别发送至主机和设备，主机的部分会自动生成对 CUDA 运行时的调用，以方便像 nvcc 中用特别的 3 对尖括号 <<>>> 启动内核这样的操作。

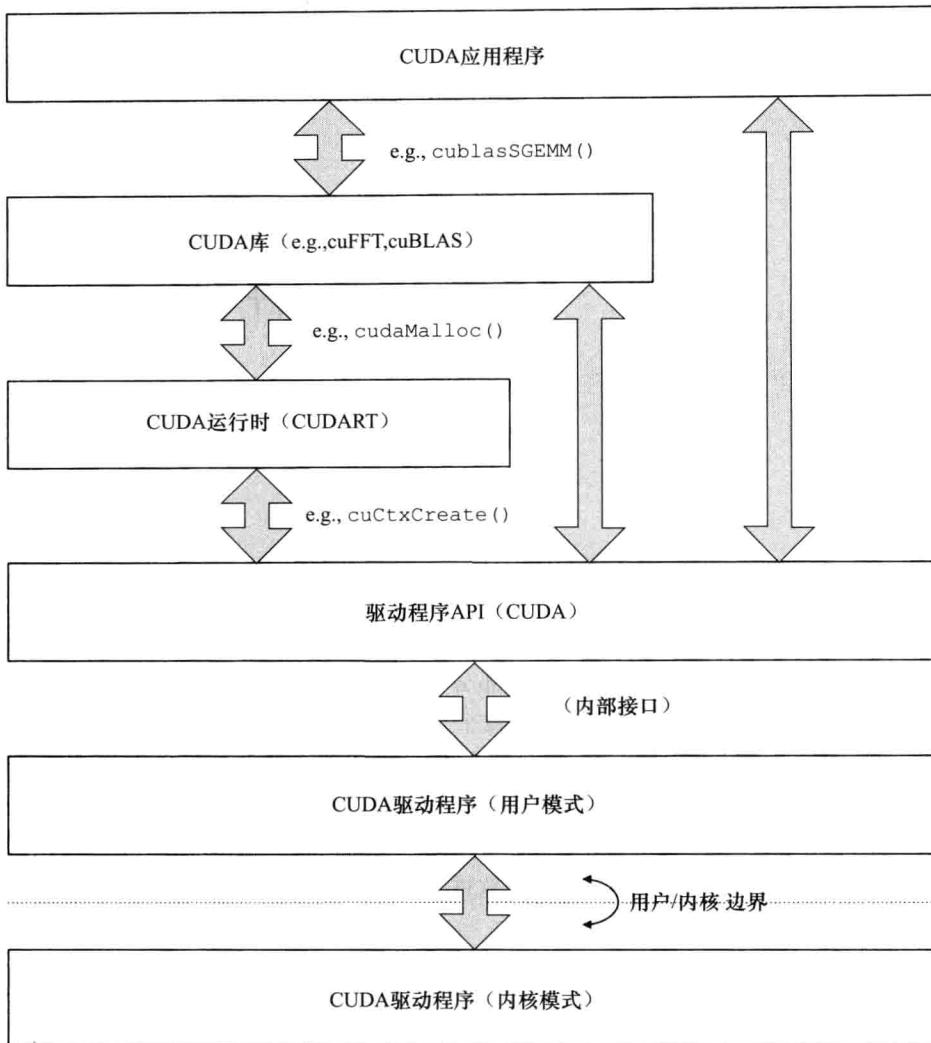


图 3-1 CUDA 软件层次结构

CUDA 驱动程序 API，由 CUDA 用户模式驱动程序直接导出，是 CUDA 应用程序可用的最底层 API。驱动程序 API 直接调用用户模式驱动程序，之后可能会继续调用内核模式做分配内存等操作。驱动程序 API 和 CUDA 运行时中的函数名称分别以 cu*() 和 cuda*() 打头，许多函数，像 cudaEventElapsedTime()，在本质上是相同的，只是在前缀上有一些区别。

3.1.1 CUDA 运行时和驱动程序

CUDA 运行时（经常简写为 CUDART）是一个由 CUDA 语言集成环境使用的库。每一个版本的 CUDA 工具集（CUDA toolchain）都有其特定的 CUDA 运行时版本，应用程序会自

动链接和运行时兼容的工具集，程序只有当匹配版本的 CUDART 在路径中可用时才能正确运行。

CUDA 驱动程序是后向兼容的，支持所有同版本 CUDA 编写的程序和旧版本程序。它提供一个相对低层的“驱动程序 API”（在 `cuda.h` 中），使开发者能够更紧凑地整理资源和为初始化定时。驱动程序的版本可由 `cuDriverGetVersion()` 函数获取。

```
CUresult CUDAAPI cuDriverGetVersion(int *driverVersion);
```

函数返回一个十进制数，给出由驱动程序支持的 CUDA 版本，例如，3010 代表 CUDA3.1，5000 代表版本 5.0。

表 3-1 总结了 CUDA 每个版本中的特性。在 CUDA 运行时应用程序中，版本信息由 `cudaDeviceProp` 结构体中的 `major` 和 `minor` 成员提供。我们会在 3.3.2 小节中详细审视 `cudaDeviceProp`。

表 3-1 CUDA 驱动程序每版本新特性

CUDA 版本	引入的驱动程序新特性
1.0	CUDA
1.1	流与事件，并发 1D 内存复制和内核执行
2.0	3D 纹理操作
2.1	提升 OpenGL 互用性
2.2	可分享、映射和写结合锁定内存；等步长内存纹理操作
3.0	费米架构，多引擎复制和内核并行执行
3.1	GPUDirect
3.2	64 位寻址；CUDA 内核中的 <code>malloc()/free()</code>
4.0	统一虚拟寻址；提升的线程支持；主机内存注册；GPUDirect 2.0（点对点内存复制和映射）；分层纹理
4.1	立体纹理；进程间点对点映射
4.2	开普勒架构（Kepler）
5.0	动态并行；GPUDirect RDMA

CUDA 运行时需要机器中安装的驱动程序版本高于或等于由运行时支持的 CUDA 版本。如果驱动程序的版本低于运行时版本，CUDA 应用程序会初始化失败，弹出错误 `cudaErrorInsufficientDriver(35)`。CUDA 5.0 引入了设备运行时（device runtime）概念，这是 CUDA 运行时的一个子集，可被 CUDA 内核直接调用。有关设备运行时的更详细描述在第 7 章给出。

3.1.2 驱动程序模型

除了 Windows Vista 和其后续版本，所有 CUDA 运行的操作系统（Linux、MacOS 和 Windows XP）通过用户模式客户端驱动程序（user mode client driver）访问硬件。这些驱动程序都不约而同地回避了一个需求，这一情况在所有现代操作系统中都普遍存在，即硬件资源

可被内核代码操纵。现代的硬件，例如 GPU 可以巧妙地解决这个需求——通过分配特定的寄存器（例如给硬件提交工作的硬件寄存器）进入用户模式。因为用户模式不被操作系统信任，所以对用户模式下的硬件寄存器，硬件必须包含对抗流氓写入的保护程序。这样做的目的是防止用户模式代码通过直接内存存取（DMA）修改不应该修改的代码（例如操作系统内核代码）。

硬件在用户模式和命令流之间使用一个中间层来防止内存损坏，这样 DMA 技术只能在之前已经被验证且已被内核代码映射的内存上使用。驱动程序开发者必须小心验证编写的内核代码，以确保代码只会访问可以使用的内存。这样的好处是程序在向硬件提交工作时不会招致多余的内核模式转换花销，我们的驱动程序可以以峰值的效率执行。

许多操作，例如内存分配，仍然需要内核模式转换，因为编辑 GPU 页表只能在内核模式下执行。因此，用户模式驱动程序采取了一系列步骤来减少内核模式转换的次数，例如，CUDA 内存分配器尽量从内存池取出内存满足内存分配。

1. 统一虚拟寻址

在 2.4.5 小节中详细描述了统一虚拟寻址（unified virtual addressing, UVA），在 64 位 Linux，64 位 XPDDM 和 MacOS 操作系统上均可使用。在这些平台上，统一虚拟寻址是内置可用的。但在本书编写时，UVA 还不能够在 WDDM 上运行。

2. Windows 显示驱动模型

在 Windows Vista 操作系统中，微软公司采用了新的桌面演示模型，在这个模型中，屏幕的输出由后台缓冲区与页面翻转协作完成，正如同电子游戏一样。这个新的“Windows 桌面管理器”（WDM）有能力比以往 Windows 版本更广泛地使用 GPU，所以微软公司决定修改 GPU 驱动模型以适应现行的桌面演示模型。新的 GPU 驱动模型——Windows 显示驱动模型（Windows display driver model, WDDM）已经是 Windows Vista 及其后续版本中的默认驱动模型。而 XPDDM 则继续使用在 Windows 的以往版本中^①。

就 CUDA 而言，WDDM 有两个主要的改变：

1) WDDM 不再允许硬件寄存器被分配进入用户模式。硬件命令——甚至是启动 DMA 操作的指令——必须通过内核模式来调用。用户→内核转换成本对机器来说是一个十分昂贵的行为，机器中可以使用用户模式驱动程序缓冲区来暂存指令，延迟提交。

2) WDDM 的设计初衷是为了允许多应用程序同时使用同一 GPU，但 GPU 并不支持请求式调页，WDDM 中包括了在内存对象基础上模拟换页的工具。对图形应用程序，目标包括内存对象、Z 缓冲区或纹理内存；对 CUDA，内存对象包括全局内存和 CUDA 数组。由于驱动需要在内核调用之前建立好对 CUDA 数组的访问，所以 CUDA 数组可以被 WDDM 交

^① 特斯拉板（Tesla boards）（没有任何显示输出的能运行 CUDA 的电路板）可以在 Windows 上使用 XPDDM，称为特斯拉计算集群（Tesla compute cluster, TCC）驱动程序，可被 nvidia-smi 工具触发。

换。全局内存是驻留在一段连续的地址空间（可以存储指针的地方）里的，为了让内核成功启动，给定 CUDA 上下文中每一个内存对象都会驻留在这段地址空间上。

上述第一点的主要影响：对 CUDA 请求的工作，例如，内核启动或异步内存复制操作，一般来说不会立即提交给硬件去操作。

提交被挂起的工作的一个惯例操作是查询默认流（NULL stream）：cudaStreamQuery(0) 或 cuStreamQuery(NULL)。如果没有挂起的工作，这一调用会迅速返回；如果有工作正在挂起，这项工作将被提交，因为这个调用是异步的，执行可能在硬件处理结束之前返回给调用者。而在非 WDDM 平台上，查询 NULL 流总是非常快的。

上述第二点的主要作用是使 CUDA 内存分配控制更加灵活。在用户模式客户端驱动程序中，成功的内存分配意味着：一旦内存空间被分配，那么它无法为操作系统中的其他客户端程序（例如游戏或正在运行的 CUDA 应用程序）再次占用。在 WDDM 下，如果有应用程序在相同的 GPU 中竞争使用时间，Windows 能够交换出内存对象使每一个应用程序顺利运行。Windows 操作系统会尽其所能提高这项工作的效率。当然，对于所有的可能有换页操作的程序，尽量不让换页操作发生是最好的选择。

3. 超时检测与恢复

因为 Windows 使用 GPU 同用户交互，所以计算程序不能占用 GPU 过多的时间。在 WDDM 中，Windows 设定了强制延时界限（默认为 2 秒），这意味着，如果出现了持续的时间超过设定延时的情况，系统会弹出带有“显示驱动程序停止响应并已恢复”（display driver stopped responding and has recovered）的对话框，随后显示驱动程序重新启动。如果这种情况发生，在所有 CUDA 上下文所做的工作都会丢失。查看 <http://bit.ly/16mG0dX> 获取详细内容。

4. 特斯拉计算集群驱动程序

因为计算程序不需要 WDDM，英伟达提供**特斯拉计算集群**（TCC）驱动程序，只在特斯拉级板上可用（Tesla-class board）。特斯拉计算集群驱动程序（Tesla compute cluster driver）是一个用户模式下的客户端驱动程序，所以这并不需要内核转换（kernel thunk）去给硬件提交工作，我们可以使用 nvidia-smi 工具来启用或禁用它。

3.1.3 nvcc、PTX 和微码

nvcc 是 CUDA 开发者使用的编译器驱动程序，它可以使 CUDA 源代码转成一个可执行的 CUDA 应用程序。nvcc 可以做很多事，如编译、链接和用一条指令执行一个示例程序（本书中很多程序鼓励使用的一种用法），编译一个只在 GPU 上执行的 .cu 文件。

图 3-2 展示了使用 nvcc 的推荐工作流程，包括 CUDA 运行时和驱动程序 API 应用程序。对于规模很大的应用程序，nvcc 最好严格地编译 CUDA 代码并封装 CUDA 功能到可以被其他工具调用的代码，这是由于 nvcc 存在如下限制：

- nvcc 只在特定的编译器上工作。许多 CUDA 开发者不曾注意到这一限制，因为他们只使用支持 CUDA 的编译器。但是在软件生产环境中，CUDA 的代码量与其他代码相比简直是微不足道，所以编译器是否支持 CUDA 未必是决定编译器使用的因素。
- nvcc 所改变的编译环境可能与其他大量应用程序的生成环境不兼容。
- nvcc 会使用一些非标准内建类型（如 int2）和固有名称（如 __popc()）“破坏”命名空间的纯洁性。只有在最近的 CUDA 版本中，固有的符号才变成可选项，且可以通过引入适当的 sm_*_intrinsics.h 头文件来使用。

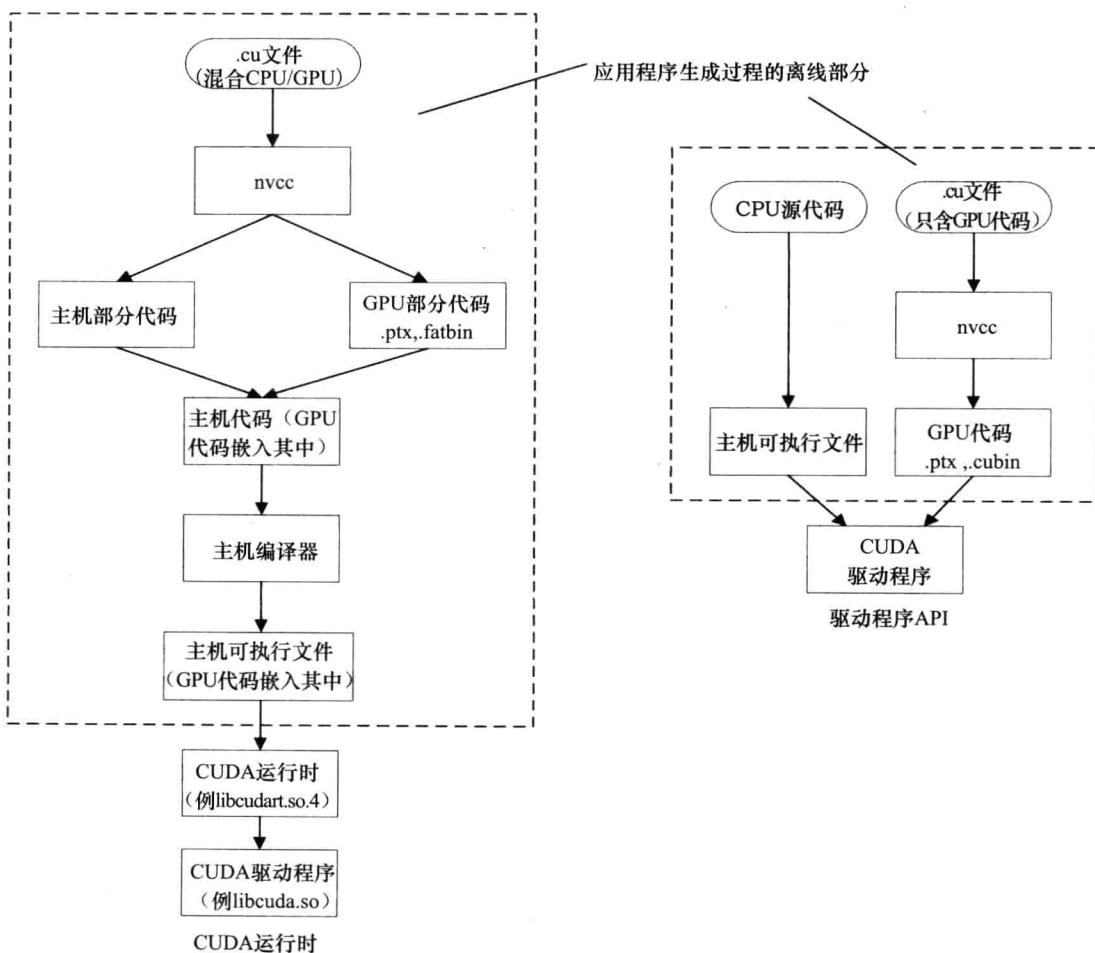


图 3-2 nvcc 工作流程图

对于 CUDA 运行时应用程序，nvcc 会把 GPU 代码转化为字符串字面值，嵌入可执行文件。如果选用 --fatbin 选项，可执行文件会自动加载目标 GPU 的微码，如果没有微码可用，会让驱动程序自动编译 PTX 为微码。

nvcc 与 PTX

并行线程执行 (Parallel Thread eXecution, PTX) 代码是编译后的 GPU 代码的一种中间形式，它可以再次编译为原生的 GPU 微码。这是英伟达公司为 CUDA 程序面向未来指令集所建立的一个创新机制，这意味着只要给定的 CUDA 内核的 PTX 是可用的，CUDA 驱动程序便可将其编译为在 GPU 上执行的程序的微码（即使在编写代码时 GPU 对其并不可用）。

PTX 可以离线或在线方式编译为 GPU 微码。离线编译指的是生成未来可在计算机上执行的程序。在图 3-2 中，我们圈出了 CUDA 编译过程中的离线编译部分。对于在线编译方式，我们可以理解为“即时”(just-in-time, JIT) 编译，指的是程序可以在运行时将 PTX 代码在线编译为 GPU 微码。

nvcc 可以通过调用 PTX 编译器 ptxas 离线编译 PTX，这将会把 PTX 编译为特定 GPU 版本的原生微码。生成的微码存储在 CUDA 二进制形式的 ‘cubin’（发音类似 “Cuban”）文件中，cubin 文件可以使用 cuobjdump --dump-sass 命令进行反汇编，这将根据特定 GPU 的微码生成 SASS 代码^②。

PTX 同样可以通过 CUDA 驱动程序在线编译 (JITted)。当我们运行使用 --fatbin 选项编译的（这是默认选项）CUDART 程序时，在线编译是自动启动的。每个内核的 .cubin 和 PTX 都包含在可执行文件里，如果硬件不支持可执行文件里的 .cubin，驱动程序便会编译 PTX。驱动程序将其缓存在磁盘上，因为编译 PTX 可能花费大量时间。

最后，PTX 可以在运行时通过显式调用 cuModuleLoadEx() 编译。驱动程序 API 不会自动嵌入或加载 GPU 微码。.cubin 和 .ptx 均作为参数传给 cuModuleLoadEx() 函数，如果 .cubin 不兼容机器的 GPU 架构，会返回一个错误。对驱动程序 API 开发者来说，一个合理的处理策略是编译并嵌入 PTX，并且应该总调用 cuModuleLoadEx() 以 JIT 编译到 GPU，依赖驱动程序来缓存编译后的微码。

3.2 设备与初始化

设备就是指真实的 GPU 实体。当 CUDA 初始化完成（显式调用驱动程序接口 cuInit() 或隐式调用 CUDA 运行时函数），CUDA 驱动程序将枚举可用的设备，并创建一个全局数据结构体，这个结构体包括设备名称和一些不可变的参数，例如设备内存数量和最大时钟速率等。

在一些平台上，英伟达包含设置特定设备的工具。nvidia-smi 工具可设置 GPU 的策略，例如，它可以启用或禁用 ECC（纠错），它也可以用来控制设备创建的 CUDA 上下文数量。下面是一些可能的模式：

^② 审视 SASS 代码是程序优化的主要策略之一。

- 默认：多个 CUDA 上下文可被设备创建。
- 独占模式：只有一个 CUDA 上下文可被设备创建。
- 禁止模式：禁止设备创建 CUDA 上下文。

如果一个设备已被列出，但无法在其上创建上下文，那么设备可能是处于禁止模式或者独占模式中，对于后者，说明已有一份 CUDA 上下文被创建。

3.2.1 设备数量

应用程序可以通过调用 cuDeviceGetCount() 或 cudaGetDeviceCount() 函数查询有多少设备可用。之后，设备可以通过一个来自范围 [0..DeviceCount-1] 的索引来引用它。驱动程序 API 需要应用程序调用 cuDeviceGet() 来映射设备的索引到一个设备句柄 (CUdevice) 上。

3.2.2 设备属性

驱动程序 API 应用程序可以通过调用 cuDeviceGetName() 来查询设备名称，调用 cuDeviceTotalMem() 查询全局内存。设备的主要与次要计算能力 (即 SM 版本，例如第一代费米架构 GPU 的计算能力为 2.0) 可以使用 cuDeviceComputeCapability() 函数查询。

CUDA 运行时应用程序可以调用 cudaGetDeviceProperties()，返回包括设备名称与属性信息的结构体 cudaDeviceProp，表 3-2 给出了结构体中成员的描述。

CUDA 驱动程序 API 接口函数 cuDeviceGetAttribute() 可以查询设备属性，根据 CUdevice_attribute 参数一次返回一个属性。CUDA 5.0 中，CUDA 运行时添加了相同的功能函数：cudaDeviceGetAttribute()，想必因为基于结构体的接口在设备上运行时很笨重。

表 3-2 cudaDeviceProp 成员

cudaDeviceProp 成员	描 述
char name[256];	ASCII 字符串设备标识
size_t totalGlobalMem;	设备可用全局内存 (字节)
size_t sharedMemPerBlock;	每线程块可用共享内存 (字节)
int regsPerBlock;	每线程块可用 32 位寄存器
int warpSize;	线程束中线程容量
size_t memPitch;	最大内存复制步长
int maxThreadsPerBlock;	每线程块最大线程数
int maxThreadsDim[3];	线程块维度的最大值
int maxGridSize[3];	网格维度最大值
int clockRate;	时钟频率 (千赫)
size_t totalConstMem;	设备可用常量内存 (字节)
int major;	主计算能力

(续)

cudaDeviceProp 成员	描述
int minor;	次计算能力
size_t textureAlignment;	纹理对齐要求
size_t texturePitchAlignment;	要求绑定到等步长内存的纹理索引满足等步长对齐约束
int deviceOverlap;	设备可以并发进行内存复制和内核程序执行。已弃用，使用 asyncEngineCount 代替
int multiProcessorCount;	设备中的处理器簇数量
int kernelExecTimeoutEnabled;	指明内核程序是否有运行时限制
int integrated;	设备是否是集成 GPU (而不是独立 GPU)
int canMapHostMemory;	设备可以使用 cudaHostAlloc/cudaHostGetDevicePointer 对主机内存进行映射
int computeMode;	计算模式 (参见 ::cudaComputeMode)
int maxTexture1D;	最大 1D 纹理大小
int maxTexture1DMipmap;	最大 1D 细化纹理大小
int maxTexture1DLinear;	线性内存相关的 1D 纹理大小
int maxTexture2D[2];	最大 2D 维度
int maxTexture2DMipmap[2];	最大 2D 细化纹理维度
int maxTexture2DLinear[3];	最大维度 (宽)
int maxTexture2DGather[2];	执行纹理聚集操作时的最大纹理维度
int maxTexture3D[3];	最大 3D 纹理维度
int maxTextureCubemap;	最大立方图纹理维度
int maxTexture1DLayered[2];	最大 1D 分层纹理维度
int maxTexture2DLayered[3];	最大 2D 分层纹理维度
int maxTextureCubemapLayered[2];	最大立方图分层纹理维度
int maxSurface1D;	最大 1D 表面大小
int maxSurface2D[2];	最大 2D 表面维度
int maxSurface3D[3];	最大 3D 表面维度
int maxSurface1DLayered[2];	最大 1D 分层表面维度
int maxSurface2DLayered[3];	最大 2D 分层表面维度
int maxSurfaceCubemap;	最大立方图表面维度
int maxSurfaceCubemapLayered[2];	最大立方图分层表面维度
size_t surfaceAlignment;	表面对齐要求
int concurrentKernels;	设备可能执行多个内核并发
int ECCEnabled;	设备是否打开 ECC
int pciBusID;	设备 PCI 总线 ID

(续)

cudaDeviceProp 成员	描述
int pciDeviceID;	PCI 设备 ID
int pciDomainID;	PCI 域 ID
int tccDriver;	如果设备是一个启用 TCC 驱动程序的特斯拉设备，返回 1
int asyncEngineCount;	异步引擎数量
int unifiedAddressing;	设备与主机共享统一的地址空间
int memoryClockRate;	内存峰值时钟频率（千赫）
int memoryBusWidth;	全局内存总线宽度（位）
int l2CacheSize;	二级缓存大小（字节）
int maxThreadsPerMultiProcessor;	每个处理器簇最大驻留线程数

3.2.3 无 CUDA 支持情况

CUDA 运行时程序可以运行在不能运行 CUDA 的机器或未安装 CUDA 的机器上。如果 cudaGetDeviceCount() 函数返回 cudaSuccess 和一个非 0 的设备数，CUDA 就是可用的。

与 CUDA 运行时相反，当使用驱动程序 API 时，除非驱动程序二进制文件是可用的，否则直接链接 nvcuda.dll (Windows) 或 libcuda.so (Linux) 的可执行文件便不会成功加载。正因为驱动程序 API 需要 CUDA，所以当 CUDA 不存在时，直接启动程序会引发图 3-3 中的错误。

对于 CUDA 的应用程序，CUDA SDK 提供了一组头文件和一个 C 源文件，包装了驱动程序 API，这样，应用程序便可以检查 CUDA 而避免操作系统发出异常信号。这些文件，存放在子目录 <SDKRoot>/C/common/inc/dynlink 下，可以代替 CUDA 的核心文件。它们会插入一组中间函数，可以在 CUDA 可用时延迟加载 CUDA 库。

作为例子，让我们比较两个小程序。它们使用驱动程序 API 初始化 CUDA 并输出系统中的设备名字。代码清单 3-1 给出了 init_hardcoded.cpp 文件的代码，通过下面的命令可以使 CUDA SDK 编译这个文件：

```
nvcc -o init_hardcoded -I ../../chLib init_hardcoded.cpp -lcuda
```

使用 nvcc 编译 C++ 文件，不包含任何的 GPU 代码，是一种很方便的获取 CUDA 头文件的方式。在这行命令的开始，-o init_hardcoded 确定了输出的可执行文件的根名称。命令最后的 -lcuda 使 nvcc 链接驱动程序 API 库，缺少它，会引发一个链接错误。这个程序硬链接 (hard-link) CUDA 驱动程序 API，所以在未安装 CUDA 的机器上会编译失败。

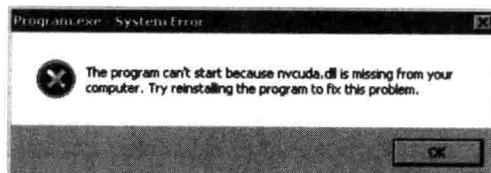


图 3-3 当 CUDA 不存在时的错误对话框 (Windows)

代码清单 3-1 初始化（硬编码）

```

/*
 * init_hardcoded.cpp
 *
 */
#include <stdio.h>

#include <cuda.h>
#include <chError.h>

int
main()
{
    CUresult status;
    int numDevices;

    CUDA_CHECK( cuInit( 0 ) );
    CUDA_CHECK( cuDeviceGetCount( &numDevices ) );

    printf( "%d devices detected:\n", numDevices );
    for ( int i = 0; i < numDevices; i++ ) {
        char szName[256];
        CUdevice device;
        CUDA_CHECK( cuDeviceGet( &device, i ) );
        CUDA_CHECK( cuDeviceGetName( szName, 255, device ) );
        printf( "\t%s\n", szName );
    }

    return 0;
Error:
    fprintf( stderr, "CUDA failure code: 0x%x\n", status );
    return 1;
}

```

代码清单 3-2 给出的程序不需要系统中装有 CUDA。可以看到，源代码几乎是相同的，仅有几行差别。其中，

```
#include <cuda.h>

被替换为：

#include "cuda_drvapi_dynlink.c"
#include "dynlink/cuda_drvapi_dynlink.h"
```

而 cuInit() 函数调用改变为指定一个 CUDA 版本。即，

```
CUDA_CHECK( cuInit(0) );
```

被替换为：

```
CUDA_CHECK( cuInit( 0, 4010 ) );
```

这里我们为 CUDA_CHECK 传入了第二个参数 4010，代表 CUDA 4.1 版本，如果系统不包括 4.1 版本的函数库，程序会编译失败。

注意你可以单独编译然后链接 cuda_drvapi_dynlink.c 到你的应用程序里，代替直接‘include’这个文件到源代码中。头文件和 C 文件一同工作，在驱动程序 API 中插入一组包装好的函数。头文件使用预处理器重新命名驱动程序 API 函数以包装在 cuda_drvapi_dynlink.h

头文件中声明的函数（例如调用 cuCtxCreate() 会实际调用 tcuCtxCreate()）。在支持 CUDA 的系统中，驱动程序动态链接库是动态加载的，包装好的函数会调用在初始化时从驱动程序动态链接库获取的函数指针；在没有 CUDA 的系统中，或者驱动程序不匹配请求的 CUDA 版本，初始化函数会返回一个错误。

代码清单 3-2 初始化 (dynlink)

```

/*
 * init_dynlink.cpp
 *
 */

#include <stdio.h>

#include "dynlink/cuda_drvapi_dynlink.h"
#include <chError.h>

int
main()
{
    CUresult status;
    int numDevices;

    CUDA_CHECK( cuInit( 0, 4010 ) );
    CUDA_CHECK( cuDeviceGetCount( &numDevices ) );

    printf( "%d devices detected:\n", numDevices );
    for ( int i = 0; i < numDevices; i++ ) {
        char szName[256];
        CUdevice device;
        CUDA_CHECK( cuDeviceGet( &device, i ) );
        CUDA_CHECK( cuDeviceGetName( szName, 255, device ) );
        printf( "\t%s\n", szName );
    }

    return 0;
Error:
    fprintf( stderr, "CUDA failure code: 0x%x\n", status );
    return 1;
}

```

CUDA 专用动态链接库 (CUDA-Only DLLs)

对 Windows 开发者，另一个在不支持 CUDA 的平台上创建 CUDA 应用程序的方式如下：

- 1) 把 CUDA 代码转移到动态链接库。
- 2) 调用 LoadLibrary() 显式加载动态链接库。
- 3) 在 __try/__except 语句中调用 LoadLibrary() 以捕捉 CUDA 不存在情况下的异常。

3.3 上下文

上下文类似于 CPU 中的进程。除了少数例外，它是管理 CUDA 程序中所有对象生命周期的容器，包括如下部分：

- 所有内存分配（包括线性设备内存、主机内存、CUDA 数组）
- 模块
- CUDA 流
- CUDA 事件
- 纹理与表面引用
- 使用本地内存的内核的设备内存
- 进行调试、分析、同步操作时，所使用的内部资源
- 换页内存复制所使用的锁定中转缓冲区

CUDA 运行时不提供对 CUDA 上下文的直接访问，它通过延迟初始化（deferred initialization）来创建上下文。每一个 CUDART 库调用和内核调用都会检查上下文是否是当前使用的，如果必要的话，创建 CUDA 上下文（使用之前用 `cudaSetDevice()`, `cudaSetDeviceFlags()`, `cudaGLSetGLDevice()` 等函数设置的状态）。

许多应用程序倾向于控制初始化延迟时间。为了让 CUDART 进行没有任何负面效果的初始化，调用

```
cudaFree(0);
```

CUDA 运行时应用程序可以通过驱动程序 API 访问当前上下文栈（后文描述）。

对驱动程序 API 中的每个指定上下文状态的函数，CUDA 运行时把上下文和设备同等对待。对于驱动程序 API 函数 `cuCtxSynchronize()`, CUDA 运行时用 `cudaDeviceSynchronize()` 取而代之；对驱动程序 API 函数 `cuCtxSetCacheConfig()`, CUDA 运行时中则有 `cudaDeviceSetCacheConfig()` 对应。

当前上下文

为了代替当前上下文栈，CUDA 运行时提供 `cudaSetDevice()` 函数，为调用的线程设置当前上下文。一个设备可以是多个 CPU 线程的当前上下文[⊖]。

3.3.1 生命周期与作用域

所有与 CUDA 上下文相关的分配资源都在上下文被销毁的同时被销毁，除了少数例外，给定 CUDA 上下文创建的资源可能不能被其他的 CUDA 上下文使用，这一限制不仅适用于内存，也同样适用于 CUDA 流与 CUDA 事件等对象。

3.3.2 资源预分配

CUDA 尽力去避免“懒惰分配”（lazy allocation）。懒惰分配只分配需要的资源以避免因缺少资源导致的失败操作。例如，换页内存复制不会因内存不足而失败，因为机器通过分配

[⊖] 早期版本的 CUDA 禁止上下文同时成为多个线程的当前状态，因为驱动程序并不是线程安全的。现在驱动程序实现了所需的同步——即使在应用程序调用同步函数，例如 `cudaDeviceSynchronize()` 的情况下。

锁页中转缓冲区以执行换页内存复制，而这个操作发生在上下文创建时。如果 CUDA 不能分配这些缓冲区，上下文创建就失败了。

在少量情况下，CUDA 不会预分配一个给定操作所需的全部资源。内核启动所需的本地内存数量可能被限制，因此 CUDA 不会预分配最大理论数量的内存。所以，当它需要比默认分配给 CUDA 上下文的内存值更多的本地内存时，内核启动可能失败。

3.3.3 地址空间

除了在上下文销毁时被自动销毁的（清理）对象外，另一个上下文中的重要抽象是地址空间：一组私有的虚拟内存地址，它可以分配线性设备内存或用以映射锁页主机内存。这些地址每一个上下文都不相同。同一地址对不同上下文可能有效也可能无效，当然也不会解析到相同的地址空间除非有特殊规定。CUDA 上下文的地址空间是与 CUDA 主机代码使用的 CPU 地址空间独立的。事实上，不同于共享内存的多 CPU，多 GPU 中的 CUDA 上下文并不共享一个内存空间。当系统使用统一虚拟地址时，CPU 和 GPU 共享相同的地址空间，其中的每个分配都有进程中唯一的地址。但在特殊情况下，CPU 和 GPU 只能读写各自的内存区，像映射的锁页内存（参见 5.1.3 小节）或点对点内存（参见 9.2.2 小节）。

3.3.4 当前上下文栈

大多数 CUDA 入口点并不含有上下文参数。取而代之的，它们在 CPU 线程的“当前上下文”上进行操作，它存储在线程的本地存储句柄里。在驱动程序 API 中，每一个 CPU 线程有一个当前上下文栈，创建一个上下文的同时将这个新上下文压入栈。

当前上下文栈有 3 个主要功能：

- 单线程应用程序可以驱动多个 GPU 上下文。
- 库可以创建并管理它们自己上下文，而不需要干涉调用者的上下文。
- 库不知晓调用它的 CPU 线程信息。

为 CUDA 建立当前上下文栈的最初动机是使单线程 CUDA 应用程序可以驱动多个 CUDA 上下文。在创建并初始化每个 CUDA 上下文后，程序可以将其中的某个上下文从栈弹出，使其变成一个飘浮上下文。由于对一个 CPU 线程，只有一个当前上下文，所以一个单线程 CUDA 应用程序可以在栈中依次压入和弹出上下文，使其在任何时间点上，只有一个飘浮上下文，实现对多个上下文的驱动。

在多数的驱动程序架构中，压入和弹出上下文的成本足够廉价，所以一个单线程应用程序可以保持多个 GPU 同时忙碌。对只在 Windows Vista 和其后续版本中的 WDDM 驱动程序，弹出当前上下文操作只在没有 GPU 命令挂起的时候才能快速运行。当有命令挂起，驱动程序会引发内核转换，以保证在弹出 CUDA 上下文之前提交挂起的命令[⊖]。

[⊖] 这一花销不仅仅局限在驱动程序 API 或当前上下文栈上，当有命令挂起时，调用 `cudaSetDevice()` 来切换设备同样会引发一次 WDDM 上的内核转换。

当前上下文栈的另一个优势是驱动来自多个 CPU 线程的 CUDA 上下文的能力。使用驱动程序 API 的应用程序可以传递 CUDA 上下文到其他 CPU 线程中：使用 cuCtxPopCurrent() 函数弹出上下文，再使用 cuCtxPushCurrent() 从另一个线程中压入上下文。库可以使用这个功能创建 CUDA 上下文，而不需要了解或干涉它们的调用者。例如，一个 CUDA 插件库可以在初始化时创建它自己的 CUDA 上下文，然后弹出并保持其“漂浮”，除非被主程序调用才会中断。漂浮上下文使库完全对哪一个 CPU 线程调用不知情。当然这种使用方式也是有利有弊：一方面，漂浮上下文内存不会被第三方 CUDA 内核的恶意写入污染；另一方面，库只能够在分配的 CUDA 资源上进行操作。

附加和分离上下文

在 CUDA 4.0 之前，每一个 CUDA 上下文都有一个“使用计数”，上下文被创建时设置为 1。cuCtxAttach() 和 cuCtxDetach() 函数分别为使用计数加 1 或减 1^①。采用“使用计数”的目的是为了使应用程序创建的上下文与链接到的库联系起来，这样程序就可以通过自己创建的 CUDA 上下文与库相互操作了^②。

当 CUDART 第一次调用时，如果一个上下文已经处于当前状态，程序会把上下文附加到 CUDART 上，而不是新建一个上下文。CUDA 运行时无法访问上下文的“使用计数”。而在 CUDA 4.0 之后，使用计数已被弃用，使用 cuCtxAttach()/cuCtxDetach() 函数没有任何作用。

3.3.5 上下文状态

类似于 CPU 中的 malloc() 和 printf() 函数，cuCtxSetLimit() 和 cuCtxGetLimit() 函数可以设置和获取内核函数中对应函数的 GPU 空间上限。cuCtxSetCacheConfig() 函数在启动内核时指定所期望的缓存配置（是否分别分配 16kb 和 48kb 给共享内存和一级缓存）。这暗示我们，任何需要 16Kb 以上共享内存空间的内核需要分配 48Kb 的共享内存。除此之外，上下文状态可以被内核特定的状态重写（cuFuncSetCacheConfig()）。这些状态存在于整个上下文范围（换句话讲，它们并不是特定于每一次内核启动的），它们改动的代价十分昂贵。

3.4 模块与函数

模块是代码与一同加载的数据的集合，类似于 Windows 中的动态链接库（DLL）和 Linux 中的动态共享对象（DSO）。CUDA 运行时和 CUDA 上下文不显式支持模块，模块只在 CUDA 驱动程序 API 中可用^③。

^① 直到 CUDA 2.2 中添加了 cuCtxDestroy()，在此之前上下文通过调用 cuCtxDetch() 分离。

^② 回头来看，英伟达公司把引用记数放在比驱动程序 API 更高的软件层会是更明智的选择。

^③ 如果 CUDA 在即时编译中添加来自源代码的多次请求能力（像 OpenCL 能做到的一样），英伟达公司可能会发现在 CUDA 运行时中暴露模块是合适的。

CUDA 中没有类似于对象文件的此类可被整合到模块中的中间结构。取而代之的是，nvcc 直接生成可以被加载为 CUDA 模块的文件：

- 面向特定 SM 版本的 .cubin 文件
- 通过驱动程序可以编译为硬件执行代码的 .ptx

数据不需要以这些形式发送给最终的用户。CUDA 提供加载模块的 API，模块以能够嵌入在可执行文件或其他地方的 NULL 结尾字符串的形式表示[⊖]。

一旦 CUDA 模块被加载，应用程序便可以查询包含在其中的资源。资源类型分如下几种：

- 全局变量
- 函数（内核）
- 纹理引用

一点值得注意：所有的资源在模块加载时就被创建，所以查询函数不会因为缺少资源而失败。

同上下文一样，CUDA 运行时隐藏了模块的管理细节。所有的模块在 CUDART 初始化时加载。对于拥有大量 GPU 代码的应用程序，选择 CUDA 驱动程序 API 而不选择 CUDA 运行时的一个首要原因是：前者可以显式的管理加载和卸载模块。

模块通过调用 nvcc 创建。nvcc 可以创建哪种类型的模块，取决于命令行的参数，参看表 3-3。由于 cubin 已被编译至特定的 GPU 架构，它们不需要即时编译，而且可以非常快地加载。但是它们既不是向后兼容的（例如被编译到 SM 2.x 的 cubin 不能在 SM 1.x 架构上运行）也不是向前兼容的（例如编译到 SM 2.x 架构上的 cubin 不能在 SM 3.x 架构上运行）。所以只有预先准确知道目标 GPU 架构（以及相应 cubin 版本），才可以用没有嵌入相同版本 PTX 代码模块的 cubin 文件作为备用。

表 3-3 nvcc 模块类型

Nvcc 参数	描 述
-cubin	编译到特定的 GPU 架构
-ptx	驱动程序即时编译可使用的中间资源
-fatbin	Cubin 和 PTX 的组合。如果可能，加载适合的 cubin，否则，编译 PTX，只在 CUDA 运行时中有效

PTX 是一种中间语言，将作为驱动程序即时编译的输入文件。因为这项编译会消耗大量的时间，如果硬件和设备没有改变过，驱动程序对给定的 PTX 模块会保存编译过的模块并复用它们。如果驱动和硬件改变，所有的 PTX 模块也需要重新编译。

通过 fatbins 选项，CUDA 运行时会自动操作使用合适的 cubin 版本，前提是 cubin 可用，不然会编译 PTX。不同的版本以字符串的形式被嵌入在 nvcc 发出的主机 C++ 代码中。应用程序使用驱动程序 API 具有对模块的细粒度控制的优势。例如，它可以作为资源嵌入在可执

[⊖] cuModuleLoadDataEx() 函数在 4.2 节中有详细描述。

行文件中，在运行时加密或生成，但是这一使用 cubins（如果不可用，则可以编译 PTX）的过程必须显式实现。

一旦模块被加载，应用程序就可以查询其中包含的资源：全局资源、函数（内核）和纹理引用。还是如前文所述：所有的这些资源在模块加载时就被创建，所以查询函数（表 3-4）不可能由于缺少资源而失败。

表 3-4 模块查询函数

函数	描述
<code>cuModuleGetGlobal()</code>	传回模块中的指针与符号大小
<code>cuModuleGetTexRef()</code>	传回模块中声明的纹理引用大小
<code>cuModuleGetFunction()</code>	传回模块中声明的一个内核函数

3.5 内核（函数）

在 .cu 文件中，用 `_global_` 标识了内核。在 CUDA 运行时中，它们可以使用由三对尖括号 (`<<< >>>`) 构成的单行代码启动。在第 7 章给出了内核怎样被启动和怎样在 GPU 上执行的细节。

模块中的 GPU 可执行代码以内核形式呈现，或者使用 CUDA 运行时的语言集成特性启动 (`<<< >>>` 符号) 或者使用驱动程序 API 中的 `cuLaunchKernel()` 函数启动。在本书写作时，CUDA 还没有任何 CUDA 模块中可执行代码的动态驻留管理。当模块加载时，内核整个地加载进设备内存。

一旦模块被加载，内核可使用 `cuModuleGetFunction()` 函数查询，内核的属性使用 `cuFuncGetAttribute()` 函数查询，内核通过 `cuLaunchKernel()` 函数启动，`cuLaunchKernel()` 包含了一整套已被废弃的 API 入口点：像 `cuFuncSetBlockShape()` 指定下次内核启动使用的线程块大小；函数 `cuParamSetv()` 指定下次内核启动传入的参数；`cuLaunch()`、`cuLaunchGrid()` 和 `cuLaunchGridAsync()` 使用此前设置的状态启动内核。这些 API 很低效，因为这花费了太多的调用来启动一次内核，并且很多参数，像线程块大小，最好在每次请求内核启动时统一指定。

函数 `cuFuncGetAttribute()` 可以被用来查询指定的属性，例如：

- 每线程块最大线程数
- 静态分配共享内存的数量
- 用户分配的常量内存大小
- 每个函数使用的本地内存数量
- 函数中每个线程使用的寄存器数量
- 被编译函数的虚拟（PTX）与二进制架构版本

当使用驱动程序 API，最好使用 `extern C` 来禁止 C++ 中的默认名称改编行为。否则，你

必须对 cuModuleGetFunction() 使用改编的名称。

1. CUDA 运行时

在 CUDA 运行时创建的可执行文件被加载的同时，它会在主机内存创建一个全局数据结构体，来描述当 CUDA 设备创建后应分配的 CUDA 资源。一旦 CUDA 设备被初始化，这些全局数据结构体就被用来一次性创建 CUDA 资源。因为这些全局数据在 CUDA 运行时中属于进程级共享，所以使用 CUDA 运行时不能递增的加载或卸载 CUDA 模块。

因为 CUDA 运行时用 C++ 语言集成，在 API 函数（像 cudaFuncGetAttributes() 和 cudaMemcpyToSymbol()）内，内核和符号必须用名字指定（即，不是字符串字面值）。

2. 缓存设置

在费米架构中，流处理器簇有一级缓存，它可以被分割为 16KB 大小的共享内存 /48KB 一级缓存或 48KB 的共享内存 /16KB 的一级缓存[⊖]。最初，CUDA 允许以每个内核为基础配置缓存，在 CUDA 运行时中使用 cudaFuncSetCacheConfig() 或驱动程序 API 中使用 cuFuncSetCacheConfig() 设置。后来，这一状态变得全局化：使用 cuCtxSetCacheConfig() / cudaDeviceSetCacheConfig() 函数设定默认缓存配置。

3.6 设备内存

设备内存（或线性设备内存）驻留在 CUDA 地址空间上，可以被 CUDA 内核通过标准 C/C++ 指针和数组解引用操作访问。大多数的 GPU 拥有一个专用的设备内存池，直接附加在 GPU 上，通过集成内存控制器进行访问。

CUDA 硬件并不支持请求式换页，所以，所有的内存分配都被真实的物理内存支持着。不同于 CPU 应用程序，可以分配超过实际内存容量的虚拟内存。CUDA 内存分配在物理内存耗尽时会失败。关于内存怎样分配、释放和访问内存的细节会在 5.2 节给出。

1. CUDA 运行时

CUDA 运行时应用程序可以通过调用 cudaGetDeviceProperties() 函数得到设备信息并检查 cudaDeviceProp::totalGlobalMem 查询给定设备的可用内存总量。使用 cudaMalloc() 和 cudaFree() 分配和释放设备内存。cudaMallocPitch() 分配等宽内存，cudaFree() 可以用来释放等宽内存，cudaMalloc3D() 用来执行一次等宽内存的 3D 分配。

2. 驱动程序 API

驱动程序 API 应用程序可以通过调用 cuDeviceTotalMem() 函数查询给定设备的可用

[⊖] SM 3.X 添加了均分缓存的能力（32KB/32KB）。

内存总量。作为选择，`cuMemGetInfo()` 函数除了可以查询总内存，也可以查询空闲的内存。`cuMemGetInfo()` 只有在 CUDA 上下文对给定 CPU 线程是当前状态时才能被调用。`cuMemAlloc()` 和 `cuMemFree()` 分配和释放设备内存。`cuMemAllocPitch()` 分配等宽内存，`cuMemFree()` 可以用来释放它。

3.7 流与事件

在 CUDA 中，流与事件使主机与设备之间的内存复制与内核操作可并发执行。后续的 CUDA 版本通过扩展流的能力来支持在一个 GPU 上的多内核并发执行，并支持多 GPU 并发执行。

CUDA 流被用来粗粒度管理多个处理单元的并发执行：

- GPU 与 CPU
- 在 SM 处理时可以执行 DMA 操作的复制引擎
- 流处理器簇 (SM)
- 并发内核
- 并发执行的独立 GPU

对给定流中的请求操作是串行执行的。狭义上来讲，CUDA 流很像 CPU 的线程，因为一个 CUDA 流中的操作按顺序执行。

3.7.1 软件流水线

因为只有一个 DMA 引擎服务于 GPU 中各种各样的粗粒度硬件资源，应用软件必须对多流中执行的操作进行软件流水线操作。否则，DMA 引擎会因为每个流中的不同引擎间的同步而中断并发。6.5 节中给出怎样使用流水线技术发挥出 CUDA 流的全部优势，在 11 章中会给出更多的范例。

开普勒架构减少了软件流水线流操作的需求，使用英伟达的 Hyper-Q 技术（在 SM 3.5 中首次使用）实际上消除了软件流水线需求。

3.7.2 流回调

CUDA 5.0 引入了另一个 CPU/GPU 同步机制，这补充了现有的机制，这个新机制可使 CPU 线程等待，直到流空闲或事件被记录。流回调函数由应用程序提供，被 CUDA 注册，随后，当流到达 `cuStreamAddCallback()` 函数调用的节点时，被 CUDA 调用。

在流回调期间流执行被挂起，所以，为性能考虑，开发者应该小心确认，在回调时，其他流是否正处于忙碌状态。

3.7.3 NULL 流

任何一个异步的内存复制函数都可以使用 NULL 流作为参数，而且内存复制不会开始，

直到 GPU 之前的操作全部完成。实际上，NULL 流是 GPU 上所有引擎的集结地。除此之外，所有的流内存复制函数都是异步的。可能在内存复制完成之前把控制权返回给应用程序。NULL 流最有用的场合是在不需要使用多流来利用 GPU 内部的并发性时，它可以解决应用程序的 CPU/GPU 并发。一旦流操作使用 NULL 流初始化，应用程序必须使用同步函数（像 cuCtxSynchronize() 或 cudaThreadSynchronize()）来确保操作在执行下一步之前完成。但是应用程序可能在执行同步之前请求了许多此类操作。例如，应用程序可能执行：

- 一次异步主机到设备内存复制；
- 一次或多次内核启动；
- 一次异步设备到主机内存复制。

在上下文同步之前，cuCtxSynchronize() 或 cudaThreadSynchronize() 在 GPU 执行最后一个请求操作后返回。这一方法在小型的不会运行过长时间的内存复制或内核启动中非常有用。CUDA 驱动程序使用这段宝贵的时间来写入命令到 GPU，并且让 CPU 执行和 GPU 命令处理重叠进行，以提升性能。



注意 即使在 CUDA 1.0 中，内核启动也是异步的。对任意的内核，在没有显式指定流的情况下，NULL 流被隐式指定。

3.7.4 事件

CUDA 事件表示了另一个同步机制。同 CUDA 流一同引入，记录 CUDA 事件是 CUDA 流中应用程序跟踪进度的一个方式。当之前的所有的 CUDA 流的操作执行结束后，全部的 CUDA 事件通过写入一个共享同步内存位置而起作用[⊖]。查询 CUDA 事件会引发驱动程序窥探这一内存位置，报告事件是否被记录。同步 CUDA 事件会引发驱动程序等待，直到事件被记录。

可选择的是，CUDA 事件同样可以写入一个从硬件高分辨率计时器中派生出的时间戳。基于事件的计时可以比基于 CPU 的计时更加准确，尤其对小型的操作，因为这不会受限于伪非相关事件（例如页错误或网络流量），但这类事件会影响 CPU 系统时钟计时。系统时钟时间是无法更改的，因为它是最终用户看到的时间的更好近似，因此 CUDA 事件计时最好使用在生产环境中的性能调优中[⊖]。

CUDA 事件计时最好与 NULL 流一同使用。使用这一规则的原因类似于 CPU 上的串行指令 RDTSC（读时间戳计数器）：正像 CPU 是一个超标量处理器，可以同时处理许多指令，

⊖ 指定 NULL 流给 cuEventRecord() 或 cudaEventRecord() 函数，意味着事件将会在 GPU 处理完所有待处理操作后被记录。
 ⊖ 此外，用于计时的 CUDA 事件不能在其他操作中使用，最近的 CUDA 版本允许“禁用计时”特性，便于在其他操作中使用 CUDA 事件，例如跨设备同步。

GPU 也可在多个流里同时运行。在没有显式序列化的情况下，一个计时操作可能无意的包含了本不想被计时的操作，也可能排除了一些应该被包含进的操作。正如 RDTSC 经常使用的技巧，我们为计时提供足够的工作，所以计时本身的执行时间可以忽略。

随机的 CUDA 事件可能会引发一次中断，硬件会为中断发出信号，使驱动程序执行一次所谓的阻塞等待。当驱动程序等待 GPU 的同时阻塞等待会挂起等待的 CPU 线程，节省 CPU 时钟周期和功率。在阻塞等待机制可用之前，CUDA 开发者普遍的抱怨 CUDA 驱动程序通过轮询内存位置让整个 CPU 核心空转来等待 GPU。但与此同时，阻塞等待可能由于处理中断的额外开销将消耗更多的时间，所以很多应用程序可能还是希望使用默认的轮询行为。

3.8 主机内存

主机内存就是 CPU 内存——那个在我们听说 CUDA 之前使用 `malloc()/free()` 和 `new[]/delete[]` 函数操作了好多年的东西。在所有运行 CUDA 的操作系统上，主机内存是虚拟化的。硬件实施的内存保护使进程在没有特殊规定的情况下无法互相读写对方的内存[⊖]。内存中的“页”，通常为 4KB 或 8KB 大小，可在不改变它们虚拟地址的情况下被重定位。特别的是，它们可被交换到磁盘上，有效地使电脑中有比物理内存更多的虚拟内存。当一个页被标记为“非驻留的”，一次访问这个页的操作会发送给操作系统“页错误”的信号，这会提示操作系统寻找一块可用的内存，复制磁盘上的数据到内存中，使虚拟的内存页指向新的物理位置从而恢复请求操作。

操作系统中管理虚拟内存的部件叫做“虚拟内存管理器”或 VMM。除了别的之外，VMM 监视内存活动并使用启发式方法决定什么时候从磁盘驱换出一个页，以及在换出页发生引用时解决“缺页故障”。

VMM 作为提供给硬件驱动的服务，使硬件直接访问主机内存变得容易。在现代计算机中，许多外设，包括控制器、网络控制器和 GPU，可以通过 DMA 读写主机内存。DMA 的前两个优点：它避免了数据复制[⊖]并且使硬件可以与 CPU 并发操作。第三个优点是硬件通过 DMA 可以得到更好的总线表现。

为了便利 DMA，操作系统 VMM 提供一个页面锁定（page-locking）功能。被 VMM 标记为锁页的内存就不能被换出了，所以物理内存地址不能被修改。一旦内存被锁页，驱动程序便可以使用 DMA 硬件引用内存的物理地址。这一硬件设置是与锁页操作本身独立且不同的操作。因为页面锁定使其低层物理地址不再为其他分配所用，太多的锁页反而会影响计算机性能。

没被锁页的内存称作可换页的（pageable）。锁页后的内存被称作锁页内存（pinned），由

[⊖] 实现进程间共享的 API 包括 Windows 的 `MapViewOfFile()` 或 Linux 的 `mmap()`。

[⊖] 这一额外复制对使用 GPU 的开发者来说很显眼，它会比磁盘或网络控制器这样的设备占用更多的带宽。

无论设备是什么类型的，没有 DMA，驱动程序就必须使用 CPU 或特殊的硬件缓冲区来复制数据。

于其物理内存不能被操作系统改变（它被固定在那个位置）。

3.8.1 锁页主机内存

锁页主机内存由 CUDA 使用函数 `cuMemHostAlloc()` / `cudaHostAlloc()` 分配。该内存是页锁定的并且由当前 CUDA 上下文为 DMA 设置[⊖]。

CUDA 跟踪分配的内存范围并且自动加速引用锁页内存的内存复制操作。异步内存复制操作只在锁页内存上工作。应用程序可以通过 `cuMemHostGetFlags()` 函数判断一个给定的主机内存地址范围是否被锁定。

在操作系统文档中，页面锁定（page-locked）和锁页（pinned）是同义的。但是对 CUDA，锁页内存（pinned memory）是经过页锁定后且为硬件访问而映射后的主机内存，而页面锁定（page-locking）只是指操作系统的一种不允许换页的机制。

3.8.2 可分享的锁页内存

可分享锁页内存页锁定后映射给所有的 CUDA 上下文。这一操作的底层机制十分复杂：当一个可分享锁页内存分配操作执行，它会在返回前映射到所有的 CUDA 上下文。此外，无论何时 CUDA 上下文被创建，所有的可分享锁页内存分配都会被映射到新的 CUDA 上下文中。无论是对可分享内存分配还是上下文创建，任何映射的失败都会引发内存分配错误或上下文创建错误。可喜的是，在 CUDA 4.0 中，如果 UVA 有效，所有的锁页内存分配全部可分享。

3.8.3 映射锁页内存

映射锁页内存映射到 CUDA 上下文中的地址空间中，所以内核可能读写这块内存。默认的，锁页内存不被映射到 CUDA 地址空间，所以它不能被内核的不合法写入毁坏。对集成 GPU，映射的锁页内存带来“零复制”：由于主机（CPU）和设备（GPU）共享一块内存池，它们可以不用显式复制来交换数据。

对于独立 GPU，映射锁页内存使主机内存可被内核直接读写。对于小型数据，这一特点可以消除显式内存复制命令的花销。映射的锁页内存存在写入方面尤其有优势，因为这不会有延时存在。在 CUDA 4.0 中，如果 UVA（统一虚拟编址）有效，所有的锁页分配都会被映射。

3.8.4 主机内存注册

由于开发者（尤其是库开发者）有时无法分配他们想要访问的内存，CUDA 4.0 添加了注册已存在的虚拟内存地址范围的能力。`cuMemHostRegister()`/`cudaHostRegister()` 函数

[⊖] CUDA 开发者经常会有疑问：页锁定内存与 CUDA 的锁页内存是否有区别？答案是：有！锁页内存由 CUDA 进行分配或注册，进行映射之后可为 GPU 直接访问，普通的页锁定内存则不是。

会接受虚拟内存范围，之后将页面锁定，并为当前 GPU（或为所有的 GPU，如果 CU_MEMHOSTREGISTER_PORTABLE 或 cudaHostRegisterPortable 被指定）映射它。主机内存注册与 UVA 有着反常关系，由于任何的适合注册的地址范围必须不包含在当 CUDA 驱动程序被初始化时为 UVA 保留的虚拟地址空间里。

3.9 CUDA 数组与纹理操作

CUDA 数组与设备内存从相同的物理内存池中分配，但是前者拥有一个细节不明的布局：为 2D 和 3D 局部性做了优化。图形驱动程序使用这些布局保存纹理，从地址中分离出数组索引，硬件可以在 2D 或 3D 元素块上操作，取代 1D 寻址。对展示出稀疏访问模式的应用程序，特别是有维度局部性的程序（例如，计算机视觉领域的应用程序），CUDA 数组无疑会胜出；对拥有常规访问模式的应用程序，尤其是那些没有重用或那些重用可以被应用程序在共享内存上显式管理的，设备内存指针是显而易见的选择。

有些应用程序，像图像处理程序，会处在选择设备指针还是 CUDA 数组的两难处境中。如果所有的因素都相同，设备内存会是更好的选择，但是如下的几点考虑可能会对选择产生影响：

- CUDA 数组不消耗 CUDA 地址空间。
- 在 WDDM 驱动程序中，系统可以自动管理 CUDA 数组的驻留。
- CUDA 可以只在设备内存上驻留，GPU 在总线上传输数据时在这两种形式之间转化。
对于某些应用程序，在主机内存上保持一个等宽形式内存，在设备内存上建立 CUDA 数组是最好的处理办法。

3.9.1 纹理引用

纹理引用是 CUDA 用来设置纹理硬件解释实际内存内容的对象[⊖]。这个间接层存在的一部分原因是：这会使多个纹理引用使用不同的属性引用相同的内存变得有效。

纹理引用的属性可能是不可变的，也可能是可变的。前者在编译时间指定，不会导致应用程序表现的不正确或不定，后者中的应用程序以编译器不可见的方式改变纹理行为（见表 3-5）。例如，纹理维数（1D、2D 和 3D）是不可变的，这必须由编译器预先获取并使用正确数量的输入参数，发出正确的机器指令。相对之下，滤波与寻址模式是可变的，它们隐式改变应用程序的行为，而编译器毫不知情。

表 3-5 可变的与不可变纹理属性

不可变属性	可变属性	不可变属性	可变属性
维度	滤波模式	返回类型	标准化坐标
类型（格式化）	寻址模式		色彩空间（sRGB）转换

[⊖] 在 CUDA 3.2 前，纹理引用是除了显式的内存复制之外读取 CUDA 数组的唯一方式。

CUDA 运行时（语言集成特性）和 CUDA 驱动程序 API 处理纹理引用的行为十分不同。对双方来说，纹理引用都需要调用一个名为 texture 的模板来声明：

```
texture<Type, Dimension, ReadMode> Name;
```

Type 指的是纹理所要在内存中读取的对象类型，Dimension 是纹理的维度（1、2 或 3），ReadMode 指定了在纹理读取时整数纹理类型是否转化为归一化浮点数。

纹理引用必须在使用前绑定在实际内存上。硬件从 CUDA 数组绑定纹理会更有优势，但在下述情况下，应用程序可以从设备内存绑定纹理中获利：

- 作为带宽聚合器支持纹理缓存。
- 使应用程序满足合并访问限制。
- 对那些最好通过设备内存写入的应用，可以避免从内存读取时的多余副本。例如，一个视频解码器或许希望发送给设备内存一个帧，而通过纹理方式读取它们。

一旦纹理引用与实际内存绑定，CUDA 内核便可以调用 tex* 内建函数读取内存，例如表 3-6 中的 tex1D() 函数。

表 3-6 纹理内建函数

纹理类型	内建函数
设备内存	<code>tex1Dfetch(int)</code>
1D	<code>tex1D(float x, float y)</code>
2D	<code>tex2D(float x, float y)</code>
3D	<code>tex3D(float x, float y, float z)</code>
立方图纹理	<code>texCubemap(float x, float y, float z)</code>
分层纹理（1D）	<code>tex1DLayered(float x, int layer)</code>
分层纹理（2D）	<code>tex2DLayered(float x, float y, int layer)</code>
分层纹理（立方图）	<code>texCubemapLayered(float x, float y, float z, int layer)</code>

 **注意** 在这里对通过全局加载 / 存储或表面加载 / 存储执行的纹理读和写之间没有一致性保证。因此，CUDA 内核当心不要从那些可以使用其他方式访问的内存读取纹理。

1. CUDA 运行时

为了绑定内存到纹理上，应用程序必须调用表 3-7 中的一个函数。CUDA 运行时应用程序可以通过直接指定结构体成员修改可变的纹理引用属性。

```
texture<float, 1, cudaReadModeElementType> tex1;
...
tex1.filterMode = cudaFilterModeLinear; // enable linear filtering
tex1.normalized = true; // texture coordinates will be normalized
```

表 3-7 绑定设备内存到纹理的函数

内 存	函 数
1D 设备内存	<code>cudaBindTexture()</code>
2D 设备内存	<code>cudaBindTexture2D()</code>
CUDA 数组	<code>cudaBindTextureToArray()</code>

指定这些结构体成员会立即生效，且不用重新绑定。

2. 驱动程序 API

由于在使用驱动程序 API 时，CPU 代码和 GPU 代码之间存在严格的分区，任何 CUDA 模块中声明的纹理引用都必须通过 `cuModuleGetTexRef()` 查询，这个函数传回一个 CUtexref。不同于 CUDA 运行时，纹理引用必须使用所有正确的属性初始化——可变的和不变的——因为编译器不会编码纹理引用的不可变属性到 CUDA 模块中。表 3-8 总结了驱动程序 API 函数——在绑定纹理引用与内存中使用。

表 3-8 绑定内存到纹理的驱动程序 API 函数

内 存	函 数
1D 设备内存	<code>cuTexRefSetAddress()</code>
2D 设备内存	<code>cuTexRefSetAddress2D()</code>
CUDA 数组	<code>cuTexRefSetArray()</code>

3.9.2 表面引用

表面引用可以使 CUDA 内核使用表面加载 / 存储内建函数读写 CUDA 数组。它是最近添加到 CUDA 中的，在特斯拉架构 GPU 上不可用。设置表面引用的初衷是使 CUDA 内核可以直接写入数组。在表面加载 / 存储函数实现之前，内核必须先写入设备内存，再执行一次从设备到数组的内存复制并转化输出为 CUDA 数组。

纹理引用与表面引用对比明显。前者可以把输入的任何坐标转化到设置的输出格式，后者则暴露了一个常规的逐位操作接口来读写 CUDA 数组内容。

你可能会想，为什么 CUDA 没有实现表面加载 / 存储内建函数来直接在 CUDA 数组上操作（像 OpenGL 一样）。原因是想要表面操作与时俱进，能够以更加高级的方式使图形加载 / 存储操作转化为底层表达，例如将采样使用分数坐标放入 CUDA 数组，或与抗锯齿图形表面互操作。目前，CUDA 开发者将需要通过软件来完成这些操作。

3.10 图形互操作性

图形互操作性函数族使得 CUDA 可以读写属于 OpenGL 或 Direct3D API 的内存。如果

应用程序可以通过主机内存共享数据获取可以接受的性能，这些 API 就没有存在的必要了。但是本地内存的带宽可以高达 140G/s，而事实上 PCIe 总线带宽却很少能达到 6G/s，在可能的情况下让应用程序保持数据在 GPU 上是十分重要的。使用这些图形交互操作 API，CUDA 内核可以写入数据到图像和纹理中，随后合并到由 OpenGL 和 Direct 3D 执行的图形输出。

因为图形和 CUDA 应用程序必须在低层协调以开启互操作性。应用程序必须发出信号表明它们的意图来尽早执行图形交互操作。特别是，必须通过调用特殊的上下文创建 API，例如 cuD3D10CtxCreate() 或 cudaGLSetDevice()，通知 CUDA 上下文，让它知道会与给定的 API 交互。

在驱动程序之间的协调同样会促成在图形 API 和 CUDA 之间的资源共享，按以下的两个步骤进行：

1) 注册：是一个成本高昂的底层操作，该操作把开发者需要共享资源的意图告诉给底层驱动程序，可能会引发所需要资源的移动或锁定。

2) 映射：一个预期会频繁出现的轻量级操作。

在 CUDA 的早期版本中，图形互操作 API 按四种不同的图形 API (OpenGL、Direct3D 9、Direct3D 10 和 Direct3D 11) 严格的分离开。例如，对 Direct3D 9 互操作中，下列的函数可以配合使用：

- cuD3D9RegisterResource() / cudaD3D9RegisterResource()
- cuD3D9MapResources() / cudaD3D9MapResources()
- cuD3D9UnmapResources() / cudaD3D9UnmapResources()
- cuD3D9UnregisterResource() / cudaD3D9UnregisterResource()

因为实际的硬件能力是相同的，忽略访问它们的 API，许多函数被在 CUDA 3.2 中予以合并。注册函数仍保持着 API 分离性，因为它们在绑定时需要区分 API。但映射、解映射和移除注册资源这些函数变为公用的了，对应上述函数的 CUDA 3.2 API 如下：

- cuD3D9RegisterResource() / cudaD3D9RegisterResource()
- cuGraphicsMapResources() / cudaGraphicsMapResources()
- cuGraphicsUnmapResources() / cudaGraphicsUnmapResources()
- cuGraphicsUnregisterResource() / cudaGraphicsUnregisterResource()

Direct3D 10 中的互操作 API 与上述函数全部类同，唯一不同的是开发者必须使用 cuD3D10RegisterResource() / cudaD3D10RegisterResource() 代替 cuD3D9* 形式。

CUDA 3.2 同样增加了使用图形 API 访问 CUDA 数组形式的纹理功能。在 Direct 3D 中，纹理只是资源的一种，可以被 IDirect3DResource9 * (或 IDirect3DResource10 *，等) 引用。而在 OpenGL 中，提供了一个独立的函数：cuGraphicsGLRegisterImage()。

3.11 CUDA 运行时与 CUDA 驱动程序 API

CUDA 运行时提供了方便的语言集功能，可以使程序很轻松的从零开始。通过自动进行像初始化上下文或加载模块这样的任务，并且用其他的 C++ 代码以内联方式启动内核。CUDART 使开发者专注于怎样使代码工作的更快。少数的 CUDA 抽象，像 CUDA 模块，在 CUDART 中并不可用。

另一方面，驱动程序 API 暴露了所有的 CUDA 抽象，使开发者可以在应用程序中操作它们。驱动程序 API 没有提供任何性能优势。但它为应用程序提供了显式的资源管理，这在需要资源管理的应用程序中十分重要，例如拥有插件架构的大规模商业应用。

驱动程序 API 的运行速度并不比 CUDA 运行时显著提升，如果你正在寻找改进 CUDA 应用程序性能的方法，请到其他章节去找。

CUDA 的大部分特性对 CUDA 运行时和驱动程序 API 都可使用，只有几个例外。表 3-9 给出了详细信息：

表 3-9 CUDA 运行时 vs 驱动程序 API

特 性	CUDART	驱动程序 API
设备内存分配	*	*
锁页主机内存分配	*	*
内存复制	*	*
CUDA 流	*	*
CUDA 事件	*	*
图形互用性	*	*
纹理支持	*	*
表面支持	*	*
cuMemGetAddressRange		*
语言集成	*	
胖二进制 (Fat binary)	*	
显式 JIT 选项		*
简化内核调用	*	
显式上下文与模块管理		*
上下文迁移		*
16 位浮点纹理		*
16 位与 32 位内存赋值		*
独立编译		*

在两种 API 之间，像内存复制这样的操作趋向于功能性相同，但是接口却全然不同。但流 API 是几乎相同的。

CUDART	驱动程序 API
cudaStream_t stream;	CUstream stream;
cudaError_t status = cudaStreamCreate(&stream);	CUresult status = cuStreamCreate(&stream, 0);
...	...
status = cudaStreamSynchronize(stream);	status = cuStreamSynchronize(stream);

事件 API 有一些细微的区别，如果开发者想要指定一个标志词（创建一个阻塞事件时需要），CUDART 提供了一个独立的 `cudaEventCreateWithFlags()` 函数。

CUDART	驱动程序 API
cudaEvent_t eventPolling;	CUevent eventPolling;
cudaEvent_t eventBlocking;	CUevent eventBlocking;
cudaError_t status = cudaEventCreate(&eventPolling);	CUresult status = cuEventCreate(&eventPolling, 0);
cudaError_t status = cudaEventCreateWithFlags(&eventBlocking, cudaEventBlockingSync);	CUresult status = cuEventCreate(&eventBlocking, CU_EVENT_BLOCKING_SYNC);
...	...
status = cudaEventSynchronize(event);	status = cuEventSynchronize(event);

内存复制函数的接口大部分都是不同的，尽管它们的实际功能是相同的。CUDA 支持 3 种类型的内存——主机内存、设备内存、CUDA 数组——内存复制发生在它们的全排列间，并可以发生在 1D、2D 或 3D 三种结构上。所以内存复制函数要么包括大量的不同函数，要么包括小数量却能支持不同内存类型的函数。

CUDA 中的最简单的内存复制是主机与设备内存间的复制，但是这些函数接口仍然大不相同：CUDART 使用 `void *` 既代表主机指针又能表示设备指针，并且一个内存复制函数接受一个目标参数；而驱动程序 API 使用 `void *` 代表主机内存，`CUdeviceptr` 代表设备内存，3 个独立的函数 (`cuMemcpyHtoD()`、`cuMemcpyDtoH()` 和 `cuMemcpyDtoD()`) 对应不同的复制目标。下表给出 CUDART 和驱动程序 API 主机与内存间的相关参数列表：

CUDART	驱动程序 API
void *dptr; void *hptr; void *dptr2; status = cudaMemcpy(dptr, hptr, size, cudaMemcpyHostToDevice);	CUdeviceptr dptr; void *hptr; CUdeviceptr dptr2; status = cuMemcpyHtoD(dptr, hptr, size);
status = cudaMemcpy(hptr, dptr, size, cudaMemcpyDeviceToHost);	status = cuMemcpyDtoH(hptr, dptr, size);
status = cudaMemcpy(dptr, dptr2, size, cudaMemcpyDeviceToDevice);	status = cuMemcpyDtoD(dptr, dptr2, size);

对于 2D 和 3D 内存复制，驱动程序 API 实现了一批接受一个描述结构体并支持所有类型的内存复制的函数，这也支持低维内存复制。例如，如果需要，`cuMemcpy3D()` 可以代替

cuMemcpyHtoD() 执行 1D 主机设备间的内存复制：

```
CUDA_MEMCPY3D cp = {0};
cp.dstMemoryType = CU_MEMORYTYPE_DEVICE;
cp.dstDevice = dptr;
cp.srcMemoryType = CU_MEMORYTYPE_HOST;
cp.srcHost = host;
cp.WidthInBytes = bytes;
cp.Height = cp.Depth = 1;
status = cuMemcpy3D( &cp );
```

CUDART 使用描述结构体的组合来描述复杂的内存复制函数（例如，cudaMemcpy3D()），同时使用不同的函数实现不同内存类型的复制。如同 cuMemcpy3D() 函数，CUDART 中的 cudaMemcpy3D() 函数可以接受一个描述结构体来描述内存复制的双方，也包括跨维度内存复制（执行 1D 与 2D CUDA 数组中的一列之间的复制，或执行 2D CUDA 数组与 3D CUDA 数组一个切片之间的复制）。它的描述结构仅有微小改变，这个改变在于另外嵌入了其他结构体。表 3-10 给出了两个 API 的 3D 复制结构体的对照表。

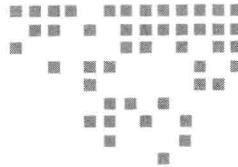
表 3-10 3D 内存复制结构体

<pre>struct cudaMemcpy3DParms { struct cudaArray *srcArray; struct cudaPos srcPos; struct cudaPitchedPtr srcPtr; struct cudaArray *dstArray; struct cudaPos dstPos; struct cudaPitchedPtr dstPtr; struct cudaExtent extent; enum cudaMemcpyKind kind; }; struct cudaPos { size_t x; size_t y; size_t z; }; struct cudaPitchedPtr { void *ptr; size_t pitch; size_t xsize; size_t ysize; }; struct cudaExtent { size_t width; size_t height; size_t depth; };</pre>	<pre>typedef struct CUDA_MEMCPY3D_st { size_t srcXInBytes; size_t srcY; size_t srcZ; size_t srcLOD; CUmemorytype srcMemoryType; const void *srcHost; CUdeviceptr srcDevice; CUarray srcArray; void *reserved0; size_t srcPitch; size_t srcHeight; size_t dstXInBytes; size_t dstY; size_t dstZ; size_t dstLOD; CUmemorytype dstMemoryType; void *dstHost; CUdeviceptr dstDevice; CUarray dstArray; void *reserved1; size_t dstPitch; size_t dstHeight; size_t WidthInBytes; size_t Height; size_t Depth; } CUDA_MEMCPY3D;</pre>
---	---

两者 3D 内存复制的用法是相似的。它们均被设计为零初始化，并且开发者需要对给定的操作设置需要的成员。例如，执行主机到 3D 数组间的复制可以按下列步骤执行：

struct cudaMemcpy3DParms cp = {0}; cp.srcPtr.ptr = host; cp.srcPtr.pitch = pitch; cp.dstArray = hArray; cp.extent.width = Width; cp.extent.height = Height; cp.extent.depth = Depth; cp.kind = cudaMemcpyHostToDevice; status = cudaMemcpy3D(&cp);	CUDA_MEMCPY3D cp = {0}; cp.srcMemoryType = CU_MEMORYTYPE_HOST; cp.srcHost = host; cp.srcPitch = pitch; cp.srcHeight = Height; cp.dstMemoryType = CU_MEMORYTYPE_ARRAY; cp.dstArray = hArray; cp.WidthInBytes = Width; cp.Height = Height; cp.Depth = Depth; status = cuMemcpy3D(&cp);
--	--

对于一个覆盖整个 CUDA 数组的 3D 复制，源和目标地址偏移量在第一行被设置为 0，且只被引用一次。不同于函数的参数，代码只需要引用复制操作需要的参数，而且如果程序执行超过一个相同的操作（例如填充不止一个设备内存区或 CUDA 数组），这个描述结构体可被复用。



软件环境

本章简要介绍 CUDA 开发工具和支持 CUDA 应用程序的软件环境。前四节专门介绍英伟达工具箱中的各种工具。

- nvcc：CUDA 编译器驱动程序
- ptxas：PTX 汇编工具
- cuobjdump：CUDA 目标文件转储工具
- nvidia-smi：英伟达系统管理接口

4.5 节介绍亚马逊的弹性计算云（Elastic Compute Cloud，EC2）服务，以及如何用它通过互联网来访问支持 GPU 的服务器。本章的目的是为了给大家一个参考，而不是教程指导。本书的第三部分中会给出使用范例。

4.1 nvcc——CUDA 编译器驱动程序

nvcc 是 CUDA 开发者用来把源代码翻译为具有一定功能的 CUDA 应用程序的编译器驱动程序。它可以完成许多功能，从仅对 .cu 文件中的纯 GPU 代码进行编译到在单条命令中（本书中许多示例程序所提倡的一种用法）执行样例程序的编译、链接并运行的一条龙服务。

作为一个编译器驱动程序，nvcc 所做的也就是搭建起生成环境，产生一组本地工具（例如安装在系统里的 C 语言编译器）以及利用 CUDA 特定的命令行工具（如 ptxas）生成 CUDA 代码。它包含了许多默认选项（这些默认选项被认为是用户在多数情况下会使用的），可以进一步被命令行选项改写。其确切的行为取决于主命令行选项指定的“编译路径”（compile trajectory）。

表 4-1 列出了 nvcc 支持的文件扩展名以及为它们实现的默认行为（注意，这里略去了一

些中间文件类型，如包含 CUDA 前台生成的主机代码的 .i/.ii 文件类型。)。表 4-2 列出了编译阶段的选项以及相应的编译路径。表 4-3 列出了 nvcc 影响系统环境的选项，如指向所包括目录的路径。表 4-4 列出了 nvcc 影响输出的选项，比如是否包含调试信息。表 4-5 列出了“传递”的选项，这些选项使 nvcc 可以把选项传递给它调用的工具，如 ptxas。表 4-6 列出了 nvcc 其他一些不易归类的选项，如指示 nvcc 不要删除它创建的临时文件的 -keep 选项。

表 4-1 nvcc 输入文件的扩展名

文件扩展名	默认行为
.c/.cc/.cpp/.cxx	预处理、编译、链接
.cu	把主机代码和设备代码分开，分别编译它们
.o(bj)	链接
.ptx	把 PTX 组合成 cubin

表 4-2 编译路径

选 项	路 径
--cuda	把所有的 .cu 输入文件编译为 .cu/.cpp/.ii 输出文件
--cubin	把所有的 .cu/.ptx/.gpu 文件编译为 .cubin 文件
--fatbin	把所有的 .cu/.ptx/.gpu 文件编译为 PTX 或仅适用于由 --arch 或 --code 指定的设备的二进制文件，并把结果输出到被 -o 选项指定的胖二进制文件中
--ptx	把所有的 .cu/.gpu 文件编译为仅适用于设备的 .ptx 文件 ^①
--gpu	把所有的 .cu 文件编译为仅适用于设备的 .gpu 文件
--preprocess (-E)	预处理 .c/.cc/.cpp/.cxx/.cu 输入文件
--generate-dependencies (-M)	为一个 .c/.cc/.cpp/.cxx/.cu 输入文件 (在此模式下不允许多个输入文件) 生成可以包含在 makefile 文件中的依赖项文件
--compile (-c)	把每个 .c/.cc/.cpp/.cxx/.cu 的输入文件编译成目标文件
--link (-link)	编译和链接所有的输入文件 (默认的处理方式)
--lib (-lib)	(如有必要) 把所有的输入文件编译成目标文件，并添加结果到指定的输出库文件
--x (-x)	明确指定输入文件的语言，而不是让编译器基于文件的后缀名来选择默认语言。允许的语言有 C, C++, cu
--run (-run)	编译并链接所有输入文件为一个可执行程序，然后运行它。如果输入可执行，则直接运行，而不再需要编译或链接

① 这些命令行选项会丢弃输入文件里的主机代码。

表 4-3 nvcc 选择项 (环境)

选 项	描 述
--output-file <file> (-o)	指定输出文件的名称和位置。当此选项在 nvcc 非链接 / 归档模式下时，只允许一个输入文件
--pre-include <include-file> (-include)	在预处理期间，必须包含指定的头文件

(续)

选 项	描 述
--library <library> (-l)	指定链接阶段使用的库。这些库在使用 -L 指定的库搜索路径下进行搜索
--define-macro <macrodef> (-D)	指定在预处理或编译过程中使用的宏定义
--undefine-macro (-U)	指定在预处理或编译过程中取消的宏定义
--include-path <include-path> (-I)	指定所包含的搜索路径
--system-include <include-path> -isystem	指定系统包含的搜索路径
--library-path (-L)	指定库搜索路径
--output-directory (-odir)	指定输出文件的目录。这个选项是为了使依赖生成步骤 (-generatedependencies) 形成一个规则，这个规则会在合适的目录中定义目标文件
--compiler-bindir <path> (--ccbin)	指定编译器可执行文件（微软 Visual Studio 的 CL 或者 gcc 的派生）所在的目录。默认情况下，这个可执行文件通常在当前可执行文件的搜索路径下。对一个不同的编译器或者为了给这些编译器指定一个不同的可执行文件名，要向编译器指定包含可执行文件名称的路径
--cl-version <cl-version-number> (-cl-version)	指定 Microsoft Visual Studio 的安装版本。此选项允许的版本：2005、2008 和 2010。如果指定了 --use-local-env，这个选项是必需的

表 4-4 编译器或链接器指定行为的选项

选 项	描 述
--profile (-pg)	为 gprof 使用而生成的代码 / 可执行文件（仅限 Linux）
--debug (-g)	生成主机代码的调试信息
--device-debug<level> (-G)	为设备代码生成调试信息，还可以为设备代码指定优化级别（0 ~ 3），以控制其“可调试性”
--optimize <level> -O	指定主机代码的优化级别
--shared -shared	在链接过程中，生成一个共享库
--machine <bits> -m	指定 32 位或 64 位架构，此选项允许的值为：32 和 64

表 4-5 nvcc 传递选项

选 项	描 述
--compiler-options <options> (-Xcompiler)	直接指定编译器 / 预处理器的选项

(续)

选 项	描 述
--linker-options <options> -Xlinker	直接指定链接器的选项
--archive-options <options> (-Xarchive)	直接指定库管理器的选项
--cudafe-options <options> -Xcudafe	直接指定 cudafe 的选项
--ptx-options <options> -Xptxas	直接指定 PTX 优化汇编的选项

表 4-6 其余的 nvcc 选项

选 项	描 述
--dont-use-profile (-noprof)	不要使用 nvcc.profiles 的文件指导编译
--dryrun (-dryrun)	禁止编译命令的执行
--verbose (-v)	列出由 nvcc 生成的命令
--keep (-keep)	保留内部编译过程中产生的中间文件
--keep-dir (-keep-dir)	指定 --keep 所指定文件应被写入的目录
--save-temp (-save-temp)	(同 --keep)
--clean-targets (-clean)	与 nvcc 创建文件的过程相反，使 nvcc 删除所有的非临时文件
--run-args <arguments> (-run-args)	如果指定了 --run，此选项指定传递给可执行文件的命令行参数
--input-drive-prefix <prefix> (-idp)	对 Windows 而言，指定输入文件所在的绝对路径。对于 Cygwin 用户而言，指定 “-idp /cygwin/”；对于 Mingw 用户而言，指定 “-idp /”
--dependency-drive-prefix <prefix> (-ddp)	对 Windows 而言，当产生依赖文件时 (--generate-dependencies)，指定绝对路径的前缀。对于 Cygwin 用户而言，指定 “-idp /cygwin/”；对于 Mingw 用户而言，指定 “-idp /”
--drive-prefix <prefix> (-dp)	指定用于输入文件和依赖文件的前缀
--no-align-double	指定在 32 位平台上 -malign-double 不应该作为编译器参数传递。注意，对于某些 64 位类型的平台，该选项使得应用程序二进制接口 (ABI) 不兼容 CUDA 的内核 ABI

表 4-7 列出了与 nvcc 代码生成有关的选项。--gpu-architecture 和 --gpu-code 选项是特别让人迷惑的。前者控制面向哪个虚拟 GPU 架构进行编译（即，产生哪个 PTX 版本），而后者控制面向哪个实际 GPU 架构进行编译（即，产生哪个 SM 微码版本）。--gpu-code 选项必须指定至少与 --gpu-architecture 中一样高的 SM 版本。

表 4-7 nvcc 代码生成选项

选 项	描 述
--gpu-architecture <gpu architecture name> -arch	指定编译到哪种虚拟英伟达 GPU 架构。此选项指定 PTX 版本。有效选项包括: compute_10、compute_11、compute_12、compute_13、compute_20、compute_30、compute_35、sm_10、sm_11、sm_12、sm_13、sm_20、sm_21、sm_30 和 sm_35
--gpu-code <gpu architecture name> -code	指定编译到哪种实际 GPU 架构。此选项指定编译的流处理器簇版本。如果未指定,这个选项应该是与 --gpu-architecture 指定 PTX 版本对应的流处理器簇版本。有效选项包括: compute_10、compute_11、compute_12、compute_13、compute_20、compute_30、compute_35、sm_10、sm_11、sm_12、sm_13、sm_20、sm_21、sm_30 和 sm_35
--generate-code (-genocode)	以元组方式指定所面向的虚拟的和实际的 GPU 架构。--generate-code arch=<arch>,code=<code> 相当于 --gpu-architecture <arch> --gpu-code <code>
--export-dir	指定所有设备代码的映像将被复制到的目录名称
--maxregcount <N> (-maxregcount)	指定 GPU 函数可以使用的最大寄存器数量
--ftz [true, false] (-ftz)	清洗为 0 模式: 当执行单精度浮点运算时, 非规格化数字 (denormal) 置为零。--use-fast-math 意味着 --ftz=true, 默认值是 false
--prec-div [true, false] (-prec-div)	精确除法: 如果值是 true, 单精度浮点除法和倒数进行全精度运算 (就近舍入使用无误差的最小精度单位)。--use-fast-math 意味着 --prec-div=false, 默认值为 true
--prec-sqrt [true, false]	精确的平方根: 如果值是 true, 单精度浮点平方根进行全精度运算 (就近舍入使用无误差的最小精度单位)。--use-fast-math 意味着 --prec-sqrt=false, 默认值为 true
--fmad [true, false]	启用或禁用浮点乘法和加法 / 减法的连写式转化为浮点乘加指令 (FMAD)。仅当 --gpu-architecture 是 compute_20、sm_20 或更高时, 这个选项才可用。对于其他的架构, 总是自动转化的。--use-fast-math 意味着 --fmad=true
--use-fast-math	使用快速数学库。除了意味着 --prec-div 为 false、--prec-sqrt 为 false、--fmad 为 true 外, 单精度运行时数学函数直接编译为 SFU 内建函数

--export-dir 选项指定所有设备的代码映像将要复制的目标目录。它的目的是在应用程序运行过程中(在这种情况下, 该目录应该在 CUDA_DEVCODE_PATH 的环境变量中)作为一个 CUDA 驱动程序可以检查的设备代码的存放地。存放地可以是目录也可以是 ZIP 文件。在这两种情况下, 为了方便 CUDA 驱动器查找代码, CUDA 将维护一个目录结构。如果指定了一个不存在的文件名, 那么该位置将创建一个目录结构(不是一个 ZIP 文件)。

4.2 ptxas——PTX 汇编工具

ptxas 是把 PTX 编译成指定的 GPU 微码的工具。它在 CUDA 生态系统中占有独特的地位, 英伟达不仅把它放到了离线工具中(开发者用来编译应用程序), 还把它作为驱动程序的一部分, 支持所谓的“在线编译”或者“即时编译”。

在离线编译时，如果 `--gpu-code` 命令行选项已经指定了实际的 GPU 架构，`ptxas` 一般由 `nvcc` 调用。在这种情况下，命令行选项（已总结到表 4-8 中）会通过传送给 `nvcc` 的 `-Xptxas` 参数传递给 `ptxas`。

表 4-8 `ptxas` 的命令行选项

选 项	描 述
<code>--abi-compile <yes no></code> <code>(-abi)</code>	启用或禁用使用应用程序二进制接口 (ABI) 编译函数。默认设置为 yes
<code>--allow-expensive-optimizations</code> <code><true false></code> <code>(-allow-expensive-optimizations)</code>	启用或禁用昂贵的编译时优化。编译时优化将使用最大可用的资源 (内存和编译时间)。如果未指定，在优化级别 $\geq O2$ 时，默认行为是启用此功能
<code>--compile-only</code> <code>(-c)</code>	生成一个可重定位的目标文件
<code>--def-load-cache [ca cg cs lu cv]</code> <code>-dlcm</code>	默认基于全局负载的缓存修饰符。默认值：ca
<code>--device-debug</code> <code>(-g)</code>	为设备代码生成调试信息
<code>--device-function-maxregcount</code> <code><archmax/archmin/N></code> <code>(-func-maxregcount)</code>	当使用 <code>--compile-only</code> 编译时，指定设备函数可以使用的最大寄存器数量。这个选项是为全程序编译所忽略的，并且这个选项不影响入口函数所使用的寄存器数量。对于设备函数，此选项将改写由 <code>-maxrregcount</code> 指定的值。如果 <code>--device-function-maxregcount</code> 与 <code>--maxrregcount</code> 都没有被指定，那么就可以假定没有最大值
<code>--dont-merge-basicblocks</code> <code>(-no-bb-merge)</code>	通常情况下，作为其优化过程的一部分， <code>ptxas</code> 尝试合并连续的基本程序块。这个选项禁止基本块合并，这样可以以牺牲一小部分性能来改善生成代码的可调试性
<code>--entry <entry function></code> <code>(-e)</code>	入口函数名称
<code>--fmad <true false></code> <code>(-fmad)</code>	启用或禁用浮点乘法和加法 / 减法转化成浮点乘加运算 (FMAD、FFMA 或 DFMA)。默认值：true
<code>--generate-line-info</code> <code>(-lineinfo)</code>	生成设备代码的行号信息
<code>--gpu-name <gpu name></code> <code>(-arch)</code>	指定生成代码的流处理器簇版本。此选项还使用虚拟的计算机架构，在这种情况下，代码生成被抑制。这个仅仅可以用来解析。这个选项的允许值： <code>compute_10</code> 、 <code>compute_11</code> 、 <code>compute_12</code> 、 <code>compute_13</code> 、 <code>compute_20</code> 、 <code>compute_30</code> 、 <code>compute_35</code> 、 <code>sm_10</code> 、 <code>sm_11</code> 、 <code>sm_12</code> 、 <code>sm_13</code> 、 <code>sm_20</code> 、 <code>sm_21</code> 、 <code>sm_30</code> 和 <code>sm_35</code> 。默认值： <code>sm_10</code>
<code>--input-as-string <ptx-string></code> <code>(-ias)</code>	指定在命令行上编译的包含 PTX 模块的字符串
<code>--machine [32 34]</code> <code>(-m)</code>	为 32 位还是 64 位架构编译
<code>--maxrregcount <archmax/archmin/N></code> <code>-maxrregcount</code>	指定 GPU 函数可以使用的寄存器的最大量
<code>--opt-level <N></code> <code>(-O)</code>	指定优化级别 (0-3)

(续)

选 项	描 述
--options-file <filename> (-optf)	从指定的文件中载入命令行选项
--output-file <filename> (-o)	指定输出文件的名称 (默认: elf.o)
--return-at-end -ret-end	为提高程序的可调试性, 禁止在程序结束处优化掉返回指令的默认 ptxas 行为
--sp-bounds-check (-sp-bounds-check)	生成一个堆栈指针的边界检查的代码序列。当指定 --device-debug (-g) 或 --generate-line-info (-lineinfo) 时自动启用
--suppress-double-demote-warning -suppress-double-demote-warning	在碰到 PTX 中双精度指令欲编译为不支持双精度的流处理器簇版本时, 禁止给出警告
--verbose -v	启用详细模式
--version	打印版本信息
--warning-as-error -Werror	把所有警告转化为错误

开发人员也可以通过调用 cuModuleLoadDataEx() 动态地加载 PTX 代码, 如下。

```
CUresult cuModuleLoadDataEx (
    CUmodule *module,
    const void *image,
    unsigned int numOptions,
    CUjit_option *options,
    void **optionValues
);
```

cuModuleLoadDataEx() 使用一个指针映像, 并且可以把相应的模块载入到当前的上下文。这个指针可以通过以下三种方式获得: 从 CUBIN 或 PTX 或 fatbin 文件映射而来; 作为一个 NULL 结尾的文本字符串传递 CUBIN 或 PTX 或 fatbin 文件; 把一个 cubin 或 fatbin 目标文件合并到可执行资源中, 并且通过操作系统调用函数 (如 Windows 的 FindResource() 函数) 来获取指针。选项是通过 options 传递的数组, 并且任何相关参数都可以通过 optionValues 传递。总的选项的数目是由 numOptions 指定的。任何输出将通过 optionValues 返回。表 4-9 中给出了支持的选项。

表 4-9 cuModuleLoadDataEx() 的选项

选 项	描 述
CU_JIT_MAX_REGISTERS	指定每个线程的寄存器的最大数量
CU_JIT_THREADS_PER_BLOCK	输入是每线程块允许的线程的最小数目。输出是编译器实际使用的线程数。此参数使调用者限定寄存器的数量, 以确保被指定了线程数的线程块应该能够在寄存器限制下正常启动。请注意, 此选项对寄存器以外的其他资源 (如共享内存) 无效
CU_JIT_WALL_TIME	传回一个浮点数, 包含了编译 PTX 代码的整个过程的系统时间 (以毫秒为单位)

(续)

选 项	描 述
CU_JIT_INFO_LOG_BUFFER	输入是一个指向缓冲区的指针，在这个缓冲区可以从 PTX 汇编代码打印任何有用的消息（缓冲区大小是通过 CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES 选项指定的）
CU_JIT_INFO_LOG_BUFFER_SIZE_BYTES	输入缓冲区的大小（以字节为单位）；传回的是消息的字节数
CU_JIT_ERROR_LOG_BUFFER	输入是一个指向缓冲区指针，在这个缓冲区可以从 PTX 汇编代码打印任何出错的消息（缓冲区大小是通过 CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES 选项指定的）
CU_JIT_ERROR_LOG_BUFFER_SIZE_BYTES	输入是缓冲区的大小（以字节为单位）；输出的是消息的字节数
CU_JIT_OPTIMIZATION_LEVEL	可应用到生成代码上的优化级别（0-4），其中默认的级别是 4
CU_JIT_TARGET_FROM_CUCONTEXT	从当前 CUDA 上下文推断编译目标。如果 CU_JIT_TARGET 没有指定，这是默认的行为
CU_JIT_TARGET	CUjit_target_enum：指定编译目标，选自如下之一：CU_TARGET_COMPUTE_10、CU_TARGET_COMPUTE_11、CU_TARGET_COMPUTE_12、CU_TARGET_COMPUTE_13、CU_TARGET_COMPUTE_20、CU_TARGET_COMPUTE_21、CU_TARGET_COMPUTE_30 和 U_TARGET_COMPUTE_35
CU_JIT_TARGET_FALLBACK_STRATEGY	CUjit_fallback_enum：指定在无法找到匹配的 cubin 时的后备策略，可能的值是 CU_PREFER_PTX 或 CU_PREFER_BINARY

4.3 cuobjdump

cuobjdump 是可用于检查 CUDA 所生成的二进制文件的实用工具。特别地，它可用于检查 nvcc 生成的微码。指定 nvcc 的 --cubin 参数产生 .cubin 文件，然后用

```
cuobjdump --dump-sass <filename.cubin>
```

从 .cubin 文件中转存反汇编的微码。表 4-10 给出了 cuobjdump 完整的命令行选项。

表 4-10 cuobjdump 命令行选项

选 项	描 述
--all-fatbin (-all)	转存所有 fatbin 部分。默认情况下，只会转存可执行 fatbin 的内容；如果没有可执行 fatbin，转存可重定位的 fatbin 的内容
--dump-cubin (-cubin)	为列出的所有设备函数转存 cubin
--dump-elf (-elf)	转存 ELF 目标部分
--dump-elf-symbols (-symbols)	转存 ELF 符号名
--dump-function-names (-fnam)	转存设备函数的名称。如果给出了 --dump-sass、--dump-cubin 或者 --dump-ptx 选项，此选项就会被隐含

(续)

选 项	描 述
--dump-ptx (-ptx)	为列出的所有设备函数转存 PTX
--dump-sass (-sass)	为列出的所有设备函数转存反汇编代码
--file <filename> (-f)	指定源文件的名称，这个源文件的胖二进制结构必须被转存。源文件可以通过 nvcc 编译的完整路径指定，也可以只通过文件名指定，或者使用基文件名指定（不使用文件扩展名）
--function <function name> (-fun)	指定设备函数的名字，其胖二进制结构必须转存
--help (-h)	打印有关此工具的帮助信息
--options-file <file> (-optf)	从指定的文件中载入命令行选项
--sort-functions (-sort)	当转存 SASS 时排序函数
--version (-V)	打印有关此工具的版本信息

4.4 nvidia-smi

nvidia-smi 意指英伟达系统管理的接口，用于管理英伟达特斯拉服务器级 GPU 板的运行环境。它可以报告 GPU 的状态和 GPU 执行的控制信息，例如是否启用 ECC 以及在一个给定的 GPU 上可以创建多少 CUDA 上下文。

当 nvidia-smi 由 --help (-h) 选项调用时，它会产生一个用法消息。消息中除了简要说明其目的和命令行选项外，还给出了一个所支持的产品列表。特斯拉服务器系列和 Quadro 系列的 GPU 是被完全支持的，而 GeForce 系列的 GPU 只能得到部分支持。

nvidia-smi 支持的许多 GPU 板包含多个 GPU。nvidia-smi 把每个 GPU 板称为“单元”。一些操作，如切换 LED（发光二极管）的状态，仅在每个单元的意义下可用。

nvidia-smi 有几种操作模式。如果没有提供其他命令行参数，它会给出可用 GPU 的摘要，可以使用表 4-11 中的命令行选项进一步处理。其他可选用的命令行选项包括以下几种内容：

表 4-11 nvidia-smi 列表选项

选 项	描 述
--list-gpus (-L)	显示可用 GPU 列表
--id=<GPU> (-i)	指定一个特定的 GPU
--filename=<name> (-f)	记录到一个给定的文件，而不是标准输出
--loop=<interval> (-l)	在指定的时间间隔（以秒计）进行检查，直到按下 Ctrl+C 键按停止

- 列表：--list-gpus (-L) 选项显示可用的 GPU 和其 UUID 的列表。对所列信息可以使用附加选项进一步细化，附加选项汇总于表 4-11 中。

- 查询：--query (-q) 选项显示 GPU 或“单元”信息。对所查询信息可以使用附加选项进一步细化，附加选项汇总于表 4-12 中。

表 4-12 nvidia-smi 查询选项

选 项	描 述
--query (-q)	显示 GPU 或“单元”信息
--unit (-u)	显示“单元”的属性，而不是 GPU 属性
--id=<GPU> (-i)	指定一个特定的 GPU
--filename=<name> (-f)	记录到一个给定的文件，而不是标准输出
--xml-format (-x)	生成 XML 输出
--display=<list> (-d)	仅显示选定的信息。下列选项可以用逗号分隔的列表来选择：MEMORY、UTILIZATION、ECC、TEMPERATURE、POWER、CLOCK、COMPUTER、PIDS、PERFORMNACE 和 SUPPORTED_CLOCKS
--loop=<interval> (-l)	在指定的时间间隔（以秒计）进行检查，直到按下 Ctrl+C 按键停止

- 文档类型定义 (DTD)：--dtd 选项产生 nvidia-smi 的 XML 格式输出的文档类型定义。可选的 --filename (-f) 选项可以指定输出文件；--unit (-u) 选项使得 GPU 板（而不是 GPU）的 DTD 被写入文件。
- 设备修改：表 4-13 中指定的选项可用于设置 GPU 的持久化状态，比如是否启用 ECC（纠错检查）。

表 4-13 nvidia-smi 设备修改选项

选 项	描 述
--application-clocks=<clocks> (-ac)	按元组格式 <memory,graphics> 指定 GPU 的时钟速度（例如：2000, 800）
--compute-mode=<mode> (-c)	设置计算模式：0/DEFAULT、1/EXCLUSIVE_THREAD、2/PROHIBITED，或者 3/EXCLUSIVE_PROCESS
--driver-model=<model> (-dm)	启用或禁用 TCC（特斯拉计算群集）驱动程序（仅适用于 Windows）：0/WDDM, 1/TCC。也可参见 --force-driver-model (-fdm)
--ecc-config=<config> (-e)	设置 ECC 模式：0/DISABLED 或者 1/ENABLED
--force-driver-model=<model> (-fdm)	启用或禁用 TCC（特斯拉计算群集）驱动程序（仅适用于 Windows）：0/WDDM, 1/TCC。这个选项使得即使显示器连接到 GPU，TCC 也可以被启动，否则会导致 nvidia-smi 报错
--gom=<mode>	设置 GPU 的运行模式：0/ALL_ON, 1/COMPUTE, 2/LOW_DP
--gpu-reset (-r)	触发 GPU 的二级总线复位。当机器可能需要重启时，此选项可以用来重设 GPU 硬件状态。需要与 --id 配合使用
--id=<GPU> (-i)	指定一个特定的 GPU
--persistence-mode=<mode> (-pm)	设定持久模式：0/DISABLED 或 1/ENABLED
--power-limit (-pl)	指定最大瓦的功耗管理限制
--reset-application-clocks (-rac)	重置应用时钟为默认值
--reset-ecc-errors=<type> (-p)	重置 ECC 错误计数：0/VOLATILE 或 1/AGGREGATE

- “单元”修改：--toggle-led 选项可以设定为 0/GREEN 或 1/AMBER。--id (-i) 选项可以用来指定目标单元。

4.5 亚马逊 Web 服务

亚马逊 Web 服务是“基础设施作为服务”(infrastructure as a service, IAAS)的云计算服务的卓越供应商。亚马逊的 Web 服务 (amazon web service, AWS) 使客户能够分配存储空间、与亚马逊的数据中心交换数据并运行亚马逊数据中心的服务器。反过来，客户应该按照他们的使用量缴费：按存储空间的字节数、按传输的字节数或按服务器时间的实例小时数计费。一方面，客户可以访问理论上无限的计算资源，而无须在自己的基础设施上面投资；另一方面，他们只需要为他们使用的资源买单。由于灵活性和对需求快速增长的适应能力（比如说，一个独立游戏开发商的游戏突然变得很火），云计算正变得越来越流行。

关于 AWS 特点的完整介绍以及如何使用它们，本书不予阐述。这里我们为那些对测试支持 CUDA 的虚拟机感兴趣的人介绍 AWS 的一些显著特点。

- 简单存储服务 (Simple Storage Service, S3)。可以上传和下载对象。
- 弹性计算云 (Elastic Compute Cloud, EC2)。可以启动、重启以及终止实例。
- 弹性块存储 (Elastic Block Storage, EBS)。卷可以被创建、复制、附加到 EC2 实例以及销毁。
- 提供安全组。类似于 EC2 实例的防火墙。
- 设有密钥对。用于身份验证。

亚马逊 Web 服务的所有功能都可以通过 AWS 管理控制台使用，也可以通过访问 aws.amazon.com 使用。除在上面列出的任务之外，AWS 管理控制台可以完成很多其他任务，但前面的几个操作都是我们需要在本书中介绍的。

4.5.1 命令行工具

AWS 的命令行工具可以通过 <http://aws.amazon.com/developertools> 下载，请查找“Amazon EC2 API Tools”。这些工具在 Linux 机器上也可能是现成的。Windows 用户可以安装 Cygwin。一旦安装完成后，你就可以使用命令，例如，你可以使用 ec2-run-instances 启动 EC2 实例、使用 ec2-describe-instances 给出正在运行的实例的代码清单，或者使用 ec2-terminate-instances 终止一系列实例。凡是可以在管理控制台中完成的，都可以使用命令行工具来完成。

4.5.2 EC2 和虚拟化

EC2 意指“弹性计算云”，是 AWS 的家庭成员。它允许客户“租用”一个支持 CUDA

的服务器一段时间，并且仅仅根据使用时间收费。这些虚拟的计算机，它们从用户的角度看起来像独立服务器，被称为实例。客户可以根据自己对实例计算资源的需要使用 EC2 的 Web 服务来启动、重启并终止实例。

EC2 上的支持技术之一是虚拟化，它使得一台服务器上可以同时支持多个“客户”操作系统。EC2 数据中心的单个服务器可以支持多个客户的运行实例，这可以提高经济规模和降低成本。不同的实例类型有不同的特点和定价。它们可能拥有不同数量的 RAM、CPU 能力、本地存储和 I/O 性能。它们按需定价，每实例小时的价格范围可能从 0.085 美元到 2.40 美元不等。截至写至此时，支持 CUDA 的 cg1.4xlarge 实例类型的成本为每实例小时 2.10 美元，它的配置如下：

- 23 GB 的 RAM。
- 33.5 的 ECU (两个四核心英特尔 Xeon X5570 Nehalem CPU)。
- 1690GB 的实例存储空间。
- 64 位平台。

由于 cg1.4xlarge 是一种“集群”实例类型，在一个给定服务器中只能运行一个实例。并且，相对于其他 EC2 实例类型，为了更好地并行工作，它需要更高带宽的网络以启用集群计算。

4.5.3 密钥对

密钥对支撑着 EC2 实例的访问。这个词引用自公钥加密的核心概念，即使用私有密钥（只提供给那些授权的人）和一个可以自由共享的公钥。

当创建了一个密钥对时，私钥会以一个 .pem 文件的形式下载。在密钥对创建后，需要小心保存，原因有以下两点：首先，任何访问 .pem 文件的人都能够用它来获取你的计算资源的访问权限；再者，没有办法获得私钥的新副本！亚马逊没有密钥保留业务，所以一旦私有密钥被下载，保存它就是你自己的事了。

代码清单 4-1 给出了一个 .pem 文件的例子。格式很方便查看，因为它有锚点行 ("BEGIN RSA PRIVATE KEY"/ "END RSA PRIVATE KEY") 并使用“7 位纯数据”编码（即只使用 ASCII 文本字符），因此它可以通过电子邮件发送、可以被复制并粘贴到文本域、可以追加到 ~/.ssh/authorized_keys 文件来启用无密码登录，也可以发表在书籍上。启动 EC2 实例时，给定的密钥对的名称需要指定。反过来，相应的私钥文件是用来获得访问该实例的权限的。为了更具体地了解如何用 .pem 文件访问 EC2 实例，请参见下面与 Linux 和 Windows 相关的小节。

代码清单 4-1 .pem 文件例子

```
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEAE2mHaXk9tTZqN7ZiUWoxhcSHjVCbHmn1SKamXqOKdLDfmqducvVkJAlB1cjIz/
NcwIHk0TxbnEPEDyPPHg8RYGya34evswzBUCOIcilibVIpVCyaTyzo4k0WKPW8znXJzQpxr/0Hzzu
```

```
tAvlq95HGoBobuGM5kaDSlkugOmTUXFKxZ4ZN1rm2kUo21N2m9jrkDDq4qTMFxuYW0H0AXeHOFNF
ImroUCN2udTWOjpdgIPCgYEzz3Cssd9QIzDyadw+wbkTYq7eeqTNKULs4/gmLIAw+EXKE2/seyl
leQeK11j1TFhDCjYRfghp0ecv4UnpAtiO6nNzod7aTAR1bXqJXbSqwIDAQABAcIBAAh2umvlUCst
zKpjG3zW6//iffKk17nZGZ1bzJDzF3xbPk1fBZghFvCmoquf21ROCB1ckqObK4vaSIksJrexTtoK
MBM0IRQHzGo8co6y0/n0QrXpcFzqOGknEHGk0D3ou6XEUUzMo8O+okwi9UaFq4aAn2FdYkFDa5X7
d4Y0id1wzPcVurOSrnFNkWL4GRu+p1u2bmSm7RUxQWGbp7bf98EyhdugOdo7R3yOCcdaaGg0L
hdTlwJ3jCP9dmnk7NqApRzkv7R1sXzOnU2v3b9+WpF0g6wCeM2eUk1IY3BP10Pg+Q4xU0jprSr0
vLDt8fUcIdH4PXTKua1NxsaBa1uECgYEAt7wC3BnL7HMIgf33yvK+/yA1z6AsAvIIAlCHJ0i9sihT
XF6dnfaJ6d12oCj1RUqG9e9Y3cW1YjcdqQBk5F8M6bPuIfzOctM/urd1ryWZ3ddSxgBaLE01h4c
3/cQWGGAvaMPpDSAih2d/CnnlVoQGiQrlWxDGzIHzu8RRV43fKcCgYEAtYDkj6kzlx4cuQwwsPVb
IfdtP6WzHe+Ro724ka3Ry+4xFPcarXj5y1s/aPHNpdPPCFr+uYNjBiTD90w+duV8LtBxJcF+i/lt
Muia116xXMBaMGQfFMS0u2+z3aZ18MXZF8gGDIr19VVfpDCi2RNKaT7KhfraZ8VzZsdAqDO8Z10C
gYEAvVq3iEvMF12ERQsPhzs1Q7G93U/YfxvcqbF2qoJIRTcPduZ90gjCwmwE/fZmxT6ELs31grBz
HBM0r8BWxteZW2B6uH8NjpbBf0FUQhk0+u+0oUedFcGy8juusRM9oijjCgOntfHMxMESSfT6a2yn
f4VL0wmkqUWQV2FMT4iMadECgYATFUGYrA9XT1KynNht3d9wyzPWe8ecTrPsWdj3rujybaj90aSo
gLaJX2eyP/C6mLDw83BX4PD6045ga46/UMnxWX-10fdxOrTXkEVq91YY01Yk1koj/F944gw1FS3o
34J6exJjfAQoaK3EUWU9sGHocAVFJdcrm+tufuI93NyM0QKBgB+kobIkJG8u0f19oWldhUWERsuo
poXZ9Kh/GvJ9u5DUwv6F+hCRotdBfhjuwKNTbutdzElxDMNHKoy/rhiqgcneMUmyHh/F0U4sOWl
XqgMD2QfKXBau0ttviPbsmm0dbjzTTd3FO1qx2K90T3u9GEUDwYqMxOyZjUoLyNr+Tar
-----END RSA PRIVATE KEY-----
```

4.5.4 可用区域 (AZ) 和地理区域

AWS 在被精心分开的可用区域 (AZ) 中提供服务，目的是防止影响一个可用区域的故障影响到任何其他可用区域。对 CUDA 开发人员来说，要牢记的主要考虑的因素是实例、EBS 卷和其他资源必须在同一 AZ 中进行互操作。

4.5.5 S3

简单存储服务 (Simple Storage Service, S3) 被设计为一种可靠的方式来存储数据供以后检索。“对象”(基本上是文件) 被存储在一个分层的结构里，结构顶层由“桶”构成，可根据需要插入“文件夹”。

除了存储和检索 (“PUT” 和 “GET”) 外，S3 还包括以下功能。

- 权限控制。默认情况下，S3 账户的所有者才能访问 S3 对象，但 S3 对象可能会公开或权限被授予特定的 AWS 账户。
- 对象可以加密。
- 每个 S3 对象可伴有元数据，例如一个文本文件的语言或对象是否是一个图像。
- 日志记录：S3 对象上执行的操作可以被记录 (日志被存储为更多的 S3 对象)。
- 减少冗余：S3 对象可以以较低的价格提供较低可靠性的存储服务。
- 通知：例如，如果检测到减少冗余对象的丢失可以设置自动通知以让顾客知道。
- 对象生命周期管理：可以在指定的一段时间后将对象按计划自动删除。

许多其他 AWS 服务使用 S3 作为持久性数据存储，例如，存储在 S3 的 AMI 和 EBS 卷的快照。

4.5.6 EBS

EBS (弹性块存储) 由基于网络的存储组成，可分配、附加和分离给正在运行的实例。

AWS 的用户还可以对 EBS 卷创建“快照”、为新的 EBS 卷创建模板。

EC2 实例经常有一个包含操作系统和驱动程序软件的根 EBS 卷。如果需要更多的存储空间，可以创建并附加 EBS 卷并且把它装载在客户操作系统里[⊖]。

4.5.7 AMI

亚马逊机器映像（AMI）是用来描述一旦 EC2 实例启动“看起来”会是什么样的，包括操作系统和附加的 EBS 卷的数量和内容的说明。大多数 EC2 用户是开始于一个由 Amazon 提供的“库存”AMI，并且不断修改它以满足用户的要求，然后对 AMI 创建快照，这样他们能够使用相同的配置启动更多实例。

启动一个实例时，EC2 需要用几分钟集合请求的资源，并启动虚拟机。一旦实例运行，就可以查询其 IP 地址并通过互联网用私钥访问它，它的名字已经在实例启动时指定了。

实例的外部 IP 地址被纳入了 DNS 名称。例如，一个 cg1.4xlarge 实例可能会被命名为 ec2-70-17-16-35.compute-1.amazonaws.com，而这台机器的外部 IP 地址是 70.17.16.35[⊖]。

EC2 实例也有可用于集群内通信的内部 IP 地址。例如，如果你启动了一批需要通过软件（如消息传递接口（MPI））相互通信的实例就要使用内部 IP 地址。

4.5.8 EC2 上的 Linux

EC2 支持许多不同类型的 Linux，包括由红帽派生的亚马逊品牌的 Linux（“Amazon Linux”）。一旦一个实例启动，人们就可以使用用来启动这个实例的密钥对通过 ssh 来访问它。使用以上的 IP 地址和代码清单 4-1 中的 .pem 文件实例，我们就可以键入 ssh -i Example.pem ec2-user@70.17.16.35（超级用户名随着 Linux 类型的不同而不同：ec2-user 是 Amazon Linux 的超级用户名，而 CentOS 使用 root，Ubuntu 使用 ubuntu）。

登录后，机器就归你使用啦！你可以添加用户并为他们设置密码、设置 SSH 登录密码、安装需要的软件（如 CUDA 工具集），附加更多的 EBS 卷并设立临时磁盘。然后，你可以为 AMI 创建快照以便能够启动更多的实例，这些实例看起来酷似你已经配置好的实例。

1. EBS

EBS 卷很容易创建，既可以从一个空白卷创建，还可以现场制作一个快照副本。一旦创建，EBS 卷可能被附加到一个实例上，它将会作为一个设备（如 /dev/sdf 或在更近期

[⊖] 如果要让来自派生 AMI 的实例使用此 EBS 卷，你必须修改操作系统的配置。请参见 4.5.8 和 4.5.9 节。

[⊖] 为保护用户，这里的 IP 地址在实际地址上做了修改。

的 Linux 内核上为 /dev/xvdf) 出现。首次附加 EBS 卷时, 它仅仅是一个原始块存储设备, 在使用命令 (如 mkfs.ext3) 前它必须被格式化。格式化之后, 驱动器可能会被装载到一个目录:

```
mount <Device> <Directory>
```

最后, 如果你想为 AMI 创建快照并为了让刚启动的派生 AMI 看到驱动器, 编辑 /etc/fstab 以载入卷。当创建一个 EBS 卷以附加到一个正在运行的实例时, 确保在实例的同一个可用区域里 (例如, us-east-1b) 创建它。

2. 短暂存储

许多 EC2 实例的类型, 包括 cgl.4xlarge, 有与它们相关联的本地硬盘。当这些磁盘可用时, 被严格作为可删除本地存储使用。不像 EBS 或 S3, 没有纠错编码或其他技术使磁盘更可靠。为了强调这种被降低了的可靠性, 这里的磁盘被称为短暂存储。

为了让短暂磁盘可用, 指定了 ec2-run-instances 的 “-b” 选项, 例如,

```
ec2-run-instances -t cgl.4xlarge -k nwiltEC2 -b /dev/sdb=ephemeral0  
/dev/sdc=ephemeral1
```

像 EBS 卷一样, 在使用之前, 短暂存储必须格式化 (如, mkfs.ext3)、装载, 并且它们必须有 fstab 入口, 为的是在实例被重启时能够再现它们。

3. 用户数据

在启动时或当一个实例运行时 (在这种情况下, 必须重新启动实例), 用户数据可以被指定到某个实例。可以在如下地址查询用户数据:

<http://169.254.169.254/latest/user-data>

4.5.9 EC2 上的 Windows

访问 Windows 实例与访问 Linux 实例的方式略有不同。一经启动, 客户必须使用自己的私钥文件检索 EC2 实例的管理员账户的密码。你可以指定你的 .pem 文件或复制并粘贴其内容到 AWS 管理控制台 (如图 4-1 所示)。

默认情况下, 此密码生成行为只对来自 AWS 的“库存”AMI 有作用。如果你对这些 AMI 之一创建“快照”, 得到的快照将保留机器上的任何密码。要创建一个新的可以生成随机密码的 Windows AMI, 运行“EC2 Config Service”工具 (可在“Start”菜单找到), 点击“Bundle”选项卡, 然后单击上面写着“Run Sysprep and Shutdown Now”的按钮 (见图 4-2)。点击这个按钮后, 任何相对于它创建的 AMI 都会产生一个随机密码, 就像库存 Windows AMI 那样。

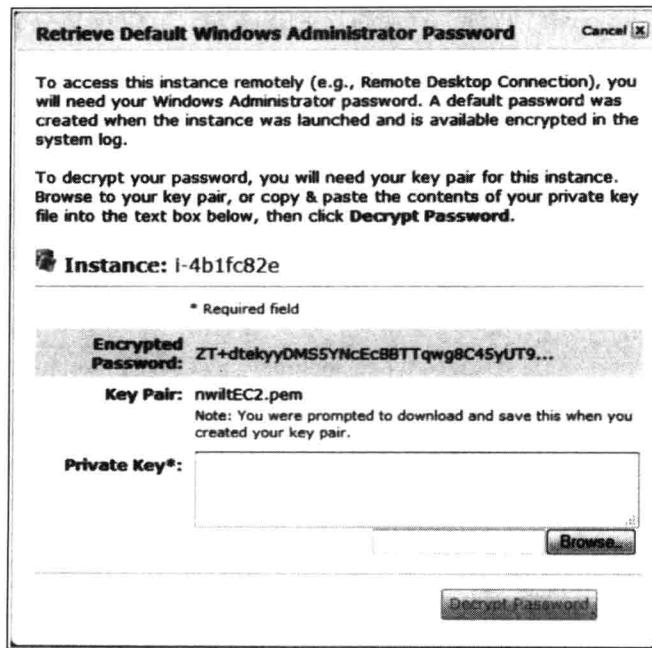


图 4-1 AWS Windows 密码检索

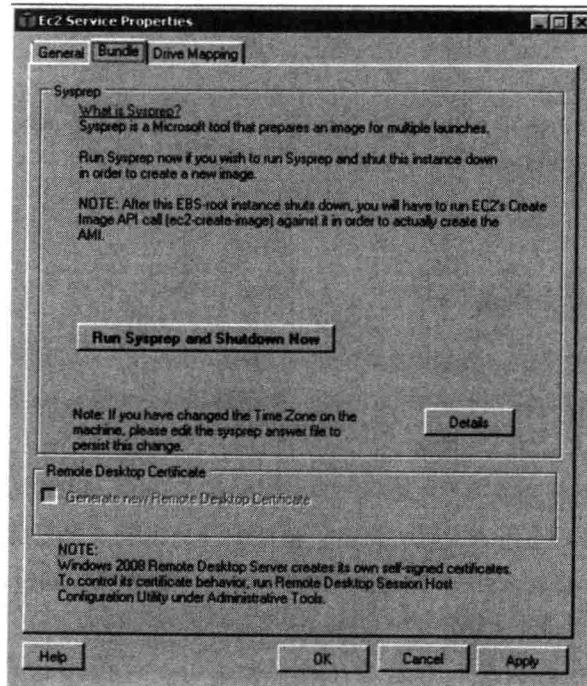


图 4-2 Windows EC2 上的 Sysprep

1. 短暂存储

为了使短暂存储可用于 Windows 实例，必须指定 ec2-run-instances 的 -b 选项，如下所示：

```
ec2-run-instances -t cg1.4xlarge -k nwiltEC2 -b /dev/sdb=ephemeral0  
/dev/sdc=ephemeral1
```

2. 用户数据

在启动时或当一个实例运行时（在这种情况下，必须重新启动实例），用户数据可以被指定到一个实例。用户数据可以到如下地址查询：

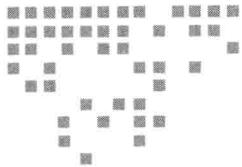
<http://169.254.169.254/latest/user-data>



第二部分 *Part 2*

CUDA 编程

- 第 5 章 内存
 - 第 6 章 流与事件
 - 第 7 章 内核执行
 - 第 8 章 流处理器簇
 - 第 9 章 多 GPU
 - 第 10 章 纹理操作
- *****



内 存

为了最大化地提升性能，CUDA 使用了多种内存类型，我们使用何种类型的内存依赖于预期的使用方法。主机内存指的是系统中 CPU 的内存。CUDA 同样提供了 API 来更快的访问主机内存，可通过页面锁定将其映射给 GPU。设备内存指的是使用一个专用内存控制器访问的 GPU 内存，并且，正像每一个 CUDA 的初学者都知道的，数据必须在主机与设备内存间显式复制，以便 GPU 处理。

设备内存可以通过多种方式分配与访问：

- 全局内存可以被静态或动态地分配，并可以通过 CUDA 内核中的指针访问，内核可将其转化为全局加载 / 存储指令。
- 常量内存是只读内存，通过被不同的指令以优化方式访问。来自多个线程的读请求可以借助缓存以广播方式实现。
- 本地内存包含栈：不能被寄存器保存、不能作为参数、也不能作为子程序返回地址的本地变量。
- 纹理内存（CUDA 数组形式）通过纹理与表面加载 / 存储指令访问。同常量内存一样，来自纹理内存的读请求由独立缓存服务，借助该缓存能够对只读访问优化。

共享内存是 CUDA 中一个重要的内存类型，它不由设备内存提供。相反，它是对片上暂存内存的抽象，在每个线程块内线程间的快速数据交换中使用。物理上来看，共享内存以 SM 上的内建内存形式出现：在 SM 1.x 硬件上，共享内存用 16KB 的 RAM 实现；在 SM 2.x 和最近的硬件中，共享内存使用 64KB 缓存实现，这块缓存可以划分成 48KB 一级缓存 /16KB 共享内存，或 48KB 共享内存和 16KB 一级缓存。

5.1 主机内存

在 CUDA 中，主机内存指的是系统中可被 CPU 访问的内存。默认来说，这个内存是可换页的，意味着操作系统可移动该内存或换出到磁盘。因为可换页内存的物理内存位置可能在不被察觉的情况下改变，它不可以被如 GPU 这样的外设访问。为了让硬件使用 DMA，操作系统允许主机内存进行页锁定，并且因为性能原因，CUDA 包含了开发者使用这些操作系统工具的 API。页锁定后的且映射为 CUDA 直接访问的锁页内存允许以下几点：

- 更快的传输性能。
- 异步内存复制（即：在必要的内存复制结束之前内存复制返回控制给调用者；GPU 复制操作与 CPU 并行地执行。）。
- 映射锁页内存可以被 CUDA 内核直接访问。

因为可换页内存的虚拟地址到物理地址的映射可能变化无常，所以 GPU 根本不能访问可换页内存。CUDA 为了能够复制可换页内存中数据，会使用一对中转缓冲区，这对缓冲区是锁页的，在 CUDA 上下文创建时由驱动程序分配。第 6 章介绍了手动编写的换页内存复制程序，其中使用 CUDA 事件进行管理这个双缓冲区需要的同步。

5.1.1 分配锁页内存

锁页内存可以通过 CUDA 提供的特殊函数分配与释放：CUDA 运行时中的 `cudaHostAlloc()`/`cudaFreeHost()`，驱动程序 API 中的 `cuMemHostAlloc()`/`cuMemFreeHost()`。这些函数同主机操作系统一同，分配锁页内存并对其进行映射使得 GPU 可以进行 DMA 操作。

CUDA 会跟踪分配的内存并在内存复制中涉及使用 `cuMemHostAlloc()`/`cudaHostAlloc()` 分配的主机内存指针时，自动加速。除此之外，一些函数（尤其是异步内存复制函数）需要使用锁页内存。

SDK 中的示例 `bandwidthTest` 使开发者能够轻松的比较锁页内存与普通换页内存的性能。选项 `--memory=pinned` 使测试使用锁页内存替代可换页内存。表 5-1 列出了亚马逊 EC2 中一个 `cg1.4xlarge` 实例的 `bandwidthTest` 数值（单位 MB/s），测试运行在 Windows 7-x64 上。分配锁页内存成本十分高昂，因为这项操作对主机来说包含了大量的工作，其中包括一个内核转换。

表 5-1 锁页带宽与可换页带宽对比

	主机到设备	设备到主机
锁页	5523	5820
可换页	2951	2705

CUDA 2.2 添加了几个锁页内存的特性。可共享锁页内存可以被任何 GPU 访问；“映射锁页内存”被映射到 CUDA 地址空间中，以便 CUDA 内核直接访问；写结合锁页内存在一

些系统上拥有更快的总线传输速度。CUDA 4.0 同样添加了两个关于主机内存的重要的特性：已存在的主机内存区间可以使用主机内存注册操作将页面锁定，统一虚拟寻址（UVA）使所有的指针在进程内变得唯一，包括主机与设备指针。当 UVA 启用，系统可以通过地址区间推测出内存是设备内存还是主机内存。

5.1.2 可共享锁页内存

默认情况下，锁页内存分配只能由使用 `cudaHostAlloc()` 或 `cuMemHostAlloc()` 的 GPU 访问。通过在函数 `cudaHostAlloc()` 中指定 `cudaHostAllocPortable` 标志，或在函数 `cuHostMemAlloc()` 中指定 `CU_MEMHOSTALLOC_PORTABLE`，应用程序可以请求锁页分配映射给所有的 GPU。可共享锁页内存能获利于前文描述的内存复制自动加速，并且可以参与到系统中任何 GPU 的异步内存复制操作中。对想要使用多 GPU 的应用程序，指定所有锁页分配为可共享的是一个非常好的方式。



注意 当统一虚拟寻址（UVA）有效，所有的锁页内存分配都是可共享的。

5.1.3 映射锁页内存

默认的，锁页内存分配被映射给 CUDA 地址空间外的 GPU。它们可以被 GPU 直接访问，但仅仅是通过内存复制函数可以做到。CUDA 内核不能直接读写主机内存。然而，在 SM 1.2 以及以上版本的 GPU 上，CUDA 内核可以直接读写主机内存。只需分配的内存被映射到设备内存地址空间即可。

为了启用映射锁页内存分配，使用 CUDA 运行时的程序必须使用 `cudaDeviceMapHost` 标志调用 `cudaSetDeviceFlags()` 函数，这个调用必须在所有的初始化执行之前进行。驱动程序 API 则在函数 `cuCtxCreate()` 中指定 `CU_CTX_MAP_HOST` 标志。

一旦映射锁页内存被启用，它就可以通过使用 `cudaHostAllocMapped` 标志调用 `cudaHostAlloc()` 函数，或者在函数 `cuMemHostAlloc()` 中使用标志 `CU_MEMALLOCHOST_DEVICEMAP` 来分配。除非 UVA 启用，应用程序随后必须使用函数 `cudaHostGetDevicePointer()` 或 `cuMemHostGetDevicePointer()` 查询关联到所分配内存的设备指针。得到的设备指针可以传入到 CUDA 内核。使用映射锁页内存的最佳实践在本章“映射锁页内存用法”一节中描述。



注意 当 UVA 启用，所有的锁页内存都是被映射的[⊖]。

[⊖] 除了标记为写结合的内存之外。

5.1.4 写结合锁页内存

写结合内存，也称为非缓存预测的写结合（uncacheable speculative write combining，USWC）内存。创建写结合内存可以使 CPU 快速写入 GPU 帧缓冲区而不用污染 CPU 缓存^①。为了实现这个目的，英特尔公司添加了新的页表种类，控制写入到特殊的写结合缓冲区而不是原来的主处理器缓存结构。之后，英特尔又添加了“非暂时”存储指令（即：MOVNTPS 和 MOVNTI），使应用程序以单个指令为单位控制写入到写结合缓冲区中。一般来说，需要使用内存栅栏指令（例如 MFENCE）保持与 WC 内存的一致性。CUDA 应用程序不需要这些操作，因为在 CUDA 驱动程序提交给硬件工作时这些操作都会被自动执行。

对 CUDA 来说，可以在调用 cudaHostAlloc() 函数时使用 cudaHostWriteCombined 标志，或者使用 cuMemHostAlloc() 函数和 CU_MEMHOSTALLOC_WRITECOMBINED 标志请求写结合内存。除了设置页表入口绕过 CPU 缓存，这一内存同样不能在 PCIe 总线传输时被窥探（snoop）。在拥有前端总线的系统上（AMD 翡翠处理器和英特尔微处理器），避免窥探提高了 PCIe 的传输性能。在 NUMA（非一致内存访问）系统上写结合（WC）内存的性能优势微弱。

使用 CPU 读 WC 内存非常慢（大约比 CPU 读慢 6 倍），除非读操作由指令 MOVNTDQA（在 SSE4 中新添加）完成。在英伟达集成 GPU 中，写结合内存与系统内存操作几乎同样快——这些系统内存从系统启动时间就被保留来为 GPU 使用，它们对 CPU 来说是不可用的。

尽管有这些前面提到的优势，只有很少的理由会使 CUDA 开发者选择写结合内存。指向写结合内存的主机内存指针很容易“泄露”到想要进行内存读操作的应用程序中。在缺少经验证据的情况下，这应当被避免。

 **注意** 当统一虚拟寻址有效，写结合锁页内存分配不会被映射到统一地址空间中。

5.1.5 注册锁页内存

CUDA 开发者不太有机会分配主机内存用于 GPU 直接访问。例如，一个大型的、可扩展应用程序可能有传递内存指针给 CUDA 插件的接口，或者应用程序可能出于与 CUDA 相同原因使用其他的 API 为其他的外设分配专用内存（特别是高速的网络）。为了应对这些使用情况，CUDA 4.0 添加了注册锁页内存能力。

锁页内存注册把内存分配与页锁定和主机内存映射分离开来。它可以操作一个已分配的虚拟地址范围，并页锁定它，然后，将其映射给 GPU。正如 cudaHostAlloc()，可根据需要让

① WC 内存最初由英特尔公司在 1997 年提出，与加速图像端口（AGP）同时提出。在 PCIe 之前，AGP 被使用在图像处理板卡上。

内存映射到 CUDA 地址空间或变成可共享的（所有 GPU 可访问）。

函数 cuMemHostRegister()/cudaHostRegister() 和 cuMemHostUnregister()/ cudaHostUnregister() 分别实现把主机内存注册为锁页内存和移除注册的功能。注册的内存范围必须是页对齐的：换句话说，无论是基地址还是大小，都必须是可被操作系统页面大小整除的。应用程序可以用两种方式分配页对齐地址范围：

- 使用操作系统提供的工具遍历所有的页来分配内存，像 Windows 上的 VirtualAlloc() 或其他平台上的 valloc() 或 mmap()[⊖]。
- 给定一段任意的地址范围（例如，使用 malloc() 或操作符 new[] 分配内存），夹取内存范围到倒数第二页页边界并补齐下一个页面。



注意 即使启用 UVA，已映射到 CUDA 地址空间的注册的锁页内存还是有着与主机指针不同的设备指针。应用程序必须调用 cudaHostGetDevicePointer()/cuMemHostGetDevicePointer() 以得到设备指针。

5.1.6 锁页内存与统一虚拟寻址

当 UVA（统一虚拟寻址）有效，所有的锁页内存分配均是映射的和可共享的。这一规则的例外是写结合内存和注册的内存。对于这两者，设备指针可能不同于主机指针，并且应用程序仍然需要使用 cudaHostGetDevicePointer()/cuMemHostGetDevicePointer() 函数查询其设备指针。

除了 Windows Vista 和 Windows 7 (Windows 8)，UVA 被所有的 64 位平台所支持。在 Windows Vista 和 Windows 7 上，只有 TCC 驱动程序（可以使用 nvidia-smi 启用或禁用）支持 UVA。应用程序查询 UVA 是否启用有两种方法：一是可以使用函数 cudaGetDeviceProperties() 并检查 cudaDeviceProp::unifiedAddressing 结构体成员，二是调用函数 cuDeviceGetAttribute() 并使用参数 CU_DEVICE_ATTRIBUTE_UNIFIED_ADDRESSING。

5.1.7 映射锁页内存用法

对依赖于 PCIe 传输性能的应用程序，映射锁页内存是一个福音。由于 GPU 可以通过内核直接读写主机内存，它减少了一些内存复制的需求，减少了花销。这里是一些常见的使用映射锁页内存的情况：

- 提交写入到主机内存：为了同其他的 GPU 进行交换，多 GPU 应用程序经常需要返回中转结果给系统内存；通过映射锁页内存写这些结果避免了一次不必要的设备到主机间的内存复制。对主机内存的只写访问模式会很吸引人，因为它没有需要隐藏的延迟。

[⊖] 或者联合使用函数 posix_memalign() 和 getpagesize()。

- 使用流：这些工作负载在其他方面会使用 CUDA 流来协调，使设备与内存间的内存与内核的处理工作并发执行。
- “重叠复制”：有些负载会得益于在计算的同时通过 PCIe 传输数据。例如，GPU 可以在为扫描而传输数据时执行子数组归约。



映射锁页内存并不是万能的，这里有一些使用映射锁页内存应该小心的方面：

- 在映射锁页内存上进行纹理操作是可能的，但速度非常慢。
- 使用合并内存事务访问映射锁页内存非常重要（参考 5.2.9 节）。未进行合并内存事务处理的访问时间会是处理后的 2 ~ 6 倍。但是即使是 SM 2.x 和之后的 GPU，它们的缓存机制已经企图将合并处理替换掉，这项性能惩罚也是巨大的。
- 不建议使用内核轮询主机内存（例如，进行 CPU/GPU 同步操作）。
- 不要尝试在映射锁页内存上进行原子操作，对主机（locked compare-exchange，加锁的比较交换操作）或设备（atomicAdd()）都是如此。CPU 中加锁操作执行互斥的工具，对 PCIe 总线上的外设来说，是不可见。反过来，对 GPU，原子操作只在本地设备内存上工作，因为它们是使用 GPU 本地内存控制器实现的。

5.1.8 NUMA、线程亲和性与锁页内存

在 AMD 鲲龙和英特尔系列处理器上，CPU 内存控制器被直接集成进 CPU。之前，内存被附加在芯片组中的北桥上的前端总线（FSB）上。在多 CPU 系统中，北桥可以为任何一个 CPU 的内存请求服务，并且 CPU 间的内存访问性能相当一致。随着集成内存控制器的引入，每一个 CPU 都拥有自己的专用内存池，专用内存池来自每一个直接附加在 CPU 上的“本地”物理内存。尽管任何 CPU 都可以访问任意其他的 CPU 内存——通过 AMD 的 HyperTransport（HT）或 Intel 的 QuickPath Interconnect（QPI），但这会招致延迟惩罚和带宽限制。相对于系统使用 FSB 的一致内存访问时间，这一系统架构称作 NUMA，即非一致内存访问（nonuniform memory access）。

多线程应用程序的性能可能高度依赖于正在运行线程的 CPU 是否引用的是本地内存。无论如何，对大多数的应用程序，高额的非本地访问开销被 CPU 上的缓存所弥补。一旦非本地内存被读取进 CPU，它会始终保留在缓存中，直到换出，或者其他 CPU 想要访问内存中的相同的页。事实上，启用系统 BIOS 选项，以便在 CPU 间交错物理内存，对 NUMA 系统是很普遍的。当这个 BIOS 选项启用，内存按缓存行大小（普遍为 64 位）为基础均匀划分。举个例子，在双 CPU 系统中，平均大约有 50% 内存是非本地访问的。

对 CUDA 应用程序，PCIe 传输性能依赖于内存引用是否为本地。如果系统有超过一个 I/O 集线器（IOH），给定 IOH 上附加的 GPU 在锁页内存是本地的情况下会有更好的性能表

现，并且会减少 QPI 带宽的需求。因为一些高级的 NUMA 系统是分级的，但是没有严格的在 CPU 与内存带宽池间构成联系，NUMA API 既适用于严格关联的节点，也适用于没有严格关联的节点上。

如果系统上的 NUMA 开启，对给定 GPU 在同一节点分配主机内存是一个好的做法，不幸的是，现在没有官方的 CUDA API 来为 GPU 连接一个给定的 CPU。拥有系统设计经验的开发者可能会知道哪一个节点关联着哪一个 GPU。之后，可以使用针对特定平台的 NUMA 感知的 API (NUMA-aware API) 分配内存，然后利用主机内存注册 (参见 5.1.5 小节) 锁定这些分配的虚拟内存，并且把它们映射给 GPU。

代码清单 5-1 给出了 Linux 上的执行 NUMA 感知的内存分配的代码片段[⊖]，代码清单 5-2 给出了 Windows 上的代码片段[⊖]。

代码清单 5-1 NUMA 感知的内存分配 (Linux)

```

bool
numNodes( int *p )
{
    if ( numa_available() >= 0 ) {
        *p = numa_max_node() + 1;
        return true;
    }
    return false;
}

void *
pageAlignedNumaAlloc( size_t bytes, int node )
{
    void *ret;
    printf( "Allocating on node %d\n", node ); fflush(stdout);
    ret = numa_alloc_onnode( bytes, node );
    return ret;
}

void
pageAlignedNumaFree( void *p, size_t bytes )
{
    numa_free( p, bytes );
}

```

代码清单 5-2 NUMA 分配 (Windows)

```

bool
numNodes( int *p )
{
    ULONG maxNode;
    if ( GetNumaHighestNodeNumber( &maxNode ) ) {
        *p = (int) maxNode+1;
        return true;
    }
    return false;
}

```

[⊖] <http://bit.ly/USy4e7>。

[⊖] <http://bit.ly/XY1g8m>。

```

void *
pageAlignedNumaAlloc( size_t bytes, int node )
{
    void *ret;
    printf( "Allocating on node %d\n", node ); fflush(stdout);
    ret = VirtualAllocExNuma( GetCurrentProcess(),
        NULL,
        bytes,
        MEM_COMMIT | MEM_RESERVE,
        PAGE_READWRITE,
        node );
    return ret;
}

void
pageAlignedNumaFree( void *p )
{
    VirtualFreeEx( GetCurrentProcess(), p, 0, MEM_RELEASE );
}

```

5.2 全局内存

全局内存是 CUDA 中的一个主要抽象，内核通过其读写设备内存^①。由于设备内存直接附加在 GPU 上，由集成在 GPU 中的内存控制器读写，所以峰值带宽会非常高：在高端的 CUDA 卡上通常会超过 100G/s。

设备内存可以被 CUDA 内核使用设备指针访问，下面是简单用于内存设置的内核的一个例子。

```

template<class T>
__global__ void
GPUMemset( int *base, int value, size_t N )
{
    for ( size_t i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*blockDim.x )
    {
        base[i] = value;
    }
}

```

设备指针 `base` 驻留在设备地址空间，独立于 CUDA 程序中主机代码使用的 CPU 地址空间。因此，CUDA 程序中的主机代码可以在设备指针上执行指针算术运算，但它们不能解引用这些指针^②。

这一内核写入整型值到由 `base` 和 `N` 给定的地址范围。在指定的线程块和网格参数来启动内核时，对 `blockIdx`、`blockDim` 和 `gridDim` 的引用使内核能够正确地操作。

- ① 最让开发者困惑的是，CUDA 使用“设备指针”术语指代驻留在全局内存上的指针（CUDA 内核可访问的设备内存）。
- ② 映射锁页指针是这一规则的一个例外。它们驻留在系统内存中，但是可以被 GPU 访问，在非 UVA 系统上，指向这一内存的主机指针和设备指针并不相同：应用程序必须调用函数 `cuMemHostGetDevicePointer()` 或 `cudaHostGetDevicePointer()` 映射主机指针到相关联的设备指针上。但是当 UVA 有效时，这两个指针就是相同的。

5.2.1 指针

当使用 CUDA 运行时时，设备指针与主机指针类型均为 `void*`。驱动程序 API 则使用一个整型值类型的 `typedef` 定义——`CUdeviceptr`，它与主机指针有相同宽度（即，32 位操作系统上 32 位宽，64 位同理），如下：

```
#if defined(__x86_64) || defined(AMD64) || defined(_M_AMD64)
typedef unsigned long long CUdeviceptr;
#else
typedef unsigned int CUdeviceptr;
#endif
```

类型 `uintptr_t` 是在头文件 `<stdint.h>` 中定义的，在 C++0x 中被引入。开发者可使用它在主机指针 (`void*`) 和设备指针 (`CUdeviceptr`) 之间方便的转换，如下：

```
CUdeviceptr devicePtr;
void *p;
p = (void *) (uintptr_t) devicePtr;
devicePtr = (CUdeviceptr) (uintptr_t) p;
```

主机可在设备指针上进行指针的算术运算，作为参数传入一个内核或被内存复制调用，但是主机不能使用这些指针读或写设备内存。

驱动程序 API 中的 32 位和 64 位指针

因为驱动程序 API 中对指针的原始定义是 32 位的，在 CUDA 中添加对 64 位指针的支持需要定义 `CUdeviceptr`，并且，所有使用 `CUdeviceptr` 作为一个参数的驱动程序 API 函数，需要有所变化来改变支持位数。[⊖] 举个例子，函数 `cuMemAlloc()`，从

```
CUresult CUDA API cuMemAlloc(CUdeviceptr *dptr, unsigned int bytesize);
```

改变为

```
CUresult CUDA API cuMemAlloc(CUdeviceptr *dptr, size_t bytesize);
```

为了同时支持旧的应用程序（函数 `cuMemAlloc()` 使用 32 位 `CUdeviceptr` 链接的）和新的应用程序，`cuda.h` 头文件包含了两个代码块，在开发者更新到新的 API 后，使用预处理技术不用改变函数名而改变绑定即可。

首先，一个代码块暗中改变函数名称来映射新的有不同语义的函数。

```
#if defined(__CUDA_API_VERSION_INTERNAL) || __CUDA_API_VERSION
>= 3020

#define cuDeviceTotalMem cuDeviceTotalMem_v2
...
#define cuTexRefGetAddress cuTexRefGetAddress_v2
#endif /* __CUDA_API_VERSION_INTERNAL || __CUDA_API_VERSION >= 3020 */
```

使用这一方式，客户端代码使用同样的旧函数名称，但是编译的代码会生成对新函数的引用，新函数名称后缀添加了 `_v2`。

在之后的头文件中，旧函数依旧使用以前的定义方式。因此，开发者应使用最新的 CUDA 版本编译程序以得到最新的函数定义与语义。`cuda.h` 使用相同的策略来处理函数语义

[⊖] 出于兼容性考虑，旧函数被保留。

因版本改变的问题，例如 `cuStreamDestroy()`。

5.2.2 动态内存分配

大多数 CUDA 中的全局内存通过动态分配得到。使用 CUDA 运行时，函数

```
cudaError_t cudaMalloc( void **, size_t );
cudaError_t cudaFree( void );
```

分别分配与释放全局内存。相关的驱动程序 API 函数是

```
CUresult CUDA API cuMemAlloc(CUdeviceptr *dptr, size_t bytesize);
CUresult CUDA API cuMemFree(CUdeviceptr dptr);
```

分配全局内存成本非常高昂。CUDA 驱动程序实现了一个 CUDA 小型内存请求的子分配器 (suballocator)，但是如果这个子分配器必须创建一个新的内存块，这需要调用操作系统的一个成本很高的内核模式驱动程序。如果这种情况发生，CUDA 驱动程序必须与 GPU 同步，这可能会中断 CPU/GPU 的并发。因此，在性能要求很高的代码中避免分配或释放全局内存是一个较好的做法。

1. 等步长内存分配

合并限制以及为纹理操作和 2D 内存复制而设的对齐限制，激发了等步长内存分配的应用。这一想法是当创建一个 2D 数组时，指向数组的指针每移动到不同的行后必须保持相同的对齐特性。数组步长是数组中每一行的字节数^①。等步长内存分配使用一个宽度（单位为字节）和高度，填补宽度到一个合适的硬件相关的宽度，传回基指针和步长给驱动程序。通过使用这些分配函数，把选择步长的任务委托给驱动程序，开发者可以让他们的代码适应未来增大对齐宽度的架构变化^②。

CUDA 应用程序经常需要满足硬件强制的对齐约束。不仅仅针对基地址，对内存复制宽度（字节）和绑定到纹理的线性内存也是同样的。因为对齐约束是由机器决定的，CUDA 提供 API 使开发者为驱动程序选择适当的对齐。这些 API 使 CUDA 应用程序实现了独立于硬件的代码，并且可以在未来适应现在尚未出现的架构。

图 5-1 展示了一个在数组上执行的等步长内存分配，这个数组的宽度为 352 字节宽。在内存分配

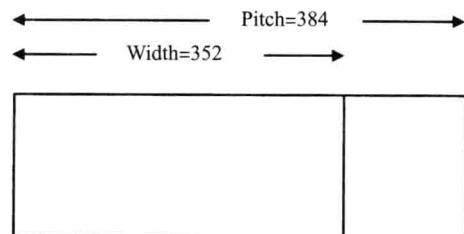


图 5-1 步长与宽度

- ① 对齐 2D 内存复制的想法要比 CUDA 出现早的多。图形 API，像苹果的 QuickDraw 和微软的 DirectX 分别暴露了“行字节”和“间距”。同时，补齐使用移位代替乘法，简化了寻址计算，甚至使用两次移位与一次与两个 2 的幂的加法来简化乘法，例如 640 (512+128)。但在今天，整数加法的速度已经很快，等宽内存分配有着其他的动机，例如避免与缓存交互带来的性能消极影响。
- ② 这不是一个意想不到的趋势，在特斯拉架构上，费米已经扩展了对不同的对齐需求的支持。

之前，宽度会填补到最近的 64 的倍数。对给定的数组宽度，除了行和列，一个数组元素的地址可以按如下代码计算：

```
inline T *
getElement( T *base, size_t Pitch, int row, int col )
{
    return (T *) ((char *) base + row*Pitch) + col;
}
```

执行等步长内存分配的 CUDA 运行时函数如下：

```
template<class T>
__inline__ __host__ cudaError_t cudaMallocPitch(
    T **devPtr,
    size_t *pitch,
    size_t widthInBytes,
    size_t height
);
```

CUDA 运行时同样包含了 cudaMalloc3D() 函数，这个函数会使用 cudaPitchPtr 和 cudaExtent 结构体分配 3D 内存。

```
extern __host__ cudaError_t CUDARTAPI cudaMalloc3D(struct
cudaPitchedPtr* pitchedDevPtr, struct cudaExtent extent);
```

cudaPitchedPtr 结构接收分配的内存，定义如下：

```
struct cudaPitchedPtr
{
    void *ptr;
    size_t pitch;
    size_t xsize;
    size_t ysize;
};
```

cudaPitchedPtr::ptr 指定指针，cudaPitchedPtr::pitch 指定了分配的步长（单位为字节），cudaPitchedPtr::xsize 和 cudaPitchedPtr::ysize 分别指定了分配的逻辑宽度和高度。cudaExtent 如下定义：

```
struct cudaExtent
{
    size_t width;
    size_t height;
    size_t depth;
};
```

cudaExtent::width 在数组与线性设备内存中分别做不同处理。在数组中，它指定数组成员的宽度；对线性设备内存，它会指定步长（用字节描述的宽度）。

驱动程序 API 使用的等步长内存分配函数如下：

```
CUresult CUDA API cuMemAllocPitch(CUdeviceptr *dptr, size_t *pPitch,
size_t WidthInBytes, size_t Height, unsigned int ElementSizeBytes);
```

ElementSizeBytes 参数可能为 4、8 或 16 字节，而且它会引发分配的等步长内存分配分别填补到 64、128 或 256 字节边界。这些“对齐”在 SM 1.0 和 SM 1.1 硬件上的 4、8、16 字节内存事务合并处理中需要。那些不关心性能的应用程序可以指定为 4。

函数 cudaMallocPitch()/cuMemAllocPitch() 返回的步长是在调用者传入的一个以字节为单位的宽度基础上，补齐到满足全局加载 / 存储操作的合并限制和纹理操作绑定 API 的对齐

限制。内存分配数量是高 * 宽。

对 3D 数组，开发者可以在执行内存分配之前用深度乘上高度。这一考虑只适用于会通过全局加载 / 存储访问的数组，这是由于 3D 纹理操作不能被绑定到全局内存。

2. 内核内分配

费米架构硬件可以使用 malloc() 函数动态的分配全局内存。由于这可能需要 GPU 来中断 CPU，可能会很慢。样例程序 mallocSpeed.cu 测量了内核中 malloc() 和 free() 函数的性能。

代码清单 5-3 展示了 mallocSpeed.cu 中的关键内核和计时子程序。一个重要的提示，cudaSetDeviceLimit() 函数必须连同 cudaLimitMallocHeapSize 参数在内核可能调用的 malloc() 函数之前调用。mallocSpeed.cu 中的调用需要一个完整的 10 亿字节 (2^{30})。

```
CUDART_CHECK( cudaDeviceSetLimit(cudaLimitMallocHeapSize, 1<<30) );
```

当调用 cudaDeviceSetLimit() 时，所请求数量的内存被分配，并且不能再被其他目的使用。

代码清单 5-3 MallocSpeed 函数和内核

```
__global__ void
AllocateBuffers( void **out, size_t N )
{
    size_t i = blockIdx.x*blockDim.x + threadIdx.x;
    out[i] = malloc( N );
}

__global__ void
FreeBuffers( void **in )
{
    size_t i = blockIdx.x*blockDim.x + threadIdx.x;
    free( in[i] );
}

cudaError_t
MallocSpeed( double *msPerAlloc, double *msPerFree,
            void **devicePointers, size_t N,
            cudaEvent_t evStart, cudaEvent_t evStop,
            int cBlocks, int cThreads )
{
    float etAlloc, etFree;
    cudaError_t status;

    CUDART_CHECK( cudaEventRecord( evStart ) );
    AllocateBuffers<<<cBlocks,cThreads>>>( devicePointers, N );
    CUDART_CHECK( cudaEventRecord( evStop ) );
    CUDART_CHECK( cudaThreadSynchronize() );
    CUDART_CHECK( cudaGetLastError() );
    CUDART_CHECK( cudaEventElapsedTime( &etAlloc, evStart, evStop ) );

    CUDART_CHECK( cudaEventRecord( evStart ) );
    FreeBuffers<<<cBlocks,cThreads>>>( devicePointers );
    CUDART_CHECK( cudaEventRecord( evStop ) );
    CUDART_CHECK( cudaThreadSynchronize() );
    CUDART_CHECK( cudaGetLastError() );
```

```
CUDART_CHECK( cudaEventElapsedTime( &etFree, evStart, evStop ) );
*msPerAlloc = etAlloc / (double) (cBlocks*cThreads);
*msPerFree = etFree / (double) (cBlocks*cThreads);
Error:
    return status;
}
```

代码清单 5-4 展示了在 Amazon cg1.4xlarge 类型实例上 mallocAlloc.cu 的运行输出。很明确的是分配器可以为小型内存分配进行优化：64 字节的内存分配平均会使用 0.39 微秒来执行，而分配 12KB 内存会消耗 3 ~ 5 微秒。第一个结果（每次分配 155 微秒）在 500 线程块上分配 1MB 缓冲区，而每线程块使用一个线程。

代码清单 5-4 样例 mallocSpeed.cu 的输出

Microseconds per alloc/free (1 thread per block):									
alloc	free	alloc	free	alloc	free	alloc	free	alloc	free
154.93	4.57								

Microseconds per alloc/free (32-512 threads per block, 12K allocations):									
32	64	128	256	512	32	64	128	256	512
alloc	free	alloc	free	alloc	alloc	free	alloc	free	alloc
3.53	1.18	4.27	1.17	4.89	1.14	5.48	1.14	10.38	1.11

Microseconds per alloc/free (32-512 threads per block, 64-byte allocations):									
32	64	128	256	512	32	64	128	256	512
alloc	free	alloc	free	alloc	alloc	free	alloc	free	alloc
0.35	0.27	0.37	0.29	0.34	0.27	0.37	0.22	0.53	0.27



内核中使用 `malloc()` 函数分配的内存，必须使用函数 `free()` 释放，这时在主机端调用 `cudaFree()` 将不会成功。

5.2.3 查询全局内存数量

系统中全局内存的大小可以在 CUDA 初始化之前查询。

1. CUDA 运行时

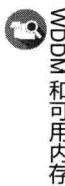
调用函数 `cudaGetDeviceProperties()`，检查 `cudaDeviceProp.totalGlobalMem`:

```
size_t totalGlobalMem; /*< Global memory on device in bytes */.
```

2. 驱动程序 API

调用驱动程序 API 函数。

```
CUresult CUDA API cuDeviceTotalMem(size_t *bytes, CUdevice dev);
```



Windows Vista 中引入的 Windows 显示驱动模型，改变了显示驱动程序的内存管理。它使视频内存的数据块可以在主机内存中被换入和换出，以满足渲染所需。因此，`cuDeviceTotalMem()` / `cudaDeviceProp::totalGlobalMem` 函数所返回的内存数量并不能准确地反映出物理内存数量。

5.2.4 静态内存分配

应用程序可以静态地分配全局内存，通过使用 `_device_` 关键字标记在内存声明中进行标记即可。这一内存是由 CUDA 驱动程序在模块加载时分配的。

1. CUDA 运行时

静态内存上的内存复制可以使用 `cudaMemcpyToSymbol()` 和 `cudaMemcpyFromSymbol()` 函数执行：

```
cudaError_t cudaMemcpyToSymbol(
    char *symbol,
    const void *src,
    size_t count,
    size_t offset = 0,
    enum cudaMemcpyKind kind = cudaMemcpyHostToDevice
);
cudaError_t cudaMemcpyFromSymbol(
    void *dst,
    char *symbol,
    size_t count,
    size_t offset = 0,
    enum cudaMemcpyKind kind = cudaMemcpyDeviceToHost
);
```

当调用函数 `cudaMemcpyToSymbol()` 或 `cudaMemcpyFromSymbol()` 时，不要将符号名称用引号括起来，即我们应该这样使用函数：

```
cudaMemcpyToSymbol(g_xOffset, poffsetx, Width*Height*sizeof(int));
```

而不是这样：

```
cudaMemcpyToSymbol("g_xOffset", poffsetx, ... );
```

这两种方式的调用都会工作，但是后者会为任何的符号名称编译（即使是没有定义的符号）。如果你想让编译器报告出非法符号错误，记住不要使用引号。

CUDA 运行时应用程序可以通过调用函数 `cudaGetSymbolAddress()` 查询关联到静态分配的内存上的指针。

```
cudaError_t cudaGetSymbolAddress( void **devPtr, char *symbol );
```

小心：传递给 CUDA 内核一个静态声明的设备内存分配的符号非常容易，但是这没有作用。你必须调用 `cudaGetSymbolAddress()` 函数并使用返回的指针。

2. 驱动程序 API

使用驱动程序 API 的开发者可以使用函数 cuModuleGetGlobal() 获取静态分配内存的指针。

```
CUresult CUDA API cuModuleGetGlobal(CUdeviceptr *dptr, size_t *bytes,
                                     CUmodule hmod, const char *name);
```

注意 cuModuleGetGlobal() 函数同时传回基指针和对象大小。如果我们不需要大小，开发者可以在 bytes 参数中传入 NULL。一旦得到这一指针，只需要传入 CUdeviceptr，内存就可以被内存复制或 CUDA 内核调用访问。

5.2.5 内存初始化 API

为了开发者方便，CUDA 提供了 1D 和 2D 内存初始化函数。由于它们使用内核实现，所以即使在没有指定流参数的情况下，它们也是异步的。然而，对必须在流内依次执行内存初始化的应用程序，这里还是有 *Async() 形式的函数接受一个流参数的。

1. CUDA 运行时

CUDA 运行时只支持以字节为单位的内存初始化：

```
cudaError_t cudaMemset(void *devPtr, int value, size_t count);
cudaError_t cudaMemset2D(void *devPtr, size_t pitch, int value,
                        size_t width, size_t height);
```

参数 pitch 指定了内存初始化中每行的字节数。

2. 驱动程序 API

驱动程序 API 支持不同大小的 1D 和 2D 内存初始化，在表 5-2 中我们予以总结。这些内存初始化函数接受一个复制目标指针、待赋的值和从地址开始写入的值的数量。参数步长是每一行的字节数（非元素数）。

表 5-2 内存初始化变种

操作数大小	1D	2D
8 位	cuMemsetD8	cuMemsetD2D8
16 位	cuMemsetD8	cuMemsetD2D8
32 位	cuMemsetD8	cuMemsetD2D8

```
CUresult CUDA API cuMemsetD8(CUdeviceptr dstDevice, unsigned char uc,
                               size_t N);
CUresult CUDA API cuMemsetD16(CUdeviceptr dstDevice, unsigned short
                                us, size_t N);
CUresult CUDA API cuMemsetD32(CUdeviceptr dstDevice, unsigned int ui,
                               size_t N);
CUresult CUDA API cuMemsetD2D8(CUdeviceptr dstDevice, size_t dstPitch,
                                 unsigned char uc, size_t Width, size_t Height);
```

```
CUresult CUDA API cuMemsetD2D16(CUdeviceptr dstDevice, size_t
dstPitch, unsigned short us, size_t Width, size_t Height);
CUresult CUDA API cuMemsetD2D32(CUdeviceptr dstDevice, size_t
dstPitch, unsigned int ui, size_t Width, size_t Height);
```

现在，在同一个应用程序中 CUDA 运行时和驱动程序 API 可以和平共存，CUDA 运行时开发者可以按照需要使用这些函数。参数 `unsigned char`, `unsigned short`, 和 `unsigned int` 只是指定了位的模式；为了使用其他类型数据填充全局内存区间，像 `float`, 使用 `volatile union` 强制转换 `float` 到 `unsigned int`。

5.2.6 指针查询

CUDA 跟踪所有的内存分配，并提供 API 使应用程序可以查询 CUDA 中的所有指针。函数库和插件可以在此基础之上使用不同的处理策略。

1. CUDA 运行时

函数 `cudaPointerGetAttributes()` 接受一个指针作为输入，并传回一个 `cudaPointerAttributes` 结构体，此结构体包含关于指针的信息：

```
struct cudaPointerAttributes {
    enum cudaMemoryType memoryType;
    int device;
    void *devicePointer;
    void *hostPointer;
}
```

当 UVA 启用，指针在进程内是唯一的，所以不会有相同的输入指针地址空间。当 UVA 未启用，输入的指针被假定为在当前设备的地址空间（表 5-3）。

表 5-3 cudaPointerAttributes 成员

结构体成员	描述
<code>enum cudaMemoryType memoryType;</code>	被输入指针引用的内存类型
<code>int device;</code>	如果 <code>memoryType==cudaMemoryTypeDevice</code> , 内存驻留于此设备上 如果 <code>memoryType==cudaMemoryTypeHost</code> , 此设备的上下文被用来分配内存
<code>void *devicePointer;</code>	与分配关联的设备指针，如果当前设备不能访问这块内存，这一结构体成员被设置为 NULL
<code>void *hostPointer;</code>	与分配关联的主机指针，如果内存不是映射锁页内存，此结构体成员被设置为 NULL

2. 驱动程序 API

使用函数 `cuMemGetAddressRange()`，开发者可以查询给定设备指针驻留的地址空间区间。

```
CUresult CUDA API cuMemGetAddressRange(CUdeviceptr *pbase, size_t
*psize, CUdeviceptr dptr);
```

函数接受一个设备指针作为输入，返回包含这个设备指针的基地址和分配空间大小。

伴随 CUDA 4.0 中新加的 UVA 功能，开发者可以使用函数 cuPointerGetAttribute() 查询 CUDA 以获取更多的关于一个地址的信息。

```
CUresult CUDA API cuPointerGetAttribute(void *data, CUpointer_
attribute attribute, CUdeviceptr ptr);
```

这一函数接受一个设备指针作为输入，并传回属性参数相关的信息，如表 5-4 所示。注意对统一寻址，使用 CU_POINTER_ATTRIBUTE_DEVICE_POINTER 或 CU_POINTER_ATTRIBUTE_HOST_POINTER 中的任意一个会传回与所传入相同的指针值。

表 5-4 cuPointerAttribute 使用方法

枚举值	传回
CU_POINTER_ATTRIBUTE_CONTEXT	分配指针或注册指针的 CUcontext
CU_POINTER_ATTRIBUTE_MEMORY_TYPE	与指针的内存类型关联的 cuMemoryType: CU_MEMORYTYPE_HOST 代表主机内存，CU_MEMORYTYPE_DEVICE 代表设备内存，CU_MEMORYTYPE_UNIFIED 代表统一内存
CU_POINTER_ATTRIBUTE_DEVICE_POINTER	Ptr 假设为映射主机内存；data 指向 void*，接受一个与分配关联的设备指针。如果当前设备不能访问内存，结构体成员被设置为 NULL
CU_POINTER_ATTRIBUTE_HOST_POINTER	Ptr 假定为设备内存；data 指向一个 void*，接受一个与分配相关的主机指针。如果分配不是映射锁页内存，这一结构体成员被设置为 NULL

3. 内核查询

在 SM 2.x (费米架构) 及之后的硬件中，开发者可以查询给定的指针是否指向了全局空间。内置函数 __isGlobal()，如下：

```
unsigned int __isGlobal( const void *p );
```

返回值为 1 时说明指针指向全局内存，返回 0 则没有指向。

5.2.7 点对点内存访问

在特定的情况下，SM 2.0 和之后的硬件可以映射属于其他有相同能力的 GPU 中的内存。在以下诸情况可以做到这一点：

- UVA 必须启用。
- 映射双方的 GPU 都必须是费米架构，并基于相同的芯片。
- 双方 GPU 必须在同一个 I/O 集线器上。

由于点对点映射本质上属于一个多 GPU 特性，所以我们在多 GPU 一章（第 9.2 节）提供了更多细节。

5.2.8 读写全局内存

CUDA 内核可以使用标准 C 语言读写全局内存，例如指针间接寻址（操作符 *、操作符 ->）与数组索引（操作符 []）。下面有一个带模板的简单内核，用于写入一个常量到内存。

```
template<class T>
__global__ void
GlobalWrites( T *out, T value, size_t N )
{
    for ( size_t i = blockIdx.x*blockDim.x+threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x ) {
        out[i] = value;
    }
}
```

这个内核在任何输入下都会运行正确：任意的元素大小，任意的线程块大小，任意的网格大小。这块代码着重于演示目的，而不是追求最大的性能。那些使用更多的寄存器与内层循环中操纵多个值的 CUDA 内核执行的较快，但对一些线程块和网格配置，性能会刚刚好。尤其是，假如基地址和线程块大小被正确指定，硬件会执行合并内存事务，最大化内存带宽。

5.2.9 合并限制

为了达到读写数据时的最佳性能，CUDA 内核必须执行内存事务合并处理。任何不满足合并所需的全套标准的内存事务称为“未合并的”。未合并内存事务的性能惩罚由 2 ~ 8 倍不等。在最近的硬件上，合并内存事务对性能的影响显著减少，见表 5-5。

表 5-5 非合并内存访问的带宽惩罚

芯 片	惩 罚	芯 片	惩 罚
SM 1.0-1.1	6x	SM 2.x (ECC 禁用)	20%
SM 1.2	2x	SM 2.x (ECC 启用)	2x

事务在每一个线程束的基础上被合并。为了由束执行内存读写事务的合并，一些简化的标准必须被满足：

- 字至少为 32 位。读写字节或 16 位字的事务总是非合并的。
- 束上的线程访问的地址是连续并递增的（即，依线程 ID 偏移）。
- 束的基地址（束中第一个线程访问的地址）需按表 5-6 对齐。

表 5-6 合并的对齐标准

字 大 小	对 齐	字 大 小	对 齐
8 位	①	64 位	128 字节
16 位	①	128 位	256 字节
32 位	64 字节		

① 8 位和 16 位字内存访问是始终未合并的。

函数 cuMemAllocPitch() 的参数 ElementSizeBytes 旨在适应大小限制，它会指定应用程序想要访问的内存大小（单位字节），所以步长不仅会保证对分配中给定行的合并内存事务，也会保证其他行的事务合并。

本书中的大多数内核执行合并的内存事务，是建立在输入的地址恰好对齐的基础上。在全局内存事务的处理方式上，英伟达提供了更多分别针对不同架构的信息，详见下文。

1. SM 1.x (特斯拉架构)

如前所述，SM1.0 和 SM1.1 硬件要求线程束中的每一个线程按顺序访问临近的内存地址。SM1.2 和 SM1.3 硬件对合并限制已经不是特别严格。为了发出合并内存请求，分割 32 个线程构成的线程束为两个“半束”：0-15 号线程与 16-31 号线程。为了满足在每个半束上的内存请求，硬件会执行以下算法：

- 1) 找到具有最小线程号的活动线程，并查找包含线程请求地址的内存段。内存段的大小由字大小决定：1 字节大小字请求会需要 32 字节的段；2 字节请求需要 64 字节段；其他的请求会需要 128 字节段。
- 2) 找到所有请求地址位于相同段的其他活动线程。
- 3) 如果可能的话，减少段事务大小，到 64 字节或 32 字节。
- 4) 完成事务并标记刚接受服务的那些线程为不活动的。
- 5) 重复 1 ~ 4 步，直到所有的半束中的线程请求全部满足。

尽管同 SM1.0 和 SM1.1 限制相比，这些需求变得更加灵活，但是高效的合并操作始终需要大量的局部性特性。在实践中，这一弱化的合并限制意味着束中的线程可以在必要时使用更少的内存段置换输入。

2. SM 2.x (费米架构)

SM2.x 和之后的硬件包含一级和二级缓存。二级缓存为整个芯片服务；一级缓存是在一个 SM 内可见的，可以设置为 16KB 或 48KB 大小。缓存行为 128 字节宽，对应设备内存上的 128 字节对齐段。当访问的是同时缓存于一级与二级缓存的内存时，使用 128 字节内存事务处理，但是当内存访问的是只在二级缓存中被缓存的对象，该访问只使用 32 字节内存事务处理。使用二级缓存可以因此减少过度读取，例如，分散模式的内存访问时。

硬件可以指定每一条指令访问的全局内存的可缓存性。默认的，编译器发出同时使用一级和二级缓存的内存访问指令（-Xptxas-dlcm=ca）。这可以通过指定 -Xptxas-dlcm=cg 改变为只使用二级缓存。不使用一级缓存，而只使用二级缓存的内存访问只采用 32 位内存事务处理，这会提升执行分散内存访问的应用程序的缓存利用率。

使用声明为 volatile 的指针读取内存会使所有的缓存被丢弃，而且为数据重新缓存。这一用法在轮询主机内存位置的情况下非常有用。表 5-7 总结了如何将一个线程束的内存请求分解为 128 字节缓存行请求。

表 5-7 SM 2.x 缓存行请求

字大小	128 字节请求	线程单元	字大小	128 字节请求	线程单元
8位	1	束	64位	2	半束
16位	1	束	128位	4	四分之一束
32位	1	束			

 在 SM 2.x 和更高的架构上，束内线程可以以任何顺序访问任何字，包括冗余字。

3. SM 3.x (开普勒)

SM 3.x 的二级缓存架构与 SM 2.x 中的架构相同。SM 3.x 不在一级缓存中缓存全局内存。在 SM 3.5 中，全局内存可以通过纹理缓存访问（每 SM 48KB 大小），使用 const restricted 指针访问；或者使用 sm_35_intrinsics 中的内置函数 __ldg()。当纹理操作直接来自设备内存，不去访问可能被其他途径并发访问的内存十分重要，因为二级缓存的存在，纹理缓存不会始终保持与二级缓存的一致性。

5.2.10 验证实验：内存峰值带宽

随同本书的源代码包含了一些验证程序，旨在以操作数大小、循环展开因素和线程块大小的最好组合最大化给定 GPU 的带宽。重写前文的 GlobalWrites 代码，变为一个接受附加参数 n（内层循环中执行写操作的数量）的模板，见代码清单 5-5。

代码清单 5-5 内核 GlobalWrites

```
template<class T, const int n>
__global__ void
GlobalWrites( T *out, T value, size_t N )
{
    size_t i;
    for ( i = n*blockIdx.x*blockDim.x+threadIdx.x;
          i < N-n*blockDim.x*gridDim.x;
          i += n*blockDim.x*gridDim.x ) {
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            out[index] = value;
        }
    }
    // to avoid the (index<N) conditional in the inner loop,
    // we left off some work at the end
    for ( int j = 0; j < n; j++ ) {
        size_t index = i+j*blockDim.x;
        if ( index<N ) out[index] = value;
    }
}
```

代码清单 5-6 中给出的函数 ReportRow()，通过调用模板函数 BandwidthWrites（未给出）写入输出的其中一行，给出了对给定类型、网格和线程块大小的带宽情况。

代码清单 5-6 函数 ReportRow()

```

template<class T, const int n, bool bOffset>
double
ReportRow( size_t N,
           size_t threadStart,
           size_t threadStop,
           size_t cBlocks )
{
    int maxThreads = 0;
    double maxBW = 0.0;
    printf( "%d\t", n );
    for ( int cThreads = threadStart;
          cThreads <= threadStop;
          cThreads *= 2 ) {
        double bw;
        bw = BandwidthWrites<T,n,bOffset>( N, cBlocks, cThreads );
        if ( bw > maxBW ) {
            maxBW = bw;
            maxThreads = cThreads;
        }
        printf( "%.2f\t", bw );
    }
    printf( "%.2f\t%d\n", maxBW, maxThreads );
    return maxBW;
}

```

参数 `threadStart` 和 `threadStop` 通常为 32 和 512, 32 为线程束大小并且是线程块中可调度到机器的最小线程数。模板参数 `bOffset` 指定 `BandwidthWrites` 是否需要在基指针上偏移, 这是由于所有的内存事务会被合并。如果程序在调用时包含命令行选项 `--uncoalesced`, 程序会在偏移指针上执行带宽测量。

注意, 受制于 `sizeof(T)`, 当内核的 `n` 高于一定值时, 会出现一个明显的性能退化。因为内层循环的临时变量数增长太多, 将会出现溢出寄存器的情况。

表 5-8 中总结的 5 个应用程序实现了这一策略, 它们测量在不同操作数大小 (8 位、16 位、32 位、64 位和 128 位)、不同线程块大小 (32、64、128、256、和 512) 和不同展开循环 (1 ~ 16) 下的内存传输带宽。CUDA 硬件不一定会对所有的这些参数作出反应。例如, 许多参数设置使 GK104 带宽通过纹理操作达到 140GB/s, 但是只有操作数在 32 位以上时这才会生效。无论如何, 对于指定的工作负载和硬件, 验证程序突出了起重要作用的那个参数。同样, 对于小的操作数大小, 验证程序突出了循环展开是怎样帮助提升性能的 (不是所有的应用程序都可以被重构来读取更大的操作数)。

表 5-8 内存带宽测试

验证程序的文件名	内存事务	验证程序的文件名	内存事务
<code>globalCopy.cu</code>	一次读, 一次写	<code>globalReadTex.cu</code>	一次通过纹理操作的读
<code>globalCopy2.cu</code>	两次读, 一次写	<code>globalWrite.cu</code>	一次写
<code>globalRead.cu</code>	一次读		

代码清单 5-7 给出了运行在 GeForce GTX 680 GPU 上的 `globalRead.cu` 的输出示例。输

出按照操作数大小进行了分组，从单字节，到2、4、8、16字节。每一组最左侧的列给出了循环展开数。32到512线程数的线程块的传输带宽在每列中依次给出，maxBW和maxThreads列分别给出了最大带宽和对应的线程块大小。

代码清单5-7 样例globalRead.cu的输出

Running globalRead.cu microbenchmark on GeForce GTX 680							
Using coalesced memory transactions							
Operand size: 1 byte							
Input size: 16M operands							
Unroll	32	64	128	256	512	maxBW	maxThreads
1	9.12	17.39	30.78	30.78	28.78	30.78	128
2	18.37	34.54	56.36	53.53	49.33	56.36	128
3	23.55	42.32	61.56	60.15	52.91	61.56	128
4	21.25	38.26	58.99	58.09	51.26	58.99	128
5	25.29	42.17	60.13	58.49	52.57	60.13	128
6	25.68	42.15	59.93	55.42	47.46	59.93	128
7	28.84	47.03	56.20	51.41	41.41	56.20	128
8	29.88	48.55	55.75	50.68	39.96	55.75	128
9	28.65	47.75	56.84	51.17	37.56	56.84	128
10	27.35	45.16	52.99	46.30	32.94	52.99	128
11	22.27	38.51	48.17	42.74	32.81	48.17	128
12	23.39	40.51	49.78	42.42	31.89	49.78	128
13	21.62	37.49	40.89	34.98	21.43	40.89	128
14	18.55	32.12	36.04	31.41	19.96	36.04	128
15	21.47	36.87	39.94	33.36	19.98	39.94	128
16	21.59	36.79	39.49	32.71	19.42	39.49	128
Operand size: 2 bytes							
Input size: 16M operands							
Unroll	32	64	128	256	512	maxBW	maxThreads
1	18.29	35.07	60.30	59.16	56.06	60.30	128
2	34.94	64.39	94.28	92.65	85.99	94.28	128
3	45.02	72.90	101.38	99.02	90.07	101.38	128
4	38.54	68.35	100.30	98.29	90.28	100.30	128
5	45.49	75.73	98.68	98.11	90.05	98.68	128
6	47.58	77.50	100.35	97.15	86.17	100.35	128
7	53.64	81.04	92.89	87.39	74.14	92.89	128
8	44.79	74.02	89.19	83.96	69.65	89.19	128
9	47.63	76.63	91.60	83.52	68.06	91.60	128
10	51.02	79.82	93.85	84.69	66.62	93.85	128
11	42.00	72.11	88.23	79.24	62.27	88.23	128
12	40.53	69.27	85.75	76.32	59.73	85.75	128
13	44.90	73.44	78.08	66.96	41.27	78.08	128
14	39.18	68.43	74.46	63.27	39.27	74.46	128
15	37.60	64.11	69.93	60.22	37.09	69.93	128
16	40.36	67.90	73.07	60.79	36.66	73.07	128
Operand size: 4 bytes							
Input size: 16M operands							
Unroll	32	64	128	256	512	maxBW	maxThreads
1	36.37	67.89	108.04	105.99	104.09	108.04	128
2	73.85	120.90	139.91	139.93	136.04	139.93	256
3	62.62	109.24	140.07	139.66	138.38	140.07	128
4	56.02	101.73	138.70	137.42	135.10	138.70	128
5	87.34	133.65	140.64	140.33	139.00	140.64	128
6	100.64	137.47	140.61	139.53	127.18	140.61	128
7	89.08	133.99	139.60	138.23	124.28	139.60	128
8	58.46	103.09	129.24	122.28	110.58	129.24	128
9	68.99	116.59	134.17	128.64	114.80	134.17	128
10	54.64	97.90	123.91	118.84	106.96	123.91	128

```

11      64.35   110.30  131.43   123.90   109.31   131.43   128
12      68.03   113.89  130.95   125.40   108.02   130.95   128
13      71.34   117.88  123.85   113.08   76.98    123.85   128
14      54.72   97.31   109.41   101.28   71.13    109.41   128
15      67.28   111.24  118.88   108.35   72.30    118.88   128
16      63.32   108.56  117.77   103.24   69.76    117.77   128
Operand size: 8 bytes
Input size: 16M operands
          Block Size
Unroll 32     64     128    256     512     maxBW  maxThreads
1       74.64  127.73  140.91  142.08  142.16  142.16  512
2       123.70 140.35  141.31  141.99  142.42  142.42  512
3       137.28 141.15  140.86  141.94  142.63  142.63  512
4       128.38 141.39  141.85  142.56  142.00  142.56  256
5       117.57 140.95  141.17  142.08  141.78  142.08  256
6       112.10 140.62  141.48  141.86  141.95  141.95  512
7       85.02   134.82  141.59  141.50  141.09  141.59  128
8       94.44   138.71  140.86  140.25  128.91  140.86  128
9       100.69  139.83  141.09  141.45  127.82  141.45  256
10      92.51   137.76  140.74  140.93  126.50  140.93  256
11      104.87 140.38  140.67  136.70  128.48  140.67  128
12      97.71   138.62  140.12  135.74  125.37  140.12  128
13      95.87   138.28  139.90  134.18  123.41  139.90  128
14      85.69   134.18  133.84  131.16  120.95  134.18  64
15      94.43   135.43  135.30  133.47  120.52  135.43  64
16      91.62   136.69  133.59  129.95  117.99  136.69  64
Operand size: 16 bytes
Input size: 16M operands
          Block Size
Unroll 32     64     128    256     512     maxBW  maxThreads
1       125.37 140.67  141.15  142.06  142.59  142.59  512
2       131.26 141.95  141.72  142.32  142.49  142.49  512
3       141.03 141.65  141.63  142.43  138.44  142.43  256
4       139.90 142.70  142.62  142.20  142.84  142.84  512
5       138.24 142.08  142.18  142.79  140.94  142.79  256
6       131.41 142.45  142.32  142.51  142.08  142.51  256
7       131.98 142.26  142.27  142.11  142.26  142.27  128
8       132.70 142.47  142.10  142.67  142.19  142.67  256
9       136.58 142.28  141.89  142.42  142.09  142.42  256
10      135.61 142.67  141.85  142.86  142.36  142.86  256
11      136.27 142.48  142.45  142.14  142.41  142.48  64
12      130.62 141.79  142.06  142.39  142.16  142.39  256
13      107.98 103.07  105.54  106.51  107.35  107.98  32
14      103.53 95.38   96.38   98.34   102.92  103.53  32
15      89.47   84.86   85.31   87.01   90.26   90.26   512
16      81.53   75.49   75.82   74.36   76.91   81.53   32

```

GeForce GTX 680 显卡最高传输速度可达 140GB/s，代码清单 5-7 让我们更加清楚一点，当在 SM 3.0 上读取 8 位或 16 位字时，全局加载不应该是我们选择程序执行的方式。字节最高传输速度 60GB/s，16 位字最高可达 101GB/s^①。对 32 位操作数，为了达到最大带宽，我们需要一次 2 倍的循环展开和至少每线程块 256 线程。

这一验证程序可以帮助开发者优化他们的带宽受限型应用程序。选择一个与你的应用程序最相似的内存访问模式，在目标 GPU 上运行验证程序，或者，可能的话，调整验证程序以与真实的工作负载相匹配，以获得最佳的参数。

^① 纹理操作表现得更好，读者可以运行 globalReadTex.cu 证实一下。

5.2.11 原子操作

SM 1.x 开始支持原子操作，但是这项操作离谱的慢。全局内存中的原子操作在 SM 2.x (费米架构) 硬件中被提升，并且在 SM 3.x (开普勒架构) 中被大幅度全面提升。

大多数的原子操作，例如 `atomicAdd()`，使代码可以使用自主导引方式替换归约（归约通常需要共享内存和同步）。SM 3.x 硬件发布之前，这种编程方式会招致巨大的性能退化，因为开普勒之前的架构在处理竞争内存位置的问题上不够高效（即：许多的 GPU 线程同时在同一内存位置做原子操作）。



注意 由于原子操作由 GPU 内存控制器实现，它们只在本地设备内存位置工作，在本书撰写时，尝试在非本地 GPU 或主机内存上执行原子操作是不可行的。

原子操作与同步

除了自主引导，原子操作同样可以使用在线程块之间的同步。CUDA 硬件支持同步的主要抽象：“对比和交换”（或 CAS，compare and swap）。在 CUDA 上，对比和交换（也可为 compare and exchange——即，x86 中的 CMPXCHG 指令）被定义如下：

```
int atomicCAS( int *address, int expected, int value);⊖
```

这一函数从 `address` 读取的值放入 `old`，计算 (`old ==expected ? value : old`)，结果传回 `address` 并返回 `old` 值。换句话说，内存地址保持不变，除非它与调用者指定的期望值相等，这种情况下，内存地址会更新为 `value`。

可以基于 CAS 建立一个称为“自旋锁”的简单临界区 (critical section)，如下所示：

```
void enter_spinlock( int *address )
{
    while atomicCAS( address, 0, 1 );
}
```

假设自旋锁的值被初始化为 0，当函数 `atomicCAS()` 执行，`while` 循环会不断迭代直到自旋锁的值为 0。这执行之后，`*address` 自动变为 1 (函数 `atomicCAS()` 的第三个参数)，并且其他的尝试得到这一临界区自旋锁的线程都必须等到这一值再次变为 0。

拥有自旋锁的线程可以放弃这个自旋锁，通过自动换回 0 即可：

```
void leave_spinlock( int *address )
{
    atomicExch( m_p, 0 );
}
```

在 CPU 上，对比和交换指令用于实现所有形式的同步。操作系统使用这些指令（一些时候与内核级线程上下文转换代码联合使用）来实现高级的同步原语。CAS 同样使用在直接实

[⊖] 函数 `atomicCAS()` 对无符号和 64 位变量同样有效。

现“无锁”队列和其他的数据结构中。

CUDA 执行模型，在使用全局内存的原子操作进行同步时，施加了限制。不同于 CPU 线程，一次内核启动内的一些 CUDA 线程可能要等到同一内核中其他线程退出后才开始执行。在 CUDA 硬件上，每一个 SM 可切换一定数量线程块的上下文，所以对多于 MaxThreadBlocksPerSM*NumSMs 数量线程块的内核启动，都需要第一批线程块退出，才能让更多的线程块开始执行。因此，开发者不假定给定内核中的所有线程都是活动的是十分重要的。

除此之外，上面给出的函数 `enter_spinlock()` 在使用块内同步时很容易导致死锁[⊖]，说明它不适合用在这种情况下。实际上，CUDA 的硬件支持很多更好的块内线程间通信和同步的方式（可以分别基于共享内存和函数 `_syncthreads()`）。

代码清单 5-8 给出了 `cudaSpinlock` 类的实现，使用了上述的算法，并且也受上述的限制。

代码清单 5-8 `cudaSpinlock` 类

```
class cudaSpinlock {
public:
    cudaSpinlock( int *p );
    void acquire();
    void release();
private:
    int *m_p;
};

inline __device__ cudaSpinlock::cudaSpinlock( int *p )
{
    m_p = p;
}

inline __device__ void
cudaSpinlock::acquire()
{
    while ( atomicCAS( m_p, 0, 1 ) );
}

inline __device__ void
cudaSpinlock::release()
{
    atomicExch( m_p, 0 );
}
```

样例 `spinlockReduction.cu` 给出了 `cudaSpinlock` 的使用。程序要计算一个 `double` 数组的和，先在每一个线程块中基于共享内存执行一次归约，之后使用自旋锁在计算总和时进行同步。代码清单 5-9 给出了这个样例的 `SumDouble` 函数。注意累加部分和的操作是怎样只由每个线程块的 0 号线程执行的。

[⊖] 推荐用法是让每个线程块中某个线程尝试去获取自旋锁。否则，分支的代码执行会趋向于死锁。

代码清单 5-9 函数 SumDouble

```

__global__ void
SumDoubles(
    double *pSum,
    int *spinlock,
    const double *in,
    size_t N,
    int *acquireCount )
{
    SharedMemory<double> shared;
    cudaSpinlock globalSpinlock( spinlock );

    for ( size_t i = blockIdx.x*blockDim.x+threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x ) {
        shared[threadIdx.x] = in[i];
        __syncthreads();
        double blockSum = Reduce_block<double,double>();
        __syncthreads();

        if ( threadIdx.x == 0 ) {
            globalSpinlock.acquire();
            *pSum += blockSum;
            __threadfence();
            globalSpinlock.release();
        }
    }
}

```

5.2.12 全局内存的纹理操作

对于不能方便地满足合并访问限制的应用程序，纹理映射硬件是一个令人满意的选择。硬件支持来自全局内存的纹理操作（通过 `cudaBindTexture()`/`cuTexRefSetAddress()` 函数），这不会拥有比合并全局读写更快的峰值执行性能，但是对不太规则的访问会有更高的性能。纹理缓存资源同样同其他缓存资源独立。在包含 `TEX` 指令的内核调用前^①，驱动程序会有一个软件一致性机制使纹理缓存无效。查看第 10 章获取更多细节。

`SM 3.x` 硬件添加了通过纹理缓存层读取全局内存的能力，这不需要设置和绑定纹理引用。这一功能可使用标准 C++ 语言构建：关键字 `const restrict`。另一方法是使用在 `sm_35_intrinsics.h` 头文件中定义的 `__ldg()` 内置函数。

5.2.13 ECC (纠错码)

`SM 2.x` 以及之后的服务器级特斯拉系列 GPU 拥有运行中纠错功能。以少量的内存（一些内存被使用在记录冗余信息）和稍低的带宽作为交换，启用 `ECC` 的 GPU 便可以自动地纠正单个比特位（single-bit）错误并且可以报告双位（double-bit）错误。

`ECC` 有以下特点：

- 减少大约 12.5% 的可用内存量。例如，在亚马逊 EC2 中的 `cg1.4xlarge` 实例上，内存

① `TEX` 是一个 SASS 微码指令助记符，执行纹理读取。

数量从 3071MB 减少到了 2678MB。

- 它使上下文同步的成本变得更高。
- 非合并内存处理的成本同样更高。

ECC 可以使用 nvidia-smi 命令行工具（小节 4.4）或 NVML（英伟达管理库）启用或禁用。

当发现一个无法纠正的 ECC 错误，同步的错误报告机制会返回 cudaErrorECCUncorrectable（CUDA 运行时）和 CUDA_ERROR_ECC_UNCORRECTABLE（驱动程序 API）。

5.3 常量内存

常量内存是为多个线程只读类型的广播操作而优化的内存。正如这个名字所传达给我们的一样，编译器使用常量内存存储不能被轻易计算得出的常量或直接被编译为机器码的常量。虽然常量内存驻留在设备内存上，但是机器使用独特的指令来访问。其间，GPU 会使用一个特殊的“常量缓存”来访问它。

常量的编译器拥有一个 64KB 的可用内存，编译器可自由地使用它。开发者同样有另一个 64KB 的可用内存空间，可以用关键字 `_constant_` 声明使用。这一限制被应用在每一个模块（驱动程序 API 应用程序）或每一个文件（CUDA 运行时应用程序）。

很多人会天真地想，关键字 `_constant_` 与 C/C++ 中的关键字 `const` 应该是相似的，即在声明后不可更改。但是 `_constant_` 关键字声明的内存内容是可以被更改的。通过内存复制或通过查询 `_constant_` 内存指针并让内核写入常量内存均可以达到更改常量内存的目的。CUDA 内核不应该写入正在被访问的常量内存，因为，内核执行时常量缓存并不是始终与其他内存结构保持一致。

5.3.1 主机与设备常量内存

使用预定义宏 `_CUDA_ARCH_` 来支持主机与设备的常量内存复制，这会方便 CPU 和 GPU 对内存的读取。Mark Harris 曾描述过这个使用宏的理念[⊖]。

```
_constant_ double dc_vals[2] = { 0.0, 1000.0 };
const double hc_vals[2] = { 0.0, 1000.0 };

_device_ __host__ double f(size_t i)
{
#ifdef __CUDA_ARCH__
    return dc_vals[i];
#else
    return hc_vals[i];
#endif
}
```

5.3.2 访问常量内存

除了使用 C/C++ 操作符隐式访问常量内存，开发者可以从常量内存复制数据或复制数据

⊖ <http://bit.ly/OpMdN5>。

到常量内存，甚至可以查询常量内存的指针。

1. CUDA 运行时

CUDA 运行时应用程序可以使用函数 `cudaMemcpyToSymbol()` 和 `cudaMemcpyFromSymbol()` 分别复制数据到常量内存和从常量内存复制数据。常量内存的指针可以使用 `cudaGetSymbolAddress()` 函数查询。

```
cudaError_t cudaGetSymbolAddress( void **devPtr, char *symbol );
```

这一指针可以用在内核写入常量内存的操作，但开发者必须注意不要在其他内核读取常量内存时写入数据。

2. 驱动程序 API

驱动程序 API 应用程序可以使用函数 `cuModuleGlobal()` 查询常量内存的设备指针。由于驱动程序 API 不包括 CUDA 运行时中的语言集成特性，驱动程序 API 不包括像 `cudaMemcpyToSymbol()` 这样的特殊内存复制函数。应用程序必须使用函数 `cuModuleGetGlobal()` 查询地址，之后调用 `cuMemcpyHtoD()` 或 `cuMemcpyDtoH()`。

内核使用的常量内存数量可以使用函数 `cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_CONSTANT_SIZE_BYTES)` 获取。

5.4 本地内存

本地内存包括 CUDA 内核中每一个线程的栈，在实现以下的目的中使用：

- 实现应用程序二进制接口（ABI）——这是一个调用惯例。
- 容纳寄存器溢出的数据。
- 保存编译器不能解析其索引的数组。

在早期的 CUDA 硬件中，任何使用本地内存的操作都是一次“死亡之吻”。它使机器的速度大幅减慢，所以提倡开发者动用一切手段避免使用本地内存。随着费米架构引入一级缓存，只要本地内存传输仅限于在一级缓存中进行[⊖]，这一性能问题就变得不那么严重。

为了让编译器报告给定内核需要的本地内存数量，开发者可以使用 nvcc 选项：`-Xptxas -v,abi=no`。在运行时中，内核使用的本地内存数量可以使用函数查询：

```
cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_LOCAL_SIZE_BYTES);
```

英伟达的 Paulius Micikevicius 介绍过一个关于确定本地内存是否对性能产生了影响和解决这个问题的方法[⊖]。寄存器溢出会导致两项开销：指令数增加和内存传输数量增加。

[⊖] 一级缓存在每个 SM 上独立存在，与共享内存有着同样的物理内存实现。

[⊖] <http://bit.ly/ZAeHc5>。

一级和二级缓存性能计数器可以用来确定内存传输是否影响到了性能。这里有一些在这种情况下提升性能的策略。

- 在编译时，对 `-maxregcount` 指定一个较高的限制。通过增加线程中可用的寄存器数量，指令数和内存传输数量均会减少。当内核使用 PTXAS 在线编译时，`_launch_bounds_` 指令可以用来调整这一参数。
- 为全局内存使用无缓存加载，例如 `nvcc -Xptxas -dlcm=cg`。
- 增加一级缓存到 48KB。（调用函数 `cudaFuncSetCacheConfig()` 或 `cudaDeviceSetCacheConfig()`。）

当启动一个内核，如果使用超过了默认的本地内存分配数量，那么在内核可以启动前，CUDA 驱动程序必须分配一块新的本地内存缓冲区。因此，内核启动可能需要额外的时间；可能会导致预想不到的 GPU/CPU 同步；并且，如果驱动程序不能为本地内存分配缓冲区，内核启动会以失败告终^①。默认的，CUDA 驱动程序会在内核启动后释放这些分配的本地内存。这一行为可以在函数 `cuCtxCreate()` 中指定标志 `CU_CTX_RESIZE_LMEM_TO_MAX` 予以禁止，或者同样可以使用标志 `cudaDeviceLmemResizeToMax` 调用函数 `cudaSetDeviceFlags()` 达到同样目的。

建立一个函数模板描述寄存器溢出时发生的“性能突然退化”并不困难。代码清单 5-10 中的内核模板 `GlobalCopy` 实现了一个简单的内存复制程序，使用了本地数组 `temp` 来中转全局内存引用。模板参数 `n` 指定了 `temp` 中的元素数量，这也决定了内存复制的内循环中执行加载 / 存储的数量。

如果我们快速的浏览由编译器提交的 SASS 微码，我们会证实，编译器可以在寄存器中保存 `temp` 直到 `n` 变得非常大。

代码清单 5-10 GlobalCopy 内核

```
template<class T, const int n>
__global__ void
GlobalCopy( T *out, const T *in, size_t N )
{
    T temp[n];
    size_t i;
    for ( i = n*blockIdx.x*blockDim.x+threadIdx.x;
          i < N-n*blockDim.x*gridDim.x;
          i += n*blockDim.x*gridDim.x ) {
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            temp[j] = in[index];
        }
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            out[index] = temp[j];
        }
    }
    // to avoid the (index<N) conditional in the inner loop,
    // we left off some work at the end
}
```

^① 在运行时中，由于大多数的资源是预分配的，无法分配本地内存是少数几个可以使内核启动失败的情况。

```

for ( int j = 0; j < n; j++ ) {
    for ( int j = 0; j < n; j++ ) {
        size_t index = i+j*blockDim.x;
        if ( index<N ) temp[j] = in[index];
    }
    for ( int j = 0; j < n; j++ ) {
        size_t index = i+j*blockDim.x;
        if ( index<N ) out[index] = temp[j];
    }
}

```

代码清单 5-11 展示了在 GK104 GPU 上来自 globalCopy.cu 的输出的一部分摘要：64 位操作数的复制性能。由寄存器溢出导致的性能退化在循环展开到 12 时变得尤为明显，这时，传输带宽由 117GB/s 锐减到不足 90GB/s，并在循环展开到 16 时已退化到 30GB/s 以下。

代码清单 5-11 globalCopy.cu 输出 (64 位)

	Block Size							
Unroll	32	64	128	256	512	maxBW	maxThreads	
1	75.57	102.57	116.03	124.51	126.21	126.21	512	
2	105.73	117.09	121.84	123.07	124.00	124.00	512	
3	112.49	120.88	121.56	123.09	123.44	123.44	512	
4	115.54	122.89	122.38	122.15	121.22	122.89	64	
5	113.81	121.29	120.11	119.69	116.02	121.29	64	
6	114.84	119.49	120.56	118.09	117.88	120.56	128	
7	117.53	122.94	118.74	116.52	110.99	122.94	64	
8	116.89	121.68	119.00	113.49	105.69	121.68	64	
9	116.10	120.73	115.96	109.48	99.60	120.73	64	
10	115.02	116.70	115.30	106.31	93.56	116.70	64	
11	113.67	117.36	111.48	102.84	88.31	117.36	64	
12	88.16	86.91	83.68	73.78	58.55	88.16	32	
13	85.27	85.58	80.09	68.51	52.66	85.58	64	
14	78.60	76.30	69.50	56.59	41.29	78.60	32	
15	69.00	65.78	59.82	48.41	34.65	69.00	32	
16	65.68	62.16	54.71	43.02	29.92	65.68	32	

表 5-9 总结了循环展开中内核的寄存器和本地内存使用情况。复制的性能退化与本地内存的使用情况有关。在这种情况下，每一个线程总是在内层循环中溢出；推测来看，当其中的一些线程在溢出时，性能不会退化的太多（例如执行一个分支的代码路径）。

表 5-9 globalCopy 寄存器和本地内存使用情况

循环展开因子	寄存器数目	本地内存(字节)	循环展开因子	寄存器数目	本地内存(字节)
1	20	None	9	62	None
2	19	None	10	63	None
3	26	None	11	63	None
4	33	None	12	63	16
5	39	None	13	63	32
6	46	None	14	63	60
7	53	None	15	63	96
8	58	None	16	63	116

5.5 纹理内存

在 CUDA 中，纹理内存的实现包含两个方面：CUDA 数组和纹理引用 / 表面引用[⊖]。前者包含物理内存分配，后者包含一个视图，使用这个视图可以读写 CUDA 数组。CUDA 数组只是一个使用存储配置的无类型“位集合”，其内存结构为 1D、2D 和 3D 访问做了特别优化。一个纹理引用包含有 CUDA 数组怎样编址和数组的内容怎样被解释方面的信息。

当使用纹理引用来从 CUDA 数组读取内容，硬件使用独立的，只读缓存来解析内存引用。当内核执行时，纹理缓存不保持与其他内存子系统的一致性，所以千万不要使用纹理引用对将在内核上使用的内存进行别名操作（这块缓存在内核启动之间是无效的）。

在 SM 3.5 硬件中，通过纹理的读操作可以通过开发者使用 `const restricted` 关键词显式请求。关键字 `restricted` 只会做出刚刚所述的“不进行别名操作”的保证，这样该内存不会被内核以任何方式引用。当使用表面引用读写 CUDA 数组，内存数据传输与全局加载与存储使用相同的内存结构。第 10 章包含了在 CUDA 中怎样分配和使用纹理操作的细节。

5.6 共享内存

共享内存被使用来在同一线程块内的 CUDA 线程间交换数据。物理上，它属于 SM 可见的内存，可以被非常快地访问。在速度上，共享内存的访问速度比寄存器访问速度慢十倍，但比全局内存访问快十倍。因此，共享内存通常作为一个减少 CUDA 内核所需要外部带宽的重要资源。

由于开发者显式分配与引用共享内存，共享内存可以被视作“手动管理缓存”或“暂存器”内存。开发者可以在内核和设备两个级别上请求不同的缓存配置：`cudaDeviceSetCacheConfig()`/`cuCtxSetCacheConfig()` 为设备指定优先缓存配置，而 `cudaFuncSetCacheConfig()`/`cuFuncSetCacheConfig()` 则为给定的内核指定优先缓存配置。如果两项同时指定，内核级别的请求会取得优先级，但是，在任何的情况下，内核的需求可能会重写开发者的优先请求。

使用共享内存 的内核由以下过程执行写入。

- 加载共享内存和 `_syncthreads()`
- 处理共享内存和 `_syncthreads()`
- 写入结果

使用 nvcc 选项 `-Xptxas-v,abi=no`，开发者可以让编译器报告给定内核使用的共享内存数量。在运行时中，内核使用的共享内存数量可以使用函数 `cuFuncGetAttribute(CU_FUNC_ATTRIBUTE_SHARED_SIZE_BYTES)` 查询。

[⊖] 表面引用只能在 SM2.x 和之后的硬件上使用。

5.6.1 不定大小共享内存声明

内核声明的每一个共享内存，都会在内核启动时为线程块自动分配。如果内核包含了一个未确定大小的共享内存声明，在内核启动时，该声明所需的内存数量必须被指定。

如果存在多个 `extern __shared__` 内存声明，它们互为别名，所以声明：

```
extern __shared__ char sharedChars[];
extern __shared__ int sharedInts[];
```

使相同的共享内存根据需要以 8 或 32 位整数寻址。一个使用这种别名的动机是，当可能读写全局内存时可使用更宽的类型，而使用更窄的类型进行内核计算。



注意 如果你有超过一个使用不定大小共享内存的内核，它们必须在独立的文件中编译。

5.6.2 束同步编码

在束同步编程中使用的共享内存变量必须声明为 `volatile`，来保护程序代码不被编译器优化，从而避免代码错误。

5.6.3 共享内存指针

使用指针引用共享内存是有效的，并且这很方便。使用这一方法的内核样例包括第 12 章的归约内核（见代码清单 12-3）和第 13 章中的 `scanBlock` 内核（见代码清单 13-3）。

5.7 内存复制

CUDA 中含有 3 种类型的内存——主机内存、设备内存和 CUDA 数组，并且 CUDA 中实现了它们之间内存复制的全套函数。对主机与设备间内存复制，CUDA 还提供了一个额外的内存复制函数集合来支持锁页主机内存和设备内存或 CUDA 数组间的异步内存复制。除此之外，还有一个点对点内存复制函数集合来支持 GPU 间的内存复制。

CUDA 运行时和驱动程序 API 使用不同的方式实现内存复制。对 1D 内存复制，驱动程序 API 定义了一个使用强类型参数的函数集。主机到设备、设备到主机和设备到设备内存复制函数是相互独立的。

```
CUresult cuMemcpyHtoD(CUdeviceptr dstDevice, const void *srcHost,
size_t ByteCount);
CUresult cuMemcpyDtoH(void *dstHost, CUdeviceptr srcDevice, size_t
ByteCount);
CUresult cuMemcpyDtoD(CUdeviceptr dstDevice, CUdeviceptr srcDevice,
size_t ByteCount);
```

与此不同，CUDA 运行时倾向于定义接受一个额外的“内存复制类型”参数，这个参数

取决于主机内存类型和复制目的地指针。

```
enum cudaMemcpyKind
{
    cudaMemcpyHostToHost = 0,
    cudaMemcpyHostToDevice = 1,
    cudaMemcpyDeviceToHost = 2,
    cudaMemcpyDeviceToDevice = 3,
    cudaMemcpyDefault = 4
};
```

对于更加复杂的内存复制操作，2 种 API 都使用描述符结构体来指定内存复制参数。

5.7.1 同步内存复制与异步内存复制

因为内存复制的大多数方面（维度、内存类型）是独立于内存复制是异步还是同步的，本节会详细检视其中的不同，下一节会简要介绍异步内存复制的相关内容。

默认的，任何涉及主机内存的复制是同步的：函数直到所有的操作执行完毕，否则不会返回控制权给调用者^①。即使是在锁页内存（例如，由函数 `cudaMallocHost()` 分配的内存）上操作，同步内存复制程序必须等待这一操作全部完成，因为应用软件会依赖于这一内存行为^②。

出于性能原因，在可能的情况下，我们需要避免同步内存复制。即使在未使用流的情况下，使所有的操作异步的进行会使 CPU 和 GPU 并发执行，会提升程序的性能。如果没有其他的操作同时执行，在 GPU 运行的同时，CPU 可以启动更多的 GPU 操作，像内核启动和其他的内存复制。如果 CPU/GPU 的并发是我们唯一的目标，我们就没有必要去创建任何的 CUDA 流，使用 NULL 流来调用一次异步内存复制就足够了。

虽然涉及主机内存的内存复制默认是同步的，但任何不涉及主机内存的复制（设备与设备、设备与数组之间）全部都是异步的。GPU 硬件在内部让这些操作顺序进行，所以对函数来说，没有必要等待 GPU 结束所有的操作再返回。

异步内存复制函数都用 `Async()` 作为后缀。例如，异步主机到设备的内存复制在驱动程序 API 中的函数是 `cuMemcpyHtoDAsync()`，在 CUDA 运行时中是 `cudaMemcpyAsync()`。

实现异步内存复制的硬件随时间更迭已经更新了很多代。最初的支持 CUDA 的 GPU（GeForce 8800 GTX）不含有任何的复制引擎，所以异步内存复制只能使 CPU/GPU 并发。之后的 GPU 添加了复制引擎，在 SM 运行时可以执行 1D 传输。此后，全功能的复制引擎被添加到 GPU 中，可以加速 2D 和 3D 传输，即使内存复制中包含了等步长布局与 CUDA 数组中使用的块线性布局的转换。除此之外，早期的 CUDA 硬件只有一个内存复制引擎，然而在今天，有些时候硬件中有两个复制引擎。而超过两个的复制引擎不会真正起作用。因为单个复

^① 这是因为硬件不能直接访问主机内存，除非它被页锁定并且映射给 GPU。对可换页内存的一次异步内存复制可以通过生成另一个 CPU 线程实现，但是至今为止，CUDA 团队选择了回避这一额外的复杂问题。

^② 当锁页内存被指定到一个同步内存复制操作中，驱动程序会让硬件使用 DMA，这通常会提升速度。

制引擎可以使 PCIe 总线单向达到饱和，为了最大化双向总线性能以及 GPU 计算和总线传输间的并发，只需要 2 个复制引擎。

复制引擎的数量可以通过调用函数 cuDeviceGetAttribute() 并使用标志 CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT 查询，或者也可以检查 cudaDeviceProp::asyncEngineCount。

5.7.2 统一虚拟寻址

统一虚拟寻址使 CUDA 可以对地址范围上的内存类型进行推断。因为 CUDA 寻址范围中包含了设备地址和主机地址，所以在函数 cudaMemcpy() 中没有必要指定参数 cudaMemcpyKind。驱动程序 API 中添加了函数 cuMemcpy()，此函数同样可以通过地址推断内存类型。

```
CUresult cuMemcpy(CUdeviceptr dst, CUdeviceptr src, size_t ByteCount);
```

CUDA 运行时中相对应的函数调用被命名为 cudaMemcpy：

```
cudaError_t cudaMemcpy( void *dst, const void *src, size_t bytes );
```

5.7.3 CUDA 运行时

表 5-10 总结了 CUDA 运行时中可用的内存复制函数。

表 5-10 内存复制函数 (CUDA 运行时)

目的对象类型	源对象类型	DIM (维度)	函 数
主机	设备	1D	cudaMemcpy(, ...cudaMemcpyHostToDevice);
主机	数组	1D	cudaMemcpyFromArray(, ...cudaMemcpyDeviceToHost);
设备	主机	1D	cudaMemcpy(..., cudaMemcpyDeviceToHost);
设备	设备	1D	cudaMemcpy(..., cudaMemcpyDeviceToDevice);
设备	数组	1D	cudaMemcpyFromArray
数组	主机	1D	cudaMemcpyToArray
数组	设备	1D	cudaMemcpyToArray
数组	数组	1D	cudaMemcpyArrayToArray
主机	设备	2D	cudaMemcpy2D(..., cudaMemcpyHostToDevice);
主机	数组	2D	cudaMemcpy2DToArray(..., cudaMemcpyHostToDevice);
设备	主机	2D	cudaMemcpy2D(..., cudaMemcpyDeviceToHost);
设备	设备	2D	cudaMemcpy2D(..., cudaMemcpyDeviceToDevice);
设备	数组	2D	cudaMemcpy2DToArray(..., cudaMemcpyDeviceToDevice);
数组	主机	2D	cudaMemcpy2DToArray(, ...cudaMemcpyHostToDevice);
数组	设备	2D	cudaMemcpy2DToArray(, ...cudaMemcpyHostToDevice);
数组	数组	2D	cudaMemcpy2DArrayToArray();

(续)

目的对象类型	源对象类型	DIM (维度)	函 数
主机	设备	3D	cudaMemcpy3D
主机	数组	3D	cudaMemcpy3D
设备	主机	3D	cudaMemcpy3D
设备	设备	3D	cudaMemcpy3D
设备	数组	3D	cudaMemcpy3D
数组	主机	3D	cudaMemcpy3D
数组	设备	3D	cudaMemcpy3D
数组	数组	3D	cudaMemcpy3D

1D 和 2D 内存复制函数使用基指针、步长和所需的大小。3D 内存复制操作接受一个描述结构体 `cudaMemcpy3DParms`, 定义如下:

```
struct cudaMemcpy3DParms
{
    struct cudaArray *srcArray;
    struct cudaPos srcPos;
    struct cudaPitchedPtr srcPtr;

    struct cudaArray *dstArray;
    struct cudaPos dstPos;
    struct cudaPitchedPtr dstPtr;

    struct cudaExtent extent;
    enum cudaMemcpyKind kind;
};
```

表 5-11 总结了 `cudaMemcpy3DParms` 结构体中的每一个成员。`cudaPos` 和 `cudaExtent` 结构体中的定义如下。

表 5-11 `cudaMemcpy3DParms` 结构体成员

结构体成员	描 述
<code>srcArray</code>	源数组, 如果 kind 需要
<code>srcPos</code>	源偏移
<code>srcPtr</code>	源指针, 如果 kind 需要
<code>dstArray</code>	目标数组, 如果 kind 需要
<code>dstPos</code>	目标偏移
<code>dstPtr</code>	目标指针, 如果 kind 需要
<code>extent</code>	内存复制中的 width、height 和 depth
<code>kind</code>	内存复制种类: <code>cudaMemcpyHostToDevice</code> 、 <code>cudaMemcpyDeviceToHost</code> 、 <code>cudaMemcpyDeviceToDevice</code> , 或 <code>cudaMemcpyDefault</code>

```
struct cudaExtent {
    size_t width;
    size_t height;
```

```

    size_t depth;
};

struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};

```

5.7.4 驱动程序 API

表 5-12 总结了驱动程序 API 中的内存复制函数。

表 5-12 内存复制函数 (驱动程序 API)

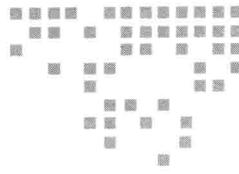
目的对象类型	源对象类型	DIM (维度)	函数
主机	设备	1D	cuMemcpyDtoH
主机	数组	1D	cuMemcpyAtoH
设备	主机	1D	cuMemcpyHtoD
设备	设备	1D	cuMemcpyDtoD
设备	数组	1D	cuMemcpyAtoD
数组	主机	1D	cuMemcpyHtoA
数组	设备	1D	cuMemcpyDtoA
数组	数组	1D	cuMemcpyAtoA
主机	设备	2D	cuMemcpy2D
主机	数组	2D	cuMemcpy2D
设备	主机	2D	cuMemcpy2D
设备	设备	2D	cuMemcpy2D
设备	数组	2D	cuMemcpy2D
数组	主机	2D	cuMemcpy2D
数组	设备	2D	cuMemcpy2D
数组	数组	2D	cuMemcpy2D
主机	设备	3D	cuMemcpy3D
主机	数组	3D	cuMemcpy3D
设备	主机	3D	cuMemcpy3D
设备	设备	3D	cuMemcpy3D
设备	数组	3D	cuMemcpy3D
数组	主机	3D	cuMemcpy3D
数组	设备	3D	cuMemcpy3D
数组	数组	3D	cuMemcpy3D

函数 cuMemcpy3D() 设计来实现以往所有内存复制函数的超集。任何的 1D、2D 或 3D

内存复制可以在主机、设备或 CUDA 数组，与复制源和复制目标的任何偏移之间使用。输入结构体 CUDA_MEMCDY_3D 的成员 WidthInBytes、Height 和 Depth，定义了内存复制的维度：Height == 0 意味着 1D 内存复制，Depth == 0 意味着一次 2D 复制。复制源和复制目标内存类型分别在结构体 srcMemoryType 和 dstMemoryType 中给出。

如果不需要，函数 cuMemcpy3D() 中结构体成员可以被忽略。例如，如果请求 1D 主机到设备内存复制，则 srcPitch、srcHeight、dstPitch 和 dstHeight 全部被忽略；如果 srcMemoryType 的值是 CU_MEMORYTYPE_HOST，srcDevice 和 srcArray 均被忽略。API 使用 C 语言中的惯例用法，为结构体赋值为 {0} 进行初始化，这也使内存复制被描述的十分简明。绝大多数的其他内存复制函数可以用几行代码实现，例如下面几行代码：

```
CUresult  
my_cuMemcpyHtoD( CUdevice dst, const void *src, size_t N )  
{  
    CUDA_MEMCPY_3D cp = {0};  
    cp.srcMemoryType = CU_MEMORYTYPE_HOST;  
    cp.srcHost = src;  
    cp.dstMemoryType = CU_MEMORYTYPE_DEVICE;  
    cp.dstDevice = dst;  
    cp.WidthInBytes = N;  
    return cuMemcpy3D( &cp );  
}
```



流与事件

CUDA 最著名的技术是能够实现细粒度并发。它带有可以组合使用共享内存和线程同步使线程在内存块内部密切合作的硬件设施。但它也有支持更粗粒并发的硬件和软件设施。

- CPU/GPU 的并发：由于它们是独立的设备，CPU 和 GPU 可以彼此独立地运作。
- 内存复制 / 内核处理并发：对于有一个或多个复制引擎的 GPU，在流处理器簇处理内核时，主机与设备间的内存复制操作也可以进行。
- 内核并发：SM 2.x 架构和更高级别的硬件可以并行运行多达 4 内核。
- 多 GPU 的并发：针对有足够的计算密度的问题，多 GPU 可以并行操作。(第 9 章是专门介绍多 GPU 程序设计的。)

CUDA 流支持这些类型的并发。在一个给定的流中，操作顺序进行，但在不同的流上的操作可以并行地执行。CUDA 流需要同步机制来协调并行执行，而这些机制由 CUDA 事件提供。CUDA 事件可以被异步“记录”到流中，并且当该 CUDA 事件之前的操作完成时，CUDA 事件就会收到通知信号。

对于在 GPU 上的多个引擎之间的同步以及多个 GPU 之间的同步而言，CUDA 事件可用于 CPU/GPU 的同步。它们还提供了一个基于 GPU 的计时机制，并且这个机制不会被系统事件（如页面错误、磁盘或网络控制器的中断）扰乱。对于整体计时而言，使用系统时间是最佳的，但要优化内核程序或者定位一系列流水线化的 GPU 操作中最耗时部分，CUDA 事件是大有用处的。本章报告的全部性能结果都是在亚马逊 EC2 云服务器的 cg1.4xlarge 实例上得到的，关于该实例的描述见 4.5 节。

6.1 CPU/GPU 的并发：隐藏驱动程序开销

CPU/GPU 的并发是指 CPU 在已经发送一些请求给 GPU 之后能够继续处理的能力。可以说，CPU/GPU 并发性最重要的用处就是隐藏来自 GPU 的请求任务的开销。

内核启动

内核的启动一直是异步的。一系列内核启动，如果它们之间没有其他 CUDA 操作干扰，将导致 CPU 把内核启动提交到 GPU，并在 GPU 处理完毕之前将控制权返回给调用者。

我们可以通过发射一系列用计时操作包围的空内核启动来测量驱动程序的开销。代码清单 6-1 展示了 nullKernelAsync.cu，它是一个测量执行内核启动所需时间的小程序。

代码清单 6-1 nullKernelAsync.cu.

```
#include <stdio.h>
#include "chTimer.h"

__global__
void
NullKernel()
{
}

int
main( int argc, char *argv[] )
{
    const int cIterations = 1000000;
    printf( "Launches... " ); fflush( stdout );
    chTimerTimestamp start, stop;

    chTimerGetTime( &start );
    for ( int i = 0; i < cIterations; i++ ) {
        NullKernel<<<1,1>>>();
    }
    cudaThreadSynchronize();
    chTimerGetTime( &stop );

    double microseconds = 1e6*chTimerElapsed( &start, &stop );
    double usPerLaunch = microseconds / (float) cIterations;

    printf( "% .2f us\n", usPerLaunch );
    return 0;
}
```

正如在附录 A 中描述的，chTimerGetTime() 使用主机操作系统的高分辨率计时设施，如 QueryPerformanceCounter() 或 gettimeofday()。在第 23 行的 cudaThreadSynchronize() 调用对于精确的计时而言是需要的。如删去它，当 stop 通过下面的函数调用记录时间时，GPU 并没有完成对最后一个内核的调用。

```
chTimerGetTime( &stop );
```

如果运行这个程序，你会看到，调用一个内核甚至一个什么都不做的内核将花费 2.0 ~ 8.0 毫秒的时间。大多数的时间花费在驱动程序上。CPU/GPU 并发性仅仅当内核运行时间比让驱动程序启动它的时间长时才有作用。为了强调 CPU/GPU 并发性对于小内核启动的重要性，让我们把 `cudaThreadSynchronize()` 调用移动到内循环中。[⊖]

```
chTimerGetTime( &start );
for ( int i = 0; i < cIterations; i++ ) {
    NullKernel<<<1,1>>>();
    cudaThreadSynchronize();
}
chTimerGetTime( &stop );
```

这里唯一的区别是，CPU 等待 GPU 处理完每一个空内核启动才进行下一个内核启动，如图 6-1 所示。作为一个例子，在禁用 ECC 的 Amazon EC2 实例中，`nullKernelASync` 在每次启动时，报告的时间为 3.4 毫秒，而 `nullKernelSync` 在每次启动时报告一个 100 毫秒的时间。因此，不考虑 CPU/GPU 的并发性，同步本身也应该避免。

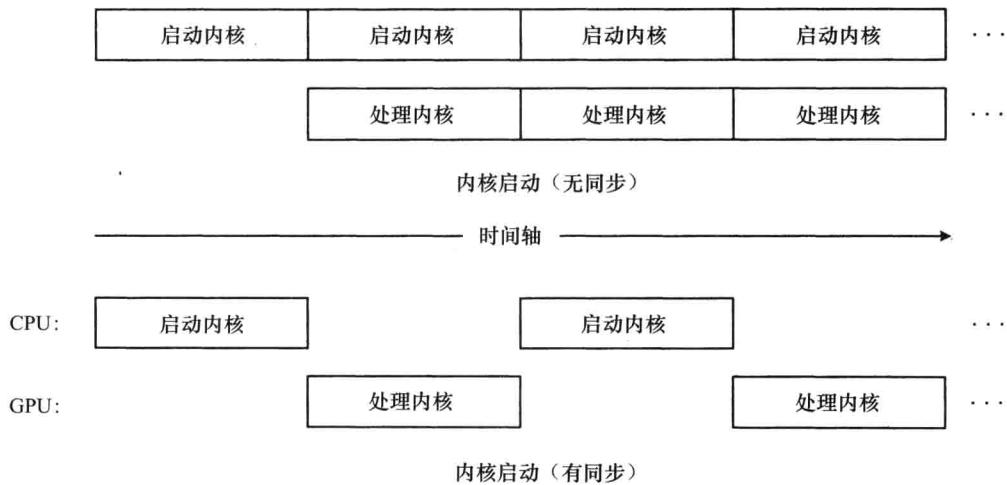


图 6-1 CPU/GPU 的并发性

即使不进行同步操作，如果内核运行时间不比启动内核所花的时间（3.4 毫秒）长，GPU 可能会在 CPU 提交更多工作之前闲置。为了知道一个内核需要做多少工作才能使启动是值得的，我们切换到一个忙等待一定时钟周期（使用 `clock()` 内置函数）的内核。

```
__device__ int deviceTime;
__global__
void
WaitKernel( int cycles, bool bWrite )
{
    int start = clock();
```

[⊖] 这个程序的源代码在 `nullkernel sync.cu` 里，这里没有详细列出是因为它与代码清单 6-1 几乎等同。

```

int stop;
do {
    stop = clock();
} while ( stop - start < cycles );
if ( bWrite && threadIdx.x==0 && blockIdx.x==0 ) {
    deviceTime = stop - start;
}
}
}

```

通过在满足条件时把结果写到 deviceTime 中，这个内核可以避免编译器优化掉忙等待延时。编译器不知道我们会把 false 作为第二个参数传递[⊖]。然后我们的 main() 函数里面的代码就会检查各种循环值的启动时间，从 0 ~ 2500。

```

for ( int cycles = 0; cycles < 2500; cycles += 100 ) {
printf( "Cycles: %d - ", cycles ); fflush( stdout );
chTimerGetTime( &start );
for ( int i = 0; i < cIterations; i++ ) {
    WaitKernel<<<1,1>>( cycles, false );
}
cudaThreadSynchronize();
chTimerGetTime( &stop );
double microseconds = 1e6*chTimerElapsedTime( &start, &stop );
double usPerLaunch = microseconds / (float) cIterations;

printf( "%.2f us\n", usPerLaunch );
}
}

```

该程序可以在 waitKernelAsync.cu 中找到。在我们的 EC2 实例中，输出正如图 6-2 所示。在这个主机平台上，打破势均力敌的标志（内核启动时间超过一个空内核启动时间的 2 倍）大约是在第 4500 个 GPU 时钟周期。

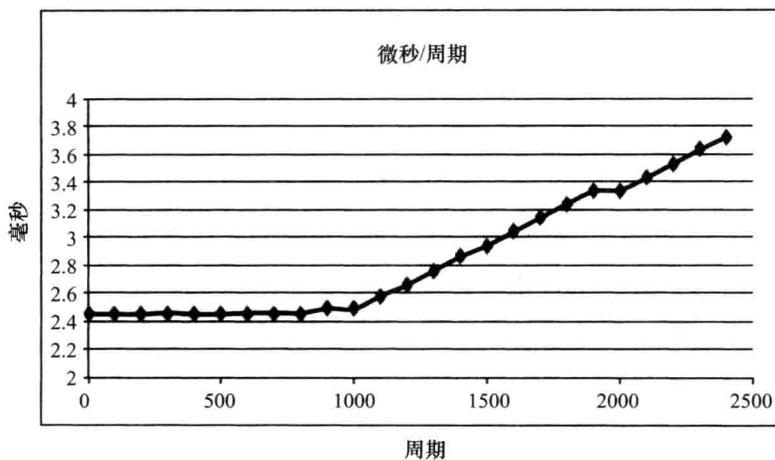


图 6-2 waitKernelAsync.cu 的毫秒 / 周期示意图

这些性能特征可能在很宽的范围内变化，并依赖于许多因素，其中包括以下几点。

[⊖] 如果 bWrite 为 false，编译器将通过每次循环里的条件分支，同时带来计时结果的失效。当计时结果看起来比较可疑时，我们应该使用 cuobjdump 检查微码以确认这一种情况是否发生。

- 主机 CPU 的性能
- 主机操作系统
- 驱动程序版本
- 驱动程序模型（是 TCC 还是 Windows 上的 WDDM）
- 是否在 GPU 上启用 ECC[⊖]

但大多数 CUDA 应用程序共同的主旨就是，开发人员应该尽最大努力避免破坏 CPU/GPU 并发。只有非常计算密集的应用程序和进行大量数据传输的应用程序能忽视这方面的开销。为了在进行内存复制和内核启动时充分利用 CPU/GPU 的并发性，开发人员必须使用异步的内存复制。

6.2 异步的内存复制

像内核启动一样，异步的内存复制调用在 GPU 完成待处理的内存复制之前就会返回。由于 GPU 能够自主运行，并且可以在没有任何操作系统介入的情况下对主机的内存进行读或写操作，所以，只有锁页内存有资格进行异步的内存复制。

最早期 CUDA 包含异步内存复制的应用程序是隐藏在 CUDA 1.0 驱动程序下的。GPU 不可以直接访问分页内存，所以驱动程序利用 CUDA 上下文分配的一对锁页的“临时缓冲区”实现分页的内存复制。图 6-3 显示了这个过程是如何工作的。

要执行主机端到设备端的内存复制，驱动程序首先通过复制到一个暂存缓冲区来“启动泵”，然后开始一个通过 GPU 读取那个数据的 DMA 操作。当 GPU 开始处理该请求时，驱动程序将更多的数据复制到其他中转缓冲区。CPU 和 GPU 在 2 个中转缓冲区之间保持来回交互，以及适当的同步，直到 GPU 完成最后的内存复制。除了复制数据之外，在复制数据时 CPU 自然也会换进非驻留的页面。

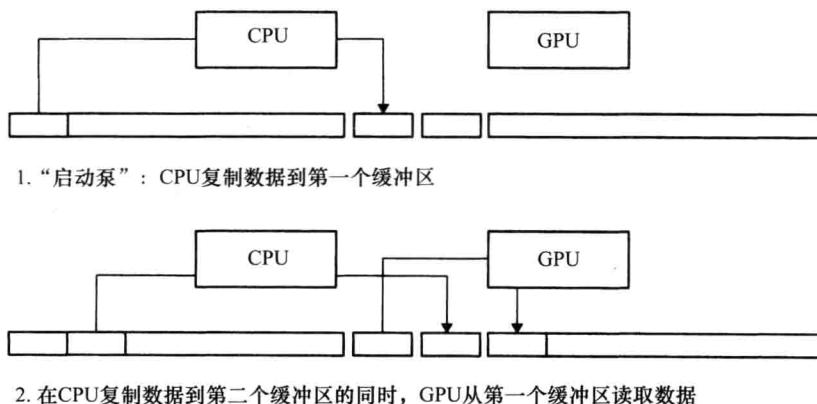
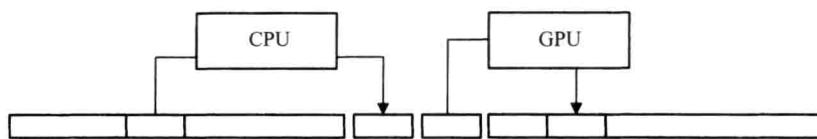
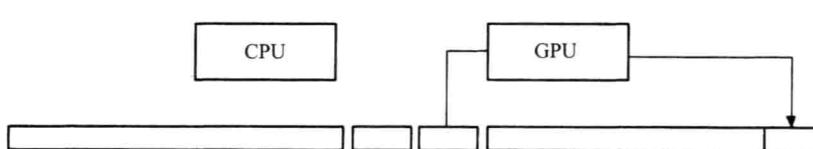


图 6-3 分页的内存复制

[⊖] 当 ECC 启用时，驱动程序必须执行内核转换来检查是否发生了内存错误。最终，即使在包含用户模式客户端驱动程序的平台上，cuda Thread Synchronize() 函数也是代价较高的。



3. 在CPU复制数据到第一缓冲区时GPU从第二个缓冲区读取数据



4. 最终的GPU内存复制操作

图 6-3 (续)

6.2.1 异步的内存复制：主机端到设备端

正如内核启动一样，异步内存复制会引发驱动程序上的固定 CPU 开销。在主机到设备的内存复制情况下，所有小于一定尺寸的内存复制都是异步的，因为驱动程序直接把源数据复制到它用来控制硬件的命令缓冲器中。

我们可以写一个应用程序，测量异步的内存复制开销，就像我们早些时候测量内核启动开销一样。下面的代码，是在一个叫做 nullHtoDMemcpyAsync.cu 文件中的，报告了一个亚马逊 EC2 上的 cg1.4xlarge 实例的结果，每个内存复制花费 3.3 毫秒。由于 PCIe 几乎可以在那段时间里运行 2000 次传输，测量一个内存复制的时间如何随着尺寸变大而增长是有意义的。

```
CUDART_CHECK( cudaMalloc( &deviceInt, sizeof(int) ) );
CUDART_CHECK( cudaHostAlloc( &hostInt, sizeof(int), 0 ) );

chTimerGetTime( &start );
for ( int i = 0; i < cIterations; i++ ) {
    CUDART_CHECK( cudaMemcpyAsync( deviceInt, hostInt, sizeof(int),
        cudaMemcpyHostToDevice, NULL ) );
}
CUDART_CHECK( cudaThreadSynchronize() );
chTimerGetTime( &stop );
```

breakevenHtoDmemcpy.cu 程序测量内存复制大小从 4K 变到 64K 的性能。在一个亚马逊 EC2 中的 cg1.4xlarge 实例上，它会产生图 6-4 的效果。这个程序产生的数据非常清晰，足以符合线性回归曲线——在这种情况下，加上截距为 3.3 微秒，并且斜率为 0.000170 微秒 / 字节。斜率对应为 5.9GB/s，大约为 PCIe 2.0 的理论带宽。

6.2.2 异步内存复制：设备端到主机端

nullDtoHmemcpyNoSync.cu 和 breakevenDtoHmemcpy.cu 程序对于从设备端到主机端的小内存复制表现出相同的测量结果。在我们信赖的亚马逊 EC2 实例上，一个内存复制的最少

时间是 4.00 微秒 (见图 6-5)。

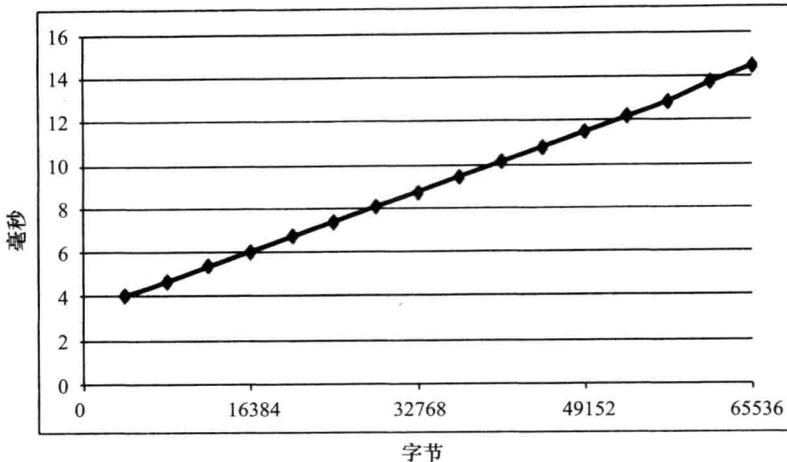


图 6-4 从主机端到设备端的小内存复制的性能

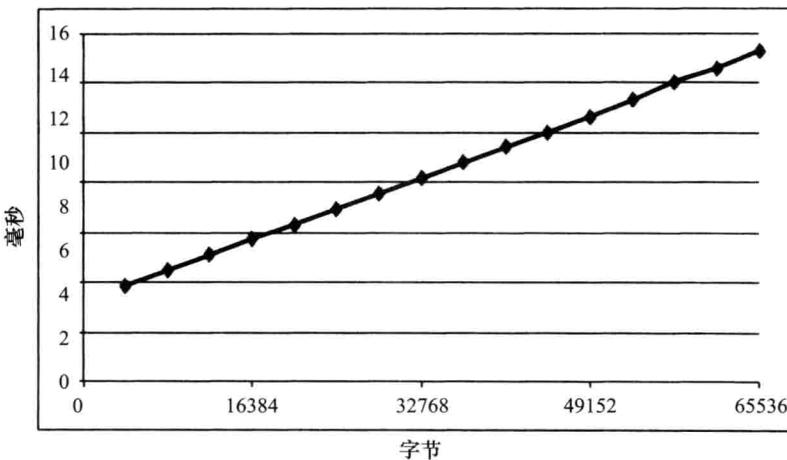


图 6-5 从设备端到主机端的小内存复制的性能

6.2.3 NULL 流和并发中断

任何流操作都可以输入 NULL 作为流参数，这些操作将等到之前的所有 GPU 操作完成后才会启动[⊖]。不需要复制引擎来让内存复制操作与内核处理重叠执行的应用程序可以使用

[⊖] 当 CUDA 流功能添加到 CUDA1.1 时，设计者有两种选择：一个是把 NULL 流相对于其他流独立出来，在它的内部只能串行执行；另一种是让它与 GPU 的所有引擎同步（“汇聚”）。他们选择了后种方案，一部分原因在于 CUDA 还没有能够执行不同流之间同步的机制。

NULL 流，以利用 CPU/GPU 并发。

一旦一个流操作已经使用 NULL 流启动了，应用程序必须使用诸如 cuCtxSynchronize() 或 cudaThreadSynchronize() 之类的同步函数来确保前面的操作在继续操作之前完成。但是应用程序可能会在进行同步之前要求很多这样的操作。例如，应用程序可能在和上下文同步之前执行一个异步的主机到设备的内存复制、一个或者多个的内核启动以及一个异步的设备到主机的内存复制。一旦 GPU 已经完成了最近一次请求的操作，cuCtxSynchronize() 或 cudaThreadSynchronize() 调用就会返回。当执行小内存复制或启动不会长期运行的内核时，这个机制是特别有用的。CUDA 驱动程序会充分利用 CPU 写命令到 GPU 的这段宝贵的时间，而且该 CPU 执行与 GPU 命令处理重叠进行可以提高性能。



注意 即使在 CUDA 1.0 中，内核启动也是异步的。其结果是，在没有指定流的情况下，所有内核启动隐式指定到 NULL 流上执行。

1. 打破并发

每当一个应用程序执行一个完整的 CPU/GPU 同步（让 CPU 等待直到 GPU 完全处于闲置状态），性能都会受损。我们可以通过把 NULL 流的内存复制调用从异步切换到同步，来测量同步操作带来的性能影响。可以通过把 cudaMemcpyAsync() 调用改为 cudaMemcpy() 调用来实现。这个 nullDtoHMemcpySync.cu 程序为设备端到主机端的内存复制专门运行上述任务。

在我们值得信赖的亚马逊 cg1.4xlarge 实例上，nullDtoHMemcpySync.cu 报告了每个内存复制大约耗时 7.9 微秒。如果一个 Windows 驱动程序必须执行内核转换（kernel thunk），或者一个启用 ECC 的 GPU 的驱动程序必须检查 ECC 错误，完全的 GPU 同步就会更昂贵。

执行此同步的显式方法包括以下几种：

- cuCtxSynchronize()/cudaDeviceSynchronize()。
- NULL 流上的 cuStreamSynchronize()/cudaStreamSynchronize()。
- 未采用流的主机和设备之间的内存复制，例如，cumemcpyHtoD()、cumemcpyDtoH() 和 cudamemcpy()。

其他将打破 CPU/GPU 并发的更微妙的方式包括以下几种：

- 运行过程是设置了 CUDA_LAUNCH_BLOCKING 环境变量的。
- 需要重新分配本地内存的内核启动。
- 执行大内存分配或主机内存分配。
- 销毁对象，如 CUDA 流和 CUDA 事件。

2. 非阻塞流

要创建一个不需要与 NULL 流同步的流（因此不太可能出现如上所述的“并发中断”），需要指定 cuStreamCreate() 的 CUDA_STREAM_NON_BLOCKING 标志或者指定 cudaStreamCreateWithFlags() 的 cudaStreamNonBlocking 标志。

6.3 CUDA 事件：CPU/GPU 同步

CUDA 事件的主要特点之一是，它们可以支持“部分”CPU/GPU 的同步。在完整的 CPU/GPU 同步中，CPU 要一直等待到 GPU 空闲，这将在 GPU 的工作任务流水线中引入“气泡”。而在部分同步中，CUDA 事件可以被记录进 GPU 命令的异步流中。然后，CPU 可以等待直到该事件之前的所有工作完成。但 GPU 可以继续执行在 cuEventRecord() / cudaEventRecord() 之后被提交的任何工作。

作为 CPU/GPU 并发的一个例子，代码清单 6-2 给出了一个分页内存的内存复制子程序。这一程序的代码实现了图 6-3 所描述的算法，可以在 pageableMemcpyHtoD.cu 中找到。它采用了 2 个锁页内存缓冲区，存储在如下方式声明的全局变量中。

```
void *g_hostBuffers[2];
```

并且两个 CUDA 事件被声明为：

```
cudaEvent_t g_events[2];
```

代码清单 6-2 chMemcpyHtoD()——分页的内存复制

```
void
chMemcpyHtoD( void *device, const void *host, size_t N )
{
    cudaError_t status;
    char *dst = (char *) device;
    const char *src = (const char *) host;
    int stagingIndex = 0;
    while ( N ) {
        size_t thisCopySize = min( N, STAGING_BUFFER_SIZE );
        cudaEventSynchronize( g_events[stagingIndex] );
        memcpy( g_hostBuffers[stagingIndex], src, thisCopySize );
        cudaMemcpyAsync( dst, g_hostBuffers[stagingIndex],
                        thisCopySize, cudaMemcpyHostToDevice, NULL );
        cudaEventRecord( g_events[1-stagingIndex], NULL );
        dst += thisCopySize;
        src += thisCopySize;
        N -= thisCopySize;
        stagingIndex = 1 - stagingIndex;
    }
Error:
    return;
}
```

chMemcpyHtoD() 是通过在 2 个主机缓冲区之间轮流操作来最大限度地提高 CPU/GPU

的并发性的。当 GPU 从一个缓冲区读数据的时候，CPU 将数据复制进另一个缓冲区。当 CPU 分别复制第一个和最后一个缓冲区时，在操作的开始和末尾会有一些不存在 CPU/GPU 并发性的“悬空区”。

在这个程序中，唯一需要的同步就是在第 11 行的 cudaEventSynchronize()，这样可以确保在开始复制数据到缓冲区之前 GPU 已经结束对该缓冲区的操作。GPU 命令一旦被排队，cudaMemcpyAsync() 就会返回。它不会等到操作完成。cudaEventRecord() 也是异步的。它会导致在刚刚请求的异步的内存复制已经完成时发信号给事件。

CUDA 事件在创建后立即被记录，因此在第 11 行的第一个 cudaEventSynchronize() 调用工作正确。

```
CUDART_CHECK( cudaEventCreate( &g_events[0] ) );
CUDART_CHECK( cudaEventCreate( &g_events[1] ) );
// record events so they are signaled on first synchronize
CUDART_CHECK( cudaEventRecord( g_events[0], 0 ) );
CUDART_CHECK( cudaEventRecord( g_events[1], 0 ) );
```

如果运行 pageablememcpyHtoD.cu，它将会报告一个比被 CUDA 驱动程序执行的分页内存复制带宽小得多的带宽。这是因为 C 运行时的 memcpy() 函数的实现没有优化到像 CPU 那样快的内存移动。为了获得最佳性能，内存必须使用可以一次性移动 16 个字节的 SSE 指令来复制。使用这些指令按照它的对齐限制来编写一个通用的内存复制程序是复杂的，但是实现一个需要源数据、目的数据和 16 字节对齐的字节数的简化版本并不困难。[⊖]

```
#include <xmmmintrin.h>
bool
memcpy16( void * _dst, const void * _src, size_t N )
{
    if ( N & 0xf ) {
        return false;
    }

    float *dst = (float *) _dst;
    const float *src = (const float *) _src;
    while ( N ) {
        _mm_store_ps( dst, _mm_load_ps( src ) );
        src += 4;
        dst += 4;
        N -= 16;
    }
    return true;
}
```

当 C 运行时中的 memcpy() 函数被这个代替时，在亚马逊 EC2 的 cg1.4xlarge 实例上的性能会从 2155MB/s 增加到 3267MB/s。更复杂的内存复制程序可以处理不严格的对齐约束，并且更高一些的性能可能通过展开内循环来实现。在 cg1.4xlarge 实例上，使用 CUDA 驱动程序中更好优化的 SSE 内存复制可以实现比 pageablememcpyHtoD16.cu 高 100MB/s 的性能。

对于分页内存复制的性能而言，CPU/GPU 的并发性有多重要呢？如果将事件同步，我

[⊖] 在某些平台上，nvcc 无法对此代码无缝地编译。在本书附带的代码中，memcpy16() 单独存为 memcpy16.cpp 文件。

们将执行主机端到设备端的同步的内存复制，如下所示。

```

while ( N ) {
    size_t thisCopySize = min( N, STAGING_BUFFER_SIZE );

< CUDART_CHECK( cudaEventSynchronize( g_events[stagingIndex] ) );
    memcpy( g_hostBuffers[stagingIndex], src, thisCopySize );
    CUDART_CHECK( cudaMemcpyAsync( dst, g_hostBuffers[stagingIndex],
        thisCopySize, cudaMemcpyHostToDevice, NULL ) );
    CUDART_CHECK( cudaEventRecord( g_events[1-stagingIndex], NULL ) );
> CUDART_CHECK( cudaEventSynchronize( g_events[1-stagingIndex] ) );
    dst += thisCopySize;
    src += thisCopySize;
    N -= thisCopySize;
    stagingIndex = 1 - stagingIndex;
}

```

此代码在 pageablememcpyHtoD16Synchronous.cu 中可以找到，并且它在 cg1.4xlarge 实例上的性能大约是原来的 70% (2334MB/s，而不是 3267MB/s)。

6.3.1 阻塞事件

CUDA 事件也可以根据需要执行“阻塞”，其中使用基于中断的 CPU 同步机制。然后 CUDA 驱动程序使用线程同步原语（挂起 CPU 线程而不是轮询事件的 32 位跟踪值）实现 cu(da)EventSynchronize() 调用。

对于延迟敏感的应用程序而言，阻塞事件可能会造成性能损失。对我们通常的分页的内存复制程序，在我们的 cg1.4xlarge 实例上使用阻塞事件会导致速度略微放缓（大约 100MB/s）。但是对于大多数 GPU 密集型应用程序，或者对于需要大量的 CPU/GPU 处理的“混合工作负载”的应用程序，使 CPU 线程空闲的好处大于处理“等待结束时处理所产生的中断”的成本。混合工作负载的一个例子是视频转码，它包含适用于 CPU 的分支代码，以及适用于 GPU 的信号和像素处理。

6.3.2 查询

CUDA 流和 CUDA 事件分别可以通过 cu(da)StreamQuery() 和 cu(da)EventQuery() 查询。如果 cu(da)StreamQuery() 返回 success，那就表示在一个给定的流里所有等待中的操作都已经完成了。如果 cu(da)EventQuery() 返回 success，那就表示事件已经记录下来了。

虽然这些查询被设计为轻量级的，但是如果启用了 ECC，它们必须执行内核转换来检查当前的 GPU 错误状态。此外，在 Windows 上，任何挂起的命令都将提交给 GPU，这也需要一个内核转换过程。

6.4 CUDA 事件：计时

当之前的命令已经完成时，CUDA 事件开始运行，提交给 GPU 一个命令。该命令使

GPU 写一个已知的值到一个 32 位的内存位置。CUDA 驱动程序通过检测该 32 位的值来实现 cuEventQuery() 和 cuEventSynchronize()。但是除了 32 位的“跟踪值”，GPU 也可以写一个源于高分辨率 GPU 时钟的 64 位时间值。

因为它们使用了 GPU 时钟，使用 CUDA 事件的计时较少受到系统事件（如页面错误或中断）的干扰，并且根据时间戳计算运行时间的函数适用于所有操作系统。所谓操作的系统时间（wall time）是指用户最终感受到的时间的流逝，所以 CUDA 事件最好有针对性地使用在内核性能调优或其他 GPU 密集型的操作上，而不是报告绝对时间给用户。

cuEventRecord() 的流参数是针对流间同步的，而不是针对计时的。当使用 CUDA 事件来计时时，最好把它们记录在 NULL 流中。其机理和在超标量 CPU 读取时间戳计数器（如基于 x86 的 RDTSC）的清空流水线的指令的理由一样：在所有的 GPU 引擎上强制执行“汇集”操作，可以避免正在被计时的操作产生任何歧义[⊖]。确保 cu(da)EventRecord() 调用包含足够的工作以便计时获得有意义的结果。

最后，请注意，CUDA 事件旨在对 GPU 操作计时。任何同步的 CUDA 操作都将导致使用 GPU 来对 CPU/GPU 同步操作进行计时。

```
CUDART_CHECK( cudaEventRecord( startEvent, NULL ) );
// synchronous memcpy - invalidates CUDA event timing
CUDART_CHECK( cudaMemcpy( deviceIn, hostIn, N*sizeof(int) ) );
CUDART_CHECK( cudaEventRecord( stopEvent, NULL ) );
```

在下一节探讨的例子中，我们将展示如何使用 CUDA 事件来计时。

6.5 并发复制和内核处理

由于为了让 GPU 操作数据，CUDA 应用程序必须通过 PCIe 总线传输数据。另一个提升性能的机会是以并发形式执行那些主机与设备间的传输和内核处理。根据阿姆达尔法则[⊖]，使用多个处理器的最大加速比是

$$\text{加速比} = \frac{1}{r_s + \frac{r_p}{N}}$$

其中 $r_s+r_p=1$ ， N 是处理器的数量。在并发执行复制和内核处理的情况下，“处理器的数量”就是 GPU 中自主硬件单元的数量：1 个或者 2 个复制引擎，加上执行内核的流处理簇。对于 $N=2$ ，图 6-6 展示了随着 r_s 和 r_p 变化的理想化的加速曲线。

所以在理论上，一个 2 倍的性能提高在有一个复制引擎的 GPU 上是可能的，但是仅当程序在流处理簇和复制引擎之间得到完美的重叠，并且仅当程序在传输和处理数据上花费相同的时间的时候是如此。

[⊖] 其他考虑：在 CUDA 的 SM 1.1 硬件上，计时事件可能只由执行内核计算的硬件单元来记录。

[⊖] <http://bit.ly/13UqBm0>。

在进行这种努力之前，你应该仔细看看这是否有利于你的应用程序。极端传输受限型（即它们把大多数时间花费在与 GPU 交换数据上）或者极端计算受限型（即它们花费大多数时间处理 GPU 上的数据）的应用程序只能从重叠传输和计算中少量受益。

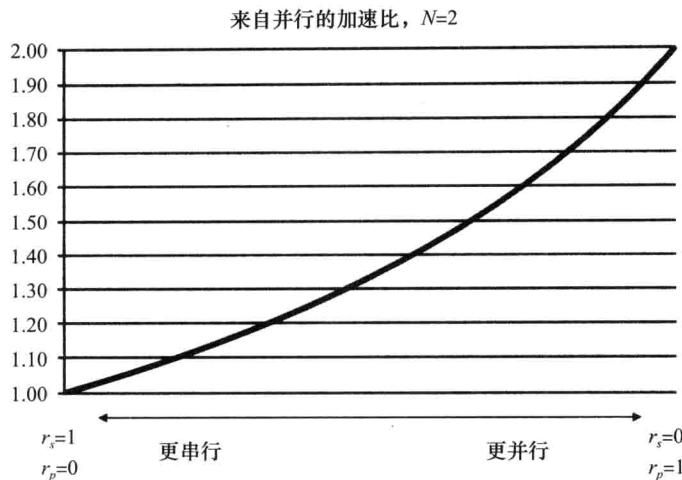


图 6-6 理想的阿姆达尔法则曲线

6.5.1 concurrencyMemcpyKernel.cu

concurrencyMemcpyKernel.cu 程序旨在说明的不仅有怎样实现并发的内存复制和内核执行，还有怎样确定这样做是否值得。代码清单 6-3 给出了 AddKernel()——一个“花哨”的内核，使用一个参数 cycles 来控制它的运行时间。

代码清单 6-3 AddKernel(), 一个带有计算密度参数的花哨内核

```
__global__ void
AddKernel( int *out, const int *in, size_t N, int addValue, int
cycles )
{
    for ( size_t i = blockIdx.x*blockDim.x+threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        volatile int value = in[i];
        for ( int j = 0; j < cycles; j++ ) {
            value += addValue;
        }
        out[i] = value;
    }
}
```

AddKernel() 流从 in 读出一个整数型数组到 out，对每个输入值循环 cycles 次。通过改变 cycles 的值，我们可以使内核涵盖从一个简单内存带宽受限型的流内核到完全的计算受限型内核的整个范围。

程序中的这两个子程序可测量 AddKernel() 的性能。

- TimeSequentialMemcpyKernel() 将输入数据复制到 GPU、调用 AddKernel() 并且从 GPU 以独立的、顺序的方式复制到输出。
- TimeConcurrentOperations() 分配了一些 CUDA 流，并且同时完成主机端到设备端的内存复制、内核处理以及设备端到主机端的内存复制。

TimeSequentialMemcpyKernel()，在代码清单 6-4 中给出，使用了 4 个 CUDA 事件来分别计时主机端到设备端的内存复制、内核处理以及设备端到主机端的内存复制。它也报告 CUDA 事件测量的总时间。

代码清单 6-4 TimeSequentialMemcpyKernel() 函数

```

bool
TimeSequentialMemcpyKernel(
    float *timesHtoD,
    float *timesKernel,
    float *timesDtoH,
    float *timesTotal,
    size_t N,
    const chShmooRange& cyclesRange,
    int numBlocks )
{
    cudaError_t status;
    bool ret = false;
    int *hostIn = 0;
    int *hostOut = 0;
    int *deviceIn = 0;
    int *deviceOut = 0;
    const int numEvents = 4;
    cudaEvent_t events[numEvents];

    for ( int i = 0; i < numEvents; i++ ) {
        events[i] = NULL;
        CUDART_CHECK( cudaEventCreate( &events[i] ) );
    }
    cudaMallocHost( &hostIn, N*sizeof(int) );
    cudaMallocHost( &hostOut, N*sizeof(int) );
    cudaMalloc( &deviceIn, N*sizeof(int) );
    cudaMalloc( &deviceOut, N*sizeof(int) );

    for ( size_t i = 0; i < N; i++ ) {
        hostIn[i] = rand();
    }

    cudaDeviceSynchronize();

    for ( chShmooIterator cycles(cyclesRange); cycles; cycles++ ) {

        printf( "." );
        fflush( stdout );

        cudaEventRecord( events[0], NULL );
        cudaMemcpyAsync( deviceIn, hostIn, N*sizeof(int),
            cudaMemcpyHostToDevice, NULL );
        cudaEventRecord( events[1], NULL );
        AddKernel<<<numBlocks, 256>>>(
            deviceOut, deviceIn, N, 0xcc, *cycles );
        cudaEventRecord( events[2], NULL );
        cudaMemcpyAsync( hostOut, deviceOut, N*sizeof(int),
            cudaMemcpyDeviceToHost, NULL );
        cudaEventRecord( events[3], NULL );
    }
}

```

```

        cudaMemcpyDeviceToHost, NULL );
cudaEventRecord( events[3], NULL );
cudaDeviceSynchronize();

cudaEventElapsedTime( timesHtoD, events[0], events[1] );
cudaEventElapsedTime( timesKernel, events[1], events[2] );
cudaEventElapsedTime( timesDtoH, events[2], events[3] );
cudaEventElapsedTime( timesTotal, events[0], events[3] );

timesHtoD += 1;
timesKernel += 1;
timesDtoH += 1;
timesTotal += 1;
}

ret = true;

Error:
for ( int i = 0; i < numEvents; i++ ) {
    cudaEventDestroy( events[i] );
}
cudaFree( deviceIn );
cudaFree( deviceOut );
cudaFreeHost( hostOut );
cudaFreeHost( hostIn );
return ret;
}

```

`cyclesRange` 参数，使用了在附录 A.4 节中描述的“shmoo”功能，它指定了当调用 `AddKernel()` 时使用的循环值的范围。在一个 EC2 中的 `cg1.4xlarge` 实例上，循环值的时间是 4 ~ 64 (以毫秒计)，如下：

<code>cycles</code> 的值	主机到设备复制时间	内核执行时间	设备到主机复制时间	总计时间
4	89.19	11.03	82.03	182.25
8	89.16	17.58	82.03	188.76
12	89.15	24.10	82.03	195.28
16	89.15	30.57	82.03	201.74
20	89.14	37.03	82.03	208.21
24	89.16	43.46	82.03	214.65
28	89.16	49.90	82.03	221.10
32	89.16	56.35	82.03	227.54
36	89.13	62.78	82.03	233.94
40	89.14	69.21	82.03	240.38
44	89.16	75.64	82.03	246.83
48	89.16	82.08	82.03	253.27
52	89.14	88.52	82.03	259.69
56	89.14	94.96	82.03	266.14
60	89.14	105.98	82.03	277.15
64	89.17	112.70	82.03	283.90

对那些 *cycles 值处于 48 (已高亮显示) 附近的结果，我们看到内核花费的时间和内存复制操作占用的时间一样多，我们可以假定并发执行操作是有利的。

TimeConcurrentMemcpyKernel() 函数会把 AddKernel() 执行的计算分为大小为 stream Increment 的片段，并且分别使用单独的 CUDA 流计算每一片段。代码清单 6-5 中来自 TimeConcurrentMemcpyKernel() 的代码片段突出了流编程的复杂性。

代码清单 6-5 TimeConcurrentMemcpyKernel() 片段

```

intsLeft = N;
for ( int stream = 0; stream < numStreams; stream++ ) {
    size_t intsToDo = (intsLeft < intsPerStream) ?
        intsLeft : intsPerStream;
    CUDART_CHECK( cudaMemcpyAsync(
        deviceIn+stream*intsPerStream,
        hostIn+stream*intsPerStream,
        intsToDo*sizeof(int),
        cudaMemcpyHostToDevice, streams[stream] ) );
    intsLeft -= intsToDo;
}
intsLeft = N;
for ( int stream = 0; stream < numStreams; stream++ ) {
    size_t intsToDo = (intsLeft < intsPerStream) ?
        intsLeft : intsPerStream;
    AddKernel<<<nunBlocks, 256, 0, streams[stream]>>>(
        deviceOut+stream*intsPerStream,
        deviceIn+stream*intsPerStream,
        intsToDo, 0xcc, *cycles );
    intsLeft -= intsToDo;
}

intsLeft = N;
for ( int stream = 0; stream < numStreams; stream++ ) {
    size_t intsToDo = (intsLeft < intsPerStream) ?
        intsLeft : intsPerStream;
    CUDART_CHECK( cudaMemcpyAsync(
        hostOut+stream*intsPerStream,
        deviceOut+stream*intsPerStream,
        intsToDo*sizeof(int),
        cudaMemcpyDeviceToHost, streams[stream] ) );
    intsLeft -= intsToDo;
}

```

除了要求应用程序来创建和销毁 CUDA 流之外，对于每一个主机到设备端的内存复制、内核处理以及设备到主机端的内存复制操作，流必须分别循环遍历。如果没有这种“软件流水线”的话，将没有不同流中工作的并发执行，因为每个流中的操作都要在前面加上“互锁”操作，以防止此操作继续执行直至同一流中之前的所有操作均完成。其结果将不仅仅是无法启动引擎间的并行执行，而且由于管理流并发性的轻微开销也将有额外的性能下降。

计算无法完全并发处理，因为没有内核处理能与第一个或者最后一个内存复制重叠执行，并且在 CUDA 流之间的同步上会有一些开销，正如我们在之前的章节知道的，在调用内存复制上和内核操作本身上也会有一些开销。其结果是，流的最佳数目取决于应用程序，并且应该根据经验来确定。在 concurrencyMemcpyKernel.cu 程序中，通过在命令行上使用 --numStreams 参数指定流的数目。

6.5.2 性能结果

带有固定的缓冲区大小和一定数量的流的 concurrencyMemcpyKernel.cu 程序会根据 cycles 值报告相应的性能特征。在亚马逊 EC2 的 cg1.4xlarge 实例中，有一个 128MB 大小的缓冲区和 8 个流，如下报告所考查的 cycles 值的范围为 4 ~ 64。

CYCLES 值	主机到设备 复制时间	内核执行 时间	设备到主机 复制时间	总计时间	并发时间	加速比
4	89.19	11.03	82.03	182.25	173.09	1.05
8	89.16	17.58	82.03	188.76	173.41	1.09
12	89.15	24.1	82.03	195.28	173.74	1.12
16	89.15	30.57	82.03	201.74	174.09	1.16
20	89.14	37.03	82.03	208.21	174.41	1.19
24	89.16	43.46	82.03	214.65	174.76	1.23
28	89.16	49.9	82.03	221.10	175.08	1.26
32	89.16	56.35	82.03	227.54	175.43	1.30
36	89.13	62.78	82.03	233.94	175.76	1.33
40	89.14	69.21	82.03	240.38	176.08	1.37
44	89.16	75.64	82.03	246.83	176.41	1.40
48	89.16	82.08	82.03	253.27	176.75	1.43
52	89.14	88.52	82.03	259.69	177.08	1.47
56	89.14	94.96	82.03	266.14	179.89	1.48
60	89.14	105.98	82.03	277.15	186.31	1.49
64	89.17	112.7	82.03	283.90	192.86	1.47

对应 cycles 从 4 ~ 256 的完整曲线见图 6-7。不幸的是，对于这些设置而言，这里仅有 50% 的加速，远低于理论上可以得到的 3 倍加速。

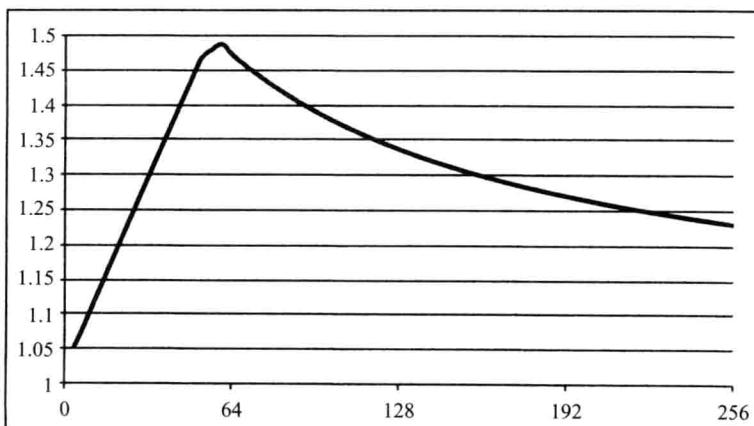


图 6-7 内存复制 / 内核并发 (Tesla M2050) 带来的加速比

仅仅包含一个复制引擎的 GeForce GTX 280 的优势更为明显。这里图 6-8 展示了 cycles 的值变至 512 的结果。从图中可以看出，它的最大加速比更接近 2 倍这一理论上的最大值。

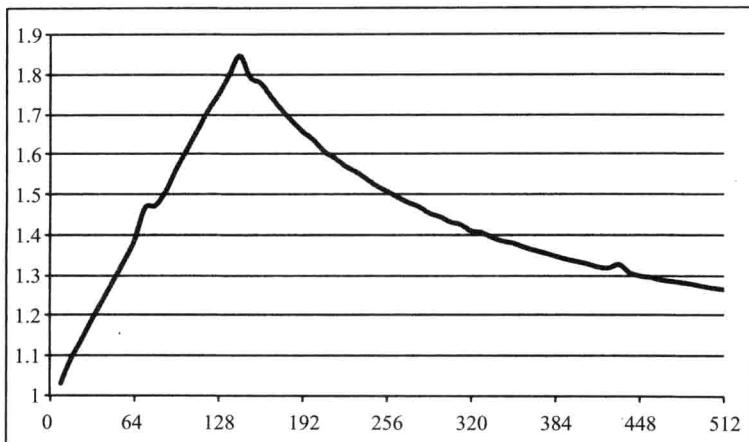


图 6-8 内存复制 / 内核并发 (GeForce GTX 280) 带来的加速比

正如已写到的那样，`concurrencyMemcpyKernel.cu` 不过就是起一个演示作用，因为 `AddValues()` 仅仅是一个花哨的内核。但是可以把自己的内核插入这个应用程序以便确定是否使用流的额外复杂性对性能提高有利。请注意，除非需要并发的内核执行（参看 6-7 节），代码清单 6-5 中的内核调用可以用同一个流中的连续的内核调用代替，并且，应用程序仍将得到理想的并发性。

附加说明，复制引擎的数量可以通过调用 `cudaGetDeviceProperties()` 并检查 `cudaDeviceProp::asyncEngineCount` 属性查询，也可以使用带有 `CU_DEVICE_ATTRIBUTE_ASYNC_ENGINE_COUNT` 标志的 `cuDeviceQueryAttribute()` 来查询。

SM 1.1 和部分 SM 1.2 硬件的复制引擎仅仅可以复制线性内存，但是更新的复制引擎全面支持 2D 内存复制，包括 2D 和 3D 的 CUDA 数组。

6.5.3 中断引擎间的并发性

使用 CUDA 流用于并发的内存复制和内核执行程序的同时会带来更多的“中断并发”的机会。在先前的小节中，CPU/GPU 并发性能够被无意中做的一些事（可以导致 CUDA 执行一个完整的 CPU/GPU 同步）中断。在这里，CPU/GPU 并发性可以被无意中执行的未指定流的 CUDA 操作中断。回想一下，NULL 流会强制所有的 GPU 引擎汇聚，所以如果指定了 NULL 流，那么即使一个异步的内存复制也将使多个引擎之间的并发停止。

除了明确指定 NULL 流外，这些无意的“并发中断”的主要来源是隐式运行在 NULL 流的函数调用，因为它们没有使用流参数。当在 CUDA 1.1 中第一次引入流时，一些函数比如

cudaMemset() 和 cuMemcpyDtoD(), 以及类似 CUFFT 和 CUBLAS 库的接口, 也没有以任何方式为应用程序指定流参数。目前 Thrust 库仍不支持这一功能。CUDA Visual Profiler 会在它的报告中定位出并发中断。

6.6 映射锁页内存

映射锁页内存可用于 PCIe 传输和内核处理的重叠执行, 尤其是对于设备端到主机端的复制(这里不需要隐藏太长时间主机内存的延迟)。映射锁页内存比原生 GPU 内存复制有更严格的对齐要求, 因为它们必须被合并读取。使用非合并的内存事务处理会比使用映射锁页内存慢 2 ~ 6 倍。

我们对 concurrencyMemcpyKernelMapped.cu 程序作一简单移植会产生一个有趣的结果: 在亚马逊 EC2 的 cg1.4xlarge 实例上, 映射锁页内存 in cycles 低于 64 的情况下运行非常缓慢。

CYCLES 的值	锁页内存下时间	采用流的时间	加速比
8	95.15	43.61	0.46
16	96.70	43.95	0.45
24	95.45	44.27	0.46
32	97.54	44.61	0.46
40	94.09	44.93	0.48
48	94.25	45.26	0.48
56	95.18	46.19	0.49
64	28.22	49.29	1.75
72	31.58	52.38	1.66
...			
208	92.59	104.60	1.13
216	96.11	107.68	1.12

对于小的 cycles, 内核需要很长的时间来运行, 就好像 cycles 大于 200 一样。虽然只有英伟达能发现这种表现异常的确切原因, 但是它并不难解决: 通过展开内核的内部循环, 为每个线程创造更多的工作, 提高性能。

请注意, 代码清单 6-6 中这个版本的 AddKernel() 与代码清单 6-3 中的那个功能相同[⊖]。它只是计算每个循环迭代的 unrollFactor 输出。既然展开因子是一个模板参数, 编译器可以使用寄存器来保存 value 数组, 并且最内层的循环可以完全展开。

[⊖] 除非 N 不可被 unrollFactor 整除。这当然可以对 for 循环做一微小改变, 事后做一清理来进行修正。

代码清单 6-6 循环展开的 AddKernel()

```

template<const int unrollFactor>
__device__ void
AddKernel_helper( int *out, const int *in, size_t N, int increment, int cycles )
{
    for ( size_t i = unrollFactor*blockIdx.x*blockDim.x+threadIdx.x;
          i < N;
          i += unrollFactor*blockDim.x*gridDim.x )
    {
        int values[unrollFactor];

        for ( int iUnroll = 0; iUnroll < unrollFactor; iUnroll++ ) {
            size_t index = i+iUnroll*blockDim.x;
            values[iUnroll] = in[index];
        }
        for ( int iUnroll = 0; iUnroll < unrollFactor; iUnroll++ ) {
            for ( int k = 0; k < cycles; k++ ) {
                values[iUnroll] += increment;
            }
        }
        for ( int iUnroll = 0; iUnroll < unrollFactor; iUnroll++ ) {
            size_t index = i+iUnroll*blockDim.x;
            out[index] = values[iUnroll];
        }
    }
}

__device__ void
AddKernel( int *out, const int *in, size_t N, int increment, int cycles, int
unrollFactor )
{
    switch ( unrollFactor ) {
        case 1: return AddKernel_helper<1>( out, in, N, increment, cycles );
        case 2: return AddKernel_helper<2>( out, in, N, increment, cycles );
        case 4: return AddKernel_helper<4>( out, in, N, increment, cycles );
    }
}

```

对于 unrollFactor==1，其实现与代码清单 6-3 中相同。对于 unrollFactor==2，映射锁页方案比流方案显示出一些改进。转折点是从 cycles=64 下降到 cycles=48。对于 unrollFactor==4，性能同样优于流版本。

CYCLES 的值	锁页内存下的时间	采用流的时间	加 速 比
8	36.73	43.77	1.19
16	34.09	44.23	1.30
24	32.21	44.72	1.39
32	30.67	45.21	1.47
40	29.61	45.90	1.55
48	26.62	49.04	1.84
56	32.26	53.11	1.65
64	36.75	57.23	1.56
72	41.24	61.36	1.49

这些值是针对 32M 的整数给出的，所以程序需要读取和写入 128MB 的数据。对于

`cycles==48`, 程序运行在 26 毫秒内。为了达到此等有效带宽 (在 PCIe 2.0 上超过 9GB/s), GPU 在执行内核处理的同时, 通过 PCIe 进行并发读和写。

6.7 并发内核处理

SM 2.x 架构和更高级别的 GPU 能够并发运行多个内核, 只要它们是在不同流中被启动的并且有适当大小的内存块 (足够小, 从而一个内核不会填满整个 GPU) 的话。假使每个内核启动中线程块的数量足够小的话, 代码清单 6-5 (9 ~ 14 行) 将导致内核并发运行。由于内核之间只能通过全局内存通信, 我们可以添加以下代码到 `AddKernel()` 来跟踪同时运行的内核的数量。可以使用下面的“内核并发跟踪”结构。

```
static const int g_maxStreams = 8;
typedef struct KernelConcurrencyData_st {
    int mask; // mask of active kernels
    int maskMax; // atomic max of mask popcorn
    int masks[g_maxStreams];
    int count; // number of active kernels
    int countMax; // atomic max of kernel count
    int counts[g_maxStreams];
} KernelConcurrencyData;
```

我们可以在函数的开始和结尾分别添加代码到 `AddKernel()` 中来“登记”和“退房”。“登记”使用“内核 ID”参数 `kid` (一个传递给内核的值, 其范围介于 0 到 `NumStreams-1` 之间), 接着计算一个对应一个全局变量的内核 ID 的掩膜 (`kid` 左移一位), 并对该值进行原子或操作存入全局变量里。请注意, `atomicOR()` 在 OR 执行之前返回在内存该位置的值。结果, 当原子或操作执行时, 返回值对每一个活跃的内核都有一个位来对应。

同样地, 这段代码通过递增 `kernelData->count` 和调用在共享的全局内存上的 `atomicMax()` 来追踪活跃内核的数量。

```
// check in, and record active kernel mask and count
// as seen by this kernel.
if ( kernelData && blockIdx.x==0 && threadIdx.x == 0 ) {
    int myMask = atomicOr( &kernelData->mask, 1<<kid );
    kernelData->masks[kid] = myMask | (1<<kid);
    int myCount = atomicAdd( &kernelData->count, 1 );
    atomicMax( &kernelData->countMax, myCount+1 );
    kernelData->counts[kid] = myCount+1;
}
```

在内核的底部, 类似的代码清除掩膜并递减活跃内核的计数。

```
// check out
if ( kernelData && blockIdx.x==0 && threadIdx.x==0 ) {
    atomicAnd( &kernelData->mask, ~(1<<kid) );
    atomicAdd( &kernelData->count, -1 );
}
```

`kernelData` 参数代表在文件作用域内声明的 `__device__` 变量。

```
__device__ KernelConcurrencyData g_kernelData;
```

请记住，指向 `g_kernelData` 的指针必须通过调用 `cudaGetSymbolAddress()` 获得。可以编写引用 `&g_kernelData` 的代码，但是 CUDA 的语言集成特性将无法正确解析该地址。

`concurrencyKernelKernel.cu` 程序增加了对命令行选项 `blocksPerSM` 的支持，来指定内核启动的线程块的数量。它将生成活跃内核数量的报告。两个 `concurrencyKernelKernel` 的调用示例如下。

```
$ ./concurrencyKernelKernel -blocksPerSM 2
Using 2 blocks per SM on GPU with 14 SMs = 28 blocks
Timing sequential operations... Kernel data:
Masks: ( 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 )
Up to 1 kernels were active: (0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 )

Timing concurrent operations...
Kernel data:
Masks: ( 0x1 0x3 0x7 0xe 0x1c 0x38 0x60 0xe0 )
Up to 3 kernels were active: (0x1 0x2 0x3 0x3 0x3 0x2 0x3 0x3 )

$ ./concurrencyKernelKernel -blocksPerSM 3
Using 3 blocks per SM on GPU with 14 SMs = 42 blocks
Timing sequential operations... Kernel data:
Masks: ( 0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 )
Up to 1 kernels were active: (0x1 0x0 0x0 0x0 0x0 0x0 0x0 0x0 )

Timing concurrent operations... Kernel data:
Masks: ( 0x1 0x3 0x6 0xc 0x10 0x30 0x60 0x80 )
Up to 2 kernels were active: (0x1 0x2 0x2 0x2 0x1 0x2 0x2 0x1 )
```

需要注意的是，`blocksPerSM` 是每个内核启动的块数，所以 `numStreams*blocksPerSM` 个块是在 `numStreams` 个独立内核里启动的。你可以看到，当内核网格较小时，硬件可以并发运行更多的内核。但是对于本章讨论的工作负载，并发内核处理没有带来性能上的好处。

6.8 GPU/GPU 同步：`cudaStreamWaitEvent()`

到现在为止，本章描述的所有同步函数都已经和 CPU/GPU 同步相关联。它们或者等待或者查询 GPU 操作的状态。`cudaStreamWaitEvent()` 函数是与 CPU 异步的，使指定的 `stream` 等待直到事件被记录。流和事件不必与同一个 CUDA 设备相关联。9-3 节介绍了如何进行 GPU 之间的同步，并且使用了这一特性来实现一个点对点的内存复制（见代码清单 9-1）。

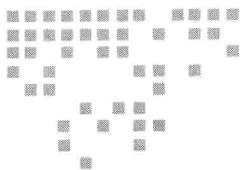
多 GPU 的流和事件：注意事项和限制

- 流和事件存在于上下文（或设备）的作用域内。当调用 `cuCtxDestroy()` 或 `cudaDeviceReset()` 时，相关的流和事件都会被破坏。
- 内核启动和 `cu(da)EventRecord()` 仅仅能在同一个上下文 / 设备中使用 CUDA 流。
- `cudaMemcpy()` 可以从任何流调用，但最好是源上下文 / 设备中调用。
- `cudaStreamWaitEvent()` 可以使用任何流、被任何事件调用。

6.9 源代码参考

本章中引用的源代码放置在 concurrency 目录。

文件名	描述
<code>breakevenDtoHMemcpy.cu</code>	在数据复制的开销超过驱动程序开销之前，测量一个异步的设备到主机端的内存复制的大小
<code>breakevenHtoDMemcpy.cu</code>	在数据复制的开销超过驱动程序开销之前，测量一个异步的主机到设备端的内存复制的大小
<code>breakevenKernelAsync.cu</code>	测量一个内核要超过驱动程序开销必须要做的工作量
<code>concurrencyKernelKernel.cu</code>	测量一个 GPU 内部内核与内核之间的并发性
<code>concurrencyKernelMapped.cu</code>	测量使用映射锁页内存的并发内存复制 / 内核处理相对于采用流的速度
<code>concurrencyMemcpyKernel.cu</code>	测量在不同大小的内核工作量下由于并发内存复制和内核处理带来的加速
<code>concurrencyMemcpyKernelMapped.cu</code>	测量由于使用映射锁页内存的并发内核执行带来的加速
<code>memcpy16.cpp</code>	SSE 优化的内存复制子程序
<code>nullDtoHMemcpyAsync.cu</code>	测量一个字节的异步的设备到主机端的内存复制时的吞吐量
<code>nullDtoHMemcpySync.cu</code>	测量一个字节的同步的设备到主机端的内存复制时的吞吐量
<code>nullHtoDMemcpyAsync.cu</code>	测量一个字节的异步的主机到设备端的内存复制时的吞吐量
<code>nullKernelAsync.cu</code>	测量异步内核启动的吞吐量
<code>nullKernelSync.cu</code>	测量同步内核启动的吞吐量
<code>pageableMemcpyHtoD.cu</code>	分页内存复制程序的示例，使用标准的 CUDA 编程框架。使用内存复制
<code>pageableMemcpyHtoD16.cu</code>	分页内存复制程序的示例，使用标准的 CUDA 编程框架
<code>pageableMemcpyHtoD16Blocking.cu</code>	与 <code>pageableMemcpyHtoD16.cu</code> 相同，但是使用阻塞事件进行同步
<code>pageableMemcpyHtoD16Broken.cu</code>	除了与事件同步之外，与 <code>pageableMemcpyHtoD16.cu</code> 相同
<code>pageableMemcpyHtoD16Synchronous.cu</code>	与 <code>pageableMemcpyHtoD16.cu</code> 相同，但在稍微不同的位置上使用了事件同步，从而中断了 CPU/GPU 的并发性
<code>peer2peerMemcpy.cu</code>	通过可共享锁页缓冲区中转的点对点内存复制



内核执行

本章将详细描述内核程序是如何在 GPU 上执行的。具体包括以下内容：它们如何启动，有哪些执行特性，如何由线程（thread）组成线程块（block）再由线程块组成网格（grid），以及资源管理涉及哪些因素。此外，本章还包括对动态并行的描述——它是 CUDA 5.0 的新特性，可以使 CUDA 内核程序为 GPU 启动工作任务。

7.1 概况

CUDA 内核程序在 GPU 上执行，并且从最早期的 CUDA 版本开始就一直与 CPU 并发执行。换句话说，内核启动是异步的：控制权会在 GPU 完成请求的操作之前返回给 CPU。在 CUDA 最初引进时，开发者并不需要关心内核启动的异步执行（或缺少此机制）。此外，数据显式地复制到 GPU 并从 GPU 复制回，而内存复制命令则会在启动内核程序的命令请求后进行排队。想要通过编写 CUDA 代码来暴露内核启动的异步特性是不可能的，异步执行的主要附带作用是在连续执行多个内核启动时隐藏驱动程序的开销。

从前面对映射锁页内存（主机内存可以直接被 GPU 访问）的介绍中可以看出，特别是对于写入主机内存的（相反的，从其中读取的）内核程序而言，内核启动的异步执行是多么的重要。如果一个内核在没有显式同步（例如采用 CUDA 事件）的情况下启动或写入主机内存，代码将面临 CPU 与 GPU 之间的竞争并会出现运行错误。对于通过映射锁页内存进行读操作的内核，一般不需要显式同步，这是因为 CPU 的任何待定写操作都会在内核启动之前被发送出来。要是内核程序通过写入映射锁页内存来给 CPU 返回结果，就必须使用同步避免读后写了。

一旦内核启动，它以一个网格的形式运行，其中该网格包含多个线程块，每个线程块又由多个线程构成。但并非线程块的运行都是并发的。每个线程块都被分配给一个 SM，而每个 SM 都可以为多线程块维持上下文。为了掩盖内存和指令带来的延时，SM 通常会需要多于单线程块可以包含的线程束（SM 2.0 及以上，一个线程块最多可以包含 1024 条线程）。每个 SM 中线程块的最大数目是无法通过 API 查询的，但在英伟达文档中已经写明，该数目在 SM 3.x 版本之前的硬件中为 8，SM 3.x 及其之后的为 16。

编程模型并不能保证执行的次序，抑或是某些线程块或线程是否可以并发运行。开发者一定不要假定内核启动中的所有线程都是并发执行的。通常很容易就启动了比机器所能维持的要多的线程，而且其中的一些在其他线程结束之前不会执行。鉴于线程次序无法确定，甚至连内核启动开始处进行全局内存的初始化都是个困难的任务。

动态并行——特斯拉 K20(GK110) (第一个支持 SM 3.5 的 GPU) 中新增的特性，使内核程序可以启动其他的内核程序，并完成它们之间的同步。这些特性解决了一些以前的硬件上 CUDA 所呈现的局限性。例如，一个动态并行内核可以通过启动和等待一个子网格来完成初始化。

7.2 语法

在使用 CUDA 运行时的时候，内核启动由我们常用的三对尖角括号语法来指定。

```
Kernel<<<gridSize, blockSize, sharedMem, Stream>>>( Parameters... )
```

其中，Kernel 指定待启动的内核名称；

gridSize 指定一个 dim3 结构形式的网格的大小；

blockSize 指定了一个 dim3 结构形式的线程块的维度；

sharedMem 指定为每个线程块预留的附加的共享内存^①；

Stream 指定内核启动所属于的流。

通常，指定网格和块大小的 dim3 结构包含有 3 个成员变量 (x、y 和 z)。当使用 C++ 进行编译时，一个带有系统默认参数的构造函数会将 x、y 和 z 均默认初始化为 1。详见代码清单 7-1，摘自 NVIDIA SDK 中的 vector_types.h 头文件。

代码清单 7-1 dim3 结构

```
struct __device_builtin__ dim3
{
    unsigned int x, y, z;
#if defined(__cplusplus)
    __host__ __device__ dim3(
        unsigned int vx = 1,
        unsigned int vy = 1,
```

^① 内核程序可用的共享内存的数目为在此参数值上加上内核中静态申请的共享内存数目。

```

    unsigned int vz = 1) : x(vx), y(vy), z(vz) {}
__host__ __device__ dim3(uint3 v) : x(v.x), y(v.y), z(v.z) {}
__host__ __device__ operator uint3(void) {
    uint3 t;
    t.x = x;
    t.y = y;
    t.z = z;
    return t;
}
#endif /* __cplusplus */
};

```

内核程序可以通过驱动程序 API 使用 cuLaunchKernel() 来启动，虽然 cuLaunchKernel() 将网格和线程块的维度作为独立参数而不是 dim3 结构。

```

CUresult cuLaunchKernel (
    CUfunction kernel,
    unsigned int gridDimX,
    unsigned int gridDimY,
    unsigned int gridDimZ,
    unsigned int blockDimX,
    unsigned int blockDimY,
    unsigned int blockDimZ,
    unsigned int sharedMemBytes,
    CUstream hStream,
    void **kernelParams,
    void **extra
);

```

对于三对尖角括号语法，cuLaunchKernel() 的参数包括了调用的内核、网格和线程块大小、共享内存数目以及流。其主要的区别在于内核本身的这些参数如何给出：因为 ptxas 发射出的内核微码包含了描述各个内核参数的元数据[⊖]，所以 kernelParams 是一个 void* 类型的数组，每个元素对应内核的一个参数。又因为参数的类型可以被驱动程序识别，所以参数所占的内存数据（int 为 4 字节、double 为 8 字节，诸如此类）将会作为用来调用内核的、具有硬件特性的命令的一部分复制到命令缓冲区中。

7.2.1 局限性

所有参与内核启动的 C++ 类都必须是带有以下特性的“简单旧数据”（plain old data, POD）。

- 无用户声明的构造函数；
- 无用户定义的复制分配操作符；
- 无用户定义的析构函数；
- 无非 POD 的非静态数据成员；
- 无私有或保护型的非静态数据；
- 无基类；

[⊖] cuLaunchKernel() 不适用于那些没有在 CUDA 3.2 或更高版本上编译的二进制映像文件，因为 CUDA 3.2 是第一个可包含内核参数元数据的版本。

- 无虚函数。

注意，违背了这些规则的类也可以用于 CUDA，甚至是 CUDA 内核程序中。只是，它们绝对不能使用于内核启动中。如此，一个 CUDA 内核使用的类可以通过使用来自内核启动的 POD 输入数据来构造。

CUDA 内核程序也没有返回值。它们必须通过设备内存（必须显式地复制回 CPU）或映射主机内存来传回结果。

7.2.2 高速缓存和一致性

GPU 包含了多个高速缓存以便于在发生重用时加速计算。其中，常量缓存（constant cache）被充分利用起来以便于广播式传输到同一个 SM 的执行单元；而纹理缓存则减少了外部带宽的使用。这两种缓存都不能很好地同 GPU 的写内存操作保持一致性。例如，没有促使这些缓存和一、二级缓存之间保持一致性的协议，所以无法减少全局内存的延迟和支持聚合带宽。这意味着两件事：

1) 当一个内核在运行时，注意不要对那些同时（或者是被一个并发运行的内核）正在通过常量内存和纹理内存进行访问的内存执行写内存的操作。

2) CUDA 驱动程序必须在每个内核启动之前使常量缓存和纹理缓存无效。

对于不含 TEX 指令的内核程序，CUDA 驱动程序不需要使常量缓存和纹理缓存无效。因此，未使用纹理的内核程序引发更少的驱动程序开销。

7.2.3 异步与错误处理

内核启动是异步的，因为一旦内核被提交到硬件就会立即与 CPU 并行执行^②。这一异步会使错误处理变得很复杂。如果一个内核遇到一个错误（例如，它读入一个无效内存位置），该错误有时会在内核启动后的一段时间才能传输到驱动程序（和应用程序）中。而检查这种错误最可靠的方法是使用 `cudaDeviceSynchronize()` 或 `cuCtxSynchronize()` 函数以使其与 GPU 同步。如果在内核执行中出现一个错误，它们将会返回“unspecified launch failure”的错误代码。

除了 `cudaDeviceSynchronize()` 或 `cuCtxSynchronize()` 等显式 CPU/GPU 同步函数外，这个错误代码还可能是来自与 CPU 隐式同步的函数，如同步的内存复制调用。

无效内核启动

有可能所请求的内核启动无法为硬件执行。例如，指定比硬件能够支持的块内线程数还要多的线程就会出现这种情况。驱动程序会尽可能检测到这些情况并报告错误，而不是试图将该启动提交给硬件。

^② 在大多数平台上，内核会在 CPU 处理完启动命令之后的数个微秒开始在 GPU 执行。但在 WDDM (Windows Display Driver Model) 平台上，它可能需要更长的时间。因为驱动程序必须执行一个内核转换以便将启动提交到硬件中，并且在用户模式下 GPU 的工作任务会进入队列，平摊用户态到内核态过渡的开销。

CUDA 运行时和驱动程序 API 以不同的方式处理这种情况。当一个无效的参数被指定时，驱动程序 API 的显式 API 调用（例如 cuLaunchGrid() 和 cuLaunchKernel() 等函数）返回错误代码。但是，当使用 CUDA 运行时的时候，由于内核是按 C/C++ 一行代码启动的，故而没有 API 调用来返回错误代码。对应地，那个错误会被“记录”到本地线程槽中，而应用程序则可以通过 cudaGetLastError() 函数来查询该错误值。与此相同的错误处理机制还被用在因其他原因（例如，内存访问冲突）导致的无效内核启动上。

7.2.4 超时

GPU 在 CUDA 内核执行期间不能进行上下文切换，所以长时间运行的 CUDA 内核可能会对使用 GPU 来与用户交互的系统的交互性产生负面影响。因此，如果 GPU 运行太久而没有进行上下文切换，许多 CUDA 系统将实施“超时”以重置它。

在 WDDM (Windows Display Driver Model) 上，超时是由操作系统执行的。微软已经阐明这个“超时检测和恢复”(Timeout Detection and Recovery, TDR) 是如何工作的，详见 <http://bit.ly/WPPSdQ> (其中包括控制 TDR 行为的注册表项)[⊖]。虽然特斯拉计算集群 (Tesla Compute Cluster, TCC) 驱动程序并非适用于所有硬件，但 TDR 确实可以通过它进行安全禁用。

在 Linux 上，NVIDIA 的驱动程序默认 2 秒超时。未用于显示的第二块 GPU 并不实施超时。开发者可以查询一个运行时限制是否在一个给定 GPU 上启用，具体通过调用带 CU_DEVICE_ATTRIBUTE_KERNEL_EXEC_TIMEOUT 标志的 cuDeviceGetAttribute() 函数，或检查 cudaDeviceProp::kernelExecTimeoutEnabled 属性来实现。

7.2.5 本地内存

由于本地内存是线程私有的，且 CUDA 中的网格可以包含数千个线程，因此 CUDA 网格需要的本地内存数目是相当多的。CUDA 开发者会用心预先分配资源以降低内核启动等操作因为缺乏资源而失败的可能性。但就本地内存而言，仅一个保守的分配就已经消耗了太多的内存。缘于此，使用大量本地内存的内核需要更长的时间且有可能是同步执行的，因为 CUDA 驱动程序必须在执行内核启动之前分配好内存。此外，如果内存分配失败，内核也将因缺乏资源而启动失败。

默认情况下，当 CUDA 驱动程序必须分配本地内存以运行内核时，它会在内核完成之后释放内存。另外，这种行为还会使得内核启动同步。但这种行为可以通过给 cuCtxCreate() 指定 CU_CTX_LMEM_RESIZE_TO_MAX 标志或在创建主上下文之前调用 cudaSetDeviceFlags() 函数置 cudaDeviceLmemResizeToMax 来禁止。它带来的结果是，在一个需要比默认更多本地内存的内核启动之后，增加的可用本地内存不会被释放。

[⊖] 当然，修改注册表只应在测试时使用。

7.2.6 共享内存

共享内存会在内核启动时进行分配，并且它会在内核执行期间而一直保持。除了可以在内核中声明的静态分配方法之外，共享内存也可以被声明为一个未指定大小的 `extern` 变量。此时，用于分配大小不定的数组的共享内存量由内核启动的第三个参数或者是 `cuLaunchKernel()` 的 `sharedMemBytes` 参数指定。

7.3 线程块、线程、线程束、束内线程

内核以线程块构成的网格进行启动。这些线程可以进一步分为 32 个线程组成的线程束（warp），而每个线程束中的单个线程被称为一个束内线程。

7.3.1 线程块网格

线程块独立被调度到 SM 中，来自于同一线程块的线程在同一 SM 中执行。如图 7-1 显示的是二维线程块 ($8W \times 8H$) 组成的一个二维网格 ($8W \times 6H$)。图 7-2 显示的是三维线程块 ($8W \times 8H \times 4D$) 组成的一个三维网格 ($8W \times 6H \times 6D$)。

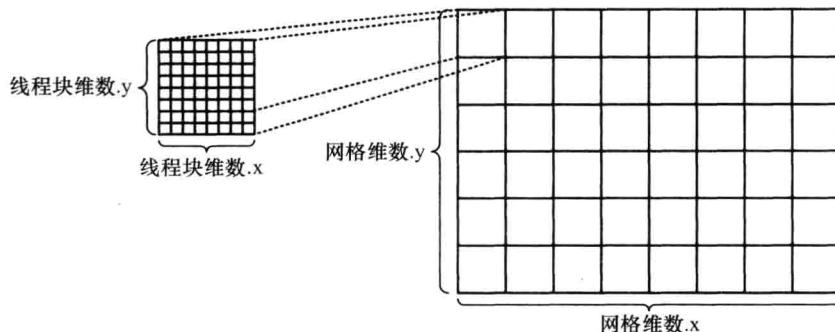


图 7-1 二维网格与线程块

网格可以由高达 65535×65535 个线程块（对于 SM1.0 的硬件）或 $65535 \times 65535 \times 65535$ 的线程块（对于 SM2.0 的硬件）组成。^① 每个线程块可以由高达 512 或 1024 个线程组成^②，而线程块中的线程之间可以通过 SM 的共享内存进行通信。一个网格中的线程块有可能会被分配给不同的 SM。为了使硬件吞吐量最大化，一个给定的 SM 可以在同一时间内运行来自不

^① 网格的最大尺寸可以通过 `CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_X`、`CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Y`、`CU_DEVICE_ATTRIBUTE_MAX_GRID_DIM_Z` 查询，亦可以通过调用 `cudaGetDeviceProperties()` 函数并检查 `cudaDeviceProp::maxGridSize` 进行查询。

^② 线程块的最大尺寸可以通过 `CU_DEVICE_ATTRIBUTE_MAX_THREADS_PER_BLOCK`，或 `deviceProp.maxThreadsPerBlock` 查询。

同线程块的线程与线程束。当所需要的资源变得可用，线程束的调度器会分派指令。

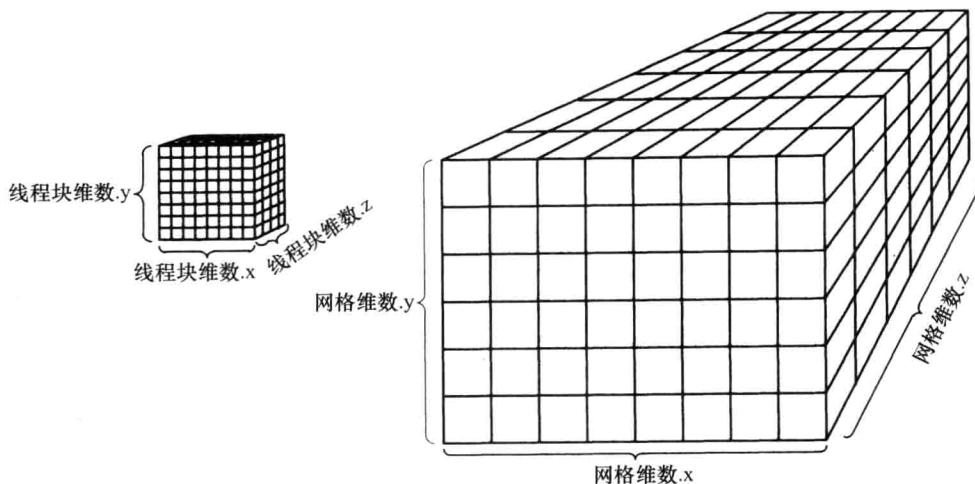


图 7-2 三维网格与线程块

1. 线程

每个线程都可以得到属于自己的一组完整的寄存器^①和一个线程块中唯一的线程 ID。为了避免传递网格和线程块的尺寸到每一个内核，这些尺寸在内核运行时是可读取的。而用来引用这些寄存器的内置变量已经在表 7-1 中给出。它们都是 dim3 类型的。

表 7-1 引用寄存器的内置变量

内置变量	说 明
gridDim	网格的维数 (线程块中)
blockDim	线程块的维数 (线程中)
blockIdx	线程块索引 (网格里)
threadIdx	线程索引 (线程块里)

综上，这些变量可以用来推断一个线程将作用到问题的哪个部分。一个线程的“全局”索引可以如下计算：

```
int globalThreadId =
    threadIdx.x+blockDim.x*(threadIdx.y+blockDim.y*threadIdx.z);
```

2. 线程束、束内线程以及 ILP

线程是成组执行的，按照 SIMD 的方式，每 32 个线程称为一个线程束，这类似于放置

① 每个线程所需的寄存器越多，在一个给定 SM 中可以分配到寄存器的线程也就越少。而一个 SM 中实际执行的线程束与其最大理论值的百分比称为占用率（详见 7-4 节）。

于“织布机”的一组平行织线^①(见图 7-3)。32 个线程都执行同一个指令，且每个线程都使用私有寄存器进行这一请求操作。针对上述比喻，一个线程处于线程束中的 ID 将称为束内线程号(lane)。

线程束 ID 和束内线程 ID 可以用如下的全局线程 ID 进行计算：

```
int warpID = globalThreadId >> 5;
int laneID = globalThreadId & 31;
```

线程束是一个很重要的执行单元，因为它们是 GPU 可以隐藏延迟的最小粒度。关于 GPU 是如何使用线程束间的并行来隐藏内存延迟的，目前已有大量文档可查。满足全局内存请求需要数百个时钟周期，所以当遇到一个纹理获取或读取时，GPU 会发出内存请求，然后在数据到达之前调度其他线程束的指令。一旦数据到达，线程束将再次变得有条件执行。

而言之较少的是 GPU 还通过发掘并行性来利用“指令级并行”(instruction level parallelism, ILP)。ILP 是指在程序执行过程中发生的细粒度的并行机制，例如，当计算 $(a+b)*(c+d)$ 时，加法运算 $a+b$ 和 $c+d$ 会在乘法运算执行之前并行地执行。因为 SM 已经包含有大量的用来跟踪依赖性和隐藏延迟的逻辑，它们都非常善于通过并行(这实际上是 ILP)以及内存延迟来隐藏指令延时。GPU 对 ILP 的支持是循环展开能够成为如此有效的一个优化策略的一部分原因，除了略微减少了每个循环迭代中的指令数外，它还为线程束调度器提供了更多的并行。



图 7-3 织布机

3. 对象作用域

可能会被内核网格引用的对象的作用域，从最小的本地(每个线程中的寄存器)到最大的全局(每个网格的全局内存和纹理引用)，都已被总结于表 7-2。在动态并行出现之前，线程块主要充当在一个线程块中的线程间同步(通过 `_syncthreads()` 等内建函数)和通信(通过共享内存)的一种机制。由于在内核中创建的流和事件仅适用于同一个线程块中的线程，动态并行把资源管理添加到它们的组合之中。

表 7-2 对象作用域

对 象	作 用 域	对 象	作 用 域
寄存器	线程	全局内存	网格
共享内存	线程块 ^①	纹理引用	网格
本地内存	线程束	流 ^②	线程块
常量内存	网格	事件 ^②	线程块

^①一个内核要执行的话只需要有足够的本地内存来服务于最大数目的活跃线程束即可。

^②流与事件只能由 CUDA 内核使用动态并行来创建。

⊕ 线程束大小是可查询的，但考虑到硬件的兼容性问题，开发者在可预期的未来，可将其固定当作 32 个。

7.3.2 执行保证

开发者永远不要臆断线程块或线程执行的顺序是相当重要的。特别地，我们无法知道哪个线程块或线程会先执行，所以一般进行初始化应该由内核调用之外的代码执行。

执行保证与块间同步

在一个给定线程块中的线程保证驻留于同一 SM，所以它们可以通过共享内存交换信息以及使用 `_syncthreads` 等内建函数进行同步执行。但线程块没有任何类似进行数据交换或同步的机制。

富有经验的 CUDA 开发者可能会问——“在全局内存中使用原子操作会如何呢？”全局内存可以在一个使用原子操作的线程下安全地更新，所以它倾向于构建类似 `_syncthreads()` 的 `_syncblocks()` 函数，让内核启动时的所有线程块在此处汇聚。也许在一个全局内存位置上还会执行一次 `atomicInc()`，若该函数没有返回结果，则会轮询内存位置到返回为止。

问题是，内核的执行模式（例如，线程块映射到 SM）会随着硬件配置的变化而变化。例如，SM 的数量就是个限制因素，除非 GPU 上下文大到足够容纳所有的网格，有些线程块将会在其他线程块还没开始运行时已完成执行。这会导致死锁：因为并非所有的线程块都要驻留于 GPU，所以正在轮询共享内存位置的线程块会阻止内核启动中的其他线程块执行。

这有几个块间同步有效的特殊情况。`atomicCAS()` 可以用来实现简单的互斥。另外，线程块可以在它们完成时使用原子来发信号，所以网格中的最后一个线程块可以在退出之前执行一些操作，以表明其他线程块都已执行完毕。采用这种策略的 `threadFenceReduction` SDK 示例和 `reduction4SinglePass.cu` 示例，可以在这本书中找到（详见 12.2 节）。

7.3.3 线程块与线程 ID

一组特殊的只读寄存器会以线程 ID 和线程块 ID 的形式提供每个线程的上下文。线程与线程块 ID 会在一个 CUDA 内核开始执行时分配。对于 2D、3D 网格和线程块，它们会以行优先顺序（row-major order）进行分配。

线程块的大小最好是 32 的倍数，因为线程束是在 GPU 上执行的最细粒度。图 7-4 显示了线程 ID 在 $16W \times 2H$ 、 $32W \times 1H$ 和 $8W \times 4H$ 规格的 32 线程构成的线程块中分别是如何分配的。

对于那些线程数不是 32 的倍数的线程块，某些线程束中并非所有的线程都是活跃的。图 7-5 就显示了 $14W \times 2H$ 、 $28W \times 1H$ 和 $7W \times 4H$ 规格的 28 线程构成的线程块中线程 ID 的分配情况。在上述的每一种情况下，32 线程的束中有 4 个线程在内核启动期间是无效的。对于任何大小不是 32 的倍数的线程块而言，一些执行资源会被浪费掉，因为某些线程束会包含整个内核执行期间禁用的束内线程进行启动。2D 或 3D 的线程块或网格并没有太大的性能优势，但它们有时会有助于与应用程序更好的匹配。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
线程块维数= (32,1,1)																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
线程块维数= (16,2,1)																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

图 7-4 32 线程构成的线程块

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
线程块维数= (28,1,1)																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
线程块维数= (14,2,1)																															
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

图 7-5 28 线程块

如例 7-2 所示，reportClocks.cu 程序说明了线程 ID 是如何分配的以及基于线程束的执行通常是如何工作的。

例 7-2 WriteClockValues 内核

```
global__ void
WriteClockValues(
    unsigned int *completionTimes,
    unsigned int *threadIDs
)
{
    size_t globalBlock = blockIdx.x+blockDim.x*
        (blockIdx.y+blockDim.y*blockIdx.z);
    size_t globalThread = threadIdx.x+blockDim.x*
        (threadIdx.y+blockDim.y*threadIdx.z);

    size_t totalBlockSize = blockDim.x*blockDim.y*blockDim.z;
```

```

size_t globalIndex = globalBlock*totalBlockSize + globalThread;
completionTimes[globalIndex] = clock();
threadIDs[globalIndex] = threadIdx.y<<4|threadIdx.x;
}

```

WriteClockValues() 函数中使用由线程块和线程 ID 以及网格和线程块大小计算的全局索引来写入 2 个输出指针。第一个输出指针用来接收内建 clock() 的返回值。该返回值为一个高分辨率计时器值，随每个线程束不断增加。在这个程序中，我们使用 clock() 来确定是哪个线程束处理给定值的。clock() 的返回值是每个 SM 私有的时钟周期计数器，通过和从所有的时钟周期值计算出的最小值做减法来使该值规格化。我们称该结果值为线程的“完成时间”。

让我们来看看在一个 $16W \times 8H$ 规格的线程块中线程的完成时间（例 7-3），并与 $14W \times 8H$ 规格中的完成时间（例 7-4）进行比较。正如预期的那样，它们被分为 32 组，对应于线程束的大小。

例 7-3 $16W \times 8H$ 规格线程块的完成时间

```

0.01 ms for 256 threads = 0.03 us/thread
Completion times (clocks):
Grid (0, 0, 0) - slice 0:
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
  6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6
  6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6
  8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8
  8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8   8
  a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a
  a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a   a
Grid (1, 0, 0) - slice 0:
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
  2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
  4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4   4
  6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6
  6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6   6
Thread IDs:
Grid (0, 0, 0) - slice 0:
  0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
  10  11  12  13  14  15  16  17  18  19  1a  1b  1c  1d  1e  1f
  20  21  22  23  24  25  26  27  28  29  2a  2b  2c  2d  2e  2f
  30  31  32  33  34  35  36  37  38  39  3a  3b  3c  3d  3e  3f
  40  41  42  43  44  45  46  47  48  49  4a  4b  4c  4d  4e  4f
  50  51  52  53  54  55  56  57  58  59  5a  5b  5c  5d  5e  5f
  60  61  62  63  64  65  66  67  68  69  6a  6b  6c  6d  6e  6f
  70  71  72  73  74  75  76  77  78  79  7a  7b  7c  7d  7e  7f
Grid (1, 0, 0) - slice 0:
  0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
  10  11  12  13  14  15  16  17  18  19  1a  1b  1c  1d  1e  1f
  20  21  22  23  24  25  26  27  28  29  2a  2b  2c  2d  2e  2f
  30  31  32  33  34  35  36  37  38  39  3a  3b  3c  3d  3e  3f
  40  41  42  43  44  45  46  47  48  49  4a  4b  4c  4d  4e  4f
  50  51  52  53  54  55  56  57  58  59  5a  5b  5c  5d  5e  5f
  60  61  62  63  64  65  66  67  68  69  6a  6b  6c  6d  6e  6f
  70  71  72  73  74  75  76  77  78  79  7a  7b  7c  7d  7e  7f

```

例 7-4 14W×8H 规格线程块的完成时间

```

Completion times (clocks):
Grid (0, 0, 0) - slice 0:
 6 6 6 6 6 6 6 6 6 6 6 6 6 6
 6 6 6 6 6 6 6 6 6 6 6 6 6 6
 6 6 6 6 8 8 8 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 a a a a a a a
a a a a a a a a a a a a a a a
a a a a a a a a a a a a c c
c c c c c c c c c c c c c c
Grid (1, 0, 0) - slice 0:
 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 2 2 2 2 2 2
 2 2 2 2 2 2 2 2 4 4 4 4 4 4
 4 4 4 4 4 4 4 4 4 4 4 4 4 4
 4 4 4 4 4 4 4 4 4 4 4 4 6 6
 6 6 6 6 6 6 6 6 6 6 6 6 6 6
Thread IDs:
Grid (0, 0, 0) - slice 0:
 0 1 2 3 4 5 6 7 8 9 a b c d
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d
Grid (1, 0, 0) - slice 0:
 0 1 2 3 4 5 6 7 8 9 a b c d
10 11 12 13 14 15 16 17 18 19 1a 1b 1c 1d
20 21 22 23 24 25 26 27 28 29 2a 2b 2c 2d
30 31 32 33 34 35 36 37 38 39 3a 3b 3c 3d
40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d
60 61 62 63 64 65 66 67 68 69 6a 6b 6c 6d
70 71 72 73 74 75 76 77 78 79 7a 7b 7c 7d

```

例 7-4 中给出的 14W×8H 规格线程块的完成时间强调了线程 ID 是如何映射到线程束的。在 14W×8H 规格下，每一个线程束中只有 28 个线程。在内核执行的整个过程中，可能的线程束内 12.5% 的线程数目是空闲的。为了避免这种浪费，开发者始终应该尽量确保线程块包含 32 倍数的线程。

7.4 占用率

占用率 (Occupancy) 指的是在一个给定的内核启动内，SM 中运行的线程数与其可容纳的最大线程数的比值。

$$\frac{\text{每 SM 包含的线程束数目}}{\text{每 SM 可容纳的最大线程束数目}}$$

该表达式的分母（每个 SM 中线程束的最大数目）是一个常数，仅由设备的计算能力决

定。而式中的分子则决定了占用率的大小，为下列各项的函数。

- 计算能力 (1.0, 1.1, 1.2, 1.3, 2.0, 2.1, 3.0, 3.5);
- 每个线程块中的线程;
- 每个线程的寄存器;
- 共享内存配置^①;
- 每个线程块的共享内存。

为了帮助开发者评估这些参数之间的权衡，CUDA 工具包以 Excel 电子表格^②的形式包含了一个占用率计算器。如果提供以上的输入，电子表格将会对如下各项进行计算。

- 活跃线程数;
 - 活跃线程束数;
 - 活动线程块数;
 - 占用率 (活跃线程束数目除以硬件可支持的线程束的最大数目)。
- 该电子表格还可以识别是哪个参数限制了占用率。
- 每个 SM 中的寄存器;
 - 每个 SM 中线程束或线程块的最大数量;
 - 每个 SM 的共享内存。

请注意，占用率并不是 CUDA 性能的全部或者终结^③。通常，更多地使用每个线程中的寄存器和依靠指令级并行 (ILP) 可以提升性能。在讨论线程束和占用率的权衡方面，英伟达公司已经作出很好的陈述^④。

在第 5.2.10 小节的代码清单 5-5 中，给出了一个低占用率的内核却实现近乎全局内存最大带宽的例子。GlobalReads 内核的内部循环可以根据一个模板参数展开。随着展开的迭代次数的增加，所需的寄存器数目会增加，且占用率会下降。例如，对于实例 cg1.4xlarge 下的 Tesla M2050，读取带宽的峰值显示 (ECC 禁用下) 为 124GiB/s，占用率却只有 66%。VolRov 报告了内核的占用率不到 10%，却实现接近峰值的内存带宽。

7.5 动态并行

动态并行 (Dynamic parallelism) 是一个仅适用于 SM3.5 架构硬件的新功能，可以使 CUDA 内核启动其他 CUDA 内核，同时也可以在 CUDA 运行时中调用各种函数。当使用动

^① 仅对应于 SM 2.x 或更高级的版本。开发者可以将 SM 中 64KB 的一级缓存分为 16KB 的共享内存与 48KB 的一级缓存，抑或是 48KB 的共享内存与 16KB 的一级缓存 (SM 3.x 新增了划分为 32KB 共享内存与 32KB 一级缓存的功能)。

^② 通常它位于工具的子目录中——例如，%CUDA_PATH%/tools(Windows) 或 \$CUDA_PATH/tools 中。

^③ Vasily Volkov 在他的报告 “Better Performance at lower occupancy” 中着重强调了这点。详见 <http://bit.ly/YdScNG>。

^④ <http://bit.ly/WHTb5m>。

态并行时，CUDA 运行时的一个子集（即所谓的“设备运行时”）可以为设备上运行的线程使用。

动态并行引入“父”和“子”网格的术语。任何被另一个 CUDA 内核启动的内核（而不是主机代码，正如所有以前的 CUDA 版本）称为一个“子内核”，而调用它的为“父内核”。默认情况下，CUDA 支持两级嵌套（一个给父网格，另一个给子网格），这个数目可以通过调用带 `cudaLimitDevRuntimeSyncDepth` 标志的 `cudaSetDeviceLimit()` 函数来增加。

动态并行为处理应用程序而设计。而在它出现之前 GPU 必须将结果发送给 CPU 以便于 CPU 可以指定在 GPU 上要执行哪些工作。这样的“握手”会扰乱第 2.5.1 小节中描述的执行流水线中的 CPU/GPU 并发，其中 CPU 会产生为 GPU 使用的命令。GPU 的时间很宝贵，GPU 不会在分配到更多工作之前一直等待 CPU 读取和分析结果。动态并行会让 GPU 在内核中为自己启动任务以避免这些流水线中的“气泡”。

动态并行可以提高如下性能。

- 在内核可以开始执行之前，给内核所需要的数据结构进行初始化。在此之前，这样的初始化要在主机代码中处理，或由之前启动的一个单独内核处理。
- 在诸如 Barnes-Hut 引力积分或空气动力学模拟的分层网格评估等应用中，它可以简化递归。



注意 动态并行只在一个给定 GPU 中有效。内核可以调用内存复制操作或其他内核，但却无法把工作提交给其他 GPU。

7.5.1 作用域和同步

子网格会继承其父网格的线程块和网格大小之外的绝大部分内核配置参数，如共享内存配置（由 `cudaDeviceSetCacheConfig()` 设置）。线程块是作用域的基本单位：一个线程块创建的流和事件仅能被该线程块使用（它们甚至不会被子网格继承），并且它们会在该线程块退出时自动销毁。



注意 通过动态并行在设备上创建的资源与在主机上创建的资源是严格分开的。在主机上创建的流和事件不能通过动态并行在设备上使用，反之亦然。

CUDA 确保在所有子网格完成之前，父网格是不会完成的。虽然父网格与子网格可以并发执行，但这却不能保证一个子网格会在它的父网格调用 `cudaDeviceSynchronize()` 之前就开始执行。

如果线程块中的所有线程都退出，该线程块的执行将会被挂起，直到所有子网格完成为止。如果默认同步不够用，开发者可以使用 CUDA 流和事件来显式同步。与主机上一样，在

一个给定流中的操作以子任务的顺序执行。当这些操作被指定到不同的流中，它们就只能并发执行了，并且还不能保证它们实际上会并发执行。如果需要，可以使用 `_syncthreads()` 等同步原语来协调给定流中子任务的顺序。



注意 在设备上创建的流和事件不能在创建它们的线程块之外使用。

`cudaDeviceSynchronize()` 函数对由块中任何线程启动的挂起工作进行同步。然而，它不执行任何线程间的同步。因此，如果要在其他线程启动的工作上同步的话，开发者必须使用 `_syncthreads()` 或其他块级同步原语（详见 8.6.2 小节）。

7.5.2 内存模型

父与子网格共享相同的全局与常量内存存储，但是它们有不同的本地和共享内存。

1. 全局内存

如下两种情况下，子网格在执行时，其内存视图与父网格具有完全的一致性：子网格被父网格调用，以及子网格被父线程中同步 API 调用产生的信号导致结束。

所有父线程中的全局内存操作对子网格来说是可见的。父网格与子网格同步完成后，所有子网格中的内存操作对其父网格来说是可见的。零复制内存具有与全局内存相同的连贯性和一致性保证。

2. 常量内存

常量是不变的，且不得在内核执行时从设备中进行修改。从一个内核线程内获取一个常量内存对象的地址与所有的 CUDA 程序具有一样的语义[⊖]，并完全支持在父网格与子网格之间传递该指针。

3. 共享与本地内存

共享和本地内存对于线程块和线程分别是私有的，并且在父与子网格之间不是可见或一致的。当这些位置的一个对象于它的作用域之外被引用，该行为是未定义的并且有可能会导致错误。

如果 nvcc 检测到一个滥用共享或本地内存指针的操作，它将会发出警告。开发者可以使用内建 `_isGlobal()` 来确定一个给定指针是否被全局内存引用。共享或本地内存的指针对

[⊖] 请注意，在设备代码中，地址必须采用“地址”运算符（一元运算符 `&`），这是因为 `cudaGetSymbolAddress()` 并不支持设备运行时。

于 `cudaMemcpy*Async()` 或 `cudaMemset*Async()` 来说是无效的参数。

4. 本地内存

本地内存是一个处于执行中的线程的私有存储空间，且在该线程之外是不可见的。启动子内核时传递一个指向本地内存的指针作为启动参数是非法的。从一个子内核中解引用这样一个本地内存指针的结果将会是未定义的。要保证这个规则不会在无意中被编译器引用，所有传递给子内核的存储空间应从全局内存堆上进行显示分配。

5. 纹理内存

父和子内核的并发访问有可能会导致数据的不一致，因此，应当避免这种行为。也就是说，父与子内核之间的一致程度由运行时决定。一个子内核可以使用纹理操作来访问其父内核写入的内存，但子内核的内存写操作并不会在父内核的纹理内存访问中反映出来，直到父与子完成同步之后为止。纹理对象在设备运行时上得到很好支持。它们不能够被创建或销毁，但可以进行传递和在层次结构中被任何网格使用（例如父和子网格）。

7.5.3 流与事件

设备运行时所创建的流和事件仅能在创建它们的线程块中使用。NULL 流在设备运行时与在主机运行时具不同语义。在主机上，与 NULL 流的同步会强制让 GPU 上其他所有的数据流操作一同“汇聚”（如第 6.2.3 节中描述的那样）；而在设备上，NULL 流是一条独立的流，并且任何流之间的同步都必须使用事件进行。

在使用设备运行时的时候，流必须以 `cudaStreamNonBlocking` 标志（`cudaStreamCreateWithFlags()` 的一个参数）创建。`cudaStreamSynchronize()` 调用是不支持的，要同步的话必须根据事件和 `cudaStreamWaitEvent()` 进行。

CUDA 事件中，只有流之间的同步功能是支持的。其结果是 `cudaEventSynchronize()`、`cudaEventElapsedTime()` 和 `cudaEventQuery()` 不被支持。此外，由于不支持计时功能，事件必须通过将 `cudaEventDisableTiming` 标志传递给 `cudaEventCreateWithFlags()` 来创建。

7.5.4 错误处理

设备运行时上的任何函数都有可能会返回一个错误（`cudaError_t`）。该错误会被记录在线程私有的某插槽中，并且可以通过调用 `cudaGetLastError()` 来进行查询。与基于主机的运行时一样，CUDA 会区分可以立即返回的错误（例如，如果一个无效参数被传递到一个内存复制函数）和必须以异步方式返回的错误（例如，如果启动执行了一个无效的内存访问）之间的不同。如果一个子网格在运行时导致一个错误，CUDA 会返回一个错误给主机而不是父网格。

7.5.5 编译和链接

与主机运行时不同，在使用设备运行时的时候，开发者必须对设备运行时的静态库进行显式链接。在 Windows 中，设备运行时为 `cudaDevrt.lib`；而在 Linux 和 MacOS 上，为 `cudaDevrt.a`。在用 nvcc 构建时，这可能会通过给命令行追加 `lcudadevrt` 来完成。

7.5.6 资源管理

无论一个内核什么时候启动子网格，该子网格都将会被视为一个新的嵌套层次，而层次的总数视为该程序的嵌套深度。与此相反，子网格启动时程序显式同步的最深层次被称为“同步深度”。通常情况下，同步深度比程序的嵌套深度小 1，但如果程序并不总是需要调用 `cudaDeviceSynchronize()` 的话，那么它可能会远小于嵌套深度。

理论上的最大嵌套深度为 24，但实际上，它受设备限制项 `cudaLimitDevRuntimeSyncDepth` 所控制。深于最大深度的任何内核启动都将失败。默认的最大同步深度是 2。而该限制必须在最顶层的内核从主机上启动之前配置好。



注意 调用一个形如 `cudaMemcpyAsync` 的设备运行时函数可以启动一个内核，同时嵌套深度增加 1。

对于从不调用 `cudaDeviceSynchronize()` 的父内核，系统并不需要为父内核预留空间。在这种情况下，一个程序所需的内存使用量将远远小于预留的最大值。这种程序可以指定一个较小的最大同步深度，以避免过度分配后备存储。

1. 内存使用量

设备运行时的系统软件会预留设备内存以作如下用途。

- 跟踪挂起网格的启动；
- 在同步过程中保存父网格状态；
- 为内核调用的 `malloc()` 和 `cudaMalloc()` 充当可分配的堆。

这一内存不能被应用程序使用，所以有些应用程序可能会降低默认分配，还有某些应用程序可能会为了操作准确而增加默认值。要更改默认值，如表 7-3 中总结的那样，开发者可以调用 `cudaDeviceSetLimit()`。限制项 `cudaLimitDevRuntimeSyncDepth` 是特别重要的，因为每一个嵌套层次将消耗高达 150MB 的设备内存。

2. 挂起的内核启动

在内核启动时，所有相关的配置和参数数据都会被跟踪，直到内核完成为止。这批数据被存在一个系统管理的启动池中。该池的大小是可通过从主机中调用 `cudaDeviceSetLimit()`，

并指定 `cudaLimitDevRuntimePendingLaunchCount` 来配置。

3. 配置选项

设备运行时系统软件的资源分配是通过从主机程序的 `cudaDeviceSetLimit()` API 来控制的。配置必须在任何内核启动之前设置，并且在 GPU 运行中不再改变。

设备运行时分配的内存必须由设备运行时释放。此外，由设备运行时分配的内存来自预分配堆，其大小由设备限制项 `cudaLimitMallocHeapSize` 指定。表 7-3 中列出的限制项可根据需要设置。

表 7-3 `cudaDeviceSetLimit()` 值表

限 制	行 为
<code>cudaLimitDevRuntimeSyncDepth</code>	设定 <code>cudaDeviceSynchronize()</code> 调用的最大深度。启动有可能深于此值，但显式同步深度一旦超过这一限制将会返回 <code>cudaErrorLaunchMax DepthExceeded</code> 的错误。默认的最大同步深度为 2
<code>cudaLimitDevRuntimePendingLaunchCount</code>	为由于无法解析的依赖或执行资源缺乏而还没有开始执行的内核启动控制预留的缓冲内存量。当缓冲区满时，启动将设置线程的最后一个错误为 <code>cudaErrorLaunchPendingCountExceeded</code> 。默认的挂起启动数是 2048 个
<code>cudaLimitMallocHeapSize</code>	设置设备运行时堆的大小，可以通过从内核调用 <code>malloc()</code> 或 <code>cudaMalloc()</code> 来分配

7.5.7 小结

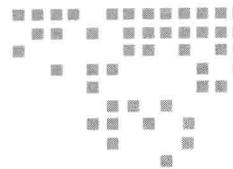
表 7-4 总结了设备运行时和主机运行时之间的主要差异和限制。表 7-5 中列出了有可能被设备运行时调用的功能子集，以及所有相关限制。

表 7-4 设备运行时限制

性 能	限制和差异
事件	仅限于线程块范围 不支持查询 不支持计时，必须由 <code>cudaEventCreateWithFlags</code> (<code>cudaEventDisableTiming</code>) 创建 有限的同步支持，要使用 <code>cudaStreamWaitEvent()</code>
本地内存	只对网格是可用的，不能传递到子网格
空流	不强制参与其他流
共享内存	只对网格是可用的，不能传递到子网格
流	仅限于线程块范围 不支持查询 有限的同步支持，使用 <code>cudaStreamWaitEvent()</code>
纹理	纹理引用（CUDA 5.0 之前为模块范围纹理）只可用于由主机启动的顶层内核启动
纹理与表面对象	只有 SM 3.x 的纹理和表面对象不能被设备运行时创建或销毁，但它们在设备上可以自由使用

表 7-5 CUDA 设备运行时函数

运行时 API 函数	说 明
<code>cudaDeviceSynchronize</code>	仅使线程所在线程块的启动任务同步
<code>cudaDeviceGetCacheConfig</code>	
<code>cudaDeviceGetLimit</code>	
<code>cudaGetLastError</code>	最后一个错误是线程私有的状态，而不是线程块私有的
<code>cudaPeekAtLastError</code>	
<code>cudaGetErrorString</code>	
<code>cudaGetDeviceCount</code>	
<code>cudaGetDeviceProperty</code>	可以返回任何设备的属性
<code>cudaGetDevice</code>	总是返回当前设备 ID 且主机可见
<code>cudaStreamCreateWithFlags</code>	必须传递 <code>cudaStreamNonBlocking</code> 标志
<code>cudaStreamDestroy</code>	
<code>cudaStreamWaitEvent</code>	
<code>cudaEventCreateWithFlags</code>	必须传递 <code>cudaEventDisableTiming</code> 标志
<code>cudaEventRecord</code>	
<code>cudaEventDestroy</code>	
<code>cudaFuncGetAttributes</code>	
<code>cudaMemcpyAsync</code>	
<code>cudaMemcpy2DAsync</code>	
<code>cudaMemcpy3DAsync</code>	
<code>cudaMemsetAsync</code>	
<code>cudaMemset2DAsync</code>	
<code>cudaMemset3DAsync</code>	
<code>cudaRuntimeGetVersion</code>	
<code>cudaMalloc</code>	只能由设备释放
<code>cudaFree</code>	可以释放仅由设备分配的内存



流处理器簇

流处理器簇（SM）是 GPU 中运行 CUDA 内核函数的部分。每一个流处理器簇包含以下部分：

- 上千上万计的可以被划分到执行线程的寄存器；
- 几种类型的缓存：
 - 用以在线程之间快速交换数据的共享内存；
 - 用以快速分发从常量内存中读取数据的常量缓存；
 - 用以从纹理内存中聚合带宽的纹理缓存；
 - 用以减少访问本地或全局内存造成延迟的一级缓存；
- 线程束调度器，它可以很快地在线程之间切换上下文，并将就绪状态的指令发给线程束；
- 整型变量与浮点型变量操作的执行核心，它可以进行：
 - 整形和单精度浮点数操作；
 - 双精度浮点数操作；
 - 特殊函数单元（Special Function Unit, SFU），用来执行单精度浮点数的超越函数（Transcendental Functions）操作。

之所以要有这么多寄存器，并且硬件可以如此高效地在不同的线程之间快速切换上下文，是为了最大化硬件的吞吐量。GPU 拥有足够的状态来应对：在一个读指令执行后，数据从设备读出的数百时钟周期的执行延迟和访存延迟。

SM 是多用途处理器，但是 SM 的设计与 CPU 执行核心有很多不同：SM 的时钟频率要低很多；支持指令并行，但不支持分支预测和预测执行；它们没有或只有很小的缓存。

在适合的工作量下，GPU 的超强计算能力不仅足以完全弥补这些劣势，甚至会表现得更加强大。

从 2006 年的第一款支持 CUDA 的设备面世以来，SM 的设计在不断的演化发展，产生了 3 次主要更迭：分别是代号为特斯拉、费米和开普勒的架构。开发人员可以通过调用 `cudaGetDeviceProperties()` 函数查看 `cudaDeviceProp.major` 和 `cudaDeviceProp.minor`，或者调用驱动程序 API 函数 `cuDeviceComputeCapability()` 来查看设备的计算能力。计算能力 1.x、2.x 和 3.x 分别对应着特斯拉架构、费米架构和开普勒架构硬件。表 8-1 汇总了每一代 SM 硬件中新添加的功能。

表 8-1 SM 功能

计算等级	介 绍
SM 1.1	全局内存原子操作；映射锁页内存；可调试（例如断点指令）
SM 1.2	放松的合并限制；线程束投票（ <code>any()</code> 和 <code>all()</code> 内置指令）；共享内存上的原子操作
SM 1.3	支持双精度
SM 2.0	64 位寻址；一级和二级缓存；内核并发执行；可配置的 16KB 或 48KB 共享内存；位操作指令（ <code>_clz()</code> 、 <code>_popc()</code> 、 <code>_ffs()</code> 、 <code>_brev()</code> 内置指令）；可控制的单精度浮点数取整；融合的乘加；64 位时钟计数器；表面加载 / 存储；64 位全局内存单精度浮点数原子加法；线程束投票（ <code>ballot()</code> 内置指令）；断言和格式化输出（ <code>printf()</code> ）
SM 2.1	内核内函数直接调用和间接调用
SM 3.0	增加最大网格大小；线程束洗牌（ <code>shuffle</code> ）；排列；32KB/32KB 共享内存配置；可配置共享内存（32 位或 64 位模式）无须绑定的纹理（“纹理对象”）；更快的全局原子操作
SM 3.5	64 位原子 <code>min</code> 、 <code>max</code> 、 <code>AND</code> 、 <code>OR</code> 和 <code>XOR</code> ；64 位漏斗移位（ <code>funnel shift</code> ）；通过纹理操作读取全局内存；动态并行

在第 2 章中，从图 2-29 ~ 图 2-32 展示了不同 SM 的框图。CUDA 核心可以执行整型和单精度浮点型指令；如果有一个双精度单元的话，还可以支持双精度运算；特殊函数单元实现求倒数、平方根倒数、正 / 余弦以及对数 / 指数的函数。线程束调度器在这些指令执行需要的资源变得可用时，将指令分发到这些执行单元。

本章主要聚焦于 SM 指令集的能力。有时，这也称“SASS”指令，这是 CUDA 驱动程序或 ptxas 翻译 PTX 中间码得到的本地指令。开发者不能直接修改 SASS 代码，但英伟达公司通过 `cuobjdump` 开发工具使这些指令对开发者可见，这样他们便可以通过检查已编译的微码来直接优化源代码。

8.1 内存

8.1.1 寄存器

每个 SM 包含了成千上万个 32 位寄存器，当内核被启动时，这些寄存器会被分配到指定的线程中。在 SM 中，寄存器是速度最快的，也是数目最多的存储资源。例如，开

普勒架构 (SM 3.0) 的 SMX 包含 65,536 个寄存器，容量共 256KB，而纹理寄存器只有 48KB。

CUDA 寄存器可以装入整型或浮点型数据。如果硬件支持的话 (SM 1.3 及更高) 可以进行双精度算术运算，操作数被放在偶数寄存器对中。在 SM 2.0 及更高版本的硬件中，一对寄存器同样也可以装入 64 位地址。

CUDA 硬件还支持更宽的内存事务：内置的 int2/float2 和 int4/float4 数据类型。它们分别存储在对齐的一对寄存器中或对齐的 4 个寄存器中，可以通过 64 或 128 位宽的单条存取指令读写。只要数据存储在寄存器中，那么数据元素便可以以 .x/.y (int2/float2) 或 .x/.y/.z/.w (int4/float4) 形式引用。

开发人员可以使用命令行选项 --ptxas 和 --verbose 来让 nvcc 报告一个内核使用的寄存器数量。一个内核使用的寄存器数量可以影响每一个 SM 内可装入的线程数。所以要得到优化的性能，必须仔细的设置。每一次编译使用的最大寄存器数，可以通过 --ptxas-options--maxregcount N 来指定。

寄存器别名使用

因为寄存器既可以存储整型数据，也可以存储浮点型数据，有些内置函数只能在强制编译器改变对一个变量的解释方式后工作。`_int_as_float()` 和 `_float_as_int()` 这两个内部函数会使变量在 32 位整型和单精度浮点数之间“改变身份”。

```
float __int_as_float( int i );
int __float_as_int( float f );
```

`_double2loint()`、`_double2hiint()` 和 `_hiloint2double()` 内部函数可以类似地让寄存器数据改变身份 (通常原地转换)。`_double_as_longlong()` 和 `_longlong_as_double()` 强制使寄存器原地配对；`_double2loint()` 和 `_double2hiint()` 分别返回输入的最低和最高的 32 位输入操作数；`_hiloint2double()` 则由高位和低位两部分构造一个双精度浮点数。

```
int double2loint( double d );
int double2hiint( double d );
int hiloint2double( int hi, int lo );
double long_as_double(long long int i );
long long int __double_as_longlong( double d );
```

8.1.2 本地内存

本地内存是用来容纳寄存器溢出的数据，并存储着被索引的局部变量，这些局部变量的索引在编译时是不能计算的。本地内存同全局内存是由设备内存上同一个内存池所支持的，所以在费米及之后的硬件上，本地内存与 L1、L2 缓存层次有着同样的延迟特性和优点。本地内存的寻址方式使内存事务可以自动合并 (coalesced)。硬件包括了用来加载和存储本地内存的特殊指令：对于特斯拉架构的设备，对应的 SASS 指令是 LLD/LST，而对于费米和开普勒架构的设备是 LDL/STL。

8.1.3 全局内存

SM 可以用 GLD/GST 指令（在特斯拉架构设备上）或 LD/ST 指令（在费米和开普勒架构的设备上）读写全局内存。开发者可以用标准 C 的操作符来计算和解引用地址，包括指针算术运算和解引用操作符 *、[] 和 ->。对 64 位或 128 位的内置数据类型（int2/float2/int4/float4）的操作会让编译器自动调用 64 或 128 位的加载和存储指令。通过合并内存事务，可以获得最佳的内存性能，详情参见第 5.2.9 小节。

特斯拉架构的硬件（SM 1.x）使用特殊地址的寄存器来存储指针，之后更新的硬件实现了一个加载和存储架构，对整型与浮点型值的指针使用同样的寄存器文件，对常量内存、共享内存与全局内存使用相同的地址空间^①。

费米架构的硬件包括了许多在更早的设备中不支持的特性

- 通过宽加载存储指令支持 64 位寻址，地址被放在偶数编号的一对寄存器中。在 32 位的主机平台上不支持 64 位寻址。在 64 位主机平台上，64 位寻址被自动打开。因此，由同样的内核生成的分别针对 32 位与 64 位主机平台的代码可能在寄存器数量与性能上有一些差别。
- 一级缓存可以配置成 16KB 或 48KB 大小^②。（开普勒架构另外可以将缓存分为 32KB 一级缓存和 32KB 共享内存。）加载指令可以包含高速缓存提示（来告诉硬件是将数据存入一级缓存，还是越过一级缓存仅将数据保留在二级缓存中）。这些设置可以通过内联的 PTX 或者命令行选项 -X ptxas-dlcm=ca（缓存在一级缓存和二级缓存中，这是默认设置）或者 -X ptxas-dlcm=cg（只缓存在二级缓存中）指定。

原子操作（或者仅称为原子）可以在多个 GPU 线程同时对一个内存位置进行操作时，保证内存更新的正确性。在整个操作期间，硬件会强制在该内存位置上执行互斥访问。因为操作的前后顺序不能保证，所以被支持的操作普遍是符合结合律的^③。

在 SM 1.1 版本中，首次支持了全局内存的原子操作，在更高版本中也同样支持。在 SM 1.2 及之后的版本中，实现了共享内存的原子操作。在开普勒架构的设备之前，全局内存的原子操作实际上慢得没有太多使用价值。

在表 8-2 中总结的全局原子内置函数，当适当的 GPU 架构通过 nvcc 选项 --gpu-architecture 指定后，会自动变得可用。所有的这些函数支持 32 位整型变量。SM 1.2 以后加入了对 64 位 atomicAdd()、atomicExch() 和 atomicCAS() 的支持。SM 2.0 中 atomicAdd() 加入了对 32 位浮点数（float）的支持，在 SM 3.5 中加入了 atomicMin()、atomicMax()、

^① 常量与共享内存都存在于地址窗口中，它们即使在 64 位架构上，也可以被 32 位地址引用。

^② 硬件可以在每次内核启动时对此进行配置，但是改变这个状态的代价很大，并且会破坏并发内核启动中的并发性。

^③ 唯一的例外是单精度浮点加法。话又说回来，面对缺少结合律性质的单精度浮点数操作，一般的浮点代码必须是鲁棒的；移植到不同的硬件，甚至只是使用不同的编译器选项重新编译相同的代码，都可以改变单精度浮点数操作的顺序，从而改变结果。

atomicAnd()、atomicOr() 和 atomicXor() 的 64 位支持。

表 8-2 原子操作

助记符	描述	助记符	描述
atomicAdd	加	atomicDec	自减(减一)
atomicSub	减	atomicCAS	比较并交换
atomicExch	交换	atomicAnd	与
atomicMin	最小值	atomicOr	或
atomicMax	最大值	atomicXor	异或
atomicInc	自增(加一)		

 注意 由于原子操作的实现使用了 GPU 集成内存管理器的硬件，它们无法穿越 PCIe 总线工作，所以当设备指针指向主机内存或点对点内存时不能正常工作。

在硬件层面，原子操作分为 2 种：一种是返回在操作执行前的特定内存位置的值的原子操作；另一种是开发者可以“启动后就不理”的忽略返回值的归约操作。因为在不需要返回值的情况下，硬件可以更有效地执行操作，编译器会检测返回值是否被使用，如果没有使用的话，发射不同的指令。例如在 SM 2.0 中，这种指令分别叫 ATOM 和 RED。

8.1.4 常量内存

常量内存驻留在设备内存上，但是，是被另一种只读的缓存支持的，这种缓存经过了优化，可以将读请求的结果广播到所有引用同一内存位置的线程。每个 SM 包括一个小的、延迟优化的缓存来为那些读请求提供服务。设置内存（和缓存）为只读属性，简化了缓存管理，因为硬件无须实现写回策略来处理被更新的内存。

SM 2.X 和后续硬件包括一个对内存进行特别优化的策略，这种优化针对的是并没有被指定为常量内存却被编译器标示为如下两种属性的内存：1) 只读；2) 地址不依赖线程块或线程的 ID。“统一加载”(load uniform, LDU) 指令使用常量缓存层次读取内存，并将数据广播到线程中去。

8.1.5 共享内存

共享内存是速度非常快的，是 SM 的芯片级内存。线程可以使用它在一个线程块中的线程间进行数据交换。每个 SM 有自己的共享内存，所以共享内存可以影响线程占用率，即一个 SM 上可常驻的线程束数量。SM 使用特殊的指令来存取共享内存：在 SM 1.X 上使用 G2R/R2G，在 SM 2.X 及以后的设备上使用 LDS/STS。

共享内存由交替排列的存储片(bank)构成，并且通常是针对 32 位访问来优化的。如果一个线程束中多于一个线程引用同一存储片，一个存储片冲突(bank conflict)就发生了，所

以硬件必须连续不断的处理内存请求，直到所有请求被服务。典型的，为了避免存储片冲突，应用程序根据线程编号按照交替模式来访问共享内存，就像下面这样：

```
extern __shared__ float shared[];
float data = shared[BaseIndex + threadIdx.x];
```

让线程束中的所有线程从相同的 32 位共享内存位置读取数据也是非常快的。硬件为处理这种情况，使用一种广播机制。向同一存储片写入的指令将由硬件做序列化处理，降低了性能。向同一个地址的写入操作会引起资源竞争问题，应当被避免。

对 2D 访问模式（像图像处理内核中的像素分块），最好在分配共享内存时进行补齐，这样内核就可以引用相邻的行，而不至于引起存储片冲突。SM 2.x 和之后的硬件拥有 32 个内存存储片^②，所以对 2D 分块访问，一个线程束中的线程可以按照每一行来访问，补齐分块大小为若干 33 个 32 位字是一个很好的策略。

在 SM 1.x 硬件上，共享内存大约 16KB 大小^③，在之后的硬件上，共有 64KB 的一级缓存，可以划分出 16KB 或者 48KB 的共享内存，剩余的部分继续用作一级缓存^④。

在最近的几代硬件中，英伟达公司提升了硬件处理非 32 位操作数能力。在 SM 1.x 硬件中，相同存储片的 8 位和 16 位操作数的读取会导致存储片冲突，而 SM 2.x 和之后的硬件可以在相同存储片上广播任意大小的读取。相似的，SM 1.x 的共享内存上的 64 位操作数（像 double）操作要远远慢于 32 位操作数，因此开发者有时不得不把它分割成高 32 位和低 32 位来存储。对主要在共享内存使用 64 位操作数的内核，SM 3.x 添加了一个新的特性：一个提升内存存储片大小到 64 位的模式。

共享内存原子操作

SM 1.2 添加了在共享内存上执行原子操作的功能。不同于全局内存中，使用单指令实现原子操作（根据是否使用返回值分成 GATOM 或 GRED），共享内存原子操作使用显式的加锁/解锁语义实现，编译器生成的代码会在原子操作进行处使每一个线程循环，直到原子操作执行完毕。

代码清单 8-1 给出了程序 atomic32Shared.cu 的源代码，意在突出共享内存原子操作中的编译代码生成。代码清单 8-2 给出了为 SM 2.0 生成的微码。注意指令 LDSLK（加锁加载共享内存）是如何返回一个断定结果以告诉程序锁是否获得，执行更新的代码进行了分支断定，并且代码将循环等待，直至获得锁并完成更新。

锁在每 32 位字上执行，锁的索引由共享内存地址的 2-9 位决定。注意避免资源竞争，否则代码清单 8-2 中的循环会迭代多达 32 次。

^② SM 1.x 硬件有着 16 个内存存储片（一个线程束的前 16 个线程到后 16 个线程的内存传输是分别独立服务的），但是在后续硬件上使用的策略对 SM 1.x 同样有效。

^③ 256 字节的共享内存被保留，用来进行参数传递；在 SM 2.x 及其之后，参数改由常量内存传递。

^④ SM 3.x 硬件添加了均分一级缓存的功能，即 32KB 一级缓存 /32KB 共享内存的划分。

代码清单 8-1 atomic32Shared.cu

```

__global__ void
Return32( int *sum, int *out, const int *pIn )
{
    extern __shared__ int s[];
    s[threadIdx.x] = pIn[threadIdx.x];
    __syncthreads();
    (void) atomicAdd( &s[threadIdx.x], *pIn );
    __syncthreads();
    out[threadIdx.x] = s[threadIdx.x];
}

```

代码清单 8-2 atomic32Shared.cubin (SM 2.0 版本上编译后的微码)

```

code for sm_20
Function : _Z8Return32PiS_PKi
/*0000*/    MOV R1, c [0x1] [0x100];
/*0008*/    S2R R0, SR_Tid_X;
/*0010*/    SHL R3, R0, 0x2;
/*0018*/    MOV R0, c [0x0] [0x28];
/*0020*/    IADD R2, R3, c [0x0] [0x28];
/*0028*/    IMAD.U32.U32 RZ, R0, R1, RZ;
/*0030*/    LD R2, [R2];
/*0038*/    STS [R3], R2;
/*0040*/    SSY 0x80;
/*0048*/    BAR.RED.POPC RZ, RZ;
/*0050*/    LD R0, [R0];
/*0058*/    LDSLK P0, R2, [R3];
/*0060*/    @P0 IADD R2, R2, R0;
/*0068*/    @P0 STSUL [R3], R2;
/*0070*/    @!P0 BRA 0x58;
/*0078*/    NOP.S CC.T;
/*0080*/    BAR.RED.POPC RZ, RZ;
/*0088*/    LDS R0, [R3];
/*0090*/    IADD R2, R3, c [0x0] [0x24];
/*0098*/    ST [R2], R0;
/*00a0*/    EXIT;
.....

```

8.1.6 栅栏和一致性

我们熟悉的 `__syncthreads()` 内置函数会等待线程块中的所有线程到达以后才继续执行。这是维持一个线程块中共享内存一致性所必需的^①。类似的内存栅栏指令可以被用来在更广范围内的内存上按一定次序执行指令，见表 8-3。

表 8-3 内存栅栏函数

内 函 数	描 述
<code>__syncthreads()</code>	等待，直到所有的由线程发起的所有共享内存访问对线程块中所有的线程可见
<code>threadfence_block()</code>	等待，直到所有的由线程发起的所有共享内存和全局内存访问对线程块中所有的线程可见

① 注意：对一个线程束内按“锁步”方式运行的线程，有时允许开发者自己编写“线程束同步”代码，而不用调用 `__syncthreads()`，第 7.3 小节描述了线程和线程束执行的细节，在本节第三部分中包含了几个不同的线程束同步的代码。

(续)

内 函 数	描 述
<code>threadfence()</code>	等待, 直到所有的由线程发起的所有共享内存和全局内存访问对以下可见: <ul style="list-style-type: none"> 线程块中所有访问共享内存的线程 设备中所有访问全局内存的线程
<code>threadfence_system()</code> (仅针对 SM 2.x)	等待, 直到所有的由线程发起的所有共享内存和全局内存访问对以下线程可见: <ul style="list-style-type: none"> 线程块中所有访问共享内存的线程 设备中所有访问全局内存的线程 主机中访问锁页主机内存的线程

8.2 整型支持

SM 支持完整的 32 位整型数运算：

- 加上操作数的负数实现减法运算；
- 乘法与乘加法；
- 整型除法；
- 逻辑运算；
- 条件代码操作；
- 浮点数相互转换；
- 其他操作（例如，针对窄整型的 SIMD 操作、总体计数（population count）和寻找前导零）。

CUDA 中的这些功能大多可以用标准 C 操作符调用。非标准的运算，比如 24 位乘，可以使用内联 PTX 汇编或内置函数访问[⊖]。

8.2.1 乘法

乘法在特斯拉架构和费米架构的硬件上实现的方法不同。特斯拉实现了一个 24 位乘法器，而费米实现了一个 32 位乘法器。因此，完整的 32 位乘法在 SM 1.x 的硬件上需要 4 条指令。对于面向特斯拉架构硬件的性能敏感的代码，使用 24 位乘法内置函数性能会比较高[⊖]。表 8-4 给出了有关乘法的内置函数。

表 8-4 乘法内置函数

内置函数	描 述
<code>_[_u]mul24</code>	返回 32 位乘积中的低 24 位，高 8 位被忽略
<code>_[_u]mulhi</code>	返回乘积中的高 32 位
<code>_[_u]mul64hi</code>	返回 64 位乘积中的高 64 位

[⊖] 在 SM 2.x 和之后的硬件上使用 `_mul24()` 或 `_umul24()` 会有一定程度的性能损失。

8.2.2 其他操作 (位操作)

CUDA 编译器实现了许多“位操作”的内置函数，总结在表 8-5 中。在 SM 2.x 及之后的架构中，这些内置函数对应着单条指令。在费米之前的架构中，它们是可用的，但会被编译成多个指令。当我们对此不确定，最好的办法是反汇编并查看微码！64 位变量有“ll”（代表“long long”）追加在内置函数名之后，如 `_clzll()`、`ffsll()`、`popcll()` 和 `brevll()`。

表 8-5 位操作内置函数

内 函 数	总 结	描 述
<code>_brev(x)</code>	位取反	一个字中的位的顺序取反
<code>_byte_perm(x, y, s)</code>	排列字节	根据选择参数 s 从两个输入中选择字节，构成一个 32 位的字并返回
<code>_clz(x)</code>	计算前导零的数目	返回第一个有效数字前有多少零位 (0 ~ 32)
<code>_ffs(x)</code>	查找第一个符号位	返回最低的有效位，最低的有效位为位置 1。若输入 0，则返回 0
<code>_popc(x)</code>	总体计数	返回“1”的个数
<code>_*[u]sad(x, y, z)</code>	差的绝对和	将 x-y 加到 z 并返回结果

8.2.3 漏斗移位 (SM 3.5)

GK110 增加了一个 64 位的“漏斗移位”指令来连接 2 个 32 位值（高 32 位和低 32 位被分成独立的 2 个输入，但是硬件在对齐的一对寄存器上进行操作），将 64 位值左移或右移，然后返回最高的（对左移来说）或最低的（对右移来说）32 位。

漏斗移位可以通过表 8-6 中的内置函数访问。这些内置函数被实现成设备的内联函数（使用内联 PTX 汇编器），定义在 `sm_35_intrinsics.h` 中。默认地，最低 5 位的移位计数被屏蔽，`_lc` 和 `_rc` 函数将移位值控制在 0 ~ 32 之间。

漏斗移位的应用包括：

- 多字的移位操作。
- 在非对齐的缓冲区之间使用对齐的加载和存储指令操作内存。
- 旋转。

表 8-6 漏斗移位内置函数

内置函数	描 述
<code>_funnelshift_l(hi, lo, sh)</code>	将 [hi:lo] 连接成一个 64 位量，左移 (sh&31) 位，并返回高 32 位
<code>_funnelshift_lc(hi, lo, sh)</code>	将 [hi:lo] 连接成一个 64 位量，左移 min (sh,31) 位，并返回高 32 位
<code>_funnelshift_r(hi, lo, sh)</code>	将 [hi:lo] 连接成一个 64 位量，右移 (sh&31) 位，并返回低 32 位
<code>_funnelshift_rc(hi, lo, sh)</code>	将 [hi:lo] 连接成一个 64 位量，右移 min (sh,31) 位，并返回低 32 位

为了实现数据的大于 64 位的右移，重复调用函数 `_funnelshift_r()`，从低位向高位递进

操作。最高的字使用操作符 `>>` 计算，根据整数的类别，决定移出的位是补 0 还是补符号位。为了对数据进行大于 64 位的左移，重复调用 `_funnelshift_l()` 函数，从高位到低位渐次操作。最低的字使用操作符 `<<` 计算。如果 `hi` 和 `lo` 参数相同，漏斗移位构成旋转操作。

8.3 浮点支持

快速的本地浮点硬件是 GPU 一大优势，而且在许多方面，它们都等同于或优于 CPU 的浮点实现。可以全速支持非归一化的浮点数，^②可在每一条指令级别执行定向舍入操作，并提供特殊函数单元，支持高性能的近似 6 种流行的单精度超越函数。相比之下，对非归一化的浮点数，x86 CPU 执行它们的微码可能会比归一化浮点指令慢 100 倍。取整方向由一个控制字指定，改变取整方向可能需要几十个时钟周期，SSE 指令集中唯一的超越近似函数是取倒数和平方根倒数，且仅提供 12 位的近似结果，必须使用牛顿 - 拉夫逊迭代完善后方可使用。

由于受其较低的时钟频率影响，GPU 在核心数量上的优势有所抵消，在相同的条件下开发者可以预期 10 倍（左右）加速。如果有论文指出从优化的 CPU 实现移植到 CUDA 会产生 100 倍或更大的加速，很可能是以上描述的“指令集不匹配”的原因。

8.3.1 格式

CUDA 支持图 8-1 所示的 3 种 IEEE 标准浮点格式：双精度（64 位）、单精度（32 位）和半精度（16 位）。每一个数被分成 3 个字段：符号、指数和尾数。

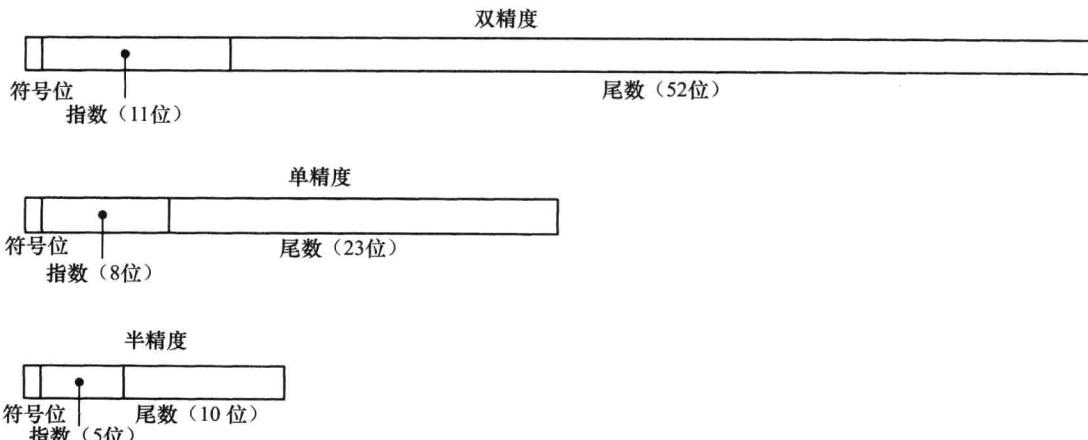


图 8-1 浮点数格式

^② 有一例外，单精度非归一化浮点数在 SM 1.x 上根本不被支持。

对于双精度、单精度和半精度来说，指数位数分别为 11、8 和 5 位，相应的尾数位数分别是 52、23 和 10 位。

指数组段更改了浮点数的解释方式。最常见的（“归一化”）编码表示是，对尾数加入一个隐含“1”位并用尾数乘以 $2^{e-\text{bias}}$ ，其中 bias 是在编码为浮点数表示之前加入到实际指数值的偏移。例如，bias 对于单精度来说是 127。

表 8-7 总结了浮点值如何进行编码。大多数指数值（即“归一化”的浮点值），假定尾数有一个隐含的“1”，并将它乘以偏移后的指数值。最大指数值被保留为无穷大和 NaN (Not A Number) 值。被 0 除（或除数溢出）结果为无限大；执行无效操作（如使用负数的平方根或对数）会产生一个 NaN。最小的指数值为那些非常小的数保留，这些数无法用隐含的首位“1”格式表示。随着非归一化浮点数[⊖]趋向零，它们失去了有效位的精度，这种现象被称为渐近下溢。表 8-8 给出了 3 种格式的某些临界值的编码和具体值。

表 8-7 浮点数的表示

双 精 度			
指 数	尾 数	值	名 称
0	0	± 0	零
0	非 0	$\pm 2^{-1022}$ (0. 尾数)	非归一化数
1-2046	任意	$\pm 2^{e-1023}$ (1. 尾数)	归一化数
2047	0	$\pm \infty$	无穷大
2047	非 0		Not-A-Number (非数)
单 精 度			
指 数	尾 数	值	名 称
0	0	± 0	零
0	非 0	$\pm 2^{-126}$ (0. 尾数)	非归一化数
1-254	任意	$\pm 2^{e-127}$ (1. 尾数)	归一化数
255	0	$\pm \infty$	无穷大
255	非 0		Not-A-Number (非数)
半 精 度			
指 数	尾 数	值	名 称
0	0	± 0	零
0	非 0	$\pm 2^{-14}$ (0. 尾数)	非归一化数
1-30	任意	$\pm 2^{e-15}$ (1. 尾数)	归一化数
31	0	$\pm \infty$	无穷大
31	非 0		Not-A-Number (非数)

⊖ 有时也被称为次归一化。

表 8-8 浮点数的临界值

双 精 度		
	十六进制值	准 确 值
最小非归一化数	0...0001	2^{-1074}
最大非归一化数	000F...F	$2^{-1022}(1-2^{-52})$
最小归一化数	0010...0	2^{-1022}
1.0	3FF0...0	1
最大整数	4340...0	2^{53}
最大整数	7F7FFFFF	$2^{1024}(1-2^{-53})$
无穷大	7FF00000	无穷大
单 精 度		
	十六进制值	准 确 值
最小非归一化数	0...0001	2^{-149}
最大非归一化数	007F...F	$2^{-126}(1-2^{-23})$
最小归一化数	0080...0	2^{-126}
1.0	3F80...0	1
最大整数	4B80...0	2^{24}
最大整数	7F7FFFFF	$2^{128}(1-2^{-24})$
无穷大	7F800000	无穷大
半 精 度		
	十六进制值	准 确 值
最小非归一化数	0001	2^{-24}
最大非归一化数	07FF	$2^{-14}(1-2^{-10})$
最小归一化数	0800	2^{-14}
1.0	3c00	1
最大整数	6800	2^{11}
最大整数	7BFF	$2^{16}(1-2^{-11})$
无穷大	7C00	无穷大

1. 舍入

IEEE 标准提供四种舍入模式。

- 舍入到最近偶数（也称为“最近舍入”）
- 向 0 舍入（也被称为“截断”或“斩断”）
- 向下舍入（或“向负无穷大舍入”）
- 向上舍入（或“向正无穷大舍入”）

最近舍入是中间值四舍五入到每次操作后最接近的可表示的浮点数值，是迄今为止最常用的舍入模式。向上舍入和向下舍入（“定向舍入模式”），用于区间算术，其中一对浮点值是用来界定一个计算的中间结果。要正确界定结果，区间的下限和上限的值必须分别向负无穷大（“向下”）和向正无穷大（“向上”）舍入。

C 语言没有为每条指令提供任何方式来指定舍入模式，而且 CUDA 硬件也没有提供一个可以隐式指定舍入模式的控制字。因此，CUDA 提供了一套内置函数来指定一个操作的舍入模式，见表 8-9。

表 8-9 舍入操作的内置函数

内置函数	操作	内置函数	操作
<code>_fadd_[rn rz ru rd]</code>	加	<code>_dadd_[rn rz ru rd]</code>	加
<code>_fmul_[rn rz ru rd]</code>	乘	<code>_dmul_[rn rz ru rd]</code>	乘
<code>_fmaf_[rn rz ru rd]</code>	乘加	<code>_fma_[rn rz ru rd]</code>	乘加
<code>_frcp_[rn rz ru rd]</code>	倒数	<code>_drcp_[rn rz ru rd]</code>	倒数
<code>_fdiv_[rn rz ru rd]</code>	除	<code>_ddiv_[rn rz ru rd]</code>	除
<code>_fsqrt_[rn rz ru rd]</code>	平方根	<code>_dsqrt_[rn rz ru rd]</code>	平方根

2. 转换

一般而言，开发者可以在不同的浮点表示和整型之间使用标准 C 语言的转换：隐式的或显式的类型转换。然而，如果需要的话，开发者可以使用表 8-10 中列出的函数来执行转换，这些转换不属于 C 语言标准，例如，它们涉及定向取整的函数。

因为 half 不是标准 C 语言的类型，CUDA 使用 unsigned short 作为函数 `_half2float()` 和 `_float2half()` 的接口。其中 `_float2half()` 只支持最近舍入模式。

```
float _half2float( unsigned short );
unsigned short _float2half( float );
```

表 8-10 转换函数

指 令	操 作
<code>_float2int_[rn rz ru rd]</code>	float 到 int
<code>_float2uint_[rn rz ru rd]</code>	float 到 unsigned int
<code>_int2float_[rn rz ru rd]</code>	int 到 float
<code>_uint2float_[rn rz ru rd]</code>	unsigned int 到 float
<code>_float2ll_[rn rz ru rd]</code>	float 到 64 位 int
<code>_ll2float_[rn rz ru rd]</code>	64 位 int 到 float
<code>_ull2float_[rn rz ru rd]</code>	64 位 unsigned int 到 float
<code>_double2float_[rn rz ru rd]</code>	double 到 float
<code>_double2int_[rn rz ru rd]</code>	double 到 unsigned int
<code>_double2ll_[rn rz ru rd]</code>	double 到 64 位 int
<code>_double2ull_[rn rz ru rd]</code>	double 到 64 位 unsigned int
<code>_int2double_rn</code>	int 到 double
<code>_uint2double_rn</code>	unsigned int 到 double
<code>_ll2double_[rn rz ru rd]</code>	64 位 int 到 double
<code>_ull2double_[rn rz ru rd]</code>	64 位 unsigned int 到 double

8.3.2 单精度 (32 位)

单精度浮点支持是 GPU 计算的主力所在。GPU 做了许多优化，以在传输这种数据类型上拥有很高的性能表现[⊖]，不仅仅表现在进行核心的 IEEE 标准操作，例如加和乘，还表现在不标准的操作，像 sin() 和 log() 这类超越函数的近似。32 位浮点数存储在与整型值相同的寄存器中，所以在单精度浮点数与整型之间的转换（使用 __float_as_int() 和 __int_as_float()）是非常轻松的。

1. 加、乘和乘加

编译器会自动的将应用在浮点数上的操作符 +、- 和 * 翻译为加、乘和乘加指令。函数 __fadd_rn() 和 __fmul_rn() 可以被用来支持将加法和乘法操作转换为乘加指令。

2. 倒数与除

对于计算能力 2.x 或更高的机器，使用选项 --prec-div=true 编译会使除法运算符遵从 IEEE 标准。对于计算能力 1.x 或者使用编译选项 --prec-div=false 的计算能力为 2.x 的机器，除法操作符和 __fdividef(x,y) 有着相同精度，但是对于 $2^{126} < y < 2^{128}$ ，函数 __fdividef(x,y) 会得到 0，而除操作符会得到正确结果。同样，对于 $2^{126} < y < 2^{128}$ ，如果 x 为无穷大，__fdividef(x,y) 返回一个 NaN，而除法操作符返回无穷大。

3. 超越函数 (SFU)

SM 中的特殊函数单元 (Special Function Unit, SFU) 实现了 6 种常用的超越函数，并且函数的运算速度非常快。

- 正弦与余弦
- 对数与指数
- 倒数与平方根倒数

表 8-11 节选自关于特斯拉架构的一篇论文[⊖]，这篇论文总结了特斯拉架构支持的操作和兼容的精度。SFU 并没有实现全精度，但是合理的近似了这些函数的结果，并取得了很好的运行速度。对于大多数依赖 SFU 运算的代码，CUDA 可以远远的领先于优化的 CPU 版本的性能表现（25 倍或更多）。

表 8-11 SFU 精度

函 数	精 度 (精确位)	ULP 误差
1/x	24.02	0.98
1/sqrt (x)	23.40	1.52

⊖ 事实上，GPU 在具备完整的 32 位整型数支持之前就有完整的 32 位浮点支持。因此，一些早期的著作解释怎样用浮点硬件实现整型操作。

⊖ Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, March-April 2008, p. 47.

(续)

函 数	精度 (精确位)	ULP 误差
2^x	22.51	1.41
$\log_2 x$	22.57	n/a
sin/cos	22.47	n/a

SFU 通过表 8-12 中给出的内置函数调用。指定编译器选项 `--fast-math` 会让编译器使用兼容的 SFU 函数替代传统的 C 函数。

表 8-12 SFU 内置函数

内置函数	操作	内置函数	操作
<code>_cosf(x)</code>	<code>cosx</code>	<code>_log10f(x)</code>	$\log_{10} x$
<code>_exp10f(x)</code>	10^x	<code>_powf(x, y)</code>	x^y
<code>_expf(x)</code>	e^x	<code>_sinf(x)</code>	$\sin x$
<code>_fdividef(x, y)</code>	x/y	<code>_sincosf(x, sptr, cptr)</code>	$*s = \sin(x)$ $*c = \cos(x)$
<code>_logf(x)</code>	$\ln x$	<code>_tanf(x)</code>	$\tan x$
<code>_log2f(x)</code>	$\log_2 x$		

4. 其他函数

`_saturate(x)` 在 $x < 0$ 时返回 0，在 $x > 1$ 时返回 1，而 x 为其他值时返回其本身。

8.3.3 双精度 (64 位)

在 SM1.3(首次在 GeForce GTX 280 中实现)中添加了对 CUDA 双精度浮点计算的支持，并在 SM 2.0 中有了大幅度的提升(功能和性能上)。CUDA 硬件支持非归一化双精度浮点数的全速计算，并且，从 SM 2.x 开始，一个原生的乘加指令(FMAD)，兼容 IEEE 754c.2008 标准，只需要一个指令就可实现取整。除了作为一个有用操作，FMAD 使那些可以在牛顿 - 拉尔逊迭代中聚合的特定函数实现全精度运算。

在单精度操作中，编译器自动地将标准 C 操作符翻译为乘法、加法和乘加指令。内置函数 `_dadd_rn()` 和 `_dmul_rn()` 可以用来防止乘和加运算合并为乘加指令。

8.3.4 半精度 (16 位)

半精度拥有 5 位指数和 10 位尾数，所以 half 值拥有足够的高动态域 (high dynamic range, HDR) 图像精度，并且可以用来存储其他类型中不需要达到 float 精度的值，例如角度。半精度值的出发点在于节省存储，而不是快速计算，所以硬件仅仅提供与 32 位浮点数之间的转换指令[⊖]。这些指令被包装为函数 `_halftofloat()` 和 `_floattohalf()`。

⊖ half 浮点值为纹理结构所支持，TEX 指令会返回 float 型值，并且半浮点到浮点的转换是由纹理硬件自动执行的。

```
float __halftofloat( unsigned short );
unsigned short __floattohalf( float );
```

这些内置函数使用 `unsigned short` 数据类型，因为 C 语言中没有标准的 `half` 浮点类型。

8.3.5 案例分析：float 到 half 的转换

分析 `float` 到 `half` 之间的转换操作对于理解浮点编码和取整的细节非常有用。因为它是简单的一元操作，我们可以把主要精力集中于编码和取整，而不去被浮点运算的细节和中间表示的精度所分心。

当从 `float` 转换为 `half` 时，对任意的 `float`，如果太大，以至于 `half` 形式表现不了时，正确的输出应为无穷大的 `half`。任意的 `float` 值，如果太小，以至于 `half` 形式表现不了的（即使是一个非正规 `half`），也应该被取整为 0.0。取整为 `half` 的 0.0 值的 `float` 为 0x32FFFFFF 或 2.98^{-8} ，最小的取为 `half` 的无穷大的 `float` 为 65520.0。在此区间的 `float` 值可以被转化为 `half`：通过传递符号位，重新对指数进行偏移（由于 `float` 有 8 位指数，偏移为 127，而 `half` 有 5 位指数，偏移为 15），最后取整 `float` 尾数值到最近的 `half` 尾数值。除非待转换的浮点值恰好落于两个可能输出值的正中间，取整是非常容易理解的。对上述这种势均力敌的情形，IEEE 标准制定取整到最近的偶数。在十进制算术运算中，这意味着 1.5 取整为 2.0，而且 2.5 同样取整为 2.0，但 0.5 取整到 0.0。

代码清单 8-3 展示了 `float` 到 `half` 转换操作的完整的 CUDA 硬件实现。变量 `exp` 和 `mag` 分别存储了输入的指数和“幅度”（magnitude）尾数与指数的符号位都屏蔽掉了。许多操作，像比较与取整操作，可以直接通过参数 `mag` 执行，而不必分离指数与尾数。

代码清单 8-3 中的宏 `LG_MAKE_MASK`，根据给定的位计数创建一个掩码：`#define LG_MAKE_MASK(bits)((1<<bits)-1)`。`volatile union` 使得同一个 32 位的值既可以当作 `float` 又可以当做 `unsigned int` 类型来处理；像 `*((float*)(&u))` 这样的使用方式是不可移植的。转换会首先传递符号位，并用掩码将其过滤掉。

在提取幅度和指数后，函数会处理输入 `float` 为 INF 和 NaN 的特殊情况，并作提前退出处理。注意 INF 是有符号的，但是 NaN 是一个典型的无符号值。代码的 50 ~ 80 行夹取输入的 `float` 值为 `half` 可表现的最小或最大值，并重计算幅度的夹取值。不要被构造 `f32MinRInfin` 和 `f32MaxRf16_zero` 的代码迷惑住，它们分别是常数 `0x477ff000` 和 `0x32fffff`。

程序的其余部分处理了输出的归一化和非归一化情况（输入非归一化的情况在之前的代码做了夹取处理，所以 `mag` 对应一个归一化的 `float`）。在这段夹取代码中，`f32Minf16Normal` 是一个常量，值为 `0x38fffff`。

为了构造一个归一化数，必须计算一个新的指数（92 ~ 93 行），并且正确移出已经取整的 10 位尾数到输出。为了构造一个非归一化数，输出的尾数要与隐含的 1 做 OR 操作，得到的尾数要根据输入指数的数值进行移位。无论是归一化还是非归一化数，输出尾数的取整由 2 个步骤完成。取整由恰好比输出的有效低位（LSB）短的一个由 1 组成的掩码序列实现，如图 8-2 所示。

这一操作在位 12 被置为 1 时将输出的尾数增 1。如果所有的输入尾数都为 1，溢出会使输出指数正确增加。如果我们添加另外一个 1 到有效高位（MSB），会用传统的取整模式，即

在势均力敌时，取整到最接近的大于真实数的整数。相反，为了实现最近偶数取整，我们在10位输出的有效低位（LSB）被置1的情况下，增加输出的尾数值（见图8-3）。注意这些步骤可以顺序实现，也可以通过许多不同的方式实现。

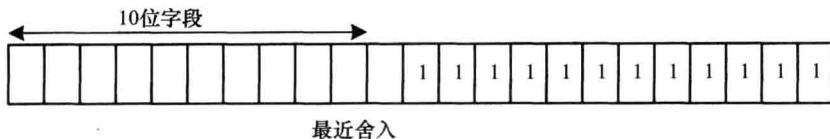


图 8-2 舍入操作的掩码

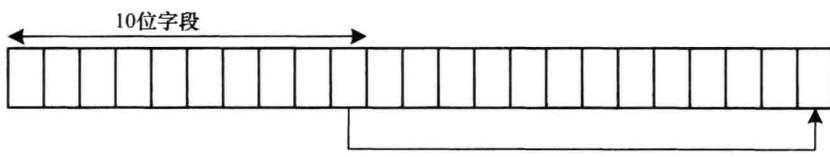


图 8-3 舍入到最近偶数（半精度）

代码清单 8-3 ConvertToHalf()

```
/*
 * exponent shift and mantissa bit count are the same.
 * When we are shifting, we use [f16|f32]ExpShift
 * When referencing the number of bits in the mantissa,
 * we use [f16|f32]MantissaBits
 */
const int f16ExpShift = 10;
const int f16MantissaBits = 10;

const int f16ExpBias = 15;
const int f16MinExp = -14;
const int f16MaxExp = 15;
const int f16SignMask = 0x8000;

const int f32ExpShift = 23;
const int f32MantissaBits = 23;
const int f32ExpBias = 127;
const int f32SignMask = 0x80000000;

unsigned short
ConvertFloatToHalf( float f )
{
    /*
     * Use a volatile union to portably coerce
     * 32-bit float into 32-bit integer
     */
    volatile union {
        float f;
        unsigned int u;
    } uf;
    uf.f = f;

    // return value: start by propagating the sign bit.
    unsigned short w = (uf.u >> 16) & f16SignMask;

    // Extract input magnitude and exponent
}
```

```

unsigned int mag = uf.u & ~f32SignMask;
int exp = (int) (mag >> f32ExpShift) - f32ExpBias;

// Handle float32 Inf or NaN
if ( exp == f32ExpBias+1 ) {      // INF or NaN

    if ( mag & LG_MAKE_MASK(f32MantissaBits) )
        return 0x7fff; // NaN

    // INF - propagate sign
    return w|0x7c00;
}

/*
 * clamp float32 values that are not representable by float16
 */
{
    // min float32 magnitude that rounds to float16 infinity

    unsigned int f32MinRInfin = (f16MaxExp+f32ExpBias) <<
        f32ExpShift;
    f32MinRInfin |= LG_MAKE_MASK( f16MantissaBits+1 ) <<
        (f32MantissaBits-f16MantissaBits-1);

    if (mag > f32MinRInfin)
        mag = f32MinRInfin;
}

{
    // max float32 magnitude that rounds to float16 0.0

    unsigned int f32MaxRf16_zero = f16MinExp+f32ExpBias-
        (f32MantissaBits-f16MantissaBits-1);
    f32MaxRf16_zero <= f32ExpShift;
    f32MaxRf16_zero |= LG_MAKE_MASK( f32MantissaBits );

    if (mag < f32MaxRf16_zero)
        mag = f32MaxRf16_zero;
}

/*
 * compute exp again, in case mag was clamped above
 */
exp = (mag >> f32ExpShift) - f32ExpBias;

// min float32 magnitude that converts to float16 normal
unsigned int f32Minf16Normal = ((f16MinExp+f32ExpBias)<<
    f32ExpShift);
f32Minf16Normal |= LG_MAKE_MASK( f32MantissaBits );
if ( mag >= f32Minf16Normal ) {
    //
    // Case 1: float16 normal
    //

    // Modify exponent to be biased for float16, not float32
    mag += (unsigned int) ((f16ExpBias-f32ExpBias)<<
        f32ExpShift);

    int RelativeShift = f32ExpShift-f16ExpShift;

    // add rounding bias
    mag += LG_MAKE_MASK(RelativeShift-1);

    // round-to-nearest even
    mag += (mag >> RelativeShift) & 1;
}

```

```

        w |= mag >> RelativeShift;
    }
else {
/*
 * Case 2: float16 denormal
 */
    // mask off exponent bits - now fraction only
    mag &= LG_MAKE_MASK(f32MantissaBits);

    // make implicit 1 explicit
    mag |= (1<<f32ExpShift);

    int RelativeShift = f32ExpShift-f16ExpShift+f16MinExp-exp;

    // add rounding bias
    mag += LG_MAKE_MASK(RelativeShift-1);

    // round-to-nearest even
    mag += (mag >> RelativeShift) & 1;

    w |= mag >> RelativeShift;
}
return w;
}

```

实际上，开发者应该使用内置函数 `_floattohalf()` 将 `float` 转换为 `half`，编译器会将其翻译为单条 F2F 机器指令。这一样例程序只是单纯的用来解释浮点结构和取整；同样，检查所有的 INF/NAN 特殊情况和非归一化值的代码，帮助我们展示了“为什么 IEEE 标准的这些特性从它诞生以来就争议不断”：由于它不仅仅增加了硅的面积，还增加了工程师验证正确性的付出，它使硬件变得更慢、代价更大或者两者兼而有之。

在随同本书一同发布的代码中，代码清单 8-3 中的程序 `ConvertFloatToHalf()` 被整合在程序 `float_to_float16.cu` 文件中，并对每一个 32 位浮点值测试了它的输出。

8.3.6 数学函数库

CUDA 包含了一个类似于 C 语言运行时库的内置数学函数库，但有一些差别：CUDA 硬件不包括取整模式寄存器（相反，取整模式由每条指令来编码）[⊖]。所以，像 `rint()` 函数会默认使用最近取整模式。除此之外，硬件不会抛出浮点异常；异常操作的结果，例如对负数取平方根，会被编码为 NaN。

表 8-13 列出了所有的数学函数库函数和每个函数的最大 ulp 误差。大多数使用 `float` 进行运算的函数，会在函数名之后追加一个 `f`，例如，计算正弦函数的函数有：

```
double sin( double angle );
float sinf( float angle );
```

在表 8-13 中，它们被表示为 `sin[f]`。

[⊖] 让每条指令编码取整模式和将其保存在控制寄存器中并不矛盾，Alpha 处理器拥有一个 2 位编码来指定每一个指令的取整模式，其中一个设置是使用控制寄存器中指定的取整模式！CUDA 硬件只使用了两个位编码来实现 IEEE 标准中的 4 种取整模式。

表 8-13 数学函数

函 数	操 作	表 达 式	ULP 误 差	
			32	64
$x+y$	加	$x+y$	0 ^①	0
$x*y$	乘	$x*y$	0 ^①	0
x/y	除	x/y	2 ^②	0
$1/x$	倒数	$1/x$	1 ^②	0
$\text{acos}[f](x)$	反余弦	$\cos^{-1}x$	3	2
$\text{acosh}[f](x)$	反双曲余弦	$\ln(x + \sqrt{x^2 + 1})$	4	2
$\text{asin}[f](x)$	反正弦	$\sin^{-1}x$	4	2
$\text{asinh}[f](x)$	反双曲正弦	$\text{sign}(x) \ln(x + \sqrt{x^2 + 1})$	3	2
$\text{atan}[f](x)$	反正切	$\tan^{-1}x$	2	2
$\text{atan2}[f](y, x)$	y/x 的反正切	$\tan^{-1}\left(\frac{y}{x}\right)$ ^③	3	2
$\text{atanh}[f](x)$	反双曲正切	\tanh^{-1}	3	2
$\text{cbrt}[f](x)$	立方根	$\sqrt[3]{x}$	1	1
$\text{ceil}[f](x)$	向上取整	$[x]$	0	
$\text{copysign}[f](x, y)$	y 的符号, x 的幅度		n/a	
$\text{cos}[f](x)$	余弦	$\cos x$	2	1
$\text{cosh}[f](x)$	双曲余弦	$\frac{e^x + e^{-x}}{2}$	2	
$\text{cospi}[f](x)$	使用 π 进行缩放的余弦	$\cos \pi x$	2	
$\text{erf}[f](x)$	误差函数	$\frac{2}{\pi} \int_0^x e^{-t^2}$	3	2
$\text{erfc}[f](x)$	误差函数的补	$1 - \frac{2}{\pi} \int_0^x e^{-t^2}$	6	4
$\text{erfcinv}[f](y)$	反误差函数的补	返回满足 $y = 1 - \text{erff}(x)$ 的 x	7	8
$\text{erfcx}[f](x)$	缩放的误差函数	$e^{x^2}(\text{erff}(x))$	6	3
$\text{erfinv}[f](y)$	反误差函数	当 $y = \text{erff}(x)$ 时返回 x	3	5
$\text{exp}[f](x)$	自然指数	e^x	2	1
$\text{exp10}[f](x)$	指数 (底为 10)	10^x	2	1
$\text{exp2}[f](x)$	指数 (底为 2)	2^x	2	1
$\text{expm1}[f](x)$	自然指数, 减去 1	$e^x - 1$	2	1
$\text{fabs}[f](x)$	绝对值	$ x $	0	0
$\text{fdim}[f](x, y)$	正差值	$\begin{cases} x-y, & x>y \\ +0, & x \leqslant y \\ \text{NAN}, & x \text{或者} y \text{为NaN} \end{cases}$	0	0
$\text{floor}[f](x)$	向下取整	$[x]$	0	0
$\text{fma}[f](x, y, z)$	乘加	$xy + z$	0	0
$\text{fmax}[f](x, y)$	最大值	$\begin{cases} x, & x>y \text{或者} \text{isNaN}(y) \\ y, & \text{其他} \end{cases}$	0	0

^① 原公式有误, 已修正。——译者注

(续)

函 数	操 作	表 达 式	ULP 误差	
			32	64
fmin[f](x, y)	最 小 值	$\begin{cases} x, & x > y \text{ 或者 } \text{isNaN}(y) \\ y, & \text{其他} \end{cases}$	0	0
fmod[f](x, y)	浮点数取余		0	0
frexp[f](x, exp)	小数部分		0	0
hypot[f](x, y)	斜边长	$\sqrt{x^2 + y^2}$	3	2
ilogb[f](x)	取指数		0	0
isfinite(x)	当 x 不是 $\pm \text{INF}$ 返回非零值		n/a	
isinf(x)	当 x 是 $\pm \text{INF}$ 返回非零值		n/a	
isnan(x)	当 x 是 NaN 返回非零值		n/a	
j0[f](x)	第一类贝塞尔函数 ($n = 0$)	$J_0(x)$	9 ⁽³⁾	7 ⁽³⁾
j1[f](x)	第一类贝塞尔函数 ($n = 1$)	$J_1(x)$	9 ⁽³⁾	7 ⁽³⁾
jn[f](n, x)	第一类贝塞尔函数	$J_n(x)$	*	
ldexp[f](x, exp)	缩放为 2 的幂次	$x2^{\text{exp}}$	0	0
lgamma[f](x)	伽马函数的对数	$\ln(\Gamma(x))$	6 ⁽⁴⁾	6 ⁽⁴⁾
llrint[f](x)	向 long long 转换		0	0
llround[f](x)	向 long long 转换		0	0
lrint[f](x)	向 long 转换		0	0
lround[f](x)	向 long 转换		0	0
log[f](x)	自然对数	$\ln(x)$	1	1
log10[f](x)	对数 (10 为底)	$\log_{10}x$	3	1
log1p[f](x)	$x + 1$ 的自然对数	$\ln(x+1)$	2	1
log2[f](x)	对数 (底为 2)	\log_2x	3	1
logb[f](x)	得到指数		0	0
modff(x, iptr)	分割整数与小数部分		0	0
nan[f](cptr)	返回 NaN	NaN	n/a	
nearbyint[f](x)	取整		0	0
nextafter[f](x, y)	返回沿 y 方向最接近的浮点值		n/a	
normcdf[f](x)	标准累计分布		6	5
normcdinv[f](x)	反标准累计分布		5	8
pow[f](x, y)	乘方	x^y	8	2
rcbrt[f](x)	立方根倒数	$\frac{1}{\sqrt[3]{x}}$	2	1
remainder[f](x, y)	余数		0	0
remquo[f](x, y, iptr)	余数 (同时返回商)		0	0
rsqrt[f](x)	平方根倒数 ⁽²⁾	$\frac{1}{\sqrt{x}}$	2	1

⁽²⁾ 原文把平方根倒数误为倒数, 已修改。——译者注

(续)

函 数	操 作	表 达 式	ULP 误 差	
			32	64
<code>rint[f](x)</code>	最近整数取整		0	0
<code>round[f](x)</code>	最近整数取整		0	0
<code>scalbln[f](x,n)</code>	对 x 缩放为 2^n 倍 (n 为 long int)	$x2^n$	0	0
<code>scalbn[f](x,n)</code>	对 x 缩放为 2^n 倍 (n 为 int)	$x2^n$	0	0
<code>signbit(x)</code>	x 为负返回一个非零值		n/a	0
<code>sin[f](x)</code>	正弦	$\sin x$	2	1
<code>sincos[f](x,s,c)</code>	正弦与余弦	* $s = \sin(x)$ * $c = \cos(x)$	2	1
<code>sincospi[f](x,s,c)</code>	正弦与余弦	* $s = \sin(\pi x)$ * $c = \cos(\pi x)$	2	1
<code>sinh[f](x)</code>	双曲正弦	$\frac{e^x - e^{-x}}{2}$	3	1
<code>sinpi[f](x)</code>	正弦, 参数乘 π	$\sin \pi x$	2	1
<code>sqrt[f](x)</code>	平方根	\sqrt{x}	3 ⁽⁵⁾	0
<code>tan[f](x)</code>	正切	$\tan x$	4	2
<code>tanh[f](x)</code>	双曲正切	$\frac{\sinh x}{\cosh x}$	2	1
<code>tgamma[f](x)</code>	真伽马函数	$\Gamma(x)$	11	8
<code>trunc[f](x)</code>	截断(向原点取整)		0	0
<code>y0[f](x)</code>	第二类贝塞尔函数 ($n=0$)	$Y_0(x)$	9 ⁽³⁾	7 ⁽³⁾
<code>y1[f](x)</code>	第二类贝塞尔函数 ($n=1$)	$Y_1(x)$	9 ⁽³⁾	7 ⁽³⁾
<code>yn[f](n,x)</code>	第二类贝塞尔函数	$Y_n(x)$	**	

注: * 对于贝塞尔函数 $jnf(n,x)$ 和 $jn(n,x)$, 当 $n=128$ 时, 最大绝对误差分别为 2.2×10^{-6} 和 5×10^{-12} 。

** 对贝塞尔函数 $ynf(n,x)$, $|x|$ 的误差为 $[2 + 2.5n]$; 另一方面, 当 $n=128$ 时最大绝对误差为 2.2×10^{-6} 。对 $yn(n,x)$, 最大绝对误差为 5×10^{-12} 。

- ①在 SM 1.x 硬件上, 合并在 FMAD 指令中的加和乘指令的精度会因为中间尾数的截断而降低精度。
- ②在 SM 2.x 和之后的硬件上, 开发者可以通过指定选项 `--prec-div=true` 来使误差降低到 0ulp。
- ③对 float 类型, $|x| < 8$ 时误差为 9ulp; 否则, 最大绝对误差为 2.2×10^{-6} 。对 double, $|x| < 8$ 的误差为 7ulp; 否则, 最大绝对误差为 5×10^{-12} 。
- ④函数 `lgammaf()` 在区间 $-10.001, -2.264$ 之间的误差大于 6。`lgamma()` 在区间 $-11.001, -2.2637$ 内的误差大于 4。
- ⑤在 SM 2.x 或者之后的硬件上, 开发者可以指定 `--prec-sqrt=true`, 使误差降为 0。

1. 转换为整型

根据 C 语言的运行时库的定义, 函数 `nearbyint()` 和 `rint()` 会将浮点数按照当前方向取整到最接近的整型值, 在 CUDA 中, 会始终取整到最接近的偶数 (round-to-nearest-even)。在 C 运行时中, `nearbyint()` 和 `rint()` 的不同仅仅在处理 INEXACT 异常上。但是 CUDA 不会抛出浮点数的异常, 2 个函数的表现完全相同。

函数 `round()` 实现了最基本的取整策略: 对处于相邻 2 个整数中间的浮点数, 永远向上取整。英伟达公司反对使用这个函数, 因为这个函数使用 8 条指令完成, 而 `rint()` 函数和其

变种仅仅使用 1 条。trunc() 函数截断浮点值，向零取整，它会编译为 1 条指令。

2. 分数与指数

```
float frexpf(float x, int *eptr);
```

函数 frexpf() 会将输入的浮点数 x 分割为范围在 [0.5,1.0] 之间的浮点值 Significand 和以 2 为底的指数：

$$x = \text{Significand} \times 2^{\text{Exponent}}$$

```
float logbf( float x );
```

函数 logbf() 取出 x 的指数并返回这个浮点数。除了它更快一点，它几乎等价于 floorf(log2f(x))，如果 x 是非归一化，logbf() 返回将 x 归一化后的指数值。

```
float ldexpf( float x, int exp );
float scalbnf( float x, int n );
float scanblnf( float x, long n );
```

函数 ldexpf()、scalbnf() 和 scanblnf() 都是通过在浮点指数上进行操作计算 $x2^n$ 的函数。

3. 浮点求余

函数 modff() 会分割输入的参数为小数和整数两部分。

```
float modff( float x, float *intpart );
```

返回值是 x 的小数部分，与 x 有着相同的符号。

函数 remainderf(x,y) 计算用 x 除以 y 的浮点余数。返回值为 $x-n*y$ ，这里 n 为 x/y 并向最近的整数取整。如果 $|x-n*y|=0.5$ ， n 会向偶数取整。

```
float remquof( float x, float y, int *quo );
```

计算余数，并将 x/y 整数商的低位返回，它与 x/y 有相同符号。

4. 贝塞尔函数

n 阶贝塞尔函数与下述微分方程有关：

$$x^2 \frac{d^2y}{dx^2} + x \frac{dy}{dx} + [x^2 - n^2]y = 0$$

这里 n 可以为实数，但是为了与 C 运行时兼容， n 被限定为非负整数。

解决二阶传统微分方程的方法结合了第一和第二类贝塞尔函数，如下所示：

$$y(x) = c_1 J_n(x) + c_2 Y_n(x)$$

数学函数 jn[f]() 和 yn[f]() 分别计算 $J_n(x)$ 和 $Y_n(x)$ 。 $j0f()$ 、 $j1f()$ 、 $y0f()$ 和 $y1f()$ 计算 $n=0$ 和

$n=1$ 情况下的函数值。

5. 伽马函数

伽马函数是阶乘函数的扩展，对实数参数 x ，伽马函数对它减 1。伽马函数的形式有许多种，其中的一种形式如下：

$$\Gamma(x) = \int_0^{\infty} e^{-t} t^{x-1} dt$$

函数增长的非常快，以致即使使用相对较小输入值，其返回值也会超出精度，因此在函数 `tgamma()`（“真实伽马函数”）之外，函数库提供了另一个函数 `lgamma()`，返回伽马函数的自然对数。

8.3.7 延伸阅读

关于本节的专题，Goldberg 的综述文章（标题很吸引人 “what every computer scientist should know about floating-point arithmetic”）是一个很好的介绍。

http://download.oracle.com/docs/cd/E19957-01/806-3568/ncg_goldberg.html

英伟达公司的 Nathan Whitehead 和 Alex Fit-Florea 曾经共同撰写过白皮书：“Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs。”。<http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDAFloating-Point.pdf>

增加有效精度

Dekker 和 Kahan 研究了能提高浮点硬件有效精度几乎一倍的方法。他们用数对来换取指数范围的微小降低（源于中间数在区间两端的下溢和上溢）。阐述这一方法的论文包括：

Dekker, T.J. Point technique for extending the available precision. *Numer. Math.* 18, 1971, pp. 224–242.

Linnainmaa, S. Software for doubled-precision floating point computations. *ACM TOMS* 7, pp. 172–283(1981).

Shewchuk, J.R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry* 18, 1997, pp. 305–363.

在该议题上，Andrew Thall、Da Graça 和 Defour 做了一些直接与 GPU 相关的工作：

Guillaume, Da Graça, and David Defour. Implementation of float-float operators on graphics hardware, 7th Conference on Real Numbers and Computers, RNC7(2006).

<http://hal.archives-ouvertes.fr/docs/00/06/33/56/PDF/float-float.pdf>

Thall, Andrew. Extended-precision floating-point numbers for GPU computation. 2007.

http://andrewthall.org/papers/df64_qf128.pdf

8.4 条件代码

硬件使用“条件代码”或 CC 寄存器存储通常采用的 4 位状态向量（符号标志、进位标志、零标志和溢出标志），这个向量用于整数比较。使用比较指令，像 ISET，可以设置这些 CC 寄存器，并且这些寄存器可以通过断定（predication）或分支（divergence）指示执行方向。断定允许（或禁止）线程束内以线程为单位执行指令，而分支则是长指令序列的条件执行。因为 SM 内的处理器按照 SIMD 的风格执行指令，其指令执行粒度为线程束（每次 32 线程），分支可以用更少的代码获取结果，因为线程束内的所有线程会执行相同的代码路径。

8.4.1 断定

由于管理分支与汇聚（convergence）需要额外成本，编译器在短指令序列中使用断定技术。大多数指令的效果可以通过条件判断出来，如果条件为非真，指令即被禁用。这一禁用发生的足够早，这样经过断定的指令——像加载/存储和 TEX，将会取消本应产生的内存流量。注意断定不会对全局内存加载/存储合并操作起任何不良作用。指定给线程束内加载/存储指令的地址必须引用一段连续的内存空间，即使指令进行了断定。

在依赖条件变化的指令数比较小时往往使用断定技术。编译器使用能够支持多达约 7 条指令的启发试探法。除了能够避免在下文描述的管理分支同步栈的额外消耗，断定技术同样给编译器在提交微码时更多的优化机会（例如指令调度）。C 语言中三元操作符（即 ?:）被看作是一个偏好使用断定的编译器提示。

表 8-2 给出了一个绝佳的断定技术的例子，这个例子使用微码表示。当在共享内存位置执行原子操作时，编译器会向共享内存位置发射代码，不断地在共享内存位置循环，直到成功地执行完原子操作。指令 LDSLK（加载共享内存并加锁（load shared and lock））会返回一个条件判断代码，告诉机器是否获得了锁。随后执行操作的指令会根据这个条件代码进行分支断定。

```
/*0058*/ LDSLK P0, R2, [R3];
/*0060*/ @P0 IADD R2, R2, R0;
/*0068*/ @P0 STSUL [R3], R2;
/*0070*/ @!P0 BRA 0x58;
```

这一代码片段同样指明断定技术和分支技术有时会协同工作。最后一条指令，如果需要的话，条件分支尝试再次得到锁，同样这是进行断定。

8.4.2 分支与汇聚

断定技术在一小段条件代码上执行得很出色，尤其是 if 语句的下面不存在相关联的 else 的时候。对于大量的条件判断代码，断定技术变得低效，因为每一条指令都要被执行，而不管指令是否真正会影响计算结果。当大量的指令导致断定的执行花销超过了断定本身带来的

好处时，编译器便会使用条件分支。当线程束内代码的执行流依据条件判断而呈现几条不同的执行路径时，我们称这样的代码为分支（divergent）。

英伟达公司对于他们的硬件是如何支持分支路径的详细信息守口如瓶，并且保留在不同代次硬件上改变的权利。硬件维护一个位向量用以保存线程束中活动的线程。对于标记为不活动的线程，执行会用类似于断定技术的方式禁止。在执行分支之前，编译器执行一条特殊的指令，把这一活动线程的位压入栈。代码随后被执行两次，第一次为条件判断为真的线程执行，第二次为断定为假的线程执行。这两个阶段的执行由一个分支同步栈（branch synchronization stack）管理，Lindholm 的论文对此做了详细的描述：^②

如果线程束中的线程通过依赖于数据的条件分支而执行分支操作，那么线程束会依次执行每一个分支路径，禁用那些不在此路径上的线程。当所有的路径执行完成，线程重新汇聚到原始的执行路径。SM 使用分支同步栈来管理进行分支和汇聚的独立线程。分支只发生在一个线程束内；不同的线程束相互独立的执行，无论它们是执行共同的还是全然无关的代码路径。

PTX 标准中未提到分支同步栈，所以可以看见的证明其存在的现有证据只能在 cuobjdump 的反汇编输出中找到。SSY 指令会压入一个状态码（像程序计数器和活动线程掩码）到栈中，.S 指令的前缀弹出这一状态码，并且如果任一活动线程没有按照这一分支执行，会引起这些线程执行被指令 SSY 压入了状态的代码路径。

SSY/.S 是在线程执行发生分支时唯一需要的指令，所以如果编译器可以保证所有线程只会在某一代码路径中，你可能看不到被 SSY/.S 指令包括起来的分支。对 CUDA 中的分支来说重要的是意识到，在任何情况下，所有线程束内的线程遵循相同的执行路径是最高效的。

代码清单 8-2 中的循环同样包括一个很完整的既有分支又有汇聚的例子。SSY 指令（0x40 偏移处）和 NOP.S 指令（0x78 偏移处）分别实现了一次分支和汇聚的操作。代码在 LDSLK 和后续断定指令上循环，活动线程直到编译器知道所有的线程会汇聚后才退出，并且分支同步栈可以使用 NOP.S 指令弹出状态码。

```
/*0040*/ SSY 0x80;
/*0048*/ BAR.RED.POPC RZ, RZ;
/*0050*/ LD R0, [R0];
/*0058*/ LDSLK P0, R2, [R3];
/*0060*/ @P0 IADD R2, R2, R0;
/*0068*/ @P0 STSUL [R3], R2;
/*0070*/ @!P0 BRA 0x58;
/*0078*/ NOP.S CC.T;
```

8.4.3 特殊情况：最小值、最大值和绝对值

由于一些条件操作非常常见，硬件会直接支持它们。硬件支持取最小值和最大值的操

^② Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro, March–April 2008, p. 39-55.

作，既适用于整型，也适用于浮点型操作数，并且能够翻译为单条指令。除此之外，浮点指令还包括对源操作数的取反和取绝对值操作。

当 min/max 操作被使用时，编译器可以很好的检测出来，但是如果你想万无一失，在整型上调用函数 min()/max()，在浮点数上调用 fmin()/fmax() 即可。

8.5 纹理与表面操作

读写纹理和表面的指令相对于其他指令涉及了更多隐秘的状态。参数，例如地址、维度、格式和纹理内容的解释方式，都包含在一个 header 头结构中。header 是一个中间数据结构，它的软件抽象被称作纹理引用（texture reference）或表面引用（surface reference）。当开发者操纵纹理或表面引用时，CUDA 运行时和驱动程序必须翻译这些改变到 header 头结构中，纹理或表面指令引用会作为 header 的索引[⊖]。

在启动作用于纹理或表面的内核前，驱动程序必须确保所有的这些状态在硬件上设置正确。因此，启动一个这样的内核会花费稍长的时间。在费米架构中，纹理的读取通过一个专门的缓存子系统支持，独立于一级 / 二级缓存，同样的独立于常量内存。每一个 SM 有一个一级纹理缓存，并且每一个纹理处理集群（texture processor cluster, TPC）和图形处理集群（graphic processor cluster, GPC）都额外拥有一个二级纹理缓存。表面的读写则通过同样为全局内存传输提供服务的一级 / 二级缓存操作的。

开普勒架构有两个专门支持纹理的技术：通过纹理缓存结构读取全局内存而不需要绑定纹理引用的能力，和通过地址指定纹理 header 头结构而不是使用索引指定的能力。第二个技术也被称作“无绑定纹理”。

在 SM 3.5 和之后的硬件中，使用纹理缓存来读取全局内存可以通过使用 const-restrict 指针实现，也可以显式的调用函数 ldg()，这个函数在 sm_35_intrinsics.h 中被定义。

8.6 其他指令

8.6.1 线程束级原语

对于 CUDA 程序员来说，用不了多少时间就会意识到线程束（线程与线程块之间的一个基本单位）作为基本执行单元的重要性。在流处理器簇版本还是 1.x 的时代，英伟达公司就开始增加了一些仅作用于线程束的特殊指令。

[⊖] SM 3.x 添加了纹理对象（texture object），这使纹理和表面头信息可以由地址引用而不是索引。过去的硬件可以在内核中最多引用 128 项纹理或表面，但是在 SM 3.x 中，这一数量只受限于内存数量。

1. 投票

CUDA 架构是 32 位的，所以每一个线程束包含 32 个线程，是一个使指令能够评估条件并且广播 1 位结果给线程束中每一个线程的最适合的选择。VOTE 指令（在 SM 1.2 中首次应用）就是用于评估条件并将 1 位结果广播给线程束中所有线程的。`_any()` 函数在线程束 32 个线程中任何一个判断为真时会返回 1。`_all()` 函数在线程束 32 个线程都判断为真时会返回 1。

费米架构中增添了一个 VOTE 的变种，它可以传回一个线程束中 32 个线程的全部断定结果。函数 `_ballot()` 会为线程束中所有线程评估条件，并且返回一个 32 位的值，每一位分别代表着相同序号的线程的环境。

2. 洗牌

开普勒架构增添了洗牌（shuffle）指令，这个指令允许同一个线程束中的线程间进行数据交换而不经过共享内存中转。尽管这样的传输并没有节省时间，但这种方式有助于避免过多的读写操作，并且减少共享内存的使用。

下面的指令被包装进许多设备函数，使用了定义于 `sm_30_intrinsics.h` 中的内联 PTX 汇编代码。

```
int __shfl(int var, int srcLane, int width=32);
int __shfl_up(int var, unsigned int delta, int width=32);
int __shfl_down(int var, unsigned int delta, int width=32);
int __shfl_xor(int var, int laneMask, int width=32);
```

宽度参数 `width`，默认为线程束的宽度 32，该值一定在 2 ~ 32 的范围内，并为 2 的幂次。它使线程束细分成段。如果 `width` 小于 32，线程束的每段会作为一个单独的实体，并且起始逻辑线程编号为 0。一个线程仅能与它在同一段的线程进行数据交换。

`_shfl()` 返回的 `var` 值由编号为 `srcLane` 的线程给出。如果 `srcLane` 超过 0 到 `width-1` 的范围，线程本身的值 `var` 被返回。该指令变种可用来在一个线程束内广播值。`_shfl_up()` 通过从调用者线程编号减去 `delta`，并夹取到 0 到 `width-1` 的范围内，计算源线程编号。`_shfl_down()` 则通过在调用者线程编号上加上 `delta` 来计算源线程编号。

`_shfl_up()` 和 `_shfl_down()` 分别启用了线程束级扫描和反转扫描操作。`_shfl_xor()` 通过执行调用者线程编号与 `laneMask` 的按位异或计算源线程编号，存在源线程中的 `var` 值被返回。该指令变种可以用来执行线程束的归约（或子线程束），每一个线程使用不同顺序执行具有结合律的操作计算归约值。

8.6.2 线程块级原语

函数 `_syncthreads()` 起着栅栏的作用。它会导致所有的线程等待，直到线程块中的所有线程执行到函数 `_syncthreads()`。费米指令集（SM 2.x）增加了几个新的线程块级的栅栏指

令，这些指令均起着聚合线程块中线程信息的作用。

- `_syncthreads_count()`: 评估一个断定并返回断定为真的线程总数。
- `_syncthreads_or()`: 返回线程块中所有线程输入的数值的或运算。
- `_syncthreads_and()`: 返回线程块中所有线程输入的数值的与运算。

8.6.3 性能计数器

开发人员可以定义自己的一套性能计数器，并通过内置函数 `_prof_trigger()` 在代码中增加计数值。

```
void __prof_trigger(int counter);
```

调用这个函数会使每个线程束中相关的计数器增 1。`counter` 必须在范围 0~7 之内；计数器 8~15 是被保留的。计数器的值可以在配置文件中的 `prof_trigger_00~prof_trigger_07` 找到。

8.6.4 视频指令

本节所提到的视频指令只能通过内联 PTX 汇编代码访问。我们在这里描述它们的基本功能，来帮助开发人员决定视频指令是否对他们的应用有用。任何使用这一指令的人，应该遵循 PTX ISA 标准。

1. 标量视频指令

标量视频指令，在 SM 2.0 硬件中添加，实现了在视频处理中在短（8 位和 16 位）整数类型需要的高效操作。按照 PTX3.1 ISA 规范，这些函数的格式如下。

```
vop.dtype.atype.btype{.sat} d, a{.asel}, b{.bsel};  
vop.dtype.atype.btype{.sat}.secop d, a{.asel}, b{.bsel}, c;
```

源和目的操作数都是 32 位的寄存器。`dtype`、`atype` 和 `btype` 可能是 `.u32` 或 `.s32` 类型，这两种类型分别代表无符号和有符号 32 位整型数。选择符 `asel`/`bsel` 选择在源操作数中选取哪一个 8 或 16 位数：`b0`、`b1`、`b2` 和 `b3` 选择字节（从低有效位开始计数），`h0/h1` 分别选择最低有效和最高有效的 16 位。

一旦输入值被提取，通过符号扩展或零扩展为有符号 33 位整型数，并且基本操作被执行，产生一个 34 位的中间结果，其符号取决于 `dtype`。最后，其结果夹取到输出范围，并执行以下操作之一：

1) 在中间结果和第三个操作数上应用第二个操作符（`add`、`min` 或 `max`）。

2) 截断中间结果为 8 或 16 位值，并合并到第三个操作数指定的位置，以产生最终的结果。

然后写入低 32 位到目标操作数。

指令 `vset` 执行一个在 8、16 或 32 位输入操作数之间的比较，并生成相应的断定（1 或 0）作为输出。PTX 标量视频指令和对应操作在表 8-14 中给出。

表 8-14 标量视频指令

助记符	操作	助记符	操作
vabsdiff	$\text{abs}(a-b)$	vmin	$\text{min}(a,b)$
vadd	$a+b$	vshl	$a < b$
vavrg	$(a+b)/2$	vset	比较 a 和 b
vmad	$a*b+c$	vshr	$a>b$
vmax	$\text{max}(a,b)$	vsub	$a-b$

2. 矢量视频指令（仅适用于 SM 3.0）

在 SM 3.0 中加入的矢量视频指令，类似于标量的视频指令，将输入增强为标准整型格式，执行核心操作，并对输出进行夹取和根据需要进行合并。但通过在成对的 16 位值或 4 个一组的 8 位值上操作，矢量视频指令有着更高的执行性能。

表 8-15 总结了由这些指令实现的 PTX 指令和对应操作。它们在视频处理和特定的图像处理中（例如中值过滤）最有用处。

表 8-15 矢量视频指令

助记符	操作	助记符	操作
vabsdiff [2 4]	$\text{abs}(a-b)$	vmin [2 4]	$\text{min}(a,b)$
vadd [2 4]	$a+b$	vset [2 4]	比较 a 和 b
vavrg [2 4]	$(a+b)/2$	vsub [2 4]	$a-b$
vmax [2 4]	$\text{max}(a,b)$		

8.6.5 特殊寄存器

许多特殊寄存器的访问是通过引用内置变量 `threadIdx`、`blockIdx`、`blockDim` 和 `gridDim` 实现的。在 7.3 节我们详细的描述了这个三维结构，这些变量分别指定了线程 ID、线程块 ID、线程数和线程块数。

除了这些伪变量，另一个特殊的寄存器是流处理器簇的时钟寄存器（clock register），它在每个时钟周期增 1。这个计数器可以通过函数 `_clock()` 或 `_clock64()` 读取。时钟计数器独立地跟踪每个流处理器簇，类似于在 CPU 上的时间戳计数器，在衡量不同代码片段的性能表现时十分有用，但在试图计算挂钟时间时最好避免使用。

8.7 指令集

英伟达公司开发的架构主要有 3 种：特斯拉（SM 1.x）、费米（SM 2.x）和开普勒（SM 3.x）。随着这一系列的架构的推出，新的指令集也在英伟达公司更新他们产品时被添加。例

如，全局内存原子操作在最初的特斯拉架构的处理器中是不存在的（G80，即2006年出货的GeForce GTX 8800），但是所有随后发布的特斯拉架构GPU都包含了这一操作。所以在通过函数`cuDeviceComputeCapability()`查询SM版本时，G80的主要和次要版本信息会是1.0，其他的特斯拉级GPU会是1.1（或更高）。如果SM版本为1.1或更高，应用程序可以使用全局原子操作。

表8-16给出了SASS指令集，它们可以使用`cuobjdump`反汇编特斯拉架构（SM 1.x）硬件微码打印出来。费米与开普勒指令集彼此十分相似，只在是否支持表面加载/存储指令上有差别，所以它们的指令集在表8-17给出。在两个表格中，中间的一列给出了首次支持给定指令集的SM版本。

表8-16 SM 1.x指令集

OPCODE (操作码)	SM 版本	描述
浮点数		
COS	1.0	余弦
DADD	1.3	双精度浮点加法
DFMA	1.3	双精度浮点乘加法
DMAX	1.3	双精度浮点最大值
DMIN	1.3	双精度浮点最小值
DMUL	1.3	双精度浮点乘法
DSET	1.3	双精度浮点条件赋值
EX2	1.0	指数（2为底）
FADD/FADD32/FADD32I	1.0	单精度浮点加法
FCMP	1.0	单精度浮点比较
FMAD/ FMAD32/ FMAD32I	1.0	单精度浮点乘加法
FMAX	1.0	单精度浮点最大值
FMIN	1.0	单精度浮点最小值
FMUL/ FMUL32/ FMUL32I	1.0	单精度浮点乘法
FSET	1.0	单精度条件赋值
LG2	1.0	单精度浮点对数（2为底）
RCP	1.0	单精度浮点倒数
RRO	1.0	区间压缩（range reduction） [⊖] 运算符（在SIN/COS之前使用）
RSQ	1.0	平方根倒数
SIN	1.0	正弦
流控制		
BAR	1.0	栅栏同步 / __syncthreads()
BRA	1.0	条件分支

[⊖] 区间压缩，通过收敛于基本区间的函数近似表达一个定义于全域的函数，常用在三角函数的近似问题中。——译者注

(续)

OPCODE (操作码)	SM 版本	描述
BRK	1.0	条件跳出循环
BRX	1.0	从常量内存读取一个地址并进行分支跳转
C2R	1.0	数据寄存器的条件代码
CAL	1.0	无条件子程序调用
RET	1.0	子程序条件返回
SSY	1.0	设置同步点；在潜在分支指令之前使用
数据转换		
F2F	1.0	复制浮点值，同时转换为浮点数
F2I	1.0	复制浮点数，同时转换为整型数
I2F	1.0	复制整型数，同时转换为浮点数
I2I	1.0	复制整型数，同时转换为浮点数
整型		
IADD/ IADD32/ IADD32I	1.0	整型加
IMAD/ IMAD32/ IMAD32I	1.0	整型乘加
IMAX	1.0	整型最大值
IMIN	1.0	整型最小值
IMUL/ IMUL32/ IMUL32I	1.0	整型乘
ISAD/ ISAD32	1.0	绝对值差的整型和
ISET	1.0	整型条件赋值
SHL	1.0	左移
SHR	1.0	右移
内存操作		
A2R	1.0	移动地址寄存器到数据寄存器
ADA	1.0	把立即值加到地址寄存器
G2R	1.0	从共享内存移动到寄存器。.LCK 后缀用来实现共享内存原子操作，让存储片上锁，直到 R2G.UNL 指令执行完毕
GATOM. IADD/ EXCH/ CAS/ IMIN/ IMAX/ INC/ DEC/ IAND/ IOR/ IXOR	1.2	全局内存原子操作；执行原子操作并返回原始值
GLD	1.0	全局内存加载
GRED. IADD/ IMIN/ IMAX/ INC/ DEC/ IAND/ IOR/ IXOR	1.2	全局内存归约操作；执行没有返回值的原子操作
GST	1.0	存储到全局内存
LLD	1.0	从本地内存加载
LST	1.0	存储到本地内存
LOP	1.0	逻辑操作 (AND/OR/XOR)
MOV/ MOV32	1.0	移动源数据到目的地址
MVC	1.0	从常量内存移动数据
MVI	1.0	移动立即值

(续)

OPCODE (操作码)	SM 版本	描述
R2A	1.0	移动寄存器数据到地址寄存器
R2C	1.0	移动数据寄存器到条件代码
R2G	1.0	存储到共享内存。当使用.UNL 后缀，会释放一个过去在这个共享内存存储片上使用的锁
其他		
NOP	1.0	无操作
TEX/ TEX32	1.0	纹理获取
VOTE	1.2	线程束级投票原语
S2R	1.0	移动特殊寄存器（例如线程 ID）到寄存器

表 8-17 SM 2.x 和 SM 3.x 指令集

OPCODE (操作码)	SM 版本	描述
浮点数		
DADD	2.0	双精度加
DMUL	2.0	双精度乘
DMNMX	2.0	双精度最小值 / 最大值
DSET	2.0	双精度赋值
DSETP	2.0	双精度断定
DFMA	2.0	双精度乘加
FFMA	2.0	单精度乘加
FADD	2.0	单精度浮点加
FCMP	2.0	单精度浮点比较
FMUL	2.0	单精度浮点乘
FMNMX	2.0	单精度浮点最大值 / 最小值
FSWZ	2.0	单精度浮点混合 (swizzle)
FSET	2.0	单精度浮点赋值
FSETP	2.0	单精度浮点赋值断定
MUFU	2.0	数值模拟函数 (SFU) 操作符
RRO	2.0	区间压缩操作符 (在 MUFU sin/cos 之前使用)
整型		
BFE	2.0	位域抽取
BFI	2.0	位域插入
FLO	2.0	查找前导位
IADD	2.0	整型加
ICMP	2.0	整型比较与选择
IMAD	2.0	整型乘加
IMNMX	2.0	整型最大值 / 最小值
IMUL	2.0	整型乘

(续)

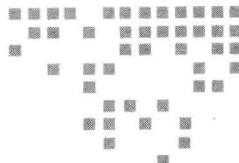
OPCODE (操作码)	SM 版本	描述
ISAD	2.0	绝对值差的整型和
ISCADD	2.0	整型加一指定值
ISET	2.0	整型赋值
ISETP	2.0	整型赋值断定
LOP	2.0	逻辑操作符 (AND/OR/XOR)
SHF	3.5	漏斗移位
SHL	2.0	左移位
SHR	2.0	右移位
POPC	2.0	总体计数
数据转换		
F2F	2.0	浮点数到浮点数
F2I	2.0	浮点数到整型数
I2F	2.0	整型数到浮点数
I2I	2.0	整型数到整型数
标量视频		
VABSDIFF	2.0	标量视频绝对差
VADD	2.0	标量视频加
VMAD	2.0	标量视频乘加
VMAX	2.0	标量视频最大值
VMIN	2.0	标量视频最小值
VSET	2.0	标量视频视频赋值
VSHL	2.0	标量视频左移位
VSHR	2.0	标量视频右移位
VSUB	2.0	标量视频减
矢量 (SIMD) 视频		
VABSDIFF2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 绝对差
VADD2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 加
VAVRG2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 平均值
VMAX2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 最大值
VMIN2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 最小值
VSET2 (4)	3.0	矢量视频 2×16 位 (4×8 位) 赋值
	3.0	矢量视频 2×16 位 (4×8 位) 减
数据移动		
MOV	2.0	移动
PRMT	2.0	排列
SEL	2.0	选择 (条件移动)
SHFL	3.0	线程束洗牌
断定 / 条件代码		

(续)

OPCODE (操作码)	SM 版本	描述
CSET	2.0	条件代码赋值
CSETP	2.0	条件代码赋值断定
P2R	2.0	断定至寄存器
R2P	2.0	寄存器到断定
PSET	2.0	断定赋值
PSETP	2.0	断定赋值断定
纹理		
TEX	2.0	纹理获取
TLD	2.0	纹理加载
TLD4	2.0	4 纹素的纹理加载
TXQ	2.0	纹理查询
内存操作		
ATOM	2.0	原子内存操作
CCTL	2.0	缓存控制
CCTLL	2.0	缓存控制 (本地)
LD	2.0	从内存加载
LDC	2.0	加载常量
LDG	3.5	非一致全局加载 (通过纹理缓存读取)
LDL	2.0	从本地内存加载
LDLK	2.0	加载并加锁
LDS	2.0	从共享内存加载
LDSLK	2.0	从共享内存加载并加锁
LDU	2.0	统一加载
LD_LDU	2.0	组合使用 LD 和 LDU
LDS_LDU	2.0	组合 LD 和 LDU
MEMBAR	2.0	内存栅栏
RED	2.0	原子内存归约操作
ST	2.0	存储到内存
STL	2.0	存储到本地内存
STUL	2.0	存储并解锁
STS	2.0	存储到共享内存
STSUL	2.0	存储到共享内存并解锁
表面内存 (费米)		
SULD	2.0	表面加载
SULEA	2.0	表面加载有效地址
SUQ	2.0	表面查询
SURED	2.0	表面归约
SUST	2.0	表面存储

(续)

OPCODE (操作码)	SM 版本	描述
表面内存 (开普勒)		
SUBFM	3.0	表面位域合并
SUCLAMP	3.0	表面夹取
SUEAU	3.0	表面有效地址
SULDGA	3.0	表面加载通用地址
SUSTGA	3.0	表面存储通用地址
流控制		
BRA	2.0	分支到相对地址
BPT	2.0	断点 / 陷阱
BRK	2.0	分支到相对索引地址
BRX	2.0	跳出循环
CAL	2.0	调用相对地址
CONT	2.0	继续循环
EXIT	2.0	退出程序
JCAL	2.0	调用绝对地址
JMP	2.0	跳转到绝对地址
JMX	2.0	跳到绝对索引地址
LONGJMP	2.0	长跳转
PBK	2.0	预中断相对地址
PCNT	2.0	预继续相对地址
PLONGJMP	2.0	预长跳转相对地址
PRET	2.0	预返回相对地址
RET	2.0	从调用中返回
SSY	2.0	设置同步点；在潜在的分支指令前使用
其他指令		
B2R	2.0	栅栏到寄存器
BAR	2.0	栅栏同步
LEPC	2.0	加载有效程序计数器
NOP	2.0	无操作
S2R	2.0	特殊寄存器到寄存器 (读时使用，例如，线程或线程块 ID)
VOTE	2.0	在整个线程束中查询条件



第 9 章

Chapter 9

多 GPU

本章介绍 CUDA 面向多 GPU 编程的设施，包括线程模型、点对点和多 GPU 间的同步。作为一个例子，我们将首先探讨多 GPU 之间的同步。为了达到此目的，需要采用可分享锁页内存，在 CUDA 流和事件的支持下实现一个点对点的内存复制。然后，我们讨论如何利用单线程和多线程的多 GPU 实现，解决 N- 体问题（该问题的详细说明见第 14 章）。

9.1 概述

多 GPU 系统通常包含多个 GPU 主板。如第 2.3 节中描述的那样，这些 GPU 主板可能带有一个 PCIe 桥接芯片（例如 GeForce GTX 690），或者插在多个 PCIe 插槽上，或者两种情况兼而有之。多 GPU 系统的每个 GPU 被 PCIe 总线分离开来，所以连接到本地 GPU（它的设备内存）的内存带宽和连接到其他的 GPU 以及 CPU 的内存带宽之间有巨大差距。

许多属于多 GPU 的 CUDA 功能，例如点对点寻址，要求这些 GPU 具有相同规格。对于那些可以指定目标硬件的应用程序（如为特定硬件配置定制的垂直应用程序），这一要求是没问题的。但当硬件系统包含各式 GPU 时（例如，同时配有日常使用低配置显卡和强大的游戏卡），应用程序可能需要使用启发式算法来决定使用哪个 GPU，或者在这些 GPU 间进行负载平衡，以使强大的 GPU 做更多的计算工作。

对所有使用多 GPU 的 CUDA 应用来说，可分享锁页内存是关键因素。正如 5.1.2 小节所述，可分享锁页内存也是锁页内存，只是它映射给所有的 CUDA 上下文，这样任何 GPU 都可以直接读取或写入该内存。

CPU 线程模型

CUDA 4.0 以前，驱动多个 GPU 的唯一办法是分别为它们创建一个 CPU 线程。在每个 CPU 线程中，必须在执行任何 CUDA 代码之前调用一次 `cudaSetDevice()` 函数。当 CPU 线程开始启动 CUDA 之际，此举意在告诉 CUDA 应该初始化哪个设备。无论是哪个 CPU 线程发出函数调用，该线程将获得独占访问 GPU 的权利，因为 CUDA 驱动程序尚无法做到线程安全（thread-safe），无法支持多个线程同时访问同一个 GPU。

在 CUDA 4.0 中，`cudaSetDevice()` 函数做了修改，实现了大家先前预期的语义：它告诉 CUDA 哪个 GPU 应该执行后续的 CUDA 操作。若有多个线程同时对同一个 GPU 进行操作，应该能够正确工作，只是可能会产生轻微的性能损失。对于我们的 N- 体示例程序，只使用一个 CPU 线程逐次在给定的设备上运行。多线程方案则拥有 N 个线程，每个线程作用在一个特定设备上；相应的，单线程方案是一个线程轮番作用于 N 个设备。

9.2 点对点机制

当 CUDA 程序使用多个 GPU 时，它们被称为“同事节点”（peer）。因为应用程序通常对它们一视同仁，仿佛它们是在一个项目上通力协作的同事。CUDA 支持两种风格的点对点模式：显式的点对点内存复制和点对点寻址[⊖]。

9.2.1 点对点内存复制

内存复制操作可以在任何两个不同设备的内存之间进行。当统一虚拟寻址（Unified Virtual Addressing，UVA）起作用时，鉴于 CUDA 可以推断出每个设备的节点内存，使用普通的内存复制函数族即可实现点对点内存复制。而如果 UVA 没有启用，点对点内存复制则必须显式调用 `cudaMemcpyPeer()`、`cudaMemcpyPeerAsync()`、`cudaMemcpy3DPeer()` 或 `cudaMemcpy3DPeerAsync()`。



注意 CUDA 不仅仅能够在两个可以直接互相寻址的设备内存间进行复制操作，而且可以在任何两个设备之间进行内存复制。如有必要，CUDA 内存复制操作将使用主机内存作为中转缓冲区，从而系统中的任何设备都可以访问它。

点对点内存复制操作无法与其他操作并发执行。点对点内存复制操作要想开始，必须保证 GPU 上之前的等待操作全部完成；同时只有点对点内存复制操作完成之后，才能执行后续的操作。一旦有可能，CUDA 将在两个指针之间进行直接点对点映射。使用直接映射，所产生的复制操作不必通过主机内存中转，速度更快。

[⊖] 对于点对点的寻址方案（peer-to-peer addressing），术语 peer 也表明所用 GPU 应该是完全相同的。

9.2.2 点对点寻址

设备内存的点对点映射，如图 2-20 所示，使得一个 GPU 上的内核可以读写驻留在另一个 GPU 上的内存。由于 GPU 只能使用点对点的方式以 PCIe 速率读写数据，开发人员必须按照下述步骤划分工作量：

- 1) 每个 GPU 拥有大约等量的工作要做。
- 2) GPU 只需交换适量的数据。

基于流水线方式的计算机视觉系统即是这类系统的典型例子。在 GPU 构成的流水线结构中，每个步骤计算一个中间数据结构（例如，所预测对象的位置），这一数据结构需要送入流水线中下一个 GPU 进一步分析。这一流水线计算方式也称为模板计算（stencil computation），其中多数计算任务是由这些独立的 GPU 来单独执行的，但在计算步骤之间必须交换边缘数据。

欲让点对点寻址起作用，有赖于下列条件：

- 启用了统一虚拟寻址。
- 双端的 GPU 的计算能力必须是 SM2.x 或更高，并且必须基于在同等芯片。
- GPU 必须位于相同的 I/O 集线器。

可以调用 cu(da)DeviceCanAccessPeer() 来查询当前设备是否可以对另一个设备的内存进行映射。

```
cudaError_t cudaDeviceCanAccessPeer(int *canAccessPeer, int device,
int peerDevice);
CURError cuDeviceCanAccessPeer(int *canAccessPeer, CUdevice device,
CUdevice peerDevice);
```

点对点的映射并不是默认启用的，必须明确调用 cudaDeviceEnablePeerAccess() 或 cuCtxEnablePeerAccess() 来启用。

```
cudaError_t cudaDeviceEnablePeerAccess(int peerDevice, unsigned int
flags);
CURError cuCtxEnablePeerAccess(CUcontext peerContext, unsigned int
Flags);
```

一旦点对点访问被启用，所有其他节点设备上的内存（包括新的内存分配），都是当前设备可以访问的。如要终止这种访问，可以调用 cudaDeviceDisablePeerAccess() 或 cuCtxDisablePeerAccess() 达到目的。

点对点访问需要占用少量额外的内存来保存更多的页表。这同样会导致更昂贵的内存分配，因为内存必须被映射到所有参与的设备。点对点功能使上下文可以通过两种方式读写属于其他上下文的内存：或者通过用内存复制函数（可能会需要通过系统内存进行中转），或者直接通过内核对全局内存指针进行读写。

cudaDeviceEnablePeerAccess() 函数对属于另一台设备的内存进行映射。点对点的内存寻址是不对称的。GPU A 可能对 GPU B 分配的内存进行映射，但 GPU A 分配的内存对 GPU B 则不可用。为了让两个 GPU 互相使用对方的内存，每个 GPU 必须显式地对对方的内存进行

映射。

```
// tell device 1 to map device 0 memory
cudaSetDevice( 1 );
cudaDeviceEnablePeerAccess( 0, cudaPeerAccessDefault );
// tell device 0 to map device 1 memory
cudaSetDevice( 0 );
cudaDeviceEnablePeerAccess( 1, cudaPeerAccessDefault );
```



注意 对基于 PCIe 3.0 桥接芯片的 GPU 主板（如特斯拉 K10），即使插到了 PCIe 2.0 的插槽上，两个处于同一显卡的 GPU 也能以 PCIe 3.0 的速率进行通信。

9.3 UVA：从地址推断设备

由于统一虚拟寻址（UVA）在具有点对点能力的系统上始终启用，不同的设备的地址范围并不存在重叠，并且驱动程序可以从一个指针值推断出它所属于的设备。可以使用 `cudaPointerGetAttribute()` 函数查询 UVA 指针的信息，包括所属上下文信息。

```
CUresult CUDA API cuPointerGetAttribute(void *data, CUpointer_
attribute attribute, CUdeviceptr ptr);
```

`cuPointerGetAttribute()` 或 `cudaPointerGetAttributes()` 函数可以用来查询一个指针的属性。表 9-1 中给出的值可以传递给 `cuPointerGetAttribute()`。由 `cudaPointerGetAttributes()` 返回的结构如下所示：

```
struct cudaPointerAttributes {
    enum cudaMemoryType memoryType;
    int device;
    void *devicePointer;
    void *hostPointer;
}
```

其中的参数说明如下。`memoryType` 可以是 `cudaMemoryTypeHost` 或 `cudaMemoryTypeDevice`。`device` 是指针分配的设备。对于设备内存，`device` 标识出对应 `ptr` 分配的内存所在的设备。对于主机内存，`device` 标识出执行分配操作时处于活跃的设备。`devicePointer` 提供了设备指针值，可以用来被当前设备引用 `ptr`。如果指针无法为当前设备访问，`devicePointer` 是 `NULL`。`hostPointer` 提供了主机指针值，该值可以用来被 CPU 引用 `ptr`。如果指针无法为当前主机访问，`hostPointer` 是 `NULL`。

表 9-1 `cuPointerGetAttribute()` 的属性

属性	返回类型	描述
<code>CU_POINTER_ATTRIBUTE_CONTEXT</code>	<code>CUcontext</code>	上下文，其中的一个指针被分配或者注册
<code>CU_POINTER_ATTRIBUTE_MEMORY_TYPE</code>	<code>CUmemorytype</code>	一个指针的物理位置
<code>CU_POINTER_ATTRIBUTE_DEVICE_POINTER</code>	<code>CUdeviceptr</code>	指针，其所在内存可以为 GPU 访问
<code>CU_POINTER_ATTRIBUTE_HOST_POINTER</code>	<code>void *</code>	指针，其所在内存可以为主机访问

9.4 多GPU间同步

CUDA事件可使用 cu(da)StreamWaitEvent() 函数对多个 GPU 进行同步。如果在两个 GPU 之间存在生产者 / 消费者的关系，应用程序可以让生产者 GPU 记录一个事件，然后让消费者 GPU 在它的命令流中插入一个流等待事件。当消费者 GPU 遇到流等待，它将停止处理命令，直到生产者 GPU 越过了 cu(da)EventRecord() 被调用时的执行点。



注意 在 CUDA 5.0 中，如 7.5 节中描述，设备运行时不支持任何 GPU 之间的同步。在未来的版本中，这个限制可能会适当放宽。

代码清单 9-1 给出了 chMemcpyPeerToPeer() 函数的具体实现[⊖]。当 GPU 之间不存在直接映射时，该段实现点对点内存复制操作的代码使用可分享内存和 GPU 间同步达到类似于 CUDA 提供的内存复制操作的功能。该函数的工作方式类似于代码清单 6-2 中的 chMemcpyHtoD() 所执行主机到设备的内存复制操作：在主机内存中分配一个中转缓冲区，复制操作先把源 GPU 的数据复制到中转缓冲区并记录事件。但是，与主机到设备的内存复制操作不同之处在于，这里不需要 CPU 执行同步，因为所有的同步均在 GPU 端完成。因为内存复制操作和事件记录是异步的，在发出初始内存复制与事件记录之后，CPU 即可请求目标 GPU 等待事件的发生并发出对同一缓冲区的一个内存复制操作。设置两个中转缓冲区和两个 CUDA 事件是必要的，因为两个 GPU 可以并发的执行复制到和复制出缓冲区的操作，就如同在主机到设备的内存复制过程中 CPU 与 GPU 并行操纵两个中转缓冲区。CPU 在输入缓冲区和输出缓冲区之间循环，发出内存复制和事件记录命令并穿梭于中转缓冲区，直到它已完成全部数据的复制请求，剩下要做的事情就是等待两个 GPU 来完成任务的处理。



注意 跟英伟达提供的 CUDA 实现一样，我们的点对点内存复制是同步的。

代码清单 9-1 chMemcpyPeerToPeer()

```
cudaError_t
chMemcpyPeerToPeer(
    void *_dst, int dstDevice,
    const void *_src, int srcDevice,
    size_t N )
{
    cudaError_t status;
    char *dst = (char *) _dst;
    const char *src = (const char *) _src;
    int stagingIndex = 0;
```

[⊖] 为清楚起见，CUDART_CHECK 错误处理代码被移除。

```

while ( N ) {
    size_t thisCopySize = min( N, STAGING_BUFFER_SIZE );

    cudaSetDevice( srcDevice );
    cudaStreamWaitEvent( 0, g_events[dstDevice][stagingIndex], 0 );
    cudaMemcpyAsync( g_hostBuffers[stagingIndex], src,
        thisCopySize, cudaMemcpyDeviceToHost, NULL );
    cudaEventRecord( g_events[srcDevice][stagingIndex] );

    cudaSetDevice( dstDevice );
    cudaStreamWaitEvent( 0, g_events[srcDevice][stagingIndex], 0 );
    cudaMemcpyAsync( dst, g_hostBuffers[stagingIndex],
        thisCopySize, cudaMemcpyHostToDevice, NULL );
    cudaEventRecord( g_events[dstDevice][stagingIndex] );
    dst += thisCopySize;
    src += thisCopySize;
    N -= thisCopySize;
    stagingIndex = 1 - stagingIndex;
}
// Wait until both devices are done
cudaSetDevice( srcDevice );
cudaDeviceSynchronize();

cudaSetDevice( dstDevice );
cudaDeviceSynchronize();

Error:
    return status;
}

```

9.5 单线程多 GPU 方案

当使用 CUDA 运行时时，一个单线程的应用程序可以驱动多个 GPU。CPU 线程通过调用 `cudaSetDevice()` 来指定欲控制的 GPU。这一模式语句用在代码清单 9-1 中，对源和目标 GPU 进行切换。该模式不仅适用于点对点内存复制，也适用于 9.5.2 节中描述的 N- 体的单线程多 GPU 实现。在驱动程序 API 中，CUDA 维护一个当前上下文栈，使子程序可以很容易地改变和重新恢复调用者的当前上下文。

9.5.1 当前上下文栈

驱动程序 API 的应用程序可以使用当前上下文栈管理当前上下文：`cuCtx PushCurrent()` 函数使得一个新上下文成为当前上下文，并把它压入栈的顶部；`cuCtxPopCurrent()` 弹出当前上下文，并恢复前一个当前上下文。代码清单 9-2 给出驱动程序 API 版本的 `chMemcpyPeerToPeer()`，它使用 `cuCtxPopCurrent()` 和 `cuCtxPushCurrent()` 在两个上下文之间执行点对点的内存复制。

当前上下文栈最初在 CUDA 2.2 中引入。当时 CUDA 运行时和驱动程序 API 不能在同一应用程序中配合使用。该限制已在后续版本中放宽。

代码清单9-2 chMemcpyPeerToPeer(驱动程序API版本)

```
CUresult
chMemcpyPeerToPeer(
    void *_dst, CUcontext dstContext, int dstDevice,
    const void *_src, CUcontext srcContext, int srcDevice,
    size_t N )
{
    CUresult status;
    CUdeviceptr dst = (CUdeviceptr) (intptr_t) _dst;
    CUdeviceptr src = (CUdeviceptr) (intptr_t) _src;
    int stagingIndex = 0;

    while ( N ) {
        size_t thisCopySize = min( N, STAGING_BUFFER_SIZE );

        CUDA_CHECK( cuCtxPushCurrent( srcContext ) );
        CUDA_CHECK( cuStreamWaitEvent(
            NULL, g_events[dstDevice][stagingIndex], 0 ) );
        CUDA_CHECK( cuMemcpyDtoHAsync(
            g_hostBuffers[stagingIndex],
            src,
            thisCopySize,
            NULL ) );
        CUDA_CHECK( cuEventRecord(
            g_events[srcDevice][stagingIndex],
            0 ) );

        CUDA_CHECK( cuCtxPopCurrent( &srcContext ) );
        CUDA_CHECK( cuCtxPushCurrent( dstContext ) );
        CUDA_CHECK( cuStreamWaitEvent(
            NULL,
            g_events[srcDevice][stagingIndex],
            0 ) );
        CUDA_CHECK( cuMemcpyHtoDAsync(
            dst,
            g_hostBuffers[stagingIndex],
            thisCopySize,
            NULL ) );
        CUDA_CHECK( cuEventRecord(
            g_events[dstDevice][stagingIndex],
            0 ) );

        CUDA_CHECK( cuCtxPopCurrent( &dstContext ) );

        dst += thisCopySize;
        src += thisCopySize;
        N -= thisCopySize;
        stagingIndex = 1 - stagingIndex;
    }

    // Wait until both devices are done
    CUDA_CHECK( cuCtxPushCurrent( srcContext ) );
    CUDA_CHECK( cuCtxSynchronize() );
    CUDA_CHECK( cuCtxPopCurrent( &srcContext ) );

    CUDA_CHECK( cuCtxPushCurrent( dstContext ) );
    CUDA_CHECK( cuCtxSynchronize() );
    CUDA_CHECK( cuCtxPopCurrent( &dstContext ) );

    Error:
        return status;
}
```

9.5.2 N- 体问题

N- 体计算问题（在第 14 章中详细描述）以 $O(N^2)$ 的时间复杂度计算 N 个作用力。其输出可独立地计算。在 k 个 GPU 构成的系统上，我们的多 GPU 实现把计算任务分割成 k 个部分。

我们的实现均假设所有 GPU 是相同的，所以我们可以均匀地划分计算任务。针对那些采用 GPU 性能不均匀的应用程序，或者那些工作量运行时较难预测的应用程序，可以更精细地划分计算任务，并让主机代码从一个任务队列中提交工作片段到 GPU。

代码清单 9-3 对代码清单 14-3 作了修改。它有两个额外的参数（一个基索引 base 和作用力的子数组的大小为 n），来计算 N- 体问题输出数组的一个子集。这个 `_device_` 函数被一个声明为 `_global_` 的外层内核调用。它组织成这种方式是为了重用代码而不发生链接错误。如果函数声明为 `_global_`，链接器会产生一个符号冗余的错误^②。

代码清单 9-3 N- 体问题的内核函数（多 GPU 配置）

```
inline __device__ void
ComputeNBodyGravitation_Shared_multiGPU(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t base,
    size_t n,
    size_t N )
{
    float4 *posMass4 = (float4 *) posMass;
    extern __shared__ float4 shPosMass[];
    for ( int m = blockIdx.x*blockDim.x + threadIdx.x;
          m < n;
          m += blockDim.x*gridDim.x )
    {
        size_t i = base+m;
        float acc[3] = {0};
        float4 myPosMass = posMass4[i];
#pragma unroll 32
        for ( int j = 0; j < N; j += blockDim.x ) {
            shPosMass[threadIdx.x] = posMass4[j+threadIdx.x];
            __syncthreads();
            for ( size_t k = 0; k < blockDim.x; k++ ) {
                float fx, fy, fz;
                float4 bodyPosMass = shPosMass[k];

                bodyBodyInteraction(
                    &fx, &fy, &fz,
                    myPosMass.x, myPosMass.y, myPosMass.z,
                    bodyPosMass.x,
                    bodyPosMass.y,
                    bodyPosMass.z,
                    bodyPosMass.w,
                    softeningSquared );
                acc[0] += fx;
```

^② 这是一个传统的解决方法。CUDA5.0 新增的链接器，使 `_global_` 函数可以编译为静态链接库并链接到应用程序。

```

        acc[1] += fy;
        acc[2] += fz;
    }
    __syncthreads();
}
force[3*m+0] = acc[0];
force[3*m+1] = acc[1];
force[3*m+2] = acc[2];
}
}

```

代码清单 9-4 给出了 N- 体问题的单线程多 GPU 版本的主机代码[⊖]。dptrPosMass 和 dptrForce 数组为每个 GPU 的输入及输出数组跟踪设备指针 (GPU 的最大数量在 nbody.h 中声明为常数; 默认是 32)。跟分派工作到 CUDA 流的过程相似, 函数在计算的不同阶段使用不同的循环: 第一个循环为每个 GPU 分配并填充输入数组; 第二个循环启动内核和输出数据的异步复制; 第三个循环依次在每个 GPU 调用 cudaDeviceSynchronize() 函数。这样组织函数可以最大化 CPU/GPU 重叠。在第一个循环中, 当 CPU 忙着为第 i 个 GPU 分配内存之际, 前 i 个 GPU (第 $0 \sim i-1$ 个 GPU) 能够执行主机到设备的异步内存复制操作。如果内核启动和设备到主机的异步复制在第一个循环中调用同步 cudaMalloc(), 会降低性能, 因为它们要与当前 GPU 进行同步。

代码清单 9-4 N- 体问题的主机代码 (单线程多 GPU 配置)

```

float
ComputeGravitation_multiGPU_singlethread(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N
)
{
    cudaError_t status;

    float ret = 0.0f;

    float *dptrPosMass[g_maxGPUs];
    float *dptrForce[g_maxGPUs];

    chTimerTimestamp start, end;
    chTimerGetTime( &start );

    memset( dptrPosMass, 0, sizeof(dptrPosMass) );
    memset( dptrForce, 0, sizeof(dptrForce) );
    size_t bodiesPerGPU = N / g_numGPUs;
    if ( !(0 != N % g_numGPUs) || (g_numGPUs > g_maxGPUs) ) {
        return 0.0f;
    }

    // kick off the asynchronous memcpy's - overlap GPUs pulling
    // host memory with the CPU time needed to do the memory
    // allocations.
    for ( int i = 0; i < g_numGPUs; i++ ) {
        cudaSetDevice( i );

```

[⊖] 为版面美观, 这里的错误检查代码已被移除。

```

        cudaMalloc( &dptraPosMass[i], 4*N*sizeof(float) );
        cudaMalloc( &dptraForce[i], 3*bodiesPerGPU*sizeof(float) );
        cudaMemcpyAsync(
            dptraPosMass[i],
            g_hostAOS_PosMass,
            4*N*sizeof(float),
            cudaMemcpyHostToDevice );
    }
    for ( int i = 0; i < g_numGPUs; i++ ) {
        cudaSetDevice( i );
        ComputeNBodyGravitation_Shared_device<<<
            300,256,256*sizeof(float4)>>>(
                dptraForce[i],
                dptraPosMass[i],
                softeningSquared,
                i*bodiesPerGPU,
                bodiesPerGPU,
                N );
        cudaMemcpyAsync(
            g_hostAOS_Force+3*bodiesPerGPU*i,
            dptraForce[i],
            3*bodiesPerGPU*sizeof(float),
            cudaMemcpyDeviceToHost );
    }
    // Synchronize with each GPU in turn.
    for ( int i = 0; i < g_numGPUs; i++ ) {
        cudaSetDevice( i );
        cudaDeviceSynchronize();
    }
    chTimerGetTime( &end );
    ret = chTimerElapsedTime( &start, &end ) * 1000.0f;
Error:
    for ( int i = 0; i < g_numGPUs; i++ ) {
        cudaFree( dptraPosMass[i] );
        cudaFree( dptraForce[i] );
    }
    return ret;
}

```

9.6 多线程多 GPU 方案

CUDA 自始至终支持多个 GPU，但在 CUDA 4.0 之前，每个 GPU 都必须由单独的 CPU 线程控制。对于需要大量 CPU 计算能力的工作负载，这一要求并不过分。因为只有通过多线程，现代多核处理器的全部潜力才可以释放。

N- 体问题的多线程多 GPU 实现为每个 GPU 创建一个 CPU 线程，它在执行给定的 N- 体途径时，委托每个线程来分派和同步工作任务。主线程在 GPU 之间均匀分割工作任务，并通过发送一个事件信号，委托每个工作线程执行任务（或在 POSIX 平台（如 Linux）的一个信号量），然后等待所有工作线程均发出完成的信号才能进行下一步。随着 GPU 数量的增长，同步的代价会受益于并行性，而逐步减少。

这里的 N- 体实现，采用了与 14.9 节中“多线程实现”相同的多线程库。附录 A 中所描述的 WorkerThread 类，使应用程序线程把工作任务委托给 CPU 线程，然后在工作线程完成

时对委托任务进行同步。

代码清单9-5给出了创建和初始化CPU线程的主机代码。有两个全局变量g_numGPUs和g_GPUThreadPool，分别保存GPU的数量和每一个GPU的工作线程。每个CPU线程被创建后，它以同步的方式调用initializeGPU()函数进行初始化，它将为CPU线程分配一个GPU，并且这一分配在应用程序的整个执行过程中都不会改变。

代码清单9-5 多线程多GPU方案的初始化代码

```

workerThread *g_CPUThreadPool;
int g_numCPUCores;

workerThread *g_GPUThreadPool;
int g_numGPUs;

struct gpuInit_struct
{
    int iGPU;

    cudaError_t status;
};

void
initializeGPU( void *_p )
{
    cudaError_t status;

    gpuInit_struct *p = (gpuInit_struct *) _p;
    CUDART_CHECK( cudaSetDevice( p->iGPU ) );
    CUDART_CHECK( cudaSetDeviceFlags( cudaDeviceMapHost ) );
    CUDART_CHECK( cudaFree(0) );

Error:
    p->status = status;
}

// ... below is from main()

if ( g_numGPUs ) {
    chCommandLineGet( &g_numGPUs, "numgpus", argc, argv );
    g_GPUThreadPool = new workerThread[g_numGPUs];
    for ( size_t i = 0; i < g_numGPUs; i++ ) {
        if ( ! g_GPUThreadPool[i].initialize( ) ) {
            fprintf( stderr, "Error initializing thread pool\n" );
            return 1;
        }
    }
    for ( int i = 0; i < g_numGPUs; i++ ) {
        gpuInit_struct initGPU = {i};
        g_GPUThreadPool[i].delegateSynchronous(
            initializeGPU,
            &initGPU );
        if ( cudaSuccess != initGPU.status ) {
            fprintf( stderr, "Initializing GPU %d failed "
                "with %d (%s)\n",
                i,
                initGPU.status,
                cudaGetStringError( initGPU.status ) );
            return 1;
        }
    }
}

```

一旦工作线程初始化完成，它们将挂起并等待一个线程同步原语，直到应用程序线程给它们分派工作。代码清单 9-6 显示了分派工作给 GPU 的主机代码：gpuDelegation 结构封装了一个给定 GPU 必须做的工作；对于每个由代码清单 9-5 中代码创建工作线程均启动 gpuWorkerThread 函数。代码清单 9-7 中显示的应用程序线程代码，为每个工作线程创建一个 gpuDelegation 结构并调用 delegateAsynchronous() 方法以启动代码清单 9-6 中代码。WaitAll() 方法将等待所有工作线程的完成。关于基于单线程和多线程的多 GPU 版本的 N- 体解决方案的性能和可扩展性，总结在 14.7 节。

代码清单 9-6 主机代码（工作线程）

```

struct gpuDelegation {
    size_t i;    // base offset for this thread to process
    size_t n;    // size of this thread's problem
    size_t N;    // total number of bodies

    float *hostPosMass;
    float *hostForce;
    float softeningSquared;

    cudaError_t status;
};

void
gpuWorkerThread( void * _p )
{
    cudaError_t status;
    gpuDelegation *p = (gpuDelegation *) _p;
    float *dptrPosMass = 0;
    float *dptrForce = 0;

    //
    // Each GPU has its own device pointer to the host pointer.
    //
    CUDART_CHECK( cudaMalloc( &dptrPosMass, 4*p->N*sizeof(float) ) );
    CUDART_CHECK( cudaMalloc( &dptrForce, 3*p->n*sizeof(float) ) );
    CUDART_CHECK( cudaMemcpyAsync(
        dptrPosMass,
        p->hostPosMass,
        4*p->N*sizeof(float),
        cudaMemcpyHostToDevice ) );
    ComputeNBodyGravitation_multiGPU<<<300,256,256*sizeof(float4)>>>(
        dptrForce,
        dptrPosMass,
        p->softeningSquared,
        p->i,
        p->n,
        p->N );
    // NOTE: synchronous memcpy, so no need for further
    // synchronization with device
    CUDART_CHECK( cudaMemcpy(
        p->hostForce+3*p->i,
        dptrForce,
        3*p->n*sizeof(float),
        cudaMemcpyDeviceToHost ) );

Error:
    cudaFree( dptrPosMass );
    cudaFree( dptrForce );
    p->status = status;
}

```

代码清单9-7 主机代码(应用线程)

```
float
ComputeGravitation_multiGPU_threaded(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N
)
{
    chTimerTimestamp start, end;
    chTimerGetTime( &start );
    {
        gpuDelegation *pgpu = new gpuDelegation[g_numGPUs];
        size_t bodiesPerGPU = N / g_numGPUs;
        if ( N % g_numGPUs ) {
            return 0.0f;
        }

        size_t i;
        for ( i = 0; i < g_numGPUs; i++ ) {
            pgpu[i].hostPosMass = g_hostAOS_PosMass;
            pgpu[i].hostForce = g_hostAOS_Force;

            pgpu[i].softeningSquared = softeningSquared;
            pgpu[i].i = bodiesPerGPU*i;
            pgpu[i].n = bodiesPerGPU;
            pgpu[i].N = N;

            g_GPUThreadPool[i].delegateAsynchronous(
                gpuWorkerThread,
                &pgpu[i] );
        }
        workerThread::waitForAll( g_GPUThreadPool, g_numGPUs );
        delete[] pgpu;
    }

    chTimerGetTime( &end );
    return chTimerElapsedTime( &start, &end ) * 1000.0f;
}
```

纹理操作

10.1 简介

在 CUDA 这个通用并行计算框架中，并不需要纹理的支持。鉴于其图形加速的传统功能，显卡上一般都配有纹理硬件。然而，在英伟达看来，支持硬件纹理单元将有助于许多操作的加速。尽管许多 CUDA 程序并没有使用纹理，但有时仍需依靠其使程序性能与 CPU 代码抗衡。

纹理映射是一种“将图片‘粉刷’到几何图形上从而创造更加丰富、看上去更加真实的物体的”技术。过去，硬件通常将纹理坐标与需要绘制的三角形 X 、 Y 、 Z 空间坐标绑定，然后以插值的方式将纹理“粉刷”到三角形面上（可选择双线性插值）。对于每个输出像素，从纹理图片中获取得到的对应值可以通过与插值得到的阴影参数进行混合，最终将混合得到的值输出到缓冲区中。随着可编程图形学的引入以及一些不包含颜色数据的类纹理数据的使用（如凹凸贴图），图形硬件变得愈加复杂。着色器程序通过使用 TEX 类似的指令来指定获取的坐标，从而使原本用来产生输出像素的着色器融入了计算功能。硬件通过使用纹理缓存以及针对局部空间而做出的内存结构优化，进一步提高了性能，并且通过特有硬件流水线将纹理坐标转换成硬件地址。

随着功能的日益丰富，程序的需求也结合了一些硬件成本的考虑，许多纹理操作特性显得并不是很灵活。例如，环绕纹理与镜面纹理的寻址模式必须在纹理坐标归一化的情况下才有效。本章对 CUDA 支持的硬件纹理单元进行了详细介绍，内容将涵盖归一化与非归一化坐标的寻址模式，线性插值的局限性，一维、二维、三维纹理和层纹理以及如何通过 CUDA 运行时 API 以及驱动程序 API 来使用这些特性。

两种使用方式

在 CUDA 中，纹理的使用有两种明显不同的方式。一种是只将纹理作为读取方式，以满足合并读取的约束或使用纹理缓存减少外部带宽需求。另一种则是利用 GPU 硬件在图形程序中具有固定功能的优势。硬件纹理单元具有一个可调控计算步骤的流水线，具体可以做的操作如下：

- 对归一化的纹理坐标进行缩放；
- 对纹理坐标执行边界条件计算；
- 依据二维或三维局部性将纹理坐标转化为地址；
- 为一维、二维或三维纹理获取 2、4 或 8 个纹理元素，并对它们进行线性插值；
- 将纹理值从整型值转化为成组的浮点值。

纹理是通过绑定到底层内存的（CUDA 数组或设备内存）纹理引用读取的。内存仅是一系列无格式的比特位。只有通过纹理引用，硬件才能正确解读数据，并在执行 TEX 指令时将数据传递到寄存器。

10.2 纹理内存

在介绍具有固定功能纹理硬件的特性之前，本章将先花一定时间介绍纹理引用将要绑定的底层内存。CUDA 可以利用设备内存或 CUDA 数组存储纹理。

10.2.1 设备内存

在设备内存中，纹理主要按行的形式寻址。图 10-1 显示了一张 1024×768 的纹理，其中 offset 是指从图像的头指针开始计算偏移（以元素为单位计数）。

$$\text{Offset} = Y \times \text{width} + X$$

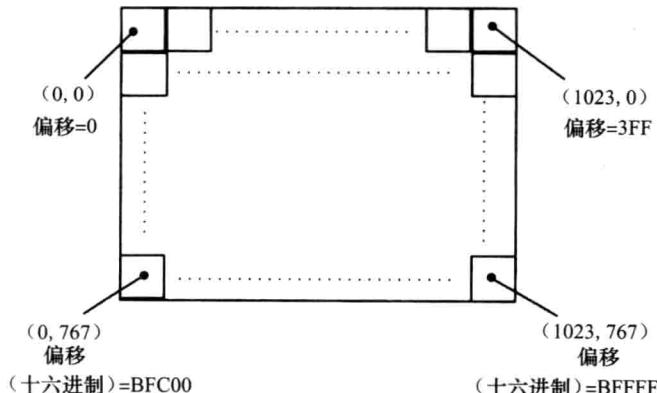


图 10-1 1024×768 的图像

如果要计算字节偏移，则需要乘以每个元素的字节大小。

$$\text{Byte offset} = \text{sizeof}[T] \times [Y \times \text{width} + X]$$

事实上，这种地址计算只适用于多数特殊的纹理宽度，例如 1024，因为 1024 为 2 的幂次，符合所有形式的对齐约束限制。为了通用化，不单单只针对特殊的纹理大小，CUDA 实现了等步长线性（pitch-linear）的寻址，纹理内存的宽度与纹理的实际宽度并不相同。针对各种非特殊的宽度，硬件将强制使用对齐限制，使得在纹理内存中，每个元素的访问将不再按照实际纹理中每个元素的宽度标准进行。例如一张宽度为 950 的纹理，如果对齐限制是 64 字节，则按字节计算的宽度需要填充到 960^①（填满下一个 64 字节）。此纹理如图 10-2 所示。

在 CUDA 中，通过填充得到的按字节计算的宽度叫做步长（pitch）。设备内存一共将使用 960×768 个元素的存储大小。此时，图像的偏移将以字节为单位计算，如下所示。

$$\text{Byte offset} = Y \times \text{Pitch} + X \text{ in Bytes}$$

应用程序可以通过调用 `cudaMallocPitch()` 或 `cuMemAllocPitch()` 委托 CUDA 驱动选择步长的大小^②。在三维纹理下，Depth 个二维切片数据连续分布在设备内存中，因此对应特定 Depth 的等步长线性图像与二维图像相似。

10.2.2 CUDA 数组与块的线性寻址

CUDA 数组是专为支持纹理操作而设计的，它的分配与设备内存一样来自物理内存池。CUDA 数组的结构对程序员而言是不透明的且不能通过指针访问。只有通过数组句柄以及一组一维、二维或三维坐标访问 CUDA 数组在内存中的位置。

CUDA 数组的寻址计算非常复杂，这样设计主要是为了让连续的地址能很好地显示二维或三维局部性。寻址计算是依赖于具体硬件的，不同代的硬件寻址方式有所不同。图 10-1 显示了其中的一种机制，在执行地址计算之前，行与列地址的低两位比特呈交错的形式。

由图 10-3 可以看出，比特交错能够使连续的地址拥有“维度局部性”，即一条缓存线上存储的是相邻的一整块像素值，而不是水平方向的所有像素^③。然而，这种方式也具有一定缺点，即在使用纹理维度时需要对纹理的维度强加一些条件，很不方便，因此，比特交错只是几种所谓的“块线性”寻址计算策略中的一种。

^① 这里应为 960，为 64 的整倍数，原文误为 964。——译者注

^② 由于根据文档对齐限制执行的存储分配操作容易产生变更，使用委托驱动程序的代码则更能适应未来硬件。

^③ 三维纹理 X、Y 以及 Z 轴的比特交错模式相似。

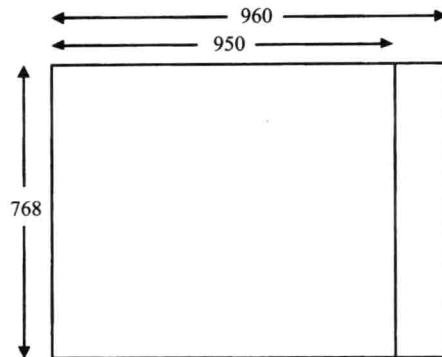


图 10-2 950 × 768 的纹理与等步长处理

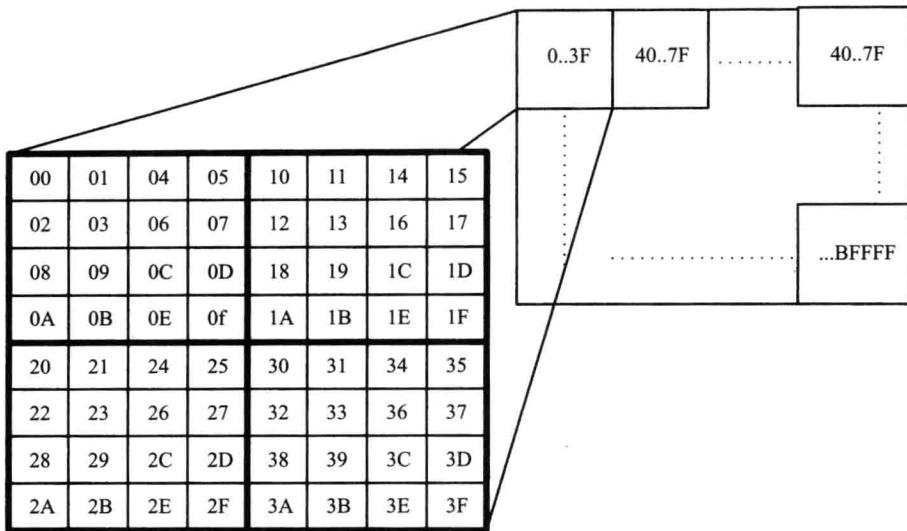


图 10-3 1024×768 图片中的交错比特

在设备内存中，一个图像元素的位置可以用以下几种方式表示：

- 基地址指针、步长、以及元组 (XInBytes,Y) 或元组 (XInBytes,Y,Z)。
- 基地址指针以及通过公式 10-1 计算出的偏移。
- 已包含偏移的设备指针。

相反，当 CUDA 数组不具备设备内存地址的时候，内存地址必须以 CUDA 数组以及一个元组 (XInBytes,Y) 或元组 (XInBytes,Y,Z) 的形式指定。

1. 创建和销毁 CUDA 数组

使用 CUDA 运行时，调用 `cudaMallocArray()` 可以创建 CUDA 数组。

```
cudaError_t cudaMallocArray(struct cudaArray **array, const struct
cudaChannelFormatDesc *desc, size_t width, size_t height __dv(0),
unsigned int flags __dv(0));
```

参数 `array` 传递回数组的句柄，`desc` 指定了每个数组元素中成分的数目以及类型（如两个浮点数），`width` 指定了数组采用字节计数的宽度，`height` 是一个可选参数，其指定了数组的高度，如果 `height` 没有被指定，`cudaMallocArray()` 将创建一个一维的 CUDA 数组。

参数 `flags` 用来表示 CUDA 数组的使用方式。当 CUDA 数组被指定用作表面读写操作时，该参数必须且只能为“`cudaArraySurfaceLoadStore`”。关于表面读写操作，之后的章节将具体介绍。

关于参数 `height` 与 `flags` 后面使用的宏 `__dv`，根据不同的语言，将发挥不同的作用。当

使用 C 编译器编译时，其只是一个简单的参数，当如果使用 C++ 编译器时，其表示一个具有指定默认值的参数。

结构体 `cudaChannelFormatDesc` 描述了一个纹理的具体内容。

```
struct cudaChannelFormatDesc {
    int x, y, z, w;
    enum cudaChannelFormatKind f;
};
```

结构体中的成员 `x`、`y`、`z` 以及 `w` 指定了纹理元素中每个成员的比特数。例如，仅包含一个浮点元素的纹理对应的 `x` 为 32，其他成员的值为 0。结构体 `cudaChannelFormatKind` 指明了该数据的类型，是带符号的整数，还是无符号整型值，或者浮点数。

```
enum cudaChannelFormatKind
{
    cudaChannelFormatKindSigned = 0,
    cudaChannelFormatKindUnsigned = 1,
    cudaChannelFormatKindFloat = 2,
    cudaChannelFormatKindNone = 3
};
```

开发人员可以通过调用函数 `CreateChannelDesc` 创建 `cudaChannnelFormatDesc` 结构体。

```
cudaChannelFormatDesc cudaCreateChannelDesc(int x, int y, int z, int w,
                                             cudaChannelFormatKind kind);
```

或者，也可以调用此类函数的模板函数，如下所示：

```
template<class T> cudaCreateChannelDesc<T>();
```

其中 `T` 可以是 CUDA 支持的任何本地格式（native format）。以下是两个使用该模板的例子。

```
template<> __inline__ __host__ cudaChannelFormatDesc
cudaCreateChannelDesc<float>(void)
{
    int e = (int)sizeof(float) * 8;

    return cudaCreateChannelDesc(e, 0, 0, 0, cudaChannelFormatKindFloat);
}

template<> __inline__ __host__ cudaChannelFormatDesc
cudaCreateChannelDesc<uint2>(void)
{
    int e = (int)sizeof(unsigned int) * 8;

    return cudaCreateChannelDesc(e, e, 0, 0,
                               cudaChannelFormatKindUnsigned);
}
```



当使用 `char` 类型的数据时，部分编译器认定 `char` 是有符号的，而有些编译器则认为 `char` 是无符号的。为了避免混淆，最好每次使用前加上 `signed` 关键字。

三维 CUDA 数组可以通过调用 `cudaMalloc3DArray()` 分配。

```
cudaError_t cudaMalloc3DArray(struct cudaArray** array, const struct
cudaChannelFormatDesc* desc, struct cudaExtent extent, unsigned int
flags __dv(0));
```

`cudaMalloc3DArray()` 接受一个名为 `cudaExtent` 的结构体，该结构体包含了 `width`、`height` 以及 `depth` 三个成员。

```
struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
```

参数 `flags` 与 `cudaMallocArray()` 中的相同，当 CUDA 数组用来进行表面读写操作时，该参数的值必须为“`cudaArraySurfaceLoadStore`”。



注意 针对数组的处理方式，CUDA 运行时 API 与驱动程序 API 是相互兼容的。通过 `cudaMallocArray()` 传递回的指针可以强制转换成 `CUarray`，然后传递给类似 `cuArrayGetDescriptor()` 的驱动程序 API。

2. 驱动程序 API

与 `cudaMallocArray()` 以及 `cudaMalloc3DArray()` 等价的驱动程序 API 分别是 `cuArrayCreate()` 与 `cuArray3DCreate()`。

```
CUresult cuArrayCreate(CUarray *pHandle, const CUDA_ARRAY_DESCRIPTOR
*pAllocateArray);
CUresult cuArray3DCreate(CUarray *pHandle, const CUDA_ARRAY3D_
DESCRIPTOR *pAllocateArray);
```

`cuArray3DCreate()` 可以通过指定 `height` 或 `depth` 的值为 0 来相应创建一维或二维 CUDA 数组。结构体 `CUDA_ARRAY3D_DESCRIPTOR` 的定义如下：

```
typedef struct CUDA_ARRAY3D_DESCRIPTOR_st
{
    size_t Width;
    size_t Height;
    size_t Depth;
    CUarray_format Format;
    unsigned int NumChannels;
    unsigned int Flags;
} CUDA_ARRAY3D_DESCRIPTOR;
```

成员 `Format` 与 `NumChannels` 共同描述了 CUDA 数组每个元素的大小：`NumChannnels` 的值可以为 1、2 或 4，`Format` 指定了数组元素每个通道的类型，具体如下：

```

typedef enum CUarray_format_enum {
    CU_AD_FORMAT_UNSIGNED_INT8 = 0x01,
    CU_AD_FORMAT_UNSIGNED_INT16 = 0x02,
    CU_AD_FORMAT_UNSIGNED_INT32 = 0x03,
    CU_AD_FORMAT_SIGNED_INT8 = 0x08,
    CU_AD_FORMAT_SIGNED_INT16 = 0x09,
    CU_AD_FORMAT_SIGNED_INT32 = 0x0a,
    CU_AD_FORMAT_HALF = 0x10,
    CU_AD_FORMAT_FLOAT = 0x20
} CUarray_format;

```



注意 CUDA_ARRAY3D_DESCRIPTOR 中指定的格式仅仅是指定 CUDA 数组中数据个数的一种便捷方式。只要每个元素的字节数相同，绑定到 CUDA 数组的纹理可以指定不同的格式。例如，可以将一个 `texture<int>` 绑定到一个包含 4 个分量、每个分量一个字节的 CUDA 数组（每个元素 32 位）。

有时候，CUDA 数组的句柄会传递给一些子程序来查询该数组的维度或格式。`cuArray3DGetDescriptor()` 就提供了该功能。

```
CUresult cuArray3DGetDescriptor(CUDA_ARRAY3D_DESCRIPTOR
    *pArrayDescriptor, CUarray hArray);
```

注意！该函数可以使用一维以及二维数组调用，甚至可以使用那些由 `cuArrayCreate()` 创建的数组调用。

10.2.3 设备内存与 CUDA 数组对比

对于那些稀疏的访存方式，尤其是具有维度局部性的访存方式的应用程序（例如计算机视觉领域的应用），使用 CUDA 数组明显更加合适。而对于那些正常访存的应用程序，尤其是那些数据复用率几乎为零，或者应用程序的数据复用可以使用共享内存来处理的情况，设备内存明显是更明智的选择。

而对于类似图像处理的部分应用程序，选择使用设备内存还是使用 CUDA 数组则显得并不那么一目了然。若其他条件都相同，选择设备内存可能比 CUDA 数组更适合。以下几条考虑因素可以帮助我们做出恰当的选择。

- CUDA 3.2 之前，CUDA 内核无法对 CUDA 数组进行写操作，程序员只能通过纹理内置指令（intrinsics）从 CUDA 数组中读取数据。CUDA 3.2 为费米架构的硬件添加了通过“表面读写”内置指令访问二维 CUDA 数组的能力。
- CUDA 数组不会消耗任何 CUDA 地址空间。
- 在 WDDM 驱动程序（Windows Vista 及其之后版本系统）上，系统能够自动管理 CUDA 数组的驻存。根据当前正在执行的 CUDA 内核是否需要使用 CUDA 数组，CUDA 数组可以从设备内存中换进换出，而且该操作对程序员而言是透明的。相反，

WDDM 则要求所有设备内存常驻以便于任一内核执行。

- CUDA 数组只能驻存在设备内存中，如果 GPU 具有复制引擎，通过总线传输数据时，数据会在两种表现形式间切换。对于部分程序而言，在主机内存保持以等步长的表现形式，而在设备内存保持以 CUDA 数组的表现形式是最好的一种选择。

10.3 一维纹理操作

为了便于说明，本节将详细介绍一维纹理，然后扩展到二维及三维纹理的讨论。

纹理设置

纹理中的数据可以由 1 个、2 个或 4 个元素组成，元素的类型可以是：

- 有符号或无符号的 8 位、16 位或 32 位整型值；
- 16 位的浮点型数值；
- 32 位的浮点型数值。

在后缀名为 .cu 的文件中（无论使用的是 CUDA 运行时还是驱动程序 API），纹理引用的声明如下所示：

```
texture<ReturnType, Dimension, ReadMode> Name;
```

其中 ReturnType 为纹理内置指令的返回值；Dimension 的值能取 1、2 或 3，分别表示一维、二维或三维；ReadMode 是一个可选参数，默认值为 cudaReadModeElementType。读取模式仅会对整型值的纹理数据产生影响。默认情况下，当纹理数据是整型值时，纹理将传回整型值，必要时，还会将其转化成 32 位的整型值。然而，当读取模式指定为 cudaReadModeNormalizedFloat 时，8 位或 16 位的整型值将转化为范围在 0.0 ~ 1.0 之间的浮点值，具体转化公式如表 10-1 所示。

表 10-1 纹理中浮点值的转化

声明格式	转化为浮点数的公式
char c	$\begin{cases} -1.0, c == 0x80 \\ c/127.0, \text{其他} \end{cases}$
short s	$\begin{cases} -1.0, s == 0x8000 \\ \frac{s}{32767.0}, \text{其他} \end{cases}$
unsigned char uc	$uc/255.0$
unsigned short us	$us/65535.0$

此类转化操作的 C 代码如代码清单 10-1 所示。

代码清单 10-1 纹理单元的浮点转换

```

float
TexPromoteToFloat( signed char c )
{
    if ( c == (signed char) 0x80 ) {
        return -1.0f;
    }
    return (float) c / 127.0f;
}

float
TexPromoteToFloat( short s )
{
    if ( s == (short) 0x8000 ) {
        return -1.0f;
    }
    return (float) s / 32767.0f;
}

float
TexPromoteToFloat( unsigned char uc )
{
    return (float) uc / 255.0f;
}

float
TexPromoteToFloat( unsigned short us )
{
    return (float) us / 65535.0f;
}

```

一旦声明纹理引用，就可以通过调用纹理内置指令在内核中对其调用。不同类型的纹理将使用不同内置指令，如表 10-2 所示。

表 10-2 纹理指令

纹理类型	纹理内置指令
线性设备内存	<code>tex1Dfetch(int index);</code>
一维 CUDA 数组	<code>tex1D(float x);</code>
二维 CUDA 数组 二维设备内存	<code>tex2D(float x, float y);</code>
三维 CUDA 数组	<code>tex3D(float x, float y, float z);</code>
一维分层纹理	<code>tex1DLayered(float x, int layer);</code>
二维分层纹理	<code>tex2DLayered(float x, float y, int layer);</code>

纹理引用类似于全局变量，具有文件范围的作用域。它们不能以参数的形式创建、销毁或传递，因此，如果要将其封装到上层抽象结构中，必须格外小心。

1. CUDA 运行时

在启动使用纹理的内核之前，必须调用 `cudaBindTexture()`、`cudaBindTexture2D()` 或 `cudaBindTextureToArray()` 将纹理绑定到 CUDA 数组或设备内存。由于 CUDA 运行时的语言

集成特性，开发人员可以通过名字引用纹理，如下所示

```
texture<float, 2, cudaMemcpyType> tex;
...
CUDART_CHECK(cudaBindTextureToArray(tex, texArray));
```

绑定纹理之后，只要纹理绑定没有改变，内核就可以使用纹理引用读取对应绑定内存里的数据。

2. 驱动程序 API

若一个纹理在一个.cu 后缀文件中声明，驱动程序必须使用 cuModuleGetTexRef() 对其进行查询。使用驱动程序 API 时，纹理的固定属性必须显式设置，且必须符合编译器生成代码时的假定条件。对于绝大多数纹理而言，这仅仅意味着纹理格式必须与 .cu 文件中声明的纹理格式一致；但也有一些例外，比如设置纹理将整型值或 16 位的浮点值增强为归一化的 32 位浮点值时。

函数 cuTexRefSetFormat 负责指定纹理中数据的格式。

```
CUresult CUDA API cuTexRefSetFormat(CUtexref hTexRef, CUarray_format
fmt, int NumPackedComponents);
```

数组格式如下所示。

枚举值	类 型
CU_AD_FORMAT_UNSIGNED_INT8	unsigned char
CU_AD_FORMAT_UNSIGNED_INT16	unsigned short
CU_AD_FORMAT_UNSIGNED_INT32	unsigned int
CU_AD_FORMAT_SIGNED_INT8	signed char
CU_AD_FORMAT_SIGNED_INT16	short
CU_AD_FORMAT_SIGNED_INT32	int
CU_AD_FORMAT_SIGNED_HALF	half (IEEE 754 “binary16” format)
CU_AD_FORMAT_SIGNED_FLOAT	float

NumPackedComponents 指定了每个纹理元素中分量的数目。该值可以为 1、2 或 4。16 位的浮点数 (half) 是一种特殊的数据类型，很适合用来高效保真地表示图像数据。[⊖] 尾数部分使用 10 位（实际上归一化的数据的尾数精度为 11 位）已可以精确表示大多数传感器生成的数据，指数部分使用 5 位也足够表示同一图像中星光以及日光亮度的动态范围。大多数浮点数架构都不包含处理 16 位浮点数的原生指令，CUDA 也不例外。硬件纹理单元可以自动将 16 位的浮点数转换为 32 位的浮点数，或者在 CUDA 内核中，使用 __float2half_rn 和 __half2float_rn 指令将数据在 16 位与 32 位浮点数之间相互转换。

[⊖] 8.3.4 节详细介绍了 16 位的浮点数。

10.4 纹理作为数据读取方式

当使用纹理作为读取数据的方式时，是为了利用硬件纹理单元避免繁锁的合并读取约束或利用纹理缓存，而不是为了利用线性插值等硬件特性。在这一方式下，许多纹理特性是不可用的。此类使用纹理的方式需注意如下要点：

- 使用 `cudaBindTexture()` 或 `cuTexRefSetAddress()` 将纹理引用绑定到设备内存上；
- 必须使用 `tex1Dfetch()` 指令，其接受一个 27 位的整型索引；[⊖]
- `tex1Dfetch()` 具有将纹理内容转换为浮点型数值的可选项。整型值将转换成范围在 0.0 ~ 1.0 之间的浮点值，16 位浮点值将增强为标准 `float` 值。

使用 `tex1Dfetch()` 读取设备内存的好处是双重的。首先，通过纹理读取内存不需要遵循针对读取全局内存的合并读取约束。其次，纹理缓存可以成为其他硬件资源甚至费米架构硬件上的二级缓存的有用补充，当传递给 `tex1Dfetch()` 的索引值越界，其返回值为 0。

10.4.1 增加有效地址范围

由于 27 位的索引指定了待读取的纹理元素，而每个纹理元素最大为 16 字节，因此使用 `tex1Dfetch()` 读取的纹理最大能覆盖 31 位 ($2^{27}+2^4$) 的内存。一个有效增加纹理覆盖到的数据量的方法是使用比数据实际大小更宽的纹理元素。例如，应用程序可以使用 `float4` 的纹理代替 `float`，然后根据需要访问元素索引的最低的若干有效位，从 `float4` 中选择相应的元素。对于整型数，也可以使用该技术，尤其是对全局内存中的 8 位或 16 位的数据进行访问（因为它们的访问无法合并）。另一种方式，应用程序可以对设备内存中不同分段上的多个纹理设置别名，并对需要的纹理进行断定读取，以此种方式每次只有其中之一处于“活跃”状态。

程序演示：`tex1dfetch_big.cu`

该程序展示了如何使用 `tex1Dfetch()` 从大数组中读取包含多分量的单个纹理或多个纹理。该程序通过以下代码进行调用。

```
tex1dfetch_big <NumMegabytes>
```

应用程序分配了指定 MB 大小的设备内存（若设备内存分配失败，将分配映射锁页主机内存），然后使用随机数对这块内存进行填充，接着使用单分量、双分量或四分量的纹理在这块数据区上计算校验和。每次最多可使用 4 个元素为 `int4` 类型的纹理，即应用程序最大可以在 8192MB 的内存上进行纹理处理。

为简单清晰起见，`tex1dfetch_big.cu` 并没有执行任何精心设计的并行归约技术。每个线程只是写回一个局部的中间和，最终的校验和计算是在 CPU 端执行的。以下是程序定义 27 位的硬件限制的代码。

[⊖] 所有支持 CUDA 的硬件均是 27 位的限制，因此目前没有查询当前设备上该限制的方法。

```
#define CUDA_LG_MAX_TEX1DFETCH_INDEX 27
#define CUDA_MAX_TEX1DFETCH_INDEX
((size_t)1<<CUDA_LG_MAX_TEX1DFETCH_INDEX)-1)
```

另外程序还定义了 4 个元素类型为 int4 的纹理。

```
texture<int4, 1, cudaReadModeElementType> tex4_0;
texture<int4, 1, cudaReadModeElementType> tex4_1;
texture<int4, 1, cudaReadModeElementType> tex4_2;
texture<int4, 1, cudaReadModeElementType> tex4_3;
```

设备函数 tex4Fetch() 接受一个索引值，并将其分离成一个纹理序号和一个 27 位的索引值，传递给 tex1Dfetch()。

```
device int4
tex4Fetch( size_t index )
{
    int texID = (int) (index>>CUDA_LG_MAX_TEX1DFETCH_INDEX);
    int i = (int) (index & (CUDA_MAX_TEX1DFETCH_INDEX_SIZE_T-1));
    int4 i4;

    if ( texID == 0 ) {
        i4 = tex1Dfetch( tex4_0, i );
    }
    else if ( texID == 1 ) {
        i4 = tex1Dfetch( tex4_1, i );
    }
    else if ( texID == 2 ) {
        i4 = tex1Dfetch( tex4_2, i );
    }
    else if ( texID == 3 ) {
        i4 = tex1Dfetch( tex4_3, i );
    }
    return i4;
}
```

该设备函数将编译成一小段使用了 4 个分支断定 TEX 指令的代码，其中 4 个纹理只有一个处于活跃状态。如果想要随机访问，应用程序也可以利用分支断定技术从返回的 int4 的 4 个分量 .x、.y、.z 或 .w 中选择。

代码清单 10-2 中所示的绑定纹理操作，用到了少量的技巧。这段代码创建了两个小数组 texSizes[] 和 texBases[]，并设定了它们能够覆盖到的设备内存的整个范围。无论需要映射到设备内存的纹理数目小于 4 还是等于 4，“for 循环”保证了所有纹理都进行了有效绑定。

代码清单 10-2 tex1Dfetch_big.cu (节选)

```
int iTexture;
cudaChannelFormatDesc int4Desc = cudaCreateChannelDesc<int4>();
size_t numInt4s = numBytes / sizeof(int4);
int numTextures = (numInt4s+CUDA_MAX_TEX1DFETCH_INDEX)>>
    CUDA_LG_MAX_TEX1DFETCH_INDEX;
size_t Remainder = numBytes & (CUDA_MAX_BYTES_INT4-1);
if ( ! Remainder ) {
    Remainder = CUDA_MAX_BYTES_INT4;
}

size_t texSizes[4];
char *texBases[4];
```

```

for ( iTexture = 0; iTexture < numTextures; iTexture++ ) {
    texBases[iTexture] = deviceTex+iTexture*CUDA_MAX_BYTES_INT4;
    texSizes[iTexture] = CUDA_MAX_BYTES_INT4;
}
texSizes[iTexture-1] = Remainder;
while ( iTexture < 4 ) {
    texBases[iTexture] = texBases[iTexture-1];
    texSizes[iTexture] = texSizes[iTexture-1];
    iTexture++;
}
cudaBindTexture( NULL, tex4_0, texBases[0], int4Desc, texSizes[0] );
cudaBindTexture( NULL, tex4_1, texBases[1], int4Desc, texSizes[1] );
cudaBindTexture( NULL, tex4_2, texBases[2], int4Desc, texSizes[2] );
cudaBindTexture( NULL, tex4_3, texBases[3], int4Desc, texSizes[3] );

```

当编译并运行之后，可以设定不同大小，调用程序以观察效果。在亚马逊 EC2 云计算环境的 CG1 实体上运行，分别以 512M、768M、1280M 以及 8192M 大小调用内核时的运行结果如下：

```

$ ./texldfetch_big 512
Expected checksum: 0x7b7c8cd3
tex1 checksum: 0x7b7c8cd3
tex2 checksum: 0x7b7c8cd3
tex4 checksum: 0x7b7c8cd3
$ ./texldfetch_big 768
Expected checksum: 0x559a1431
tex1 checksum: (not performed)
tex2 checksum: 0x559a1431
tex4 checksum: 0x559a1431
$ ./texldfetch_big 1280
Expected checksum: 0x66a4f9d9
tex1 checksum: (not performed)
tex2 checksum: (not performed)
tex4 checksum: 0x66a4f9d9
$ ./texldfetch_big 8192
Device alloc of 8192 Mb failed, trying mapped host memory
Expected checksum: 0xf049c607
tex1 checksum: (not performed)
tex2 checksum: (not performed)
tex4 checksum: 0xf049c607

```

每个 int4 类型的纹理“只能”读取 2GB 的内存，因此，当启动程序时，大小超过 8192MB 时将失败。该应用程序强调了对带索引纹理的需要，此时待读取纹理可以指定为运行时的参数，然而，CUDA 并没有提供该特性的支持。

10.4.2 主机内存纹理操作

此外，若将纹理作为一种读取方式使用，应用程序也可以通过分配映射锁页内存，获取设备指针，然后把该设备指针传给 cudaBindAddress() 或 cuTexRefSetAddress() 的方式从主机内存读取数据。虽然这种方式可行，但以纹理的形式读取主机内存的速度是很慢的。特斯拉架构的硬件设备可以以每秒 2GB 的速度从 PCIe 总线上获取纹理数据，费米架构的硬件设备的速度则要相对慢很多。因此，若要采用这种方式，最好有一定理由，例如，保持代码的简洁性。

程序演示：tex1dfetch_int2float.cu

这段代码将纹理作为一种读取方式使用，从主机内存获取数据，以确定 TexPromoteToFloat() 函数是否正常工作。即将使用的这个 CUDA 内核功能很简单，其实现了一个相当于无阻塞能力的 memcpy 函数，即从纹理中获取数据并写入到设备内存。

```
texture<signed char, 1, cudaMemcpyHostToDevice> tex;

extern "C" __global__ void
TexReadout( float *out, size_t N )
{
    for ( size_t i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*blockDim.x )
    {
        out[i] = tex1Dfetch( tex, i );
    }
}
```

由于将整型值增强至浮点型数值只对 8 位及 16 位的整型值有效，我们可以通过分配一小块缓冲区，从这块区域获取纹理数据，并判定输出是否我们的预期，以此测试每个转换是否正确。代码清单 10-3 显示了从 tex1dfetch_int2float.cu 中节选的部分代码。该代码分配了两块主机内存，其中 inHost 保存了输入缓冲区中的 256 或 65536 个输入值，fOutHost 保存了对应的浮点数输出。对应这些映射锁页主机内存的设备指针分别保存到了 inDevice 和 foutDevice 中。

代码清单 10-3 tex1d_int2float.cu (节选)

```
template<class T>
void
CheckTexPromoteToFloat( size_t N )
{
    T *inHost, *inDevice;
    float *foutHost, *foutDevice;
    cudaError_t status;

    CUDART_CHECK(cudaHostAlloc( (void **) &inHost,
                               N*sizeof(T),
                               cudaHostAllocMapped));
    CUDART_CHECK(cudaHostGetDevicePointer( (void **) &inDevice,
                                         inHost,
                                         0 ));
    CUDART_CHECK(cudaHostAlloc( (void **) &foutHost,
                               N*sizeof(float),
                               cudaHostAllocMapped));
    CUDART_CHECK(cudaHostGetDevicePointer( (void **) &foutDevice,
                                         foutHost,
                                         0 ));

    for ( int i = 0; i < N; i++ ) {
        inHost[i] = (T) i;
    }
    memset( foutHost, 0, N*sizeof(float) );

    CUDART_CHECK( cudaBindTexture( NULL,
                                  tex,
```

```

        inDevice,
        cudaCreateChannelDesc<T>(),
        N*sizeof(T)));
TexReadout<<<2,384>>>( foutDevice, N );
CUDART_CHECK(cudaDeviceSynchronize());

for ( int i = 0; i < N; i++ ) {
    printf( "% .2f ", foutHost[i] );
    assert( foutHost[i] == TexPromoteToFloat( (T) i ) );
}
printf( "\n" );
Error:
cudaFreeHost( inHost );
cudaFreeHost( foutHost );
}

```

输入值将被初始化为当前测试类型的任意值，然后调用 `cudaBindTexture()` 将纹理引用与输入对应的设备指针进行绑定。接着启动 `TexReadOut()` 内核，读取输入纹理的每一个值，并将 `tex1Dfetch()` 的返回值作为输出写到设备内存。这样，输入缓冲区与输出缓冲区均驻存在映射主机内存上了。由于内核是直接将结果写回到主机内存，因此需调用 `cudaDeviceSynchronize()` 以确保 CPU 与 GPU 之间不存在竞争条件。函数末尾调用了 `TexPromoteToFloat()`，将当前测试类型的值转换成浮点值，并与对应的内核返回值进行比较，看是否相等。若所有的比较都相等，函数则会返回 `true`，若有任何 API 调用失败或比较不相等，则返回 `false`。

10.5 使用非归一化坐标的纹理操作

除了 `tex1Dfetch()`，其他所有的纹理指令均使用浮点值指定纹理中的坐标位置。当使用非归一化的纹理坐标时，坐标的范围将为 $[0, \text{MaxDim})$ ，其中 `MaxDim` 表示纹理的宽度、高度或深度。非归一化的坐标是一种按符合人类直觉的方式对纹理进行索引的方式，使用该坐标时许多纹理特性将变得不可用。

研究纹理操作行为的一种简单方式就是将纹理中的每个元素填充为该纹理的索引值。图 10-4 显示的一个浮点值类型且只有 16 个元素的一维纹理。图中使用标识元素填充每个纹理，图的下方还注释了部分使用 `tex1D()` 得到的返回值。

尽管使用非归一化的纹理坐标时，会导致部分纹理操作特性不可用，但若结合线性过滤以及有限形式的纹理寻址模式，也可使用这些纹理特性。纹理寻址模式指明了硬件如何处理超出范围的纹理坐标的方式。图 10-4 显示了：针对于非归一化的坐标，在从纹理获取数据之前，将范围夹取在 $[0, \text{MaxDim})$ 之间的默认纹理寻址模式。图中值 16.0 超出了范围，通过夹取，获取的值为 15.0。当使用非归一化的坐标时，还可以使用另一种称为边缘寻址模式的纹理寻址选项，即超出范围的坐标对应的返回值为零。

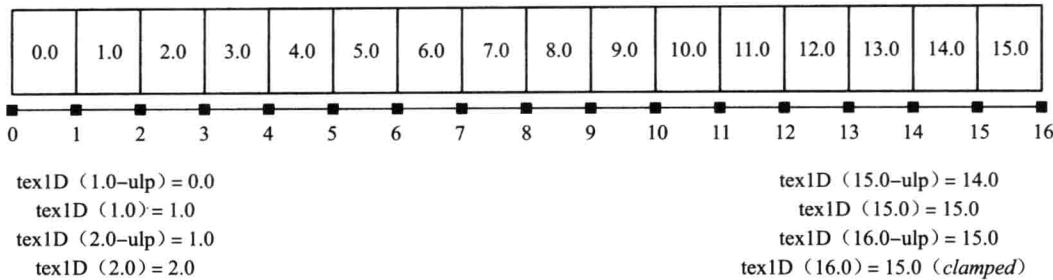


图 10-4 使用非归一化坐标进行纹理操作（不包含线性过滤）

过滤模式默认使用的是“点过滤”，其根据浮点坐标值返回一个纹理元素。相反，线性过滤将使纹理硬件获取相邻的两个纹理元素，并在它们之间根据纹理坐标作为权重进行线性插值。图 10-5 显示了含 16 个元素的一维纹理，附有返回自 tex1D() 的若干样例值。注意，纹理坐标必须加上 0.5 才能获得标识元素。

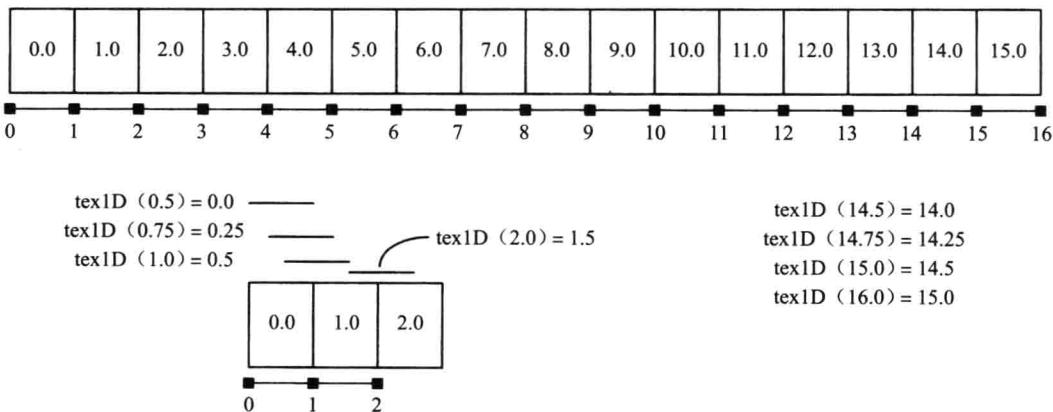


图 10-5 使用非归一化纹理坐标的纹理操作（线性过滤模式）

许多纹理操作特性可以相互结合使用，例如，线性过滤可以与之前讨论的整型增强为浮点型的操作结合使用。若这样使用，指令 tex1D() 将返回根据相邻两个增强的浮点纹理元素进行插值之后的精确结果。

程序演示：tex1d_unnormalized.cu

程序 tex1d_unnormalized.cu 就像一个显微镜，通过打印出纹理坐标以及对应 tex1D() 的返回值仔细的检视纹理操作。与程序 tex1dfetch_int3float.cu 不同，该程序的纹理数据使用的一维 CUDA 数组存储的。该程序根据指定的基地址与增量，在一定浮点数范围内执行了一系列的纹理读取操作，并将插值得到的值与 tex1D() 的返回值一起保存到 float2 类型的输出数组中。以下是该 CUDA 内核的源代码。

```

texture<float, 1> tex;

extern "C" __global__ void
TexReadout( float2 *out, size_t N, float base, float increment )
{
    for ( size_t i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += gridDim.x*blockDim.x )
    {
        float x = base + (float) i * increment;
        out[i].x = x;
        out[i].y = tex1D( tex, x );
    }
}

```

代码清单 10-4 列出了一个主机端函数 CreateAndPrintTex() 的代码，该函数接受将要创建的纹理的大小、执行纹理采样操作的次数、将传给 tex1D() 的浮点范围基地址和增量以及两个可选的纹理过滤和寻址方式等几个参数。函数创建了保存纹理数据的 CUDA 数组，然后根据调用者的需要，可以选择将该数组初始化为调用者提供的数据（若调用者传递的值为 NULL 则初始化为标识元素），接着函数将纹理绑定到 CUDA 数组上，最后打印 float2 类型的输出数组。

代码清单 10-4 CreateAndPrintTex()

```

CUDART_CHECK(cudaMemcpyToArray( texArray,
                               0, 0,
                               texContents,
                               texN*sizeof(T),
                               cudaMemcpyHostToDevice));
CUDART_CHECK(cudaBindTextureToArray(tex, texArray));

tex.filterMode = filterMode;
tex.addressMode[0] = addressMode;
CUDART_CHECK(cudaHostGetDevicePointer(&outDevice, outHost, 0));
TexReadout<<<2,384>>>( outDevice, outN, base, increment );
CUDART_CHECK(cudaThreadSynchronize());

for ( int i = 0; i < outN; i++ ) {
    printf( "%f, %f\n", outHost[i].x, outHost[i].y );
}
printf( "\n" );

Error:
if ( ! initTex ) free( texContents );
if ( texArray ) cudaFreeArray( texArray );
if ( outHost ) cudaFreeHost( outHost );
}

```

该程序的 main() 函数可以通过修改来帮助我们更好的理解纹理操作的行为。当前版本的 main() 函数创建了一个包含 8 个元素的纹理，并打印出 tex1D() 的输出值，范围从 0.0 ~ 7.0。

```

int
main( int argc, char *argv[] )
{
    cudaError_t status;
    CUDA_CHECK(cudaSetDeviceFlags(cudaDeviceMapHost));

    CreateAndPrintTex<float>( NULL, 8, 8, 0.0f, 1.0f );
    CreateAndPrintTex<float>( NULL, 8, 8, 0.0f, 1.0f,
    cudaFilterModeLinear );

    return 0;
}

```

该程序的输出结果如下所示：

```

(0.00, 0.00)      <- output from the first CreateAndPrintTex()
(1.00, 1.00)
(2.00, 2.00)
(3.00, 3.00)
(4.00, 4.00)
(5.00, 5.00)
(6.00, 6.00)
(7.00, 7.00)

(0.00, 0.00)      <- output from the second CreateAndPrintTex()
(1.00, 0.50)
(2.00, 1.50)
(3.00, 2.50)
(4.00, 3.50)
(5.00, 4.50)
(6.00, 5.50)
(7.00, 6.50)

```

如果将 main() 函数按以下方式修改函数 CreateAndPrintTex() 的调用方式，

```
CreateAndPrintTex<float>( NULL, 8, 20, 0.9f, 0.01f,
cudaFilterModePoint );
```

从输出的结果可以看出，当使用点过滤模式时，1.0 是 0 号纹理元素与 1 号纹理元素的分界线。

```
(0.90, 0.00)
(0.91, 0.00)
(0.92, 0.00)
(0.93, 0.00)
(0.94, 0.00)
(0.95, 0.00)
(0.96, 0.00)
(0.97, 0.00)
(0.98, 0.00)
(0.99, 0.00)
(1.00, 1.00)      <- transition point
(1.01, 1.00)
(1.02, 1.00)
(1.03, 1.00)
(1.04, 1.00)
(1.05, 1.00)
(1.06, 1.00)
(1.07, 1.00)
(1.08, 1.00)
(1.09, 1.00)
```

线性过滤的一个限制就是使用 9 位的权重因子。注意，插值的精度不是依赖于纹理元素的精度而是权重。以一个 10 元素的纹理为例，纹理元素初始化为归一化的标识元素，即用 (0.0,0.1,0.2,0.3…0.9) 代替 (0,1,2…9)，CreateAndPrintTex() 函数可以指定纹理内容，因此可以按照以下代码进行操作：

```
{
    float texData[10];
    for ( int i = 0; i < 10; i++ ) {
        texData[i] = (float) i / 10.0f;
    }
    CreateAndPrintTex<float>( texData, 10, 10, 0.0f, 1.0f );
}
```

调用未修改的 CreateAndPrintTex() 得到的输出，改变并不大。

```
(0.00, 0.00)
(1.00, 0.10)
(2.00, 0.20)
(3.00, 0.30)
(4.00, 0.40)
(5.00, 0.50)
(6.00, 0.60)
(7.00, 0.70)
(8.00, 0.80)
(9.00, 0.90)
```

若想让 CreateAndPrintTex() 在头两个元素（值为 0.1 和 0.2）之间进行线性插值，则需要改为如下所示的代码：

```
CreateAndPrintTex<float>( tex, 10, 10, 1.5f, 0.1f, cudaFilterModeLinear );
```

得到的输出结果如下所示：

```
(1.50, 0.10)
(1.60, 0.11)
(1.70, 0.12)
(1.80, 0.13)
(1.90, 0.14)
(2.00, 0.15)
(2.10, 0.16)
(2.20, 0.17)
(2.30, 0.18)
(2.40, 0.19)
```

通过截断为 2 位小数部分，得到的数据看起来表现很好。但若将 CreateAndPrintTex() 的输出改为十六进制，则输出结果将变为

```
(1.50, 0x3dcffffd)
(1.60, 0x3de1999a)
(1.70, 0x3df5999a)
(1.80, 0xe053333)
(1.90, 0xe0f3333)
(2.00, 0xe19999a)
(2.10, 0xe240000)
(2.20, 0xe2e0000)
(2.30, 0xe386667)
(2.40, 0xe426667)
```

很明显，大多数十进制小数无法准确描述为浮点数。不过，尽管执行插值操作时不需要过高的精度，这些值仍是以全精度（full precision）进行插值的。

程序演示：tex1d_9bit.cu

为了进一步了解精度问题，我们编写了另一个程序演示 tex1d_9bit.cu。该程序中，纹理将以 32 位的浮点数填充，每个浮点数必须以全精度才能正确表示。若以全精度进行插值操作，程序中除了传递纹理坐标的基地址和增量这对参数外，还需要传递另一对参数，即“期待”插入的值的基地址和增量。

在程序 tex1d_9bit.cu 中，函数 CreateAndPrintTex() 被修改为代码清单 10-5 中所示的代码以打印输出结果。

代码清单 10-5 tex1d_9bit.cu (节选)

```
printf( "X\tY\tActual Value\tExpected Value\tDiff\n" );
for ( int i = 0; i < outN; i++ ) {
    T expected;
    if ( bEmulateGPU ) {
        float x = base+(float)i*increment - 0.5f;
        float frac = x - (float) (int) x;
        {
            int frac256 = (int) (frac*256.0f+0.5f);
            frac = frac256/256.0f;
        }
        int index = (int) x;
        expected = (1.0f-frc)*initTex[index] +
                    frac*initTex[index+1];
    }
    else {
        expected = expectedBase + (float) i*expectedIncrement;
```

```

    }
    float diff = fabsf( outHost[i].y - expected );
    printf( "% .2f\t% .2f\t", outHost[i].x, outHost[i].y );
    printf( "%08x\t", *(int *)(&outHost[i].y) );
    printf( "%08x\t", *(int *)(&expected) );
    printf( "%E\n", diff );
}
printf( "\n" );

```

对之前包含 10 个值的纹理（以 0.1 递增），可以把调用该函数产生的真实纹理结果与预期的全精度结果进行对比。函数的调用为：

```
CreateAndPrintTex<float>( tex, 10, 4, 1.5f, 0.25f, 0.1f, 0.025f );
CreateAndPrintTex<float>( tex, 10, 4, 1.5f, 0.1f, 0.1f, 0.01f );
```

得到的输出为：

X	Y	Actual Value	Expected Value	Diff
1.50	0.10	3dcccccd	3dcccccd	0.000000E+00
1.75	0.12	3e000000	3e000000	0.000000E+00
2.00	0.15	3e19999a	3e19999a	0.000000E+00
2.25	0.17	3e333333	3e333333	0.000000E+00
X	Y	Actual Value	Expected Value	Diff
1.50	0.10	3dcccccd	3dcccccd	0.000000E+00
1.60	0.11	3de1999a	3de147ae	1.562536E-04
1.70	0.12	3df5999a	3df5c290	7.812679E-05
1.80	0.13	3e053333	3e051eb8	7.812679E-05

通过输出结果中最右侧的“Diff”列可以看出，第一组输出均是以全精度插值的，而第二组的则不是。《CUDA 编程指南》一书的附录 F 解释了产生这种差别的原因，给出了一维纹理的线性插值公式：

$$tex(x) = (1-\alpha) T(i) + \alpha T(i+1)$$

这里

$$i=\text{floor}(X_B), \alpha+\text{frac}(X_B), X_B=x-0.5$$

其中 α 保存了一个 9 位的定点值，它使用 8 位表示小数部分。

代码清单 10-5 中，该公式是基于 bEmulateGPU 的值分别模拟的。tex1d_9bit.cu 中可以通过将函数 CreateAndPrintTex() 中的参数 bEmulateGPU 值设置为 true，从而模拟 9 位的权重值。相应得到的输出如下所示：

X	Y	Actual Value	Expected Value	Diff
1.50	0.10	3dcccccd	3dcccccd	0.000000E+00
1.75	0.12	3e000000	3e000000	0.000000E+00
2.00	0.15	3e19999a	3e19999a	0.000000E+00
2.25	0.17	3e333333	3e333333	0.000000E+00
X	Y	Actual Value	Expected Value	Diff
1.50	0.10	3dcccccd	3dcccccd	0.000000E+00
1.60	0.11	3de1999a	3de1999a	0.000000E+00
1.70	0.12	3df5999a	3df5999a	0.000000E+00
1.80	0.13	3e053333	3e053333	0.000000E+00

从最右栏的一列 0 中可以看出，当采用 9 位的精度计算插值时，预期值与真实值之间不再存在差异。

10.6 使用归一化坐标纹理操作

当使用归一化的坐标进行纹理操作时，纹理寻址的坐标范围将由 $[0, \text{MaxDim})$ 变为 $[0, 1.0)$ 。对于一个 16 元素的一维纹理，其归一化的坐标如图 10-6 所示。

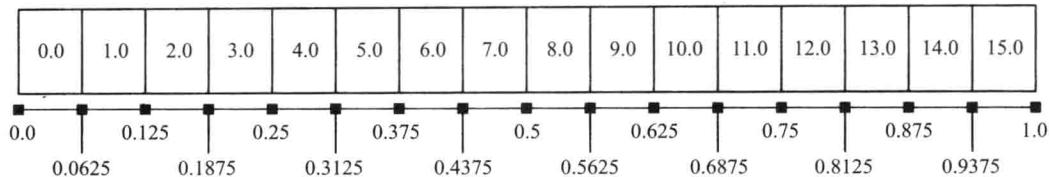


图 10-6 使用归一化坐标的纹理操作

除了失去独立于纹理维度的纹理坐标，纹理操作与先前基本是一样的。但 CUDA 纹理操作的全部功能变得可用。使用归一化的坐标进行操作时，纹理寻址除了夹取和边界寻址模式外，还有另外两种模式可供使用，即重叠寻址模式和镜像寻址模式，相应公式如下所示：

重叠寻址	$x' = x - [x]$
镜像寻址	$x' = \begin{cases} x - [x], & [x] \text{ 为偶数时} \\ 1 - x - [x], & [x] \text{ 为奇数时} \end{cases}$

图 10-7 显示了 CUDA 支持的四种纹理寻址模式。图中描述了每行头尾超出范围的两个坐标进行纹理获取的操作方式以及与正确范围内纹理元素的关系。如果读者通过此图还是无法理解这些纹理寻址模式，请参考下一小节中 `tex2d_opengl.cu` 程序演示。

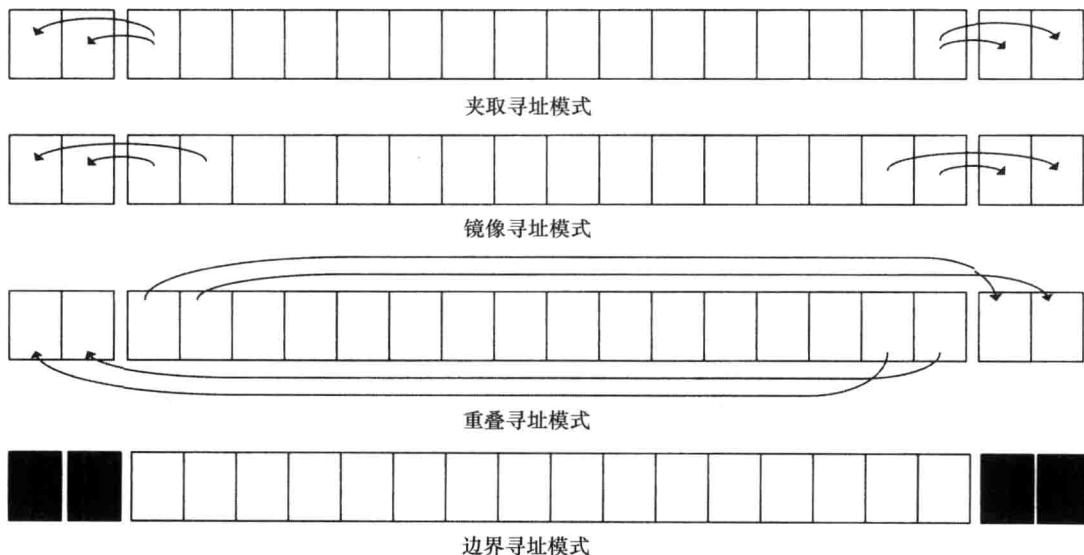


图 10-7 纹理寻址模式



在驱动程序 API 中，`cuTexRefSetArray()` 或 `cuTexRefSetAddress()` 可以完成纹理引用的改变。换言之，只有将纹理引用与内存进行绑定之后，调用诸如 `cuTexRefSetFilterMode()` 或 `cuTexRefSetAddressMode()` 之类的状态改变函数才会生效。

一维设备内存中的浮点坐标

对于那些使用浮点坐标进行纹理寻址的应用程序，或者那些使用只有归一化坐标才有的纹理操作特性的应用程序，可以调用 `cudaBindTexture2D()` 或 `cuTexRefSetAddress2D()` 指定基地址。指定高度为 1，步长为 $N * \text{sizeof}(T)$ 。内核可以接着调用 `tex2D(x, 0.0f)` 读取包含浮点坐标的一维纹理。

10.7 一维表面内存的读写

流处理器簇 2.0 出现之前，CUDA 内核只能通过纹理操作对 CUDA 数组内容进行访问。至于其他 CUDA 数组访问方式，包括所有的写访问，都只能通过如 `cudaMemcpyToArray()` 之类的内存复制函数进行。对于 CUDA 内核而言，对指定内存区域进行纹理读取以及写操作的唯一方式就是将纹理引用与线性设备内存进行绑定。

但若使用流处理器簇 2.x 上新增加的表面内存读写函数，开发者可以将 CUDA 数组与表面引用进行绑定，并调用 `surf1Dread()` 与 `surf1Dwrite()` 指令，对内核中的 CUDA 数组执行读写操作。与拥有特有硬件缓存的纹理读取不同，表面内存的读写操作将通过二级缓存实现，与进行全局加载和存储时类似。



为了将表面引用绑定到 CUDA 数组，CUDA 数组必须以 `cudaArraySurfaceLoadStore` 的标志进行创建。

一维表面内存读写指令的声明如下所示：

```
template<class Type> Type surf1Dread(surface<void, 1> surfRef, int x,
boundaryMode = cudaBoundaryModeTrap);
template<class Type> void surf1Dwrite(Type data, surface<void, 1>
surfRef, int x, boundaryMode = cudaBoundaryModeTrap);
```

从上述代码可以看出，这些指令并不是强类型的（type-strong），表面引用是以 `void` 类型声明的，调用 `surf1Dread()` 或 `surf1Dwrite()` 时内存事务的大小取决于 `sizeof(Type)`。偏移 `x` 的单位是字节，且必须按照 `sizeof(Type)` 对齐。对于如 `int` 或 `float` 类型的 4 字节操作数而言，该偏移必须能被 4 整除，而对于 `short` 类型而言，其必须能被 2 整除。其他类型以此类推。

表面读取支持的功能远远少于纹理操作的功能。^② 其只支持非格式化的读写操作，而且没有类型转换或插值功能，边界处理的模式只有两种。

表面读写的边界处理方式与纹理读取的不同。对于纹理而言，该操作受纹理引用中的寻址模式控制。而对于表面读写而言，越界偏移值的处理方式由 `surf1Dread()` 或 `surf1Dwrite()` 中的一个参数指定。越界索引可导致两种情况：若将 `surf1Dread()` 的参数设置为 `cudaBoundaryModeTrap`，处理越界索引时将抛出硬件异常；而将 `surf1Dwrite()` 的参数设置为 `cudaBoundaryModeZero` 时，`surf1Dread()` 函数读得的值为 0，`surf1Dwrite()` 函数将被忽略。

由于表面引用的无类型的特点，很容易就能写出一个一维的针对所有类型的 `memset` 模板函数。

```
surface<void, 1> surf1D;

template <typename T>
__global__ void
surf1Dmemset( int index, T value, size_t N )
{
    for ( size_t i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        surf1Dwrite( value, surf1D, (index+i)*sizeof(T) );
    }
}
```

该内核存在于案例 `surf1Dmemset.cu` 中，为演示之用，该程序创建了一个 64 字节的 CUDA 数组，并使用上述内核初始化，最后以浮点和整型数的形式打印数组。

以下是一个通用的主机端函数模板代码，其将 `cudaBindSurfaceToArray()` 与该内核的调用封装到一起。

```
template<typename T>
cudaError_t
surf1Dmemset( cudaArray *array, int offset, T value, size_t N )
{
    cudaError_t status;
    CUDART_CHECK(cudaBindSurfaceToArray(surf1D, array));
    surf1Dmemset_kernel<<<2,384>>>( 0, value, 4*NUM_VALUES );
Error:
    return status;
}
```

纹理引用的无类型特性使得其模板结构的实现要比纹理的实现简单很多。由于纹理引用既是强类型又是全局的，因此其不能通过参数列表模板化为通用的函数。若将该函数的一行调用代码由

```
CUDART_CHECK(surf1Dmemset(array, 0, 3.141592654f, NUM_VALUES));
```

改为

```
CUDART_CHECK(surf1Dmemset(array, 0, (short) 0xbeef, 2*NUM_VALUES));
```

^② 事实上，CUDA 可以利用直接对 CUDA 数组操作的指令而彻底地绕过表面引用的实现。表面引用主要是为了与纹理引用相区别，以提供与基于每个指令不同的基于每个表面引用的行为。

则程序的输出结果将由

```
0x40490fdb 0x40490fdb ... (16 times)
3.141593E+00 3.141593E+00 ... (16 times)
```

变为

```
0xbeefbeef 0xbeefbeef ... (16 times)
-4.68253E-01 -4.68253E-01 ... (16 times)
```

10.8 二维纹理操作

多数情况下，二维纹理操作与之前介绍的一维纹理操作类似。应用程序可以选择性地将整型值的纹理元素增强为浮点数，其同样可以使用归一化或非归一化的坐标。当支持线性过滤时，可以以纹理坐标小数部分位作为权重，对四个纹理值进行双线性过滤。此外，硬件可以针对每个维度采用不同的寻址模式。例如，*X*轴可以采用夹取寻址模式，而*Y*轴采用重叠寻址模式。

程序演示：TEX2D_OPENGL.CU

该程序演示利用图形展示不同纹理操作模式的效果。程序使用了具有可移植性的 OpenGL 以及 GL 应用包（GLUT）以减少辅助代码量。为了突出该程序演示的目的，减少其他因素的干扰，程序中将不会使用 CUDA 与 OpenGL 的互操作函数。相反，程序将分配映射主机内存，然后使用 glDrawPixels() 将其渲染到帧缓冲区。对 OpenGL 而言，该数据就如同来自 CPU。

程序支持归一化与非归一化的坐标，并支持分别在*X*轴方向和*Y*轴方向执行夹取、重叠、镜像以及边界这四种寻址模式的纹理操作。对非归一化的坐标，以下内核被用来将纹理内容写到输出缓冲区中。

```
__global__ void
RenderTextureUnnormalized( uchar4 *out, int width, int height )
{
    for ( int row = blockIdx.x; row < height; row += gridDim.x ) {
        out = (uchar4 *) (((char *) out)+row*4*width);
        for ( int col = threadIdx.x; col < width; col += blockDim.x ) {
            out[col] = tex2D( tex2d, (float) col, (float) row );
        }
    }
}
```

内核主要根据纹理坐标从纹理中的相应位置读出像素值，并将该值相应填充到一个具有 *width* × *height* 个像素的长方形块中。使用这种方法就能很容易地看出夹取寻址模式和边界寻址模式对超出范围的像素的处理方式。

下面这个内核使用了归一化的坐标将纹理的内容写到输出缓冲区中。

```
__global__ void
RenderTextureNormalized(
    uchar4 *out,
    int width,
    int height,
    int scale )
{
    for ( int j = blockIdx.x; j < height; j += gridDim.x ) {
```

```

        int row = height-j-1;
        out = (uchar4 *) (((char *) out)+row*4*width);
        float texRow = scale * (float) row / (float) height;
        float invWidth = scale / (float) width;
        for ( int col = threadIdx.x; col < width; col += blockDim.x ) {
            float texCol = col * invWidth;
            out[col] = tex2D( tex2d, texCol, texRow );
        }
    }
}

```

参数 scale 指定了纹理平铺到输出缓冲区中的次数，默认值为 1.0，即纹理只会平铺到缓冲区中一次。执行该程序时，我们可以按下键盘的 1 到 9 号数字键来选择纹理复制的次数。C、W、M 以及 B 键设置了当前方向的寻址模式，X 与 Y 键指定当前的方向。

按 键	操 作
I ~ 9	设置纹理复制次数
W	设置为重叠寻址模式
C	设置为夹取寻址模式
M	设置为镜像寻址模式
B	设置为边界寻址模式
N	切换使用归一化和非归一化纹理坐标
X	C、W、M 或 B 按键将在 X 方向使用相应寻址模式
Y	C、W、M 或 B 按键将在 Y 方向使用相应寻址模式
T	覆盖文本显示开关

读者最好运行该程序，并变换使用不同的参数模式，以观察不同的纹理操作设置所产生的效果。图 10-8 显示了该程序产生的 X 方向重叠与镜像以及 Y 方向重叠与镜像的四个输出，其中纹理复制了 5 次。

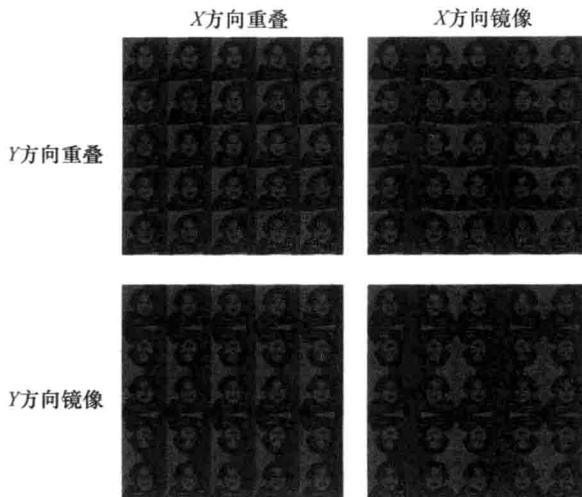


图 10-8 重叠与镜像寻址模式

10.9 二维纹理操作：避免复制

最初引入 CUDA 时，CUDA 内核只能通过纹理读取 CUDA 数组。应用程序只能通过内存复制将数据写入 CUDA 数组中。为了使 CUDA 内核能够将数据写入然后通过纹理读取，应用程序必须将数据写入设备内存中然后执行设备到数组之间的复制操作。后来，两种针对二维纹理的新机制的引入使其不再需要这些步骤。

- 二维纹理能够绑定到一块等步长分配的线性设备内存上；
- 表面内存加载及存储指令使得 CUDA 内核能够直接对 CUDA 数组执行写操作。

设备内存上的三维纹理操作以及三维表面内存的加载和存储操作不支持这两种机制。

对于那些以常规访问方式读取绝大部分或所有纹理内容的应用程序（例如视频编解码）或那些必须在特斯拉架构的硬件上运行的应用程序，数据最好存储在设备内存中。而对于那些在纹理操作时进行随机访问（但为局部访问）的应用程序，数据最好存储在 CUDA 数组中，并利用表面内存读写指令进行访问。

10.9.1 设备内存上的二维纹理操作

二维设备内存上的纹理操作并不享有“块线性”寻址所带来的益处，即存于纹理缓存的缓存行以水平的跨度拉取纹理元素（texel，也简称为纹素），而不是它们的二维或三维块。但是，除非程序是以随机方式对纹理进行访问，否则，免除将设备内存复制到 CUDA 数组所带来的优势，要远远超过因丧失块线性寻址而带来的损失。

调用 `cudaBindTexture2D()`，可以将二维纹理引用与一块设备内存进行绑定。

```
cudaBindTexture2D(
    NULL,
    &tex,
    texDevice,
    &channelDesc,
    inWidth,
    inHeight,
    texPitch );
```

上述调用可以将纹理引用绑定到由 `texDevice/texPitch` 指定的二维设备内存区间上。地址与步长必须符合硬件指定的对齐约束。[⊖] 基地址必须符合 `cudaDeviceProp.texturePitchAlignment` 的对齐限制，而步长则需要符合 `cudaDeviceProp.texturePitchAlignment` 的对齐限制。[⊖] 案例程序 `tex2d_addressing_device.cu` 与案例程序 `tex2d_addressing.cu` 几乎一模一样，只是前者使用的是设备内存保存的纹理数据。这两个程序的设计非常相似，因此我们只需要看它们不同的部分。这里是使用设备指针和步长的二元组代替 CUDA 数组。

[⊖] CUDA 数组也必须遵守相同的限制，但此时地址与步长将由 CUDA 管理，隐藏在内存层次结构中。

[⊖] 在驱动程序 API 中，对应设备属性查询为 `CU_DEVICE_ATTRIBUTE_TEXTURE_ALIGNMENT` 和 `CU_DEVICE_ATTRIBUTE_TEXTURE_PITCH_ALIGNMENT`。

```

< cudaArray *texArray = 0;
> T *texDevice = 0;
> size_t texPitch;

```

调用 `cudaMallocPitch()` 而不是 `cudaMallocArray()`。

`cudaMallocPitch()` 将委派驱动程序对基址与步长进行选择，因此该代码仍可以在未来生产的硬件上执行（未来硬件会有增加对齐要求的趋势）。

```

< CUDART_CHECK(cudaMallocArray( &texArray,
< &channelDesc,
< inWidth,
> CUDART_CHECK(cudaMallocPitch( &texDevice,
> &texPitch,
> inWidth*sizeof(T),
inHeight));

```

接着，调用 `cudaTextureBind2D()` 而不是 `cudaBindTextureToArray()`。

```

< CUDART_CHECK(cudaBindTextureToArray(tex, texArray));
> CUDART_CHECK(cudaBindTexture2D( NULL,
> &tex,
> texDevice,
> &channelDesc,
> inWidth,
> inHeight,
> texPitch));

```

最后的差别在于 CUDA 数组的释放，此处，应对 `cudaMallocPitch()` 返回的指针调用 `cudaFree()` 进行释放。

```

< cudaFreeArray( texArray );
> cudaFree( texDevice );

```

10.9.2 二维表面内存的读写

与一维表面内存的读写操作一样，费米架构的硬件允许内核直接使用表面内存读写内置函数对 CUDA 数组进行写操作。

```

template<class Type> Type surf2Dread(surface<void, 1> surfRef, int x,
int y, boundaryMode = cudaBoundaryModeTrap);
template<class Type> Type surf2Dwrite(surface<void, 1> surfRef, Type
data, int x, int y, boundaryMode = cudaBoundaryModeTrap);

```

`Surf2Dmemset.cu` 给出了表面引用的声明以及对应的二维表面内存 `memset` 函数的 CUDA 内核实现。具体代码如下：

```

surface<void, 2> surf2D;

template<typename T>
__global__ void
surf2Dmemset_kernel( T value,
                     int xOffset, int yOffset,
                     int Width, int Height )
{
    for ( int row = blockIdx.y*blockDim.y + threadIdx.y;
          row < Height;
          row += blockDim.y*gridDim.y )

```

```

    {
        for ( int col = blockIdx.x*blockDim.x + threadIdx.x;
              col < Width;
              col += blockDim.x*gridDim.x )
        {
            surf2Dwrite( value,
                         surf2D,
                         (xOffset+col)*sizeof(T),
                         yOffset+row );
        }
    }
}

```

记住, `surf2Dwrite()` 中的 *X* 偏移参数是以字节为单位计算的。

10.10 三维纹理操作

从三维纹理中读取数据与从二维纹理中读取数据类似, 但三维纹理有更多的限制条件。

- 与二维纹理的维度 65536×32768 相比, 三维纹理每个的维度较小, 为 $2048 \times 2048 \times 2048$;
- 三维纹理操作中没有避免复制的方法, CUDA 不支持设备内存上的三维纹理操作或在三维 CUDA 数组上的表面加载和存储操作。

除此之外, 三维纹理操作基本与之前介绍的纹理操作类似。内核可以使用 `tex3D()` 指令读取三维纹理, 该指令接收三个浮点参数, 对应的三维 CUDA 数组必须使用三维内存复制操作进行数据填充。三维纹理支持三线性过滤, 通过纹理坐标可读出 8 个纹理元素并进行插值, 精度限制与一维和二维纹理操作的一样, 均为 9 位。

三维纹理的大小限制可以通过调用 `cuDeviceGetAttribute()` 结合参数 `CU_DEVICE_ATTRIBUTE_TEXTURE3D_WIDTH`、`CU_DEVICE_ATTRIBUTE_TEXTURE3D_HEIGHT` 以及 `CU_DEVICE_ATTRIBUTE_TEXTURE3D_DEPTH` 进行查询, 或者调用 `cudaGetDeviceProperties()` 检查 `cudaDeviceProp.maxTexture3D` 值。由于需要的参数过多, 用来创建和操作三维 CUDA 数组的 API 集与一维或二维 CUDA 数组的有很大不同。

函数 `cudaMalloc3DArray()` 用来创建三维 CUDA 数组, 其接收一个 `cudaExtent` 结构体, 而不是参数 `width` 和 `height`。

```
cudaError_t cudaMalloc3DArray(struct cudaArray** array, const struct
cudaChannelFormatDesc* desc, struct cudaExtent extent, unsigned int
flags __dv(0));
```

结构体 `cudaExtent` 的定义如下:

```
struct cudaExtent {
    size_t width;
    size_t height;
    size_t depth;
};
```

三维内存复制操作非常复杂, CUDA 运行时与驱动程序 API 均使用结构体来指定参数。

运行时 API 使用的是 cudaMemcpy3DParms 结构体，其声明如下：

```
struct cudaMemcpy3DParms {
    struct cudaArray *srcArray;
    struct cudaPos srcPos;
    struct cudaPitchedPtr srcPtr;
    struct cudaArray *dstArray;
    struct cudaPos dstPos;
    struct cudaPitchedPtr dstPtr;
    struct cudaExtent extent;
    enum cudaMemcpyKind kind;
};
```

结构体中的大多数成员又是结构体，extent 是包含了复制的宽度、高度和深度三个值的结构体。成员 srcPos 与 dstPos 的类型均是结构体 cudaPos，其成员分别指定了复制源和目标的起始点。

```
struct cudaPos {
    size_t x;
    size_t y;
    size_t z;
};
```

结构体 cudaMemcpy3DParms 是添加到三维内存复制中，并用于存储指针和步长二元组信息的。

```
struct cudaPitchedPtr
{
    void *ptr; /**< Pointer to allocated memory */
    size_t pitch; /**< Pitch of allocated memory in bytes */
    size_t xsize; /**< Logical width of allocation in elements */
    size_t ysize; /**< Logical height of allocation in elements */
};
```

结构体 cudaMemcpy3DParms 可以通过调用函数 make_cudaPitchedPtr 创建，该函数接受待分配内存的基指针、步长、逻辑宽度和高度。函数 make_cudaPitchedPtr 仅仅是将这些参数复制到输出结构体中。

```
struct cudaPitchedPtr
make_cudaPitchedPtr(void *d, size_t p, size_t xsz, size_t ysz)
{
    struct cudaPitchedPtr s;

    s.ptr = d;
    s.pitch = p;
    s.xsize = xsz;
    s.ysize = ysz;

    return s;
}
```

SDK 中的样例 simpleTexture3D 描述了如何使用 CUDA 进行三维纹理操作。

10.11 分层纹理

分层纹理在图形学里以纹理数组著称，它可以将一维或二维纹理以数组的形式组织，并

通过整型索引值访问。针对二维或三维纹理使用分层纹理的主要优势在于其允许更大范围的分片，但使用分层纹理并不具有性能优势。

分层纹理在内存中的布局与二维和三维纹理不同，若使用针对分层纹理优化的布局，对二维或三维纹理的操作将造成性能影响。因此，当调用 `cudaMalloc3DArray()` 创建 CUDA 数组时，最后一个标志参数必须指定 `cudaArrayLayered`，或调用 `cuArray3DCreate()` 时，最后一个标志参数指定为 `CUDA_ARRAY3D_LAYERED`。SDK 中的 `simpleLayeredTexture` 样例展示了如何使用分层纹理。

10.11.1 一维分层纹理

一维分层纹理的大小限制可以通过调用 `cuDeviceGetAttribute()` 结合参数 `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH` 以及 `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS` 进行查询，或者调用 `cudaGetDeviceProperties()` 检查 `cudaDeviceProp.maxTexture1DLayered` 的值。

10.11.2 二维分层纹理

二维分层纹理的大小限制可以通过调用 `cuDeviceGetAttribute()` 结合参数 `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH`、`CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT` 以及 `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS` 进行查询，或者调用 `cudaGetDeviceProperties()` 检查 `cudaDeviceProp.maxTexture2DLayered` 的值。分层纹理大小限制可以通过调用 `cudaGetDeviceProperties()` 检查 `cudaDeviceProp.maxTexture2DLayered` 的值查询。

10.12 最优线程块大小选择以及性能

当以“显而易见”的方式生成纹理坐标，如 `tex2d_addressing.cu` 中的方式

```
row = blockIdx.y*blockDim.y + threadIdx.y;
col = blockIdx.x*blockDim.x + threadIdx.x;
... tex2D( tex, (float) col, (float) row );
```

则纹理操作的性能取决于线程块的大小。

为了找到线程块的最优大小，程序 `tex2D_shmoo.cu` 与 `surf2Dmemset_shmoo.cu` 通过计时的方式记录了当线程块的宽度和高度分别设置从 4 ~ 64 之间的任意一个值时的性能。其中部分线程块的宽高组合是无效的，因为线程的总数目超出限制。

对于该程序，为了最大程度地体现硬件纹理单元的性能，纹理操作内核将设计为“执

行尽可能少的任务”，但编译器仍将生成可执行代码。内核中每个线程将计算读出的所有值的总和，并在输出参数为非空时将总和写入输出中。然而，在执行的时候我们将不会传递一个非空指针给内核。这样设计内核的原因主要是：若直接去掉这段将结果写回输出内存的代码，编译器将发现内核没有做任何工作，因此将生成没有任何纹理操作的可执行代码。

```
extern "C" __global__ void
TexSums( float *out, size_t Width, size_t Height )
{
    float sum = 0.0f;
    for ( int row = blockIdx.y*blockDim.y + threadIdx.y;
          row < Height;
          row += blockDim.y*gridDim.y )
    {
        for ( int col = blockIdx.x*blockDim.x + threadIdx.x;
              col < Width;
              col += blockDim.x*gridDim.x )
        {
            sum += tex2D( tex, (float) col, (float) row );
        }
    }
    if ( out ) {
        out[blockIdx.x*blockDim.x+threadIdx.x] = sum;
    }
}
```

尽管我们使用了“技巧”，编译器仍可能会生成“先检查参数 `out` 的值，并在看到 `out` 为 `NULL` 时则直接退出内核的”可执行代码。因此，我们需要合成一些不会太影响性能的输出（例如，每个线程块在共享内存上进行和的归约计算，并将结果写到输出参数 `out` 中）。但在编译程序时选择 `--keep` 选项，并使用 `cuobjdump--dump-sass` 查看微码（microcode），可以看到在执行了双重嵌套的 `for` 循环之后，编译器才检查参数 `out`。

结果

在 GeForce GTX 280 (GT200) 上，线程块的最优大小为 128 个线程，得到的带宽为 35.7GB/s。线程块配置为 32×4 的速度与 16×8 或 8×16 相同，均在 1.88ms 内传输了 $4K \times 4K$ 的浮点型纹理。在特斯拉 M2050 上，线程块的最优大小为 192 个线程，带宽为 35.4GB/s。与 GT200 一样，宽高不同但整体线程数相同的线程块的执行速度相同，宽和高分别为 6×32 、 16×12 与 8×24 的线程块在传输数据上性能相同。

二维表面内存填充程序的测试结果说服力较小。线程块的线程数必须超过 128 个线程才会有较好的性能，并且需要线程数目能够被线程束的大小 32 整除。在 `cg1.4xlarge` 上不开启 ECC，二维表面内存上的内存填充的性能最快能达到 48GB/s。

对于这两个 GPU 卡均使用浮点值数据进行测试，纹理操作和表面内存写操作的峰值带宽分别是“全局加载及存储”峰值带宽的 1/4 和 1/2。

10.13 纹理操作快速参考

10.13.1 硬件能力

1. 硬件限制

能力	SM 1.X	SM 2.X
一维 CUDA 数组最大宽度	8192	32768
一维设备内存最大宽度	2^{27}	
一维分层纹理最大宽度和层数	8192×512	16384×2048
二维纹理最大范围	65536×32768	
二维分层纹理最大范围以及层数	$8192 \times 8192 \times 512$	$16384 \times 16384 \times 2048$
三维 CUDA 数组最大范围	$2048 \times 2048 \times 2048$	
能够绑定到内核上的最大纹理数目	128	
能够绑定到 CUDA 内核上的二维表面引用最大范围	n/a	8192×8192
能够绑定到内核上的最大表面内存数目		8

2. 驱动程序 API 查询

上述列出的硬件限制大多数可以通过调用 cuDeviceAttribute() 查询，具体可用的查询值如下所示：

属性	cuDeviceAttribute 查询值
一维 CUDA 数组最大宽度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH
一维分层纹理最大宽度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_WIDTH
一维分层纹理最大层数	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_LAYERED_LAYERS
二维纹理最大宽度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_WIDTH
二维纹理最大高度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_HEIGHT
二维分层纹理最大宽度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_WIDTH
二维分层纹理最大高度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_HEIGHT
二维分层纹理最大层数	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE2D_LAYERED_LAYERS
三维 CUDA 数组最大宽度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_WIDTH
三维 CUDA 数组最大高度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_HEIGHT
三维 CUDA 数组最大深度	CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE3D_DEPTH

3. CUDA 运行时查询

下列 `cudaDeviceProp` 的成员包含了上述的硬件限制：

能 力	<code>cudaDeviceProp</code> 成员
一维 CUDA 数组最大宽度	<code>int maxTexture1D;</code>
一维分层纹理最大宽度和层数	<code>int maxTextureLayered[2];</code>
二维纹理最大范围	<code>int maxTexture2D[2];</code>
二维分层纹理最大范围以及层数	<code>int maxTextureLayered[2];</code>
三维 CUDA 数组最大范围	<code>int maxTexture3D[3];</code>

10.13.2 CUDA 运行时

1. 一维纹理

操作	设备内存	CUDA 数组
分配使用……	<code>cudaMalloc()</code>	<code>cudaMallocArray()</code>
释放使用……	<code>cudaFree()</code>	<code>cudaFreeArray()</code>
绑定使用……	<code>cudaBindTexture()</code>	<code>cudaBindTextureToArray()</code>
纹理读取使用……	<code>tex1Dfetch()</code>	<code>tex1D()</code>

2. 二维纹理

操作	设备内存	CUDA 数组
分配使用……	<code>cudaMallocPitch()</code>	<code>cudaMalloc2DArray()*</code>
释放使用……	<code>cudaFree()</code>	<code>cudaFreeArray()</code>
绑定使用……	<code>cudaBindTexture2D()</code>	<code>cudaBindTextureToArray()</code>
纹理读取使用……	<code>tex2D()</code>	

注：如果要使用表面加载和存储方式，需指定 `cudaArraySurfaceLoadStore` 标志。

3. 三维纹理

操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cudaMalloc3DArray()</code>
释放使用……		<code>cudaFreeArray()</code>
绑定使用……		<code>cudaBindTextureToArray()</code>
纹理读取使用……		<code>tex3D()</code>

4. 一维分层纹理

操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cudaMalloc2DArray()</code> - specify <code>cudaArrayLayered</code> .
释放使用……		<code>cudaFreeArray()</code>
绑定使用……		<code>cudaBindTextureToArray()</code>
纹理读取使用……		<code>tex1DLayered()</code>

5. 二维分层纹理

操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cudaMalloc3DArray()</code> - specify <code>cudaArrayLayered</code> .
释放使用……		<code>cudaFreeArray()</code>
绑定使用……		<code>cudaBindTextureToArray()</code>
纹理读取使用……		<code>tex1DLayered()</code>

10.13.3 驱动 API

1. 一维纹理

操作	设备内存	CUDA 数组
分配使用……	<code>cuMemAlloc()</code>	<code>cuArrayCreate()</code>
释放使用……	<code>cuMemFree()</code>	<code>cuArrayDestroy()</code>
绑定使用……	<code>cuTexRefSetAddress()</code>	<code>cuTexRefSetArray()</code>
纹理读取使用……	<code>tex1Dfetch()</code>	<code>tex1D()</code>
大小限制……	2^{27} 元素 (128M)	65536

设备内存的纹理大小限制是不可查询的，在所有支持 CUDA 的 GPU 上均为 2^{27} 个元素。一维 CUDA 数组的纹理大小限制可以通过调用 `cuDeviceGetAttribute()` 结合参数 `CU_DEVICE_ATTRIBUTE_MAXIMUM_TEXTURE1D_WIDTH` 查询。

2. 二维纹理

操作	设备内存	CUDA 数组
分配使用……	<code>cuMemAllocPitch()</code>	<code>cuArrayCreate()</code>
释放使用……	<code>cuMemFree()</code>	<code>cudaFreeArray()</code>
绑定使用……	<code>cuTexRefSetAddress2D()</code>	<code>cuTexRefSetArray()</code>
纹理读取使用……	<code>tex2D()</code>	

3. 三维纹理

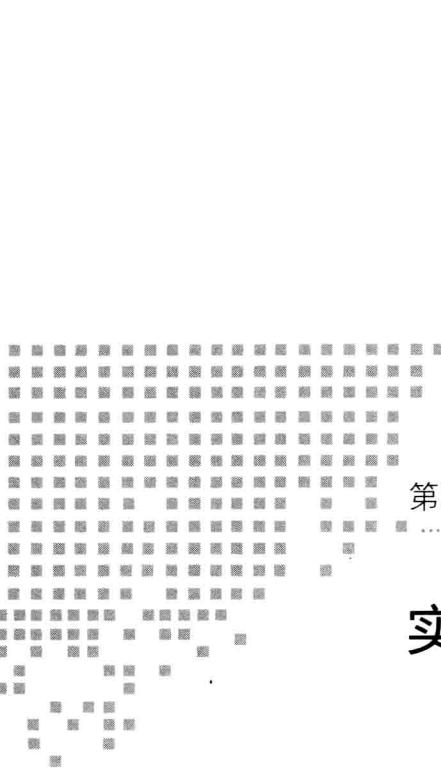
操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cudaMalloc3DArray()</code>
释放使用……		<code>cudaFreeArray()</code>
绑定使用……		<code>cudaBindTextureToArray()</code>
纹理读取使用……		<code>tex3D()</code>

4. 一维分层纹理

操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cuArray3DCreate() - specify CUDA_ARRAY3D_LAYERED</code>
释放使用……		<code>cudaFreeArray()</code>
绑定使用……		<code>cuTexRefSetArray()</code>
纹理读取使用……		<code>tex1DLayered()</code>

5. 二维分层纹理

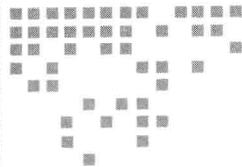
操作	设备内存	CUDA 数组
分配使用……	[不支持]	<code>cuArray3DCreate() - specify CUDA_ARRAY3D_LAYERED</code>
释放使用……		<code>cuArrayDestroy()</code>
绑定使用……		<code>cuTexRefSetArray()</code>
纹理读取使用……		<code>tex2DLayered()</code>



第三部分 *Part 3*

实 例

- 第 11 章 流式负载
 - 第 12 章 归约算法
 - 第 13 章 扫描算法
 - 第 14 章 N- 体问题
 - 第 15 章 图像处理的归一化相关系数计算
- *****



流式负载

流式负载 (streaming workload) 是每个数据元素可以被独立地计算的任务，它是可以移植到 CUDA 中的最简单的负载。这类低计算密度 (computational density) 的负载通常属于带宽受限型 (bandwidth-bound)。流式负载无须使用太多的 GPU 硬件资源，例如用于优化数据重用的高速缓存和共享存储器等。

鉴于 GPU 对于高计算密度的负载具有最大效力，我们将讨论几种情形，以助于流式负载移植到 GPU。

- 如果输入和输出均在设备内存上，让数据传回 CPU 却仅执行一次计算操作是得不偿失的。
- 如果 GPU 有比 CPU 更好的指令级操作支持（例如，频繁使用特殊功能单位指令的 Black-Scholes 期权计算任务），尽管存在额外的内存传输开销，GPU 仍可以超过 CPU。
- GPU 操作与 CPU 并发执行可以增加约一倍的性能（即使假设它们的速度相同）。
- 对于给定负载，CUDA 代码相对于高度优化的 CPU 代码可能会更可读或更可维护。
- 对于包含集成显卡的系统（即 CPU 与支持 CUDA 的 GPU 共存于同一芯片，并作用于同一内存），不存在传输开销。CUDA 技术可以使用“零复制”方法，完全避免复制操作。

本章涵盖流式负载的方方面面。针对同一负载，采用不同实现策略，以暴露可能出现的多个问题。这里关注的负载是来自 BLAS 库的 SAXPY 操作。它在一个操作中同时执行一个标量乘法和矢量加法。

代码清单 11-1 给出了一个实现 SAXPY 的简单 C 程序。每次取出两个输入数组相对应

的两个数，对其中一个施行常倍数的缩放，并加到另一个数上，然后写回输出数组的对应位置。两个输入数组和输出数组均包含 N 个元素。由于 GPU 包含原生乘加指令，SAXPY 的最内层循环在每次内存访问时，仅包含较少的指令数。

代码清单 11-1 saxpyCPU 函数

```
void
saxpyCPU(
    float *out,
    const float *x,
    const float *y,
    size_t N,
    float alpha )
{
    for ( size_t i = 0; i < N; i++ ) {
        out[i] += alpha*x[i]+y[i];
    }
}
```

代码清单 11-2 给出了 SAXPY 的一个简单 CUDA 实现版本。这个版本可以运行于任何网格或各种大小的线程块，并且其执行性能能满足大多数应用的需要。由于这个内核严重受制于带宽，对多数应用程序而言，与其在它的基础上进行优化，不如重构该程序以增加计算密度。

代码清单 11-2 saxpyGPU 内核函数

```
_global_ void
saxpyGPU(
    float *out,
    const float *x,
    const float *y,
    size_t N,
    float alpha )
{
    for ( size_t i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x ) {
        out[i] = alpha*x[i]+y[i];
    }
}
```

本章的主旨在于讨论如何将数据高效地传入和传出主机内存。但是，首先我们要花些时间研究如何从操作设备内存的角度改善这个内核的性能。

11.1 设备内存

如果输入和输出数据都在设备内存上，优化如 SAXPY 这样低计算密度的任务就变成了优化全局内存访问的问题。除了对齐和合并读取约束外，CUDA 内核的性能也对线程块数目和每块的线程数目比较敏感。globalRead、globalWrite、globalCopy 和 globalCopy2 应用程序

(在本书配套源代码的 memory/ 子目录) 生成针对多种操作数大小、块大小和循环展开因子的带宽报告。由 globalCopy2 (它遵循类似于 SAXPY 的内存访问模式: 每个循环迭代包含两次读操作和一次写操作) 生成的一个示例报告如代码清单 11-3 所示。

如果我们引用第 5 章的 globalCopy2.cu 应用程序 (参见代码清单 5-8), 在 GK104 运行它, 得到 4 个字节操作数时的输出, 如代码清单 11-3 所示。最上面一行 (展开因子为 1) 对应于简单实现 (类似于代码清单 11-2)。从中可以看到, 循环展开具有少量的性能优势。当使用的展开因子为 4 时, 得到约 10% 的加速, 提供的带宽为 128GB/s, 而不是简单实现时的 116GB/s。

有趣的是, 使用 #pragma unroll 编译器指令仅提升性能到约 118GB/s, 而修改带模板的 globalCopy2.cu 内核来执行 SAXPY, 性能则可提升至 135GB/s。代码清单 11-4 给出了最终的内核, 它在 stream1Device.cu 应用程序中实现 (见本书配套源代码 cudahandbook/streaming/ 目录)。

对于大多数应用程序, 这些小的性能差异并不能证明这种重写内核方式的价值。但是, 如果内核被编写为“跟线程块无关”(即能在任何网格或线程块大小下正确工作), 那么最优的设置可以凭经验确定, 而无须花费太多精力。

代码清单 11-3 globalCopy2 输出 (GK104)

	Block Size						
Unroll	32	64	128	256	512	maxBW	maxThreads
1	63.21	90.89	104.64	113.45	116.06	116.06	512
2	66.43	92.89	105.09	116.35	120.66	120.66	512
3	87.23	100.70	112.07	110.85	121.36	121.36	512
4	99.54	103.53	113.58	119.52	128.64	128.64	512
5	94.27	103.56	108.02	122.82	124.88	124.88	512
6	100.67	104.18	115.10	122.05	122.46	122.46	512
7	94.56	106.09	116.30	117.63	114.50	117.63	256
8	58.27	45.10	47.07	46.29	45.18	58.27	32
9	41.20	34.74	35.87	35.49	34.58	41.20	32
10	33.59	31.97	32.42	31.43	30.61	33.59	32
11	27.76	28.17	28.46	27.83	26.79	28.46	128
12	25.59	26.42	26.54	25.72	24.51	26.54	128
13	22.69	23.07	23.54	22.50	20.71	23.54	128
14	22.19	22.40	22.23	21.10	19.00	22.40	64
15	20.94	21.14	20.98	19.62	17.31	21.14	64
16	18.86	19.01	18.97	17.66	15.40	19.01	64

代码清单 11-4 saxpyGPU 内核函数 (带模板的展开)

```
template<const int n>
__device__ void
saxpy_unrolled(
    float *out,
    const float *px,
    const float *py,
```

```

size_t N,
float alpha )
{
    float x[n], y[n];
    size_t i;
    for ( i = n*blockIdx.x*blockDim.x+threadIdx.x;
          i < N-n*blockDim.x*gridDim.x;
          i += n*blockDim.x*gridDim.x ) {
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            x[j] = px[index];
            y[j] = py[index];
        }
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            out[index] = alpha*x[j]+y[j];
        }
    }
    // to avoid the (index<N) conditional in the inner loop,
    // we left off some work at the end
    for ( int j = 0; j < n; j++ ) {
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            if ( index<N ) {
                x[j] = px[index];
                y[j] = py[index];
            }
        }
        for ( int j = 0; j < n; j++ ) {
            size_t index = i+j*blockDim.x;
            if ( index<N ) out[index] = alpha*x[j]+y[j];
        }
    }
}
}

__global__ void
saxpyGPU( float *out, const float *px, const float *py, size_t N,
float alpha )
{
    saxpy_unrolled<4>( out, px, py, N, alpha );
}

```

该 stream1Device.cu 应用程序报告了从基于分页的系统内存复制数据到设备内存，在数据上运行代码清单 11-4 所示的内核，并将数据传输回去所需的系统时钟时间（wall clock time）。测试系统采用 GeForce GTX680，运行在 Intel i7 和 Windows 7 环境下，这个应用程序的输出如下所示。

```

Measuring times with 128M floats (use --N to specify number of Mfloats)
Memcpy( host->device ): 365.95 ms (2934.15 MB/s)
Kernel processing : 11.94 ms (134920.75 MB/s)
Memcpy (device->host ): 188.72 ms (2844.73 MB/s)

Total time (wall clock): 570.22 ms (2815.30 MB/s)

```

内核的执行仅占用整体执行时间的一小部分，大约 2% 的系统时钟时间。其余 98% 的时间花费在把数据传入和传出 GPU！对于像这样的传输受限型负载，如果待处理的数据的一

部分或者全部是在主机内存中，那么优化应用程序的最好方法是提高 CPU/GPU 执行的重叠程度和传输性能。

11.2 异步内存复制

除非输入和输出数据可以驻留在 GPU 上，让数据流过 GPU（即把输入和输出数据传入和传出设备内存）的策略变得尤为重要。最适合用来提高传输性能的 2 个工具是锁页内存和异步内存复制（它仅可以运行在锁页内存）。

stream2Async.cu 应用程序演示了把 stream1Device.cu 的分页内存替换成锁页内存，并启用异步内存复制的效果。

```
Measuring times with 128M floats (use --N to specify number of Mfloats)
Memcpy( host->device ) : 181.03 ms (5931.33 MB/s)
Kernel processing : 13.87 ms (116152.99 MB/s)
Memcpy (device->host ) : 90.07 ms (5960.35 MB/s)

Total time (wall clock) : 288.68 ms (5579.29 MB/s)
```

代码清单 11-5 对比了 stream1Device.cu（它执行同步传输）和 stream2Async.cu（它执行异步传输）的计时部分的差异。[⊖] 在这两种情况下，4 个 CUDA 事件用于分别记录开始时刻、主机到设备传输完成的时刻、内核执行完成的时刻以及结束时刻。对于 stream2Async.cu，所有这些操作在短时间内相继请求了 GPU，而 GPU 在执行它们的时候记录下事件的时间。对于 stream1Device.cu，GPU 的基于事件的时间是有点不够准确，因为任何对 cudaMemcpy() 的调用必须等待 GPU 完成之后再继续下一操作，会在以 evHtoD 和 evDtoH 参数执行 cudaEventRecord() 之前导致流水线出现空隙。

值得注意的是，尽管使用了较慢的、简单实现的 saxpyGPU 函数（来自代码清单 11-2），该应用程序消耗的系统时钟时间显示，它的计算速度仍然快了将近一倍：289 毫秒对比之前的 570.22 毫秒。结合使用更快的数据传输和异步执行会获得更好的性能。

尽管性能提高了，应用程序的输出指示了另一个提升性能的机会：一些内核的处理可以与数据传输并发进行。接下来的两节将介绍 2 种实现“内核执行和传输”重叠进行的方法。

代码清单 11-5 同步内核 (stream1Device.cu) 与异步内核 (stream2Async.cu) 对比

```
//  
// from stream1Device.cu  
//  
cudaEventRecord( evStart, 0 );  
cudaMemcpy( dptrX, hptrX, ..., cudaMemcpyHostToDevice );  
cudaMemcpy( dptrY, hptrY, ..., cudaMemcpyHostToDevice );  
cudaEventRecord( evHtoD, 0 );  
    saxpyGPU<<nBlocks, nThreads>>>( dptrOut, dptrX, dptrY, N, alpha );  
cudaEventRecord( evKernel, 0 );  
cudaMemcpy( hptrOut, dptrOut, N*sizeof(float), cudaMemcpyDeviceToHost );
```

[⊖] 为清晰起见，删除了进行错误检查的代码。

```

cudaEventRecord( evDtoH, 0 );
cudaDeviceSynchronize();

//  

// from stream2Async.cu  

//  

cudaEventRecord( evStart, 0 );
cudaMemcpyAsync( dptrX, hptrX, ..., cudaMemcpyHostToDevice, NULL );
cudaMemcpyAsync( dptrY, hptrY, ..., cudaMemcpyHostToDevice, NULL );
cudaEventRecord( evHtoD, 0 );
    saxpyGPU<<<nBlocks, nThreads>>>( dptrOut, dptrX, dptrY, N, alpha );
cudaEventRecord( evKernel, 0 );
cudaMemcpyAsync( hptrOut, dptrOut, N*sizeof(float), ... , NULL );
cudaEventRecord( evDtoH, 0 );
cudaDeviceSynchronize();

```

11.3 流

对于那些能够得益于内核处理与数据传输并发进行 (CPU/GPU 重叠) 的负载, CUDA 流可以用来协调它们的执行。stream3Streams.cu 应用程序将输入和输出数组分成 k 个流, 然后引发 k 个主机到设备的内存复制、内核执行和设备到主机的内存复制, 它们各自运行在自己独立的流里。把传输和计算关联到不同流, 促使 CUDA 知道这些计算是完全独立的, 这样 CUDA 将会利用硬件支持的任何并行机会。对于包含多个复制引擎的 GPU, GPU 可能在把数据传入和传出设备内存的同时, 可以让 SM 处理其他数据。

代码清单 11-6 显示了来自 stream3Streams.cu 的节选, 具有如代码清单 11-5 相同的功能。在测试系统上, 这个应用程序的输出内容如下。

```

Measuring times with 128M floats
Testing with default max of 8 streams (set with --maxStreams <count>)

Streams  Time (ms)  MB/s
1        290.77 ms  5471.45
2        273.46 ms  5820.34
3        277.14 ms  5744.49
4        278.06 ms  5725.76
5        277.44 ms  5736.52
6        276.56 ms  5751.87
7        274.75 ms  5793.43
8        275.41 ms  5779.51

```

所使用的 GPU 仅具有一个复制引擎, 因此很容易理解它在使用 2 个流的情况下, 如何达到最高的性能。如果内核执行时间超过传输时间, 它可能会得益于把数组分割为多于 2 个的子数组。根据实际情况看, 第一个内核启动只有在完成了第一个主机到设备的内存复制操作后才能开始执行, 而最后一个设备到主机的内存复制操作只有在最后一个内核启动执行完毕后才能开始。如果内核处理消耗更多的时间, 这种“突出部分”会更明显。对于我们的应用程序, 273 毫秒的系统时钟时间表明, 大多数内核处理的开销 (13.87 毫秒) 已被隐藏。

需要注意的是, 这一策略没有像代码清单 11-5 那样在操作之间插入任何 cudaEventRecord(),

部分原因是由于硬件的限制。在大多数的 CUDA 硬件上，试图在代码清单 11-6 的流式操作之间记录事件将打破并发性并降低性能。相反地，我们在所有操作之前和所有操作之后分别加入了一个 cudaEventRecord()。

代码清单 11-6 stream3Streams.cu 内核节选

```

for ( int iStream = 0; iStream < nStreams; iStream++ ) {
    CUDART_CHECK( cudaMemcpyAsync(
        dptrX+iStream*streamStep,
        hptrX+iStream*streamStep,
        streamStep*sizeof(float),
        cudaMemcpyHostToDevice,
        streams[iStream] ) );
    CUDART_CHECK( cudaMemcpyAsync(
        dptrY+iStream*streamStep,
        hptrY+iStream*streamStep,
        streamStep*sizeof(float),
        cudaMemcpyHostToDevice,
        streams[iStream] ) );
}

for ( int iStream = 0; iStream < nStreams; iStream++ ) {
    saxpyGPU<<<nBlocks, nThreads, 0, streams[iStream]>>>(
        dptrOut+iStream*streamStep,
        dptrX+iStream*streamStep,
        dptrY+iStream*streamStep,
        streamStep,
        alpha );
}

for ( int iStream = 0; iStream < nStreams; iStream++ ) {
    CUDART_CHECK( cudaMemcpyAsync(
        hptrOut+iStream*streamStep,
        dptrOut+iStream*streamStep,
        streamStep*sizeof(float),
        cudaMemcpyDeviceToHost,
        streams[iStream] ) );
}

```

11.4 映射锁页内存

对于传送受限型流式负载，例如 SAXPY，重新组织程序把映射锁页内存用于输入和输出，有许多好处。

- 从节选的 stream4Mapped.cu（代码清单 11-7）可以看出，它不需要调用复制函数 cudaMemcpy()。
- 它不需要分配设备内存。
- 对于独立 GPU，映射锁页内存执行总线传输，但最大限度地减少了上一节中提到的“突出部分”的数量。无须等待主机到设备的内存复制操作完成，而是在输入数据到达的当时被 SM 处理。无须等待内核完成后再发起设备到主机的传输，而是在 SM 处理结束就把数据发布到总线。

- 对于集成 GPU，主机和设备内存共存于同一个内存池，因此映射锁页内存支持“零复制”特性，并消除了任何总线数据传输需要。

代码清单 11-7 stream4Mapped 函数节选

```
chTimerGetTime( &chStart );
cudaEventRecord( evStart, 0 );
    saxpyGPU<<<nBlocks, nThreads>>>( dptrOut, dptrX, dptrY, N, alpha );
cudaEventRecord( evStop, 0 );
cudaDeviceSynchronize();
```

映射锁页内存写入主机内存时效果尤为突出（例如，把归约结果传给主机），因为这时与读取操作不同，没有必要等到写操作结束再继续执行。[⊖] 读取映射锁页内存的负载更容易出问题。如果在读取映射锁页内存时，GPU 无法维持全速的总线性能，较小的传输性能可能压倒较小“突出部分”带来的益处。同样，对于某些负载，让 SM 忙于更多的事情比保持（等待）PCIe 总线传输更有意义。

就我们的应用程序而言，在我们的测试系统上，使用映射锁页内存的性能优势明显。

```
Measuring times with 128M floats (use --N to specify number of Mfloats)
Total time: 204.54 ms (7874.45 MB/s)
```

它在 204.54 毫秒内完成计算任务，显著快于次佳的 273 毫秒。7.9GB/s 的高效带宽表明 GPU 充分利用了 PCIe 两个方向的传输潜能。

并不是所有组合使用系统和 GPU 的方式，都可以借助映射锁页内存维持如此高水平的性能。如果有任何异常的端倪，请把数据保持在设备内存并使用异步内存复制策略，类似于 stream2Async.cu。

11.5 性能评价与本章小结

本章介绍了 SAXPY 的四种不同的实现，强调了数据移动的不同策略：

- 设备内存的同步内存传出与传入
- 设备内存的异步内存传出与传入
- 使用流的异步内存复制
- 直接使用映射锁页内存

表 11-1 和图 11-1 总结了这些实现的相对性能。实验针对 128M 个浮点数，实验环境为带有 GK104 显卡的英特尔 i7 平台（PCIe 2.0）和英特尔至强 E5-2670 平台（PCIe 3.0）。PCIe 3.0 的好处是显而易见的，它们约快一倍。此外，E5-2670 CPU/GPU 的同步开销较高，这是因为基于分页的内存复制操作较慢。

[⊖] 硬件设计人员称之为“延迟隐藏”。

表 11-1 流的性能比较

版 本	带宽 (MB/s)	
	INTEL i7 平台	INTEL 沙桥平台
stream1Device.cu	2815	2001
stream2Async.cu	5579	10502
stream3Streams.cu	5820	14051
stream4Mapped.cu	7874	17413

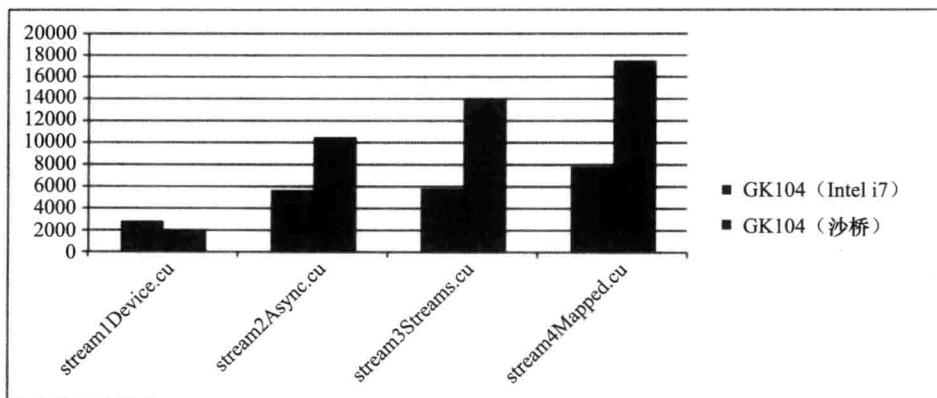
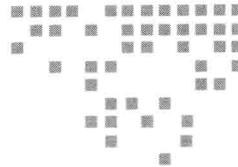


图 11-1 带宽比较 (GeForce GTX 680 在 Intel i7 平台与在沙桥平台)



归约算法

归约 (reduction) 是一类并行算法，对传入的 $O(N)$ 个输入数据，使用一个二元的符合结合律的操作符 \oplus ，生成 $O(1)$ 个结果。这类操作包括取最小、取最大、求和、平方求和、逻辑与、逻辑或、向量点积。归约也是其他高级运算中要用的基础算法，例如将在下一章介绍的扫描。

除非操作符 \oplus 的求解代价极高，否则归约倾向于带宽受限型任务 (bandwidth-bound)。本章对归约的介绍，先基于 SDK 示例 reduction 提供几个两遍方法的实现。接下来，使用 SDK 示例 threadFenceReduction 演示如何采用单遍方法执行归约，单遍方法中只需调用一个内核来执行该操作。最后，本章讨论并总结了如何使用 `_syncthreads_count()` 内置函数 (SM 2.0 加入) 实现快速的二元归约以及如何使用线程束洗牌指令 (SM 3.0 加入) 执行归约。

12.1 概述

因为该二元操作符符合结合律， $O(N)$ 个用于计算归约结果的操作可以以任意顺序执行。

$$\sum_i a_i = a_0 \oplus a_1 \oplus a_2 \oplus a_3 \oplus a_4 \oplus a_5 \oplus a_6 \oplus a_7$$

图 12-1 展示了一些处理 8 元素数组的不同方式。为了方便对比，我们给出它的串行实现。只需要具备一个可以执行 \oplus 操作符的执行单元，但是这种方法的性能比较差，因为它完成计算需要 7 步。

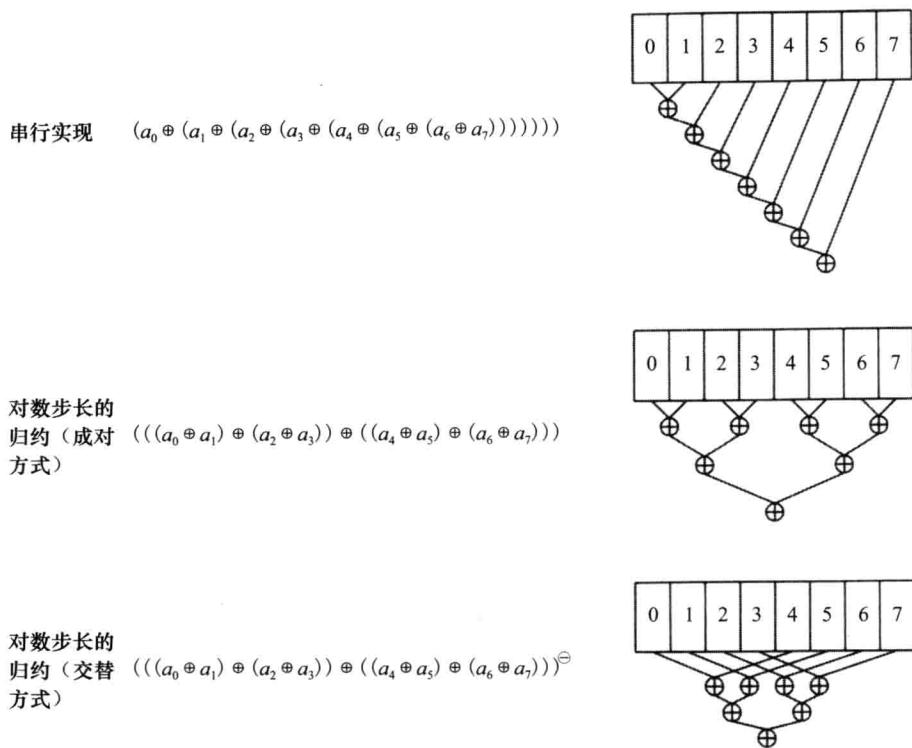


图 12-1 8 元素的归约

成对的方式是直观的，并且只需要 $O(\lg N)$ 步（这里是 3 步）来计算结果，但它在 CUDA 中性能较差。当读取全局内存时，让单个线程访问相邻的内存单元会导致非合并的内存事务。当读取共享内存时，所示的模式会引起存储片冲突（bank conflict）。

不论是对全局内存还是共享内存，基于交替策略效果更好。在图 12-1 中，交替因子为 4。对于全局内存，使用 $\text{blockDim.x} \times \text{gridDim.x}$ 的倍数作为交替因子有良好的性能，因为所有的内存事务将被合并。对于共享内存，最好的性能是按照所确定的交错因子来累计部分和，以避免存储片冲突，并保持线程块的相邻线程处于活跃状态。

一旦一个线程块处理完其交替子数组，将结果写入全局内存以备它随后启动的内核进一步处理。启动多个内核可能看起来开销很大，但内核启动是异步的，因此当 GPU 正在执行第一个内核时，CPU 可以请求下一个内核启动。每个内核启动都有机会来指定不同的启动配置。

鉴于内核的性能可随不同的线程和线程块大小而有所不同，因此编写可以正常工作于任何组合形式的线程和线程块大小的内核，是一个好主意。最佳的线程 / 线程块配置，可以根据经验来确定。

⊖ 原书这里的交替方式的公式写错了，应该改成 $((a_0 \oplus a_4) \oplus (a_1 \oplus a_5)) \oplus ((a_2 \oplus a_6) \oplus (a_3 \oplus a_7))$ 。——译者注

本章的初始归约内核展示一些大家可能熟悉的重要 CUDA 编程概念：

- 合并的内存操作，以最大限度地提高带宽；
- 可变大小的共享内存，以方便线程之间的协作；
- 避免共享内存的存储片冲突。

优化的归约内核展示更先进的 CUDA 编程惯例：

- 线程束同步代码避免了不必要的线程同步；
- 原子操作和内存栅栏避免了调用多个内核的需要；
- 洗牌指令支持线程束级别的归约，而无须使用共享内存。

12.2 两遍归约

该算法包含两个阶段。一个内核执行 NumBlocks 个并行归约，其中 NumBlocks 是指用来调用内核的线程块数，得到的结果写入一个中间数组。最终的结果通过一个线程块第二遍调用同一个内核在中间数组基础上生成。代码清单 12-1 给出了两遍归约内核，它计算一个整数数组的总和。

代码清单 12-1 两遍归约内核

```

__global__ void
Reduction1_kernel( int *out, const int *in, size_t N )
{
    extern __shared__ int sPartials[];
    int sum = 0;
    const int tid = threadIdx.x;
    for ( size_t i = blockIdx.x*blockDim.x + tid;
          i < N;
          i += blockDim.x*gridDim.x ) {
        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    for ( int activeThreads = blockDim.x>>1;
          activeThreads;
          activeThreads >>= 1 ) {
        if ( tid < activeThreads ) {
            sPartials[tid] += sPartials[tid+activeThreads];
        }
        __syncthreads();
    }

    if ( tid == 0 ) {
        out[blockIdx.x] = sPartials[0];
    }
}

void
Reduction1( int *answer, int *partial,
           const int *in, size_t N,
           int numBlocks, int numThreads )

```

```

{
    unsigned int sharedSize = numThreads*sizeof(int);
    Reduction1_kernel<<<
        numBlocks, numThreads, sharedSize>>>(
            partial, in, N );
    Reduction1_kernel<<<
        1, numThreads, sharedSize>>>(
            answer, partial, numBlocks );
}

```

位于共享内存数组用来累计每个线程块内的归约。其大小取决于块的线程数，所以当内核启动时必须指定该值。**注意：**块的线程数必须是 2 的幂次！

第一个 for 循环对输入数组落入每个线程中的元素进行求和。如果输入指针被恰当的对齐，由这段代码发起的全部内存事务将被合并，这将最大限度地提高内存带宽。然后，每个线程把它得到的累计值写入共享内存，并在执行对数步长的归约前进行同步操作。

第二个 for 循环针对共享内存中的值执行对数步长的归约操作。共享内存中前半部分的值被添加到后半部分的值上，而参与的线程数依次减半，直到 shared_sum[0] 中的一个值包含该块的输出。内核的这部分调用需要该块的线程数是 2 的幂次。

最后，线程块的输出值写入全局内存。该内核将被调用两次，如主机函数所示：第一次使用 N 个块，其中 N 选成可以在输入数组上进行最佳性能的归约操作，然后采用 1 个块调用内核来累计出最终输出。代码清单 12-1 展示了调用 Reduction1_kernel() 的主机函数。需要注意的是，这里需要分配一个用于保存部分和的数组并单独传递。还要注意，由于内核使用一个未确定大小的共享内存数组，所需的共享内存数量必须由内核指定为 <<< >>> 语法的第三个参数。

CUDA SDK 针对这个内核讨论了一些优化方案，重点是减少对数步长的归约过程中的条件代码数量。执行对数步长归约的部分 for 循环（循环的后部分，当线程数为 32 或更少时），可以用线程束同步代码实现。由于每个线程块中的线程束是按照锁步方式（lockstep）执行每条指令的，当线程块的活动线程数低于硬件线程束的大小 32 时，无须再调用 __syncthreads() 内置函数。由此产生的内核，位于 reduction2.cu 源代码文件，展示在代码清单 12-2 中。



重要事项 当编写线程束同步的代码时，必须对来自共享内存的指针使用 volatile 关键词修饰。否则，编译器引入的优化行为可能会改变内存操作的顺序并且代码将无法正确工作。

代码清单 12-2 以循环展开的线程束同步作为结束的归约

```

__global__ void
Reduction2_kernel( int *out, const int *in, size_t N )
{
    extern __shared__ int sPartials[];

```

```

int sum = 0;
const int tid = threadIdx.x;
for ( size_t i = blockIdx.x*blockDim.x + tid;
      i < N;
      i += blockDim.x*gridDim.x ) {
    sum += in[i];
}
sPartials[tid] = sum;
__syncthreads();

for ( int activeThreads = blockDim.x>>1;
      activeThreads > 32;
      activeThreads >>= 1 ) {
    if ( tid < activeThreads ) {
        sPartials[tid] += sPartials[tid+activeThreads];
    }
    __syncthreads();
}
if ( threadIdx.x < 32 ) {
    volatile int *wsSum = sPartials;
    if ( blockDim.x > 32 ) wsSum[tid] += wsSum[tid + 32];
    wsSum[tid] += wsSum[tid + 16];
    wsSum[tid] += wsSum[tid + 8];
    wsSum[tid] += wsSum[tid + 4];
    wsSum[tid] += wsSum[tid + 2];
    wsSum[tid] += wsSum[tid + 1];
    if ( tid == 0 ) {
        volatile int *wsSum = sPartials;
        out[blockIdx.x] = wsSum[0];
    }
}
}
}

```

通过把线程数变成一个模板参数，线程束同步的优化方案可以更进一步，从而实现对数步长归约的完全展开。代码清单 12-3 完整地给出了优化的内核。与 Mark Harris 的归约演示一脉相承，^④ 在编译阶段评估的代码以斜体表示。

代码清单 12-3 模板化的完全展开的对数步长的归约

```

template<unsigned int numThreads>
__global__ void
Reduction3_kernel( int *out, const int *in, size_t N )
{
    extern __shared__ int sPartials[];
    const unsigned int tid = threadIdx.x;
    int sum = 0;
    for ( size_t i = blockIdx.x*numThreads + tid;
          i < N;
          i += numThreads*gridDim.x )
    {
        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    if ( numThreads >= 1024 ) {

```

^④ <http://bit.ly/WNmH9Z>。

```

        if (tid < 512) {
            sPartials[tid] += sPartials[tid + 512];
        }
        __syncthreads();
    }
    if (numThreads >= 512) {
        if (tid < 256) {
            sPartials[tid] += sPartials[tid + 256];
        }
        __syncthreads();
    }
    if (numThreads >= 256) {
        if (tid < 128) {
            sPartials[tid] += sPartials[tid + 128];
        }
        __syncthreads();
    }
    if (numThreads >= 128) {
        if (tid < 64) {
            sPartials[tid] += sPartials[tid + 64];
        }
        __syncthreads();
    }

    // warp synchronous at the end
    if (tid < 32) {
        volatile int *wsSum = sPartials;
        if (numThreads >= 64) { wsSum[tid] += wsSum[tid + 32]; }
        if (numThreads >= 32) { wsSum[tid] += wsSum[tid + 16]; }
        if (numThreads >= 16) { wsSum[tid] += wsSum[tid + 8]; }
        if (numThreads >= 8) { wsSum[tid] += wsSum[tid + 4]; }
        if (numThreads >= 4) { wsSum[tid] += wsSum[tid + 2]; }
        if (numThreads >= 2) { wsSum[tid] += wsSum[tid + 1]; }
        if (tid == 0) {
            out[blockIdx.x] = wsSum[0];
        }
    }
}
}

```

为了实例化代码清单 12-3 中的函数模板，它必须通过一个独立的主机函数显式地调用。代码清单 12-4 显示了 Reduction3_kernel 如何被另一个函数模板调用，并且主机函数使用了一个 switch 语句以按照不同线程块大小触发该模板。

代码清单 12-4 展开的归约的模板实例化

```

template<unsigned int numThreads>
void
Reduction3_template( int *answer, int *partial,
                     const int *in, size_t N,
                     int numBlocks )
{
    Reduction3_kernel<numThreads><<<
        numBlocks, numThreads, numThreads*sizeof(int)>>>(
            partial, in, N );
    Reduction3_kernel<numThreads><<<
        1, numThreads, numThreads*sizeof(int)>>>(
            answer, partial, numBlocks );
}

```

```

void
Reduction3( int *out, int *partial,
            const int *in, size_t N,
            int numBlocks, int numThreads )
{
    switch ( numThreads ) {
        case 1: return Reduction3_template< 1>( ... );
        case 2: return Reduction3_template< 2>( ... );
        case 4: return Reduction3_template< 4>( ... );
        case 8: return Reduction3_template< 8>( ... );
        case 16: return Reduction3_template< 16>( ... );
        case 32: return Reduction3_template< 32>( ... );
        case 64: return Reduction3_template< 64>( ... );
        case 128: return Reduction3_template< 128>( ... );
        case 256: return Reduction3_template< 256>( ... );
        case 512: return Reduction3_template< 512>( ... );
        case 1024: return Reduction3_template<1024>( ... );
    }
}

```

12.3 单遍归约

两遍归约的做法是针对 CUDA 线程块无法同步这一问题的部分解决方法。在缺少块间同步机制的条件下，为了确定最终输出的处理何时可以开始，需要调用第二个内核。

使用原子操作和共享内存的组合可避免第二个内核调用，就像 CUDA SDK 的 `threadfenceReduction` 示例那样。使用一个设备内存位置跟踪哪个线程块已经写完自己的部分和。一旦所有块都完成后，一个块进行最后的对数步长的归约，将输出写回。

由于该内核执行多次来自共享内存的对数步长的归约，代码清单 12-3 中根据模板化的线程数进行条件求和的代码被提取出来，放入一个单独的设备函数，以便于重用。

`Reduction4_LogStepShared()` 函数如代码清单 12-5 所示，负责写回线程块的归约值，其部分和由 `partials` 提供给由 `out` 指定的内存位置。代码清单 12-6 给出了使用 `Reduction4_LogStepShared()` 作为子程序的单遍归约算法。

代码清单 12-5 `Reduction4_LogStepShared`

```

template<unsigned int numThreads>
__device__ void
Reduction4_LogStepShared( int *out, volatile int *partials )
{
    const int tid = threadIdx.x;
    if (numThreads >= 1024) {
        if (tid < 512) {
            partials[tid] += partials[tid + 512];
        }
        __syncthreads();
    }
    if (numThreads >= 512) {
        if (tid < 256) {
            partials[tid] += partials[tid + 256];
        }
    }
}

```

```

        __syncthreads();
    }
    if (numThreads >= 256) {
        if (tid < 128) {
            partials[tid] += partials[tid + 128];
        }
        __syncthreads();
    }
    if (numThreads >= 128) {
        if (tid < 64) {
            partials[tid] += partials[tid + 64];
        }
        __syncthreads();
    }

    // warp synchronous at the end
    if (tid < 32) {
        if (numThreads >= 64) { partials[tid] += partials[tid + 32]; }
        if (numThreads >= 32) { partials[tid] += partials[tid + 16]; }
        if (numThreads >= 16) { partials[tid] += partials[tid + 8]; }
        if (numThreads >= 8) { partials[tid] += partials[tid + 4]; }
        if (numThreads >= 4) { partials[tid] += partials[tid + 2]; }
        if (numThreads >= 2) { partials[tid] += partials[tid + 1]; }
        if (tid == 0) {
            *out = partials[0];
        }
    }
}
}

```

代码清单 12-6 单遍归约内核 (reduction4SinglePass.cuh)

```

// Global variable used by reduceSinglePass to count blocks
__device__ unsigned int retirementCount = 0;

template <unsigned int numThreads>
__global__ void
reduceSinglePass( int *out, int *partial,
                  const int *in, unsigned int N )
{
    extern __shared__ int sPartials[];
    unsigned int tid = threadIdx.x;
    int sum = 0;
    for ( size_t i = blockIdx.x*numThreads + tid;
          i < N;
          i += numThreads*gridDim.x ) {
        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    if (gridDim.x == 1) {
        Reduction4_LogStepShared<numThreads>( &out[blockIdx.x],
                                                sPartials );
        return;
    }
    Reduction4_LogStepShared<numThreads>( &partial[blockIdx.x],
                                            sPartials );
    __shared__ bool lastBlock;

    // wait for outstanding memory instructions in this thread
    __threadfence();

    // Thread 0 takes a ticket

```

```

if( tid==0 ) {
    unsigned int ticket = atomicAdd(&retirementCount, 1);

    //
    // If the ticket ID is equal to the number of blocks,
    // we are the last block!
    //
    lastBlock = (ticket == gridDim.x-1);
}
__syncthreads();

// One block performs the final log-step reduction
if( lastBlock ) {
    int sum = 0;
    for ( size_t i = tid;
          i < gridDim.x;
          i += numThreads ) {
        sum += partial[i];
    }
    sPartials[threadIdx.x] = sum;
    __syncthreads();
    Reduction4_LogStepShared<numThreads>( out, sPartials );
    retirementCount = 0;
}
}

```

内核始于熟悉的代码，每个线程计算整个输入数组中的部分归约并将结果写入共享内存。一旦完成上述操作，单个线程块的情况是经过特殊处理的，因为针对共享内存的对数步长的归约输出可以直接写回而不被写入部分和数组。该内核的其余部分只能在多线程块的内核上执行。

共享的布尔变量 `lastBlock` 是用来评估必须传达给最后一个块的所有线程的一个条件。该 `_threadfence()` 导致块中的所有线程都等待，直到任何挂起的内存事务已写回到设备内存。当 `_threadfence()` 被执行时，写入全局内存的操作不仅仅可见于被调用线程或者块内线程，而是可见于所有线程。

当每个线程块退出时，它执行一个 `atomicAdd()` 来检查自己是否是那个需要执行最终对数步长的归约的线程块。由于 `atomicAdd()` 返回内存位置上的前一个值，那个递增 `retirementCount` 并能够得到等于 `gridDim.x-1` 的值的块为“最后一个线程块”^①，它要进行最终的归约。`lastBlock` 这一共享内存位置把该结果传达给该线程块的所有线程，随后必须调用 `_syncthreads()`，这样写回 `lastBlock` 的操作将可见于该块的所有线程。最后一个线程块执行最终的针对部分和的对数步长的归约，并将结果写回。最后，将 `retirementCount` 重新置为 0，以便于后续 `reduceSinglePass()` 的调用。

12.4 使用原子操作的归约

对于那些为硬件的本地原子操作符支持的操作符 \oplus 的归约，编写更简单的归约算法是可

^① 原文误为“最后一个线程”。——译者注

能的：只要循环遍历输入数据并采取“用后即弃”的方式把输入添加到结果所在的内存位置来接收输出值。代码清单 12-7 中给出的 Reduction5_kernel 比以前的方法要简单得多。每个线程计算输入的部分和，并在尾部对输出执行一个 atomicAdd。

需要注意的是，Reduction5_kernel 不能正常工作，除非 out 指向的存储位置初始化为 0。[⊖] 像 threadFenceReduction 示例一样，这个内核的优点是只需要一个内核调用就可完成归约操作。

代码清单 12-7 使用全局内存原子操作的归约 (reduction5Atomics.cuh)

```

__global__ void
Reduction5_kernel( int *out, const int *in, size_t N )
{
    const int tid = threadIdx.x;
    int partialSum = 0;
    for ( size_t i = blockIdx.x*blockDim.x + tid;
          i < N;
          i += blockDim.x*gridDim.x ) {
        partialSum += in[i];
    }
    atomicAdd( out, partialSum );
}

void
Reduction5( int *answer, int *partial,
            const int *in, size_t N,
            int numBlocks, int numThreads )
{
    cudaMemset( answer, 0, sizeof(int) );
    Reduction5_kernel<<< numBlocks, numThreads>>>( answer, in, N );
}

```

12.5 任意线程块大小的归约

到目前为止，所有使用共享内存 的归约实现都要求线程块大小是 2 的幂次。加入少量的额外代码，这些归约算法可工作在任意大小的线程块上。代码清单 12-8 给出了派生于代码清单 12-1 中第一个两遍内核的内核，经过修改，可以在任意大小的块上运行。floorPow2 变量计算小于或等于块大小的 2 的幂次，在执行对数步长的归约循环之前，累计上那些来自超过 2 的幂次的线程的贡献。

代码清单 12-8 任意线程块大小的归约算法 (reduction6AnyBlockSize.cuh)

```

__global__ void
Reduction6_kernel( int *out, const int *in, size_t N )
{
    extern __shared__ int sPartials[];
    int sum = 0;
    const int tid = threadIdx.x;
    for ( size_t i = blockIdx.x*blockDim.x + tid;
          i < N;

```

[⊖] 内核本身不能执行这个初始化，因为 CUDA 执行模型不支持解决不同线程块之间资源竞争的机制。请参见 7.3.1 节。

```

        i += blockDim.x*gridDim.x ) {
    sum += in[i];
}
sPartials[tid] = sum;
__syncthreads();

// start the shared memory loop on the next power of 2 less
// than the block size. If block size is not a power of 2,
// accumulate the intermediate sums in the remainder range.
int floorPow2 = blockDim.x;

if ( floorPow2 & (floorPow2-1) ) {
    while ( floorPow2 & (floorPow2-1) ) {
        floorPow2 &= floorPow2-1;
    }
    if ( tid >= floorPow2 ) {
        sPartials[tid - floorPow2] += sPartials[tid];
    }
    __syncthreads();
}

for ( int activeThreads = floorPow2>>1;
      activeThreads;
      activeThreads >>= 1 ) {
    if ( tid < activeThreads ) {
        sPartials[tid] += sPartials[tid+activeThreads];
    }
    __syncthreads();
}

if ( tid == 0 ) {
    out[blockIdx.x] = sPartials[0];
}
}

```

12.6 适应任意数据类型的归约

到目前为止，我们只开发了可以计算一个整型数组总和的归约内核。为了推广这些内核来执行一组更广泛的操作，我们借助 C++ 模板。除了使用原子操作的算法外，我们见过的所有内核均可修改成带模板的。在本书的配套源代码，它们来自 CUDA 头文件 reduction1Templated.cuh、reduction2Templated.cuh，等等。代码清单 12-9 给出了代码清单 12-1 归约内核的模板化版本。

代码清单 12-9 带模板归约内核

```

template<typename ReductionType, typename T>
__global__ void
Reduction_templated( ReductionType *out, const T *in, size_t N )
{
    SharedMemory<ReductionType> sPartials;
    ReductionType sum;
    const int tid = threadIdx.x;
    for ( size_t i = blockIdx.x*blockDim.x + tid;
          i < N;
          i += blockDim.x*gridDim.x ) {

```

```

        sum += in[i];
    }
    sPartials[tid] = sum;
    __syncthreads();

    for ( int activeThreads = blockDim.x>>1;
          activeThreads;
          activeThreads >>= 1 ) {
        if ( tid < activeThreads ) {
            sPartials[tid] += sPartials[tid+activeThreads];
        }
        __syncthreads();
    }
    if ( tid == 0 ) {
        out[blockIdx.x] = sPartials[0];
    }
}

```

需要注意的是，因为我们希望能够针对一个给定类型的输入计算各种输出类型（例如，我们想构建一个内核，计算整型数组的最小值、最大值、求和或平方求和的任意组合），我们使用两种不同的模板参数：T是待归约的类型，而ReductionType是用于部分和及最终结果的类型。

代码的前几行使用`+=`操作符来检视每个输入元素，为线程块中每个线程累计一个部分和。^④然后严格按照代码清单12-1继续执行，只是这里是运行在ReductionType类型上而不是int类型上。为了避免对齐相关的编译错误，这个内核声明了一个变长的共享内存，这是一个来自CUDA SDK的惯用法。

```

template<class T>
struct SharedMemory
{
    __device__ __inline__ operator      T*()
    {
        extern __shared__ int __smem[];
        return (T*) (void *) __smem;
    }

    __device__ __inline__ operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*) (void *) __smem;
    }
};

```

代码清单12-10显示了一个类的例子，将与诸如Reduction_Templated的带模板归约函数配套使用。这个类计算整型数组的和以及平方和。^⑤除了定义操作符`+=`，还必须声明一个针对SharedMemory模板的特化；否则，编译器会生成以下错误：

```
Error: Unaligned memory accesses not supported
```

在附带的源代码reductionTemplated.cu程序中显示了如何调用来自CUDA头文件的函数模板。

^④ 我们可以很容易地定义一个函数来包装用于归约的二元操作符。Thrust库定义了一个函数操作符plus。

^⑤ 你可以在单遍中计算输入数组的一整套统计量，但作为演示，仅讨论了简单情况。

```
Reductionl<CReduction_Sumi_isq, int>( ... );
```

代码清单 12-10 CReduction_Sumi_isq 类

```
struct CReduction_Sumi_isq {
public:
    CReduction_Sumi_isq();
    int sum;
    long long sumsq;

    CReduction_Sumi_isq& operator +=( int a );
    volatile CReduction_Sumi_isq& operator +=( int a ) volatile;

    CReduction_Sumi_isq& operator +=( const CReduction_Sumi_isq& a );
    volatile CReduction_Sumi_isq& operator +=(
        volatile CReduction_Sumi_isq& a ) volatile;

};

inline __device__ __host__
CReduction_Sumi_isq::CReduction_Sumi_isq()
{
    sum = 0;
    sumsq = 0;
}

inline __device__ __host__
CReduction_Sumi_isq&
CReduction_Sumi_isq::operator +=( int a )
{
    sum += a;
    sumsq += (long long) a*a;
    return *this;
}

inline __device__ __host__
volatile CReduction_Sumi_isq&
CReduction_Sumi_isq::operator +=( int a ) volatile
{
    sum += a;
    sumsq += (long long) a*a;
    return *this;
}

inline __device__ __host__
CReduction_Sumi_isq&
CReduction_Sumi_isq::operator +=( const CReduction_Sumi_isq& a )
{
    sum += a.sum;
    sumsq += a.sumsq;
    return *this;
}

inline __device__ __host__
volatile CReduction_Sumi_isq&
CReduction_Sumi_isq::operator +=(
    volatile CReduction_Sumi_isq& a ) volatile
{
    sum += a.sum;
    sumsq += a.sumsq;
    return *this;
}
```

```

inline int
operator!=( const CReduction_Sumi_isq& a,
             const CReduction_Sumi_isq& b )
{
    return a.sum != b.sum && a.sumsq != b.sumsq;
}

//  

// from Reduction SDK sample:  

// specialize to avoid unaligned memory  

// access compile errors  

//  

template<>
struct SharedMemory<CReduction_Sumi_isq>
{
    __device__ inline operator      CReduction_Sumi_isq*()
    {
        extern __shared__ CReduction_Sumi_isq
        __smem_CReduction_Sumi_isq[];
        return (CReduction_Sumi_isq *)__smem_CReduction_Sumi_isq;
    }

    __device__ inline operator const CReduction_Sumi_isq*() const
    {
        extern __shared__ CReduction_Sumi_isq
        __smem_CReduction_Sumi_isq[];
        return (CReduction_Sumi_isq *)__smem_CReduction_Sumi_isq;
    }
};

```

12.7 基于断定的归约

断定 (predicate) 或真值 (真 / 假) 可以紧凑地表示，因为每个断定只占 1 位。在 SM 2.0 中，英伟达增加了一些指令，使断定操作更有效率。`_ballot()` 和 `_popc()` 内置函数可用于线程束级归约，而 `_syncthreads_count()` 内置函数可用于线程块级归约。

```
int __ballot( int p );
```

`_ballot()` 对线程束内所有线程评估一个条件并返回一个 32 位的字，其中每个位表示线程束相应线程的条件满足情况。由于 `_ballot()` 可以把这一结果广播给线程束的每个线程，它是针对线程束的有效归约。如果想要对线程束能够满足某个条件的线程进行计数，可以调用 `_popc()` 内置函数。

```
int __popc( int i );
```

它会返回输入字中置 1 的位数。SM 2.0 还引入了 `_syncthreads_count()`。

```
int __syncthreads_count( int p );
```

这一内置函数一直等待，直到线程块的所有线程束都到达了，然后把输入条件为真的线程数广播给线程块的所有线程。

由于经过一次线程束级或线程块级归约后，1 位断定立刻变成 5 位和 9 或 10 位。这些内

置函数仅用在最低级评估和归约中，意在减少所需共享内存消耗量。同时，它们极大地扩大了可以由单个线程块考虑的元素的个数。

12.8 基于洗牌指令的线程束归约

SM 3.0 中引入的“洗牌”(shuffle) 指令，如 8.6.1 节中所述，可用于执行对一个线程束中 32 个线程的归约。通过使用洗牌指令的“蝴蝶”(butterfly) 变形，最后 5 个步骤的对数步长的归约：

```
wsSum[tid] += wsSum[tid+16];
wsSum[tid] += wsSum[tid+8];
wsSum[tid] += wsSum[tid+4];
wsSum[tid] += wsSum[tid+2];
wsSum[tid] += wsSum[tid+1];
```

可改写为

```
int mySum = wsSum[tid];
mySum += __shuf_xor( mySum, 16 );
mySum += __shuf_xor( mySum, 8 );
mySum += __shuf_xor( mySum, 4 );
mySum += __shuf_xor( mySum, 2 );
mySum += __shuf_xor( mySum, 1 );
```

然后，线程束中的所有线程把得到的归约值保存于 mySum。图 12-2 演示了这一线程束扫描原语操作。每个线程的和显示为 $4W \times 8H$ 矩形，用深色方形指示哪些线程对每个线程的部分和做出了贡献。(除去插图，最上面一行显示了对应于每个线程的贡献的方形。) 对数步长归约每前进一步，贡献的数量加倍，直到每个线程都拥有一个完整的归约值。[⊖]

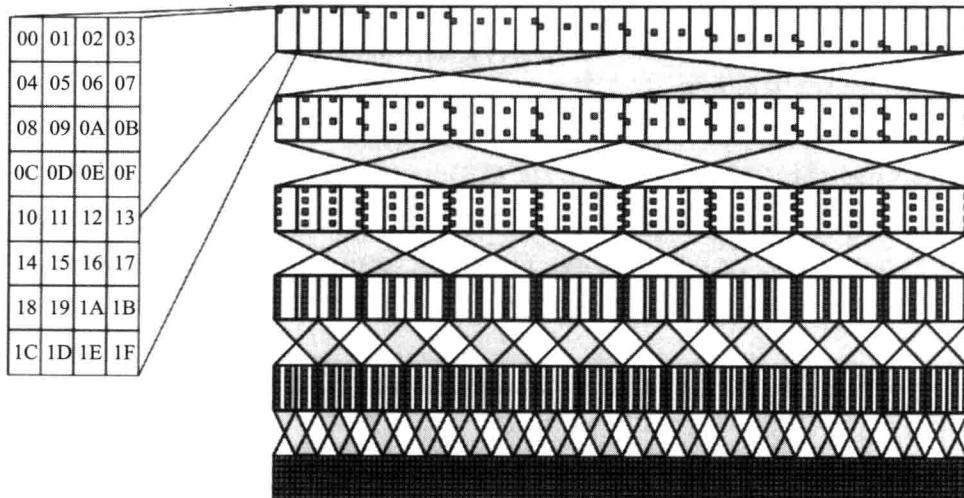


图 12-2 使用洗牌指令的归约

[⊖] 向上洗牌 (shuffle-up) 或向下洗牌 (shuffle-down) 的变体可被用来实现归约，但它们需要的次数与蝴蝶 (XOR) 变体一样，并且最终的归约值仅保存在一个线程中。

扫描算法

扫描 (Scan), 也称为前缀扫描 (prefix scan)、前缀求和 (prefix sum) 或并行前缀求和 (parallel prefix sum), 是并行编程的一个重要原语, 并作为基本模块用于许多不同的算法, 包括很多算法但不限于以下内容:

- 基数排序 (radix sort)
- 快速排序 (quicksort)
- 流压缩 (stream compaction) 和流拆分 (stream splitting)
- 稀疏矩阵与向量的乘法
- 最小生成树构建
- 面积表的求和计算

本章首先对该算法和它的一些变形予以描述, 讨论一个早期的实现策略以及扫描算法如何使用电路图进行描述, 然后提供 CUDA 扫描算法实现的细节。参考文献部分同时给出了硬件设计中的扫描算法和并行前缀求和电路问题。

13.1 定义与变形

包容性扫描 (inclusive scan) 需要一个符合结合律的二元操作符和长度为 N 的数组

$$[a_0, a_1, \dots, a_{N-1}]$$

并返回如下数组

$$[a_0, (a_0 \oplus a_1), \dots, (a_0 \oplus a_1 \oplus \dots \oplus a_{N-1})]$$

可以看出，输出的每个元素依赖于输入中前一位置元素。

排他性扫描（exclusive scan）被类似地定义，但对输出作了移位，并使用一个单位元素 $id \oplus$ （单位元素同某个值执行操作符 \oplus 的运算，不会对这个值产生任何改变，例如，0 为整数加法的单位元素，1 为乘法的单位元素等）。

$$[id \oplus, a_0, a_0 \oplus a_1, \dots, a_0 \oplus a_1 \oplus \dots \oplus a_{N-2}]$$

包容性和排他性扫描可以通过依次添加或减去输入数组的元素而相互转化，如图 13-1 所示。

流压缩是根据一定标准提取数组中元素的操作。如果一个断定（0 或 1）被用于判断输入数组的每个元素是否应该包含在输出流中，那么接着可以按照此断定执行一个排他性扫描来计算输出的元素的索引。一个流压缩的变形，被称为流拆分，按照每个断定值独立的写回紧凑的输出。分段扫描（segmented scan）也是一种变形。它在输入数组的基础上配有一组输入标志（每个数组元素对应一个标志），并在由标志分割的子阵列上执行扫描。

鉴于扫描原语的重要性，在实现优化的 CUDA 扫描算法方面，已投入了大量努力。在本章结尾处，提供了一个参考文献列表。无论是 CUDPP，还是 Thrust 库，都包括一组优化了的扫描原语，它们使用模板达到通用性和性能之间的最佳折衷。然而，总体而言，使用扫描原语的应用程序通常能够从“充分利用了具体问题的领域知识的自定义实现”中获益。

13.2 概述

一个简单的 C++ 实现如代码清单 13-1 所示。

代码清单 13-1 包容性扫描代码（C++）

```
template<class T>
T
InclusiveScan( T *out, const T *in, size_t N )
{
    T sum(0);
    for ( size_t i = 0; i < N; i++ ) {
        sum += in[i];
        out[i] = sum;
    }
    return sum;
}
```

对于代码清单 13-1 和 13-2 这些串行实现，包容性和排他性扫描之间的唯一区别在于如下行

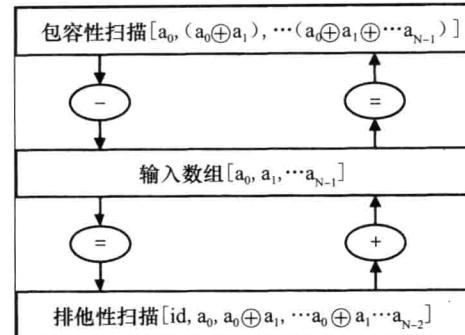


图 13-1 包容性和排他性扫描

```
out[i] = sum;
```

和

```
sum += in[i];
```

作了交换。[⊖]

代码清单 13-2 排他性扫描代码 (C++)

```
template<class T>
T
ExclusiveScan( T *out, const T *in, size_t N )
{
    T sum(0);
    for (size_t i = 0; i < N; i++) {
        out[i] = sum;
        sum += in[i];
    }
    return sum;
}
```

扫描的串行实现是如此显而易见而又微不足道，如果你拿不准它的并行实现，是可以理解的！所谓的前缀依赖性，即其中每个输出依赖于前面的所有输入。由于扫描的前缀依赖性，可能有些人甚至会怀疑它是否能够并行。但是，经过思考，你可以看到，相邻两个数的操作 ($a_i \oplus a_{i+1} \quad 0 \leq i < N-1$) 可以并行的计算，对于 $i=0$, $a_i \oplus a_{i+1}$ 计算扫描的最终输出，否则，这些针对一对数的操作计算部分和，并在后面归入最终输出，就像我们在第 12 章中那样使用部分和。

Blelloch[⊖]描述了一个两遍算法。其中自底向上阶段 (upsweep phase) 计算数组的归约，并依次存储中间结果；随后的自顶向下阶段 (downsweep phase) 计算扫描的最终输出。自底向上阶段的伪代码如下。

```
upsweep(a, N)
for d from 0 to (lg N) - 1
    in parallel for i from 0 to N - 1 by  $2^{d+1}$ 
        a[i +  $2^{d+1} - 1$ ] += a[i +  $2^d - 1$ ]
```

该操作类似于之前讨论过的对数步长的归约，不同之处在于部分和要存储起来，以备后面用到产生扫描的最终输出中去。

按照 Blelloch 的方法，图 13-2 显示了一个使用该自底向上代码运行的例子，该例子对 8 个元素构成的数组施行整数加法运算。“自底向上”这一术语源于将数组看作一颗平衡树（见图 13-3）。

一旦完成自底向上阶段，一个自顶向下阶段把“中间和”传播到树的叶子。自顶向下阶段的伪码如下。

[⊖] 成书之际，排他性扫描的实现还不支持原地计算 (in-place computation)。为了让输入和输出数组保持不变， $in[i]$ 必须保存到一个临时变量中。

[⊖] <http://bit.ly/YmTmGP>。

```

downsweep(a, N)
    a[N-1] = 0
    for d from (lg N)-1 downto 0
        in parallel for i from 0 to N-1 by 2d+1
            t := a[i+2d-1]
            a[i+2d-1] = a[i + 2d+1-1]
            a[i+2d+1-1] += t

```

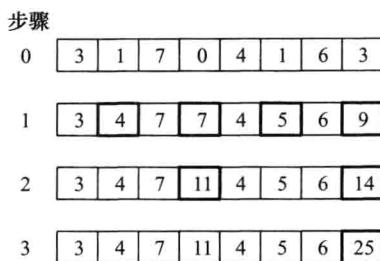


图 13-2 自底向上阶段（数组视图）

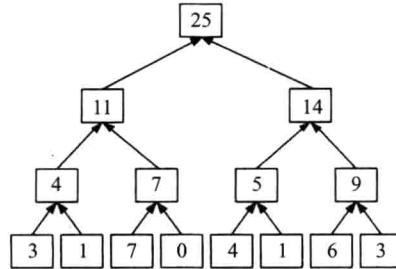


图 13-3 自底向上阶段（树视图）

图 13-4 显示了示例数组是如何在自顶向下阶段进行转化的，而图 13-5 以树的形式显示了自顶向下阶段的过程。CUDA 早期实现的扫描算法紧密遵循了这一算法，而且它在促使大家思考扫描算法的实现上起到了很好的作用。遗憾的是，这一算法跟 CUDA 架构的匹配存在瑕疵，简单地实现它会导致共享内存的存储片冲突，并且为了弥补冲突而使用的寻址方案，会导致得不偿失的开销。

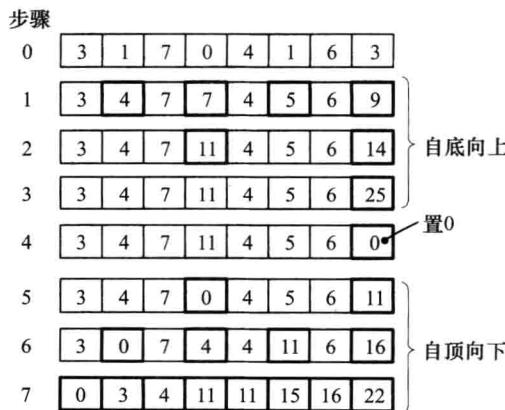


图 13-4 自顶向下阶段（数组视图）

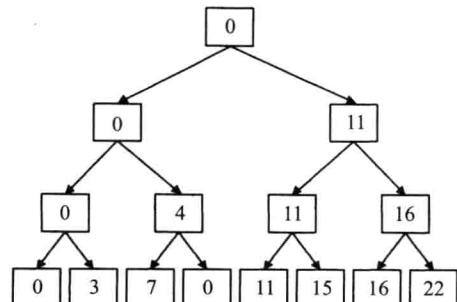


图 13-5 自顶向下阶段（树视图）

13.3 扫描和电路设计

经过分析扫描的一种并行算法，现在就变得明朗了，我们可以找到许多不同的方法来实现并行扫描算法。为指导其他可能的实现，我们可以借助执行类似功能的整型加法硬件的设

计方法：不是传递一个任意的二元可结合操作符 \oplus 给数组，然后使用归约获得最终输出，而是硬件加法器传播部分加法结果，连带进位，可以传播给多倍精度运算。

硬件设计师使用有向无环、定向图来表示扫描“电路”的不同实现。这些示意图简洁地表达出数据流和并行性。代码清单 13-1 这一串行实现的示意图在图 13-6 中给出。随时间推移，不断向下推进；垂直直线表示导线，信号通过它进行传播。在入度为 2 的节点（“操作节点”）上，对它们的输入应用操作符 \oplus 。

注意，该电路图展示的是包容性扫描，而不是排他性扫描。对于电路图而言，包容性扫描与排他性扫描之间的区别甚微。欲把图 13-6 中的包容性扫描转换成一个排他性扫描，把一个 0 连接到第一个输出，并把随后求得的和依次连接到输出，如图 13-7 所示。注意到不论是包容性还是排他性扫描都是把针对输入数组生成的归约作为输出，这一特性将用于构建高效的扫描算法。（为清楚起见，除了图 13-7 之外的所有电路图仅绘制包容性扫描。）

Blelloch 描述的扫描算法对应于 Brent-Kung 电路设计法，采用一个递归分解，其中每隔一个输出被送入只有当前一半宽度的 Brent-Kung 电路。图 13-8 展示了长度为 8 的 Brent-Kung 电路的例子，同时伴随有 Blelloch 的自底向上阶段和自顶向下阶段。请注意，下一步中把输出广播到多个节点的行为称为扇出（fan out）。Brent-Kung 电路的值得注意的特点是具有恒定的 2 个扇出分支。

Brent-Kung 电路的结构对大型电路变得更清晰。请考察如图 13-9 所示包含 16 个输入的电路。图 13-9 也加粗了电路的“脊椎”，该脊椎是用于生成扫描输出最后一个元素的最长子图。

Brent-Kung 电路的深度随输入的数量呈对数增长，展示了比串行算法更强的效率。但是由于递归分解的每一步电路深度会增加 2，Brent-Kung 电路的深度不是最小深度。Sklansky 描述了一个构建最小深度电路的方法，像图 13-10 那样递归的分解它们。

2 个输入为 $N/2$ 的电路被并行运行，并且左侧电路的脊柱的输出被添加到右侧电路中的每个元素。对于我们 16 个元素的例子，该递归的左侧子图在图 13-10 中被特别框出。

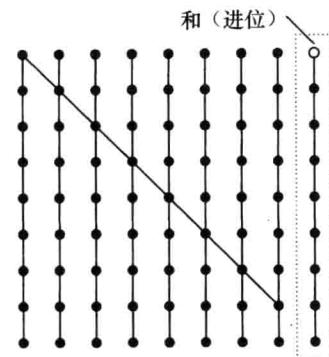


图 13-6 串行扫描

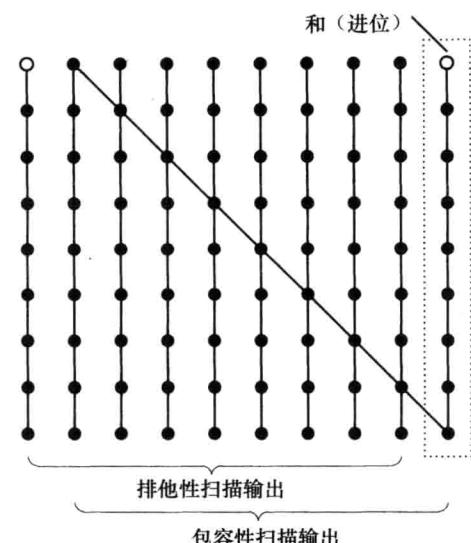


图 13-7 串行扫描（包容性和排他性）

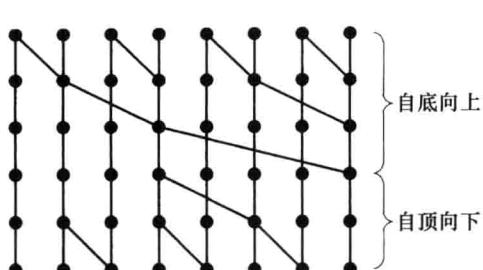


图 13-8 Brent-Kung 电路

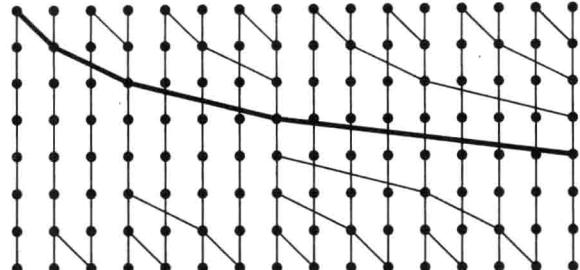


图 13-9 Brent-Kung 电路 (16 个输入数据)

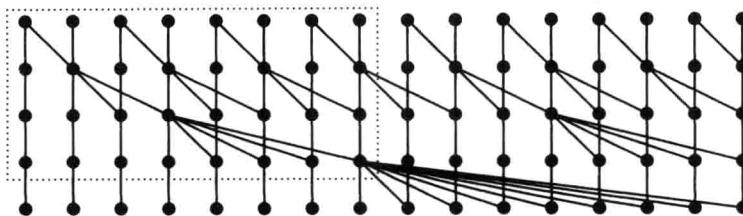


图 13-10 Sklansky (最小深度) 电路

另一种最小深度的扫描电路，称为 Kogge-Stone 电路。它具有恒定的 2 个扇出分支，这是硬件实现最期望的特性。但是，正如你在图 13-11 所见，它有许多操作节点，模拟 Kogge-Stone 电路的软件实现效率低下。

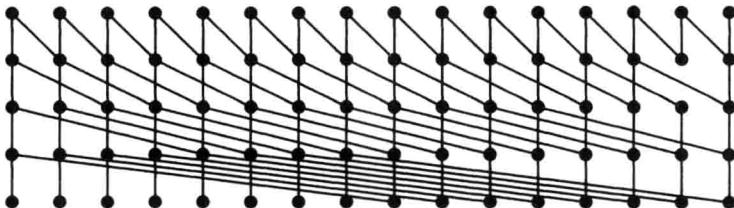


图 13-11 Kogge-Stone 电路

任何扫描电路都可以通过组合扫描（执行并行的前缀求和计算并生成输入数组的和作为输出）和扇出（把一个输入添加到剩余的每个输出）来构成。如图 13-10 所示具有最小深度的电路在它们的递归定义中大量使用了扇出操作。

要得到一个优化的 CUDA 实现，一个重要指导思想是，扇出操作不需要从扫描本身得到它的输入，任何归约都能做到。而在第 12 章，我们已经拥有了一个高度优化的归约算法。

例如，如果我们把输入数组分割成长度为 b 的子数组，并使用我们优化了的归约程序计算每个子数组的总和，我们最终得到一个由 $\lceil N/b \rceil$ 个归约值构成的数组。如果我们在该数组上执行排他性扫描，得到的数组变成了扇出操作的输入（种子），用于对应子数组的扫描。通过在全局内存的一遍扫描可以有效扫描到的值受制于 CUDA 的线程块大小和共享内存大小，因此对于较大的输入，这种方法必须递归的应用。

13.4 CUDA 实现

设计扫描算法和学习电路图是很有启发的，但为了实现 CUDA 的扫描算法，我们需要将算法映射到寄存器、内存和寻址方案以及正确的同步。扫描的最佳 CUDA 实现依赖于将要执行的扫描的规模。针对线程束大小的扫描、可以放入共享内存的扫描以及必然会溢出到全局内存的扫描，要使用不同的方案。因为线程块之间不能可靠地通过全局内存交换数据，扫描因太大而无法放入共享内存的数据时，必须执行多个内核启动。[⊖]

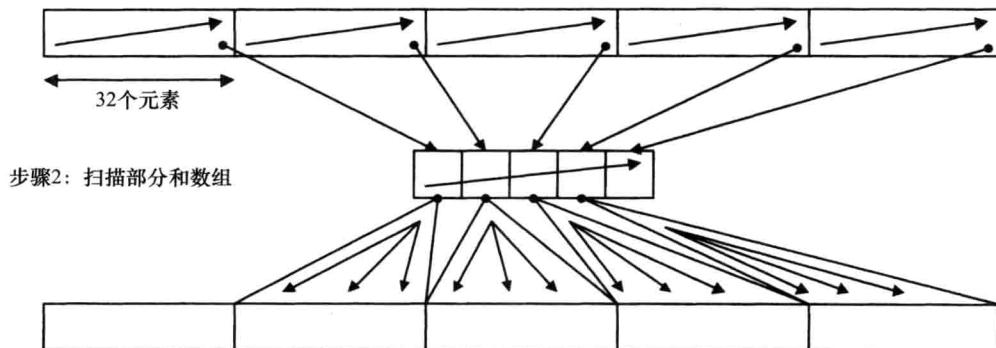
在考察特殊情况（如断定的扫描）之前，我们将研究 3 种在 CUDA 中实现扫描的方法：

- 先扫描再扇出（递归）；
- 先归约再扫描（递归）；
- 两阶段先归约再扫描。

13.4.1 先扫描再扇出

先扫描再扇出的方法对于全局内存和共享内存，均使用类似的分解方案。图 13-12 显示了用于扫描一个线程块的方法：每 32 线程构成的线程束执行一次扫描，并且该 32 个元素构成的子数组的归约值写入到共享内存。随后，单个线程束扫描数组的部分和。只需使用单个线程束就已足够，是因为 CUDA 不支持超过 1024 个线程的线程块。最后，基本和被扇出给每个线程束的输出元素。注意，图 13-12 中显示的步骤 2 所执行是包容性扫描，因此输出的第一个元素必须扇出给第二个线程束，依此类推。

步骤1：扫描线程束并把每个归约和写入部分和的共享数组



步骤3：扇出基本和到相应的子数组

图 13-12 先扫描再扇出（共享内存）

实现这个算法的代码如代码清单 13-3。假设输入数组已经加载到共享内存并且参数 `sharedPartials` 和 `idx` 分别指定了待扫描线程束的基地址和索引。（在我们第一次实现的版本，`threadIdx.x` 被作为参数 `idx` 传递）。代码的 9 ~ 13 行实现图 13-12 中的步骤 1；16 ~ 21 行实

[⊖] 使用 CUDA 5.0 和 SM 3.5 硬件，动态并行可以将多数内核启动转移到“子网格”，而不是由主机发起的内核启动。

现步骤2；31~45行实现步骤3。这个线程写回的输出值返回给调用者，但仅当它正好是该线程块的归约时才使用。

代码清单 13-3 scanBlock：针对线程块进行先扫描再扇出的线程块部分

```

template<class T>
inline __device__ T
scanBlock( volatile T *sPartials )
{
    extern __shared__ T warpPartials[];
    const int tid = threadIdx.x;
    const int lane = tid & 31;
    const int warpid = tid >> 5;

    //
    // Compute this thread's partial sum
    //
    T sum = scanWarp<T>( sPartials );
    __syncthreads();

    //
    // Write each warp's reduction to shared memory
    //
    if ( lane == 31 ) {
        warpPartials[16+warpid] = sum;
    }
    __syncthreads();

    //
    // Have one warp scan reductions
    //
    if ( warpid==0 ) {
        scanWarp<T>( 16+warpPartials+tid );
    }
    __syncthreads();
    //
    // Fan out the exclusive scan element (obtained
    // by the conditional and the decrement by 1)
    // to this warp's pending output
    //
    if ( warpid > 0 ) {
        sum += warpPartials[16+warpid-1];
    }
    __syncthreads();

    //
    // Write this thread's scan output
    //
    *sPartials = sum;
    __syncthreads();

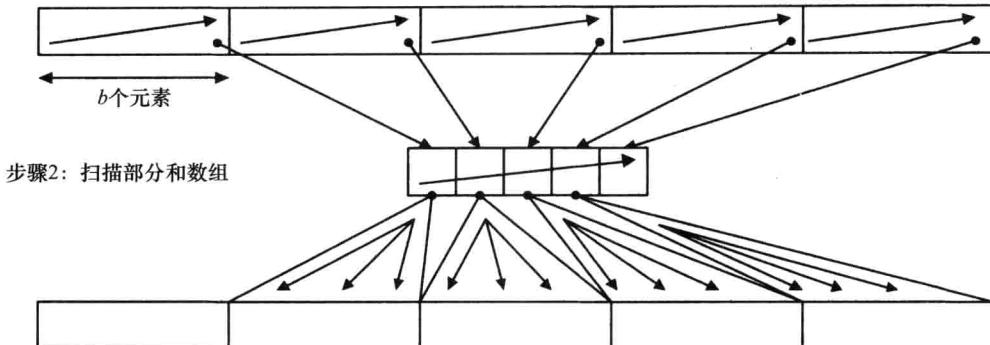
    //
    // The return value will only be used by caller if it
    // contains the spine value (i.e., the reduction
    // of the array we just scanned).
    //
    return sum;
}

```

图13-13显示了这种方法是如何应用到全局内存的。一个内核扫描 b 个元素构成的子数组，这里 b 是线程块大小。部分和被写入到全局内存；而另一个仅使用一个块启动的内核扫

描这些部分和，然后把它们扇出到全局内存的最终输出。

步骤1：扫描子数组并把每个归约和写入部分和的全局数组



步骤3：扇出基本和到最终输出数组

图 13-13 先扫描再扇出（全局内存）

代码清单 13-4 给出了图 13-13 步骤 1 中扫描内核的 CUDA 代码。它遍历全部线程块并进行处理，把输入数组通过共享内存进行中转。然后，内核根据需要，决定是否把脊柱上得到的值最终写回全局内存。在递归的底部，没有必要记录脊柱值，所以 bWriteSpine 模板参数使内核避免动态检查 partialsOut 的值。

代码清单 13-4 scanAndWritePartials

```
template<class T, bool bWriteSpine>
__global__ void
scanAndWritePartials(
    T *out,
    T *gPartials,
    const T *in,
    size_t N,
    size_t numBlocks )
{
    extern volatile __shared__ T sPartials[];
    const int tid = threadIdx.x;
    volatile T *myShared = sPartials+tid;

    for ( size_t iBlock = blockIdx.x;
          iBlock < numBlocks;
          iBlock += gridDim.x ) {
        size_t index = iBlock*blockDim.x+tid;

        *myShared = (index < N) ? in[index] : 0;
        __syncthreads();

        T sum = scanBlock( myShared );
        __syncthreads();
        if ( index < N ) {
            out[index] = *myShared;
        }
    }
    // write the spine value to global memory
}
```

```

        if ( bWriteSpine && (threadIdx.x==(blockDim.x-1)) )
        {
            gPartials[iBlock] = sum;
        }
    }
}

```

代码清单 13-5 给出了一个主机函数，它使用代码清单 13-3 和 13-4 来实现来自全局内存上数组的包容性扫描。注意，对于扫描过大无法在共享内存执行的情况，该函数递归的进行扫描。该函数的第一个条件既作为递归的标准条件，也对那些小到可以放入共享内存执行的小电路扫描问题，免去分配全局内存的需要。注意内核所需的共享内存量 ($b * \text{sizeof}(T)$) 是在内核启动时指定的。

对于更大规模的扫描，该函数计算所需的部分和的数目 $\lceil N/b \rceil$ 、分配全局内存来保存它们并遵循图 3-13 的模式，把部分和写回到全局数组，以备由后面的 scanAndWritePartials 内核（见代码清单 13-4）使用。

递归的每一层下来，待处理的元素数降低为原来的 $1/b$ ，因此对于 $b=128$ 和 $N=1048576$ 的例子中，需要两层递归：1 个是大小为 8192 规模的，另一个是大小为 64 规模的。

代码清单 13-5 scanFan 主机函数

```

template<class T>
void
scanFan( T *out, const T *in, size_t N, int b )
{
    cudaError_t status;
    if ( N <= b ) {
        scanAndWritePartials<T, false><<<1,b,b*sizeof(T)>>>( 
            out, 0, in, N, 1 );
        return;
    }

    //
    // device pointer to array of partial sums in global memory
    //
    T *gPartials = 0;
    //
    // ceil(N/b)
    //
    size_t numPartials = (N1)/b;
    //
    // number of CUDA threadblocks to use. The kernels are
    // blocking agnostic, so we can clamp to any number
    // within CUDA's limits and the code will work.
    //
    const unsigned int maxBlocks = 150;    // maximum blocks to launch
    unsigned int numBlocks = min( numPartials, maxBlocks );
    CUDART_CHECK( cudaMalloc( &gPartials,
                                numPartials*sizeof(T) ) );
    scanAndWritePartials<T, true><<<numBlocks,b,b*sizeof(T)>>>( 
        out, gPartials, in, N, numPartials );
    scanFan<T>( gPartials, gPartials, numPartials, b );
    scanAddBaseSums<T><<<numBlocks, b>>>( out, gPartials, N,
        numPartials );
    Error:
        cudaFree( gPartials );
}

```

代码清单 13-6 使用一个简单的内核，把来自全局内存的结果扇出到全局内存，完成整个算法。

代码清单 13-6 scanAddBaseSums 内核

```
template<class T>
__global__ void
scanAddBaseSums(
    T *out,
    T *gBaseSums,
    size_t N,
    size_t numBlocks )
{
    const int tid = threadIdx.x;

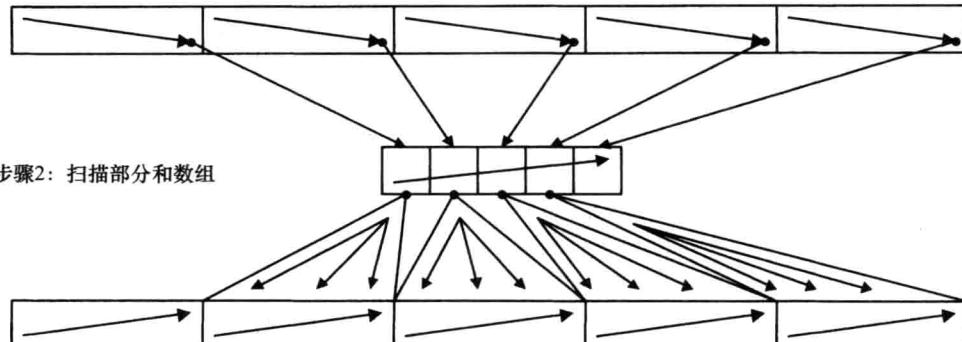
    T fan_value = 0;
    for (size_t iBlock = blockIdx.x;
        iBlock < numBlocks;
        iBlock += gridDim.x ) {
        size_t index = iBlock*blockDim.x+tid;
        if (iBlock > 0) {
            fan_value = gBaseSums[iBlock-1];
        }
        out[index] += fan_value;
    }
}
```

在递归的最高层，先扫描再扇出的策略会执行 $4N$ 次全局内存操作。初始扫描执行一次读和一次写，然后代码清单 13-4 的扇出操作执行另一次读和另一次写。我们可以通过首先只计算输入数组的归约来减少全局内存的写入次数。

13.4.2 先归约再扫描（递归）

图 13-14 显示了这一策略是如何工作的。如前，计算出输入的部分和数组 ($\lceil N/b \rceil$ 个元素)，并对它进行扫描以获得一个基本和数组。但不是在第一阶段做扫描操作，相反，我们在第一阶段只计算部分和。然后把基本和依次加入，完成最终输出的扫描。

步骤1：计算每个子数组的归约值并把写入部分和的全局数组



步骤3：扫描子数组的输出数组，把相应的部分和添加到输出数组

图 13-14 先归约再扫描

代码清单 13-7 给出了用于计算部分和数组的代码，它把来自代码清单 12-3 的归约代码作为子程序调用。与归约代码一样，内核按照线程块大小进行了模板化处理，并且一个包装模板使用一个 switch 语句来调用模板的实例。

代码清单 13-7 scanReduceBlocks

```

template<class T, int numThreads>
__global__ void
scanReduceBlocks( T *gPartials, const T *in, size_t N )
{
    extern volatile __shared__ T sPartials[];

    const int tid = threadIdx.x;
    gPartials += blockIdx.x;
    for ( size_t i = blockIdx.x*blockDim.x;
          i < N;
          i += blockDim.x*gridDim.x ) {
        size_t index = i+tid;
        sPartials[tid] = (index < N) ? in[index] : 0;
        __syncthreads();

        reduceBlock<T,numThreads>( gPartials, sPartials );
        __syncthreads();
        gPartials += gridDim.x;
    }
}

template<class T>
void
scanReduceBlocks(
    T *gPartials,
    const T *in,
    size_t N,
    int numThreads,
    int numBlocks )
{
    switch ( numThreads ) {
        case 128: return scanReduceBlocks<T, 128> ... ( ... );
        case 256: return scanReduceBlocks<T, 256> ... ( ... );
        case 512: return scanReduceBlocks<T, 512> ... ( ... );
        case 1024: return scanReduceBlocks<T,1024> ... ( ... );
    }
}

```

代码清单 13-8 给出了用于执行扫描的内核。与代码清单 13-4 的主要区别在于，不是把输入子数组之和写入全局内存，相反，内核把每个子数组对应的基本和在写入之前添加于输出元素。

代码清单 13-8 scanWithBaseSums

```

template<class T>
__global__ void
scanWithBaseSums(
    T *out,
    const T *gBaseSums,
    const T *in,
    size_t N,

```

```

        size_t numBlocks )
{
    extern volatile __shared__ T sPartials[];
    const int tid = threadIdx.x;
    for ( size_t iBlock = blockIdx.x;
            iBlock < numBlocks;
            iBlock += blockDim.x ) {
        T base_sum = 0;
        size_t index = iBlock*blockDim.x+tid;

        if ( iBlock > 0 && gBaseSums ) {
            base_sum = gBaseSums[iBlock-1];
        }
        sPartials[tid] = (index < N) ? in[index] : 0;
        __syncthreads();

        scanBlock( sPartials+tid );
        __syncthreads();
        if ( index < N ) {
            out[index] = sPartials[tid]+base_sum;
        }
    }
}

```

采用先归约再扫描策略的主机代码如代码清单 13-9 所示。在归约的最高层，先归约后扫描策略进行 $3N$ 次全局内存操作。归约的第一遍每个元素执行一次读，然后代码清单 13-9 中的扫描操作执行另一次读和一次写。与先扇出再扫描策略类似，递归的每一层处理的元素数都只是上一层的 $1/b$ 。

代码清单 13-9 scanReduceThenScan

```

template<class T>
void
scanReduceThenScan( T *out, const T *in, size_t N, int b )
{
    cudaError_t status;

    if ( N <= b ) {
        return scanWithBaseSums<T><<<1,b,b*sizeof(T)>>>( 
            out, 0, in, N, 1 );
    }

    //
    // device pointer to array of partial sums in global memory
    //
    T *gPartials = 0;

    //
    // ceil(N/b) = number of partial sums to compute
    //
    size_t numPartials = (N1)/b;
    //
    // number of CUDA threadblocks to use. The kernels are blocking
    // agnostic, so we can clamp to any number within CUDA's limits
    // and the code will work.
    //
    const unsigned int maxBlocks = 150;
    unsigned int numBlocks = min( numPartials, maxBlocks );

```

```

CUDART_CHECK( cudaMalloc( &gPartials, numPartials*sizeof(T) ) );

scanReduceBlocks<T>( gPartials, in, N, b, numBlocks );
scanReduceThenScan<T>( gPartials, gPartials, numPartials, b );
scanWithBaseSums<T><<<numBlocks,b,b*sizeof(T)>>>(
    out,
    gPartials,
    in,
    N,
    numPartials );
Error:
    cudaFree( gPartials );
}

```

13.4.3 先归约再扫描（两阶段）

Merrill[⊖]描述了另一种扫描方法，使用了少量固定大小的基本和。该算法与图 13-14 几乎完全相同，不同之处在于步骤 2 中的数组相对较小，可能只有数百个固定大小而不是 $\lceil N/b \rceil$ 个部分和构成。不论是在归约阶段还是扫描阶段，部分和的数目与使用的线程块数量相同。代码清单 13-10 显示了计算这些部分和的代码，它被更新为计算大小为 elementsPerPartial 个元素而不是线程块大小个元素的子数组的归约。⁴

代码清单 13-10 scanReduceSubarrays

```

template<class T, int numThreads>
__device__ void
scanReduceSubarray(
    T *gPartials,
    const T *in,
    size_t iBlock,
    size_t N,
    int elementsPerPartial )
{
    extern volatile __shared__ T sPartials[];
    const int tid = threadIdx.x;

    size_t baseIndex = iBlock*elementsPerPartial;

    T sum = 0;
    for ( int i = tid; i < elementsPerPartial; i += blockDim.x ) {
        size_t index = baseIndex+i;
        if ( index < N )
            sum += in[index];
    }
    sPartials[tid] = sum;
    __syncthreads();

    reduceBlock<T,numThreads>( &gPartials[iBlock], sPartials );
}

/*
 * Compute the reductions of each subarray of size
 * elementsPerPartial, and write them to gPartials.
 */

```

[⊖] <http://bit.ly/ZKtlh1>。

```

template<class T, int numThreads>
__global__ void
scanReduceSubarrays(
    T *gPartials,
    const T *in,
    size_t N,
    int elementsPerPartial )
{
    extern volatile __shared__ T sPartials[];

    for ( int iBlock = blockIdx.x;
          iBlock*elementsPerPartial < N;
          iBlock += gridDim.x )
    {
        scanReduceSubarray<T,numThreads>(
            gPartials,
            in,
            iBlock,
            N,
            elementsPerPartial );
    }
}

```

代码清单 13-11 提供了扫描代码，已被修改来在一个线程块完成扫描操作后执行对每块中和的扫描操作。代码清单 13-11 中的 bZeroPad 模板参数和使用它的 scanSharedIndex 辅助函数将在 13.5.1 节予以详描。

代码清单 13-11 scan2Level_kernel

```

template<class T, bool bZeroPad>
__global__ void
scan2Level_kernel(
    T *out,
    const T *gBaseSums,
    const T *in,
    size_t N,
    size_t elementsPerPartial )
{
    extern volatile __shared__ T sPartials[];
    const int tid = threadIdx.x;
    int sIndex = scanSharedIndex<bZeroPad>( threadIdx.x );

    if ( bZeroPad ) {
        sPartials[sIndex-16] = 0;
    }
    T base_sum = 0;
    if ( blockIdx.x && gBaseSums ) {
        base_sum = gBaseSums[blockIdx.x-1];
    }
    for ( size_t i = 0;
          i < elementsPerPartial;
          i += blockDim.x ) {
        size_t index = blockIdx.x*elementsPerPartial + i + tid;
        sPartials[sIndex] = (index < N) ? in[index] : 0;
        __syncthreads();

        scanBlock<T,bZeroPad>( sPartials+sIndex );
        __syncthreads();
        if ( index < N ) {
            out[index] = sPartials[sIndex]+base_sum;
        }
    }
}

```

```

    __syncthreads();

    // carry forward from this block to the next.
    base_sum += sPartials[
        scanSharedIndex<bZeroPad>( blockDim.x-1 ) ];
    __syncthreads();
}
}

```

代码清单 13-12 给出了 Merrill 的两阶段先归约再扫描算法的主机代码。由于计算出的部分和数目很小而且从不改变，主机代码永远不用为了执行扫描而分配全局内存，相反，我们声明一个在模块加载时分配空间的 `_device_` 数组。

`_device_ int g_globalPartials[MAX_PARTIALS];`

并通过调用 `cudaGetSymbolAddress()` 获取其地址。

```

status = cudaGetSymbolAddress(
    (void **) &globalPartials,
    g_globalPartials );

```

该程序然后计算每个部分和包含的元素数目以及要使用的线程块数目，并调用执行计算所需的 3 个内核。

代码清单 13-12 scan2Level

```

template<class T, bool bZeroPad>
void
scan2Level( T *out, const T *in, size_t N, int b )
{
    int sBytes = scanSharedMemory<T,bZeroPad>( b );

    if ( N <= b ) {
        return scan2Level_kernel<T, bZeroPad><<<1,b,sBytes>>>(
            out, 0, in, N, N );
    }

    cudaError_t status;
    T *gPartials = 0;
    status = cudaGetSymbolAddress(
        (void **) &gPartials,
        g_globalPartials );

    if ( cudaSuccess == status )
    {
        //
        // ceil(N/b) = number of partial sums to compute
        //
        size_t numPartials = (N+b-1)/b;

        if ( numPartials > MAX_PARTIALS ) {
            numPartials = MAX_PARTIALS;
        }

        //
        // elementsPerPartial has to be a multiple of b
        //
        unsigned int elementsPerPartial =
            (N+numPartials-1)/numPartials;
        elementsPerPartial = b * ((elementsPerPartial+b-1)/b);
    }
}

```

```

numPartials = (N+elementsPerPartial-1)/elementsPerPartial;

//
// number of CUDA threadblocks to use. The kernels are
// blocking agnostic, so we can clamp to any number within
// CUDA's limits and the code will work.
//
const unsigned int maxBlocks = MAX_PARTIALS;
unsigned int numBlocks = min( numPartials, maxBlocks );
scanReduceSubarrays<T>(
    gPartials,
    in,
    N,
    elementsPerPartial,
    numBlocks,
    b );
scan2Level_kernel<T, bZeroPad><<<1,b,sBytes>>>(
    gPartials,
    0,
    gPartials,
    numPartials,
    numPartials );
scan2Level_kernel<T, bZeroPad><<<numBlocks,b,sBytes>>>(
    out,
    gPartials,
    in,
    N,
    elementsPerPartial );
}
}

```

13.5 线程束扫描

到目前为止，我们集中精力于构建自上而下的扫描。在所有 3 个扫描实现的底层，隐含着完全不同的软件方法。对于大小为 32 或更小的子数组，我们使用一个基于 Kogge-Stone 电路的特殊线程束扫描（见图 13-11）。Kogge-Stone 电路效率低下，这意味着它们尽管深度不大也要花费许多运算。但是就线程束水平来说（CUDA 硬件的执行资源是可用的，不管开发者是否使用它们），Kogge-Stone 电路在 CUDA 硬件上工作良好。

代码清单 13-13 给出了一个使用 `_device_` 限定符的程序，这个程序设计在共享内存上运行，对线程来说，这是相互交换数据最快的方式。因为这里不存在共享内存的存储片冲突，并且这个程序运行在线程束粒度，在向共享内存更新数据的过程中无须线程同步。

代码清单 13-13 scanWarp

```

template<class T>
inline __device__ T
scanWarp( volatile T *sPartials )
{
    const int tid = threadIdx.x;
    const int lane = tid & 31;

    if ( lane >= 1 ) sPartials[0] += sPartials[- 1];
    if ( lane >= 2 ) sPartials[0] += sPartials[- 2];

```

```

    if ( lane >=  4 ) sPartials[0] += sPartials[- 4];
    if ( lane >=  8 ) sPartials[0] += sPartials[- 8];
    if ( lane >= 16 ) sPartials[0] += sPartials[-16];
    return sPartials[0];
}

```

13.5.1 零填充

我们可以减少实现线程束扫描必须的机器指令数：通过在线程束数组里交错放置 16 个 0 元素，免除条件判断语句。代码清单 13-14 给出了 scanWarp 的一个版本，这里假定在共享内存的基址前有 16 个零元素。

代码清单 13-14 scanWarp0

```

template<class T>
__device__ T scanWarp0( volatile T *sharedPartials, int idx )
{
    const int tid = threadIdx.x;
    const int lane = tid & 31;

    sharedPartials[idx] += sharedPartials[idx - 1];
    sharedPartials[idx] += sharedPartials[idx - 2];
    sharedPartials[idx] += sharedPartials[idx - 4];
    sharedPartials[idx] += sharedPartials[idx - 8];
    sharedPartials[idx] += sharedPartials[idx - 16];
    return sharedPartials[idx];
}

```

图 13-15 显示了交错方案如何在 256 个线程（包含 8 个线程束）组成的线程块中运行。共享内存的索引按如下方法计算。

```

const int tid = threadIdx.x;
const int warp = tid >> 5;
const int lane = tid & 31;
const int sharedIndex = 49 * warp + 32 + lane;

```

	a _{00H..1FH}	a _{20H..4FH}	a _{40H..5FH}	a _{60H..7FH}	a _{80H..9FH}	a _{A0H..BFH}	a _{C0H..DFH}	a _{E0H..FFH}
0	a _{00H..1FH}	0	a _{20H..4FH}	0	a _{40H..5FH}	0	a _{60H..7FH}	0

图 13-15 线程束扫描的交错零值

然后进行置 0 的初始化操作，如下：

```
partials[sharedIndex-16] = 0;
```

在块扫描子程序里，这种变化会影响共享内存的寻址。每个线程束的部分和的索引必须加上偏移 16，来启用单线程束的扫描操作，该操作计算基本和。最后，内核启动必须预留足够的共享内存来保存部分和以及零值。

13.5.2 带模板的版本

扫描算法的更快的零填充实现需要更多的共享内存，这一资源需求不是所有应用都能满足的。为了确保我们的代码能在两个场合中运行，代码清单 13-15 显示了一个以 bool 型变量 bZeroPad 为参数的辅助函数。该 scanSharedMemory 函数返回一个给定大小线程块所需要的共享内存量。scanSharedIndex 函数返回一个给定线程相应的共享内存索引。后面的代码清单 13-16 给出了 scanWarp 带模板的版本，可同时工作于零填充和非零填充两种场合。

代码清单 13-15 用于零填充方式的共享内存辅助函数

```
template<bool bZeroPad>
inline __device__ int
scanSharedIndex( int tid )
{
    if ( bZeroPad ) {
        const int warp = tid >> 5;
        const int lane = tid & 31;
        return 49 * warp + 16 + lane;
    }
    else {
        return tid;
    }
}

template<typename T, bool bZeroPad>
inline __device__ __host__ int
scanSharedMemory( int numThreads )
{
    if ( bZeroPad ) {
        const int warpcount = numThreads>>5;
        return (49 * warpcount + 16)*sizeof(T);
    }
    else {
        return numThreads*sizeof(T);
    }
}
```

代码清单 13-16 scanWarp (带模板版本)

```
template<class T, bool bZeroPadded>
inline __device__ T
scanWarp( volatile T *sPartials )
{
    T t = sPartials[0];
    if ( bZeroPadded ) {
        t += sPartials[- 1]; sPartials[0] = t;
        t += sPartials[- 2]; sPartials[0] = t;
        t += sPartials[- 4]; sPartials[0] = t;
        t += sPartials[- 8]; sPartials[0] = t;
        t += sPartials[-16]; sPartials[0] = t;
    }
    else {
        const int tid = threadIdx.x;
        const int lane = tid & 31;
        if ( lane >= 1 ) { t += sPartials[- 1]; sPartials[0] = t; }
```

```

        if ( lane >=  2 ) { t += sPartials[- 2]; sPartials[0] = t; }
        if ( lane >=  4 ) { t += sPartials[- 4]; sPartials[0] = t; }
        if ( lane >=  8 ) { t += sPartials[- 8]; sPartials[0] = t; }
        if ( lane >= 16 ) { t += sPartials[-16]; sPartials[0] = t; }
    }
    return t;
}

```

13.5.3 线程束洗牌

SM 3.0 指令集添加了线程束洗牌指令，该指令使寄存器能在 32 个线程组成的线程束中进行交换。线程束洗牌的“向上”和“向下”变形能够分别用于执行扫描和反向扫描。洗牌指令带两个参数，一个是用于交换的寄存器，另一个是应用于线程编号的偏移量。它返回一个断定结果，当线程是非活动的或者线程的偏移量超出了线程束范围时，断定的对应位为“假”。

代码清单 13-17 给出了 scanWarpShuffle 函数，它是使用洗牌指令实现包容性线程束扫描的设备函数。它的模板参数是整型，通常传入的值为 5，因为 5 正好是线程束大小 32 的以 2 为底的对数。鉴于编译器无法生成有效处理洗牌指令返回的断定结果的代码，scanWarpShuffle 使用一个内联实现在 PTX 中的辅助函数 scanWarpShuffle_step。

代码清单 13-17 scanWarpShuffle 设备函数

```

__device__ __forceinline__
int
scanWarpShuffle_step(int partial, int offset)
{
    int result;
    asm(
        "{.reg .u32 r0;" 
        ".reg .pred p;" 
        "shfl.up.b32 r0|p, %1, %2, 0;" 
        "@p add.u32 r0, r0, %3;" 
        "mov.u32 %0, r0;" 
        : "=r"(result) : "r"(partial), "r"(offset), "r"(partial));
    return result;
}

template <int levels>
__device__ __forceinline__
int
scanWarpShuffle(int mysum)
{
    for(int i = 0; i < levels; ++i)
        mysum = scanWarpShuffle_step(mysum, 1 << i);
    return mysum;
}

```

代码清单 13-18 显示了如何扩展 scanWarpShuffle 函数，利用共享内存扫描在一个线程块的值。采用与代码清单 13-13 一样的线程块扫描形式，scanBlockShuffle 函数使用线程束洗牌扫描每个线程束。每个线程束把它自己的部分写到共享内存中，然后再一次使用线程束洗

牌，这次仅使用单个的线程束，来扫描这些基本和。最后，每个线程束加上对应的基本和来计算最终输出值。

代码清单 13-18 scanBlockShuffle 设备函数

```

template <int logBlockSize>
__device__
int
scanBlockShuffle(int val, const unsigned int idx)
{
    const unsigned int lane    = idx & 31;
    const unsigned int warpid = idx >> 5;
    __shared__ int sPartials[32];

    // Intra-warp scan in each warp
    val = scanWarpShuffle<5>(val);

    // Collect per-warp results
    if (lane == 31) sPartials[warpid] = val;
    __syncthreads();

    // Use first warp to scan per-warp results
    if (warpid == 0) {
        int t = sPartials[lane];
        t = scanWarpShuffle<logBlockSize-5>( t );
        sPartials[lane] = t;
    }

    __syncthreads();

    // Add scanned base sum for final result
    if (warpid > 0) {
        val += sPartials[warpid - 1];
    }
    return val;
}

```

13.5.4 指令数对比

为了评价本节所讨论的不同线程束扫描算法，我们针对 SM 3.0 进行编译，并且使用 cuobjdump 来反汇编 3 个实现。得到的结果如下：

- 代码清单 13-19 给出的非零填充实现共有 30 条指令，并包括了大量的分支（配对使用的 SSY 的 /S 指令执行分支栈的压入和弹出，如本书第 8.4.2 小节所述）。
- 代码清单 13-20 给出的零填充的实现共有 17 条指令，它在读取共享内存之前不需要检查线程编号。需要注意的是，一旦共享内存上的操作都限定在一个线程束内，就无须通过调用 __syncthreads() 内置函数进行栅栏同步，该同步在 SASS 里编译成 BAR SYNC 指令。
- 代码清单 13-21 中给出的洗牌实现只有 11 条指令。

在人工合成的负载上测试（分离出线程束扫描），可以证实，基于洗牌的实现显著快（约 2 倍）于代码清单 13-19 中给出的一般情况。

代码清单 13-19 线程束扫描的 SASS 代码 (无零填充)

```

/*0070*/ SSY 0xa0;
/*0078*/ @P0 NOP.S CC.T;
/*0088*/ LDS R5, [R3+-0x4];
/*0090*/ IADD R0, R5, R0;
/*0098*/ STS.S [R3], R0;
/*00a0*/ ISETP.LT.U32.AND P0, pt, R4, 0x2, pt;
/*00a8*/ SSY 0xd8;
/*00b0*/ @P0 NOP.S CC.T;
/*00b8*/ LDS R5, [R3+-0x8];
/*00c8*/ IADD R0, R5, R0;
/*00d0*/ STS.S [R3], R0;
/*00d8*/ ISETP.LT.U32.AND P0, pt, R4, 0x4, pt;
/*00e0*/ SSY 0x10;
/*00e8*/ @P0 NOP.S CC.T;
/*00f0*/ LDS R5, [R3+-0x10];
/*00f8*/ IADD R0, R5, R0;
/*0108*/ STS.S [R3], R0;
/*0110*/ ISETP.LT.U32.AND P0, pt, R4, 0x8, pt;
/*0118*/ SSY 0x140;
/*0120*/ @P0 NOP.S CC.T;
/*0128*/ LDS R5, [R3+-0x20];
/*0130*/ IADD R0, R5, R0;
/*0138*/ STS.S [R3], R0;
/*0148*/ ISETP.LT.U32.AND P0, pt, R4, 0x10, pt;
/*0150*/ SSY 0x178;
/*0158*/ @P0 NOP.S CC.T;
/*0160*/ LDS R4, [R3+-0x40];
/*0168*/ IADD R0, R4, R0;
/*0170*/ STS.S [R3], R0;
/*0178*/ BAR.SYNC 0x0;

```

代码清单 13-20 线程束扫描的 SASS 代码 (零填充)

```

/*0058*/ LDS R4, [R3+-0x4];
/*0060*/ LDS R0, [R3];
/*0068*/ IADD R4, R4, R0;
/*0070*/ STS [R3], R4;
/*0078*/ LDS R0, [R3+-0x8];
/*0088*/ IADD R4, R4, R0;
/*0090*/ STS [R3], R4;
/*0098*/ LDS R0, [R3+-0x10];
/*00a0*/ IADD R4, R4, R0;
/*00a8*/ STS [R3], R4;
/*00b0*/ LDS R0, [R3+-0x20];
/*00b8*/ IADD R4, R4, R0;
/*00c8*/ STS [R3], R4;
/*00d0*/ LDS R0, [R3+-0x40];
/*00d8*/ IADD R0, R4, R0;
/*00e0*/ STS [R3], R0;
/*00e8*/ BAR.SYNC 0x0;

```

代码清单 13-21 线程束扫描的 SASS 代码 (基于洗牌)

```

/*0050*/ SHFL.UP P0, R4, R0, 0x1, 0x0;
/*0058*/ IADD.X R3, R3, c [0x0] [0x144];
/*0060*/ @P0 IADD R4, R4, R0;
/*0068*/ SHFL.UP P0, R0, R4, 0x2, 0x0;

```

```

/*0070*/      @P0 IADD R0, R0, R4;
/*0078*/      SHFL.UP P0, R4, R0, 0x4, 0x0;
/*0088*/      @P0 IADD R4, R4, R0;
/*0090*/      SHFL.UP P0, R0, R4, 0x8, 0x0;
/*0098*/      @P0 IADD R0, R0, R4;
/*00a0*/      SHFL.UP P0, R4, R0, 0x10, 0x0;
/*00a8*/      @P0 IADD R4, R4, R0;

```

13.6 流压缩

扫描的实现常作用于断定结果，即通过评估条件得到的真值（0或1）。正如在本章开头提及的，针对断定结果的排他性扫描可以用来实现流压缩。流压缩这类并行问题，只把一个输入数组中“感兴趣的”元素写到输出。对于断定值为1的那些“感兴趣的”元素，排他性扫描计算出该元素的输出索引。

例如，我们写一个接受int型数组并输出那些奇数值元素的扫描。[⊖]我们的实现基于Merrill的先归约再扫描方案，并使用一个固定数目的线程块数b。

- 1) 输入数据的第一遍归约给出每个大小为 $\lceil N/b \rceil$ 的子数组中满足条件的元素数。
- 2) 在大小为b的数组上进行扫描操作，得到每个子数组的输出的基索引。
- 3) 在输入数组上执行扫描，评估条件并使用“种子”值作为每个子数组输出的基索引。

代码清单13-22显示了步骤1的代码：predicateReduceSubarrays_odd()函数调用子程序predicateReduceSubarray_odd()和isOdd()来计算每个数组元素的断定值，计算归约值并把它写到基本和的数组中。

代码清单13-22 predicateReduceSubarrays_odd

```

template<class T>
__host__ __device__ bool
isOdd( T x )
{
    return x & 1;
}

template<class T, int numThreads>
__device__ void
predicateReduceSubarray_odd(
    int *gPartials,
    const T *in,
    size_t iBlock,
    size_t N,
    int elementsPerPartial )
{
    extern volatile __shared__ int sPartials[];
    const int tid = threadIdx.x;

    size_t baseIndex = iBlock*elementsPerPartial;

```

[⊖] 该代码很容易进行修改，以处理更复杂的断定。

```

int sum = 0;
for ( int i = tid; i < elementsPerPartial; i += blockDim.x ) {
    size_t index = baseIndex+i;
    if ( index < N )
        sum += isOdd( in[index] );
}
sPartials[tid] = sum;
__syncthreads();

reduceBlock<int,numThreads>( &gPartials[iBlock], sPartials );
}

/*
 * Compute the reductions of each subarray of size
 * elementsPerPartial, and write them to gPartials.
 */
template<class T, int numThreads>
__global__ void
predicateReduceSubarrays_odd(
    int *gPartials,
    const T *in,
    size_t N,
    int elementsPerPartial )
{
    extern volatile __shared__ int sPartials[];

    for ( int iBlock = blockIdx.x;
          iBlock*elementsPerPartial < N;
          iBlock += gridDim.x )
    {
        predicateReduceSubarray_odd<T,numThreads>(
            gPartials,
            in,
            iBlock,
            N,
            elementsPerPartial );
    }
}

```

调用代码清单 13-23 中的内核来计算基本和数组的扫描值。做完此步，每个基本和元素的位置是相应线程块输出数组的起始索引，它的值代表输入数组在此块之前断定为真的元素数目。

代码清单 13-23 streamCompact_odd 内核

```

template<class T, bool bZeroPad>
__global__ void
streamCompact_odd(
    T *out,
    int *outCount,
    const int *gBaseSums,
    const T *in,
    size_t N,
    size_t elementsPerPartial )
{
    extern volatile __shared__ int sPartials[];
    const int tid = threadIdx.x;
    int sIndex = scanSharedIndex<bZeroPad>( threadIdx.x );

    if ( bZeroPad ) {

```

```

        sPartials[sIndex-16] = 0;
    }
    // exclusive scan element gBaseSums[blockIdx.x]
    int base_sum = 0;
    if ( blockIdx.x && gBaseSums ) {
        base_sum = gBaseSums[blockIdx.x-1];
    }
    for ( size_t i = 0;
          i < elementsPerPartial;
          i += blockDim.x ) {
        size_t index = blockIdx.x*elementsPerPartial + i + tid;
        int value = (index < N) ? in[index] : 0;
        sPartials[sIndex] = (index < N) ? isOdd( value ) : 0;
        __syncthreads();

        scanBlock<int,bZeroPad>( sPartials+sIndex );
        __syncthreads();
        if ( index < N && isOdd( value ) ) {
            int outIndex = base_sum;
            if ( tid ) {
                outIndex += sPartials[
                    scanSharedIndex<bZeroPad>(tid-1)];
            }
            out[outIndex] = value;
        }
        __syncthreads();

        // carry forward from this block to the next.
        {
            int inx = scanSharedIndex<bZeroPad>( blockDim.x-1 );
            base_sum += sPartials[ inx ];
        }
        __syncthreads();
    }
    if ( threadIdx.x == 0 && blockIdx.x == 0 ) {
        if ( gBaseSums ) {
            *outCount = gBaseSums[gridDim.x-1];
        }
        else {
            int inx = scanSharedIndex<bZeroPad>( blockDim.x-1 );
            *outCount = sPartials[ inx ];
        }
    }
}
}

```

代码清单 13-23 显示了步骤 3 的代码，它接受输入数组以及基本和数据作为参数，计算每个输入数组元素的断定值，并在该元素断定值为真时，把此元素写入到正确索引的输出元素。主机代码类似于代码清单 13-12，它们仅有轻微变化，这里不再显示。

13.7 参考文献（并行扫描算法）

递归形式的先扫描再扇出算法，在 NVIDIA 的技术报告 NVR-2008-003 中由 Sengupta 等人提出。递归形式的先归约再扫描算法，由 Dotsenko 等人描述。两阶段先归约再扫描算法由 Merrill 给出。Merrill 的论文非常值得阅读，其中不仅包含了背景，也有失败的尝试。例如，所尝试的基于 Sklansky 的最小深度电路法的扫描方法，表现令人失望。

Blelloch, Guy E. Prefix sums and their applications. Technical Report CMU-CS-90-190.

Dotsenko, Yuri, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. Fast scan algorithms in graphics processors. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ACM, 2008, pp. 205–213.

Fellner, D., and S. Spender, eds. SIGGRAPH/Eurographics Conference on Graphics Hardware. Eurographics Association, Aire-la-Ville, Switzerland, pp. 97–106.

Harris, Mark, and Michael Garland. Optimizing parallel prefix operations for the Fermi architecture. In *GPU Computing Gems, Jade Edition*, Wen-Mei Hwu, ed. Morgan Kaufmann, Waltham, MA, 2012, pp. 29–38.

Harris, Mark, Shubhabrata Sengupta, and John Owens. Parallel prefix sum [scan] with CUDA. In *GPU Gems 3*, H. Nguyen, ed. Addison-Wesley, Boston, MA, Aug. 2007.

Merrill, Duane, and Andrew Grimshaw. Parallel scan for stream architectures. Technical Report CS2009-14. Department of Computer Science, University of Virginia.

Sengupta, Shubhabrata, Mark Harris, and Michael Garland. Efficient parallel scan algorithms for GPUs. NVIDIA Technical Report NVR-2008-003. December 2008.

<http://research.nvidia.com/publication/efficient-parallel-scan-algorithms-gpus>

Sengupta, Shubhabrata, Mark Harris, ZhangYao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proceedings of the 22nd ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*. San Diego, CA, August 4–5, 2007.

13.8 延伸阅读（并行前缀求和电路）

针对电路的并行前缀求和问题存在丰富的文献。除了 Brent-Kung、Sklansky 和 Kogge-Stone 的方法，扫描电路的其他例子包括 Ladner-Fischer 的方法和更近期 Lin 和 Hsiao 的工作。Hinze 描述了一种扫描代数，可用于指导扫描的实现。关于他工作的细节超出了本书的范围，但大力推荐阅读他的论文。

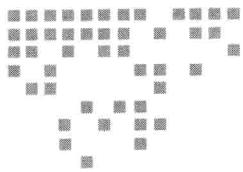
Sean Baxter 的网站 <http://www.moderngpu.com> 是优化扫描及其应用的极好资源。

Brent, Richard P., and H.T. Kung, A regular layout for parallel adders. IEEE Transactions on Computers C-31, 1982, pp. 260–264.

Hinze, Ralf. An algebra of scans. In *Mathematics of Program Construction*, Springer, 2004, Stirling, Scotland, pp. 186–210.

Kogge, Peter M., and Harold S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Transactions on Computers C-22, 1973, pp. 783–791.

Sklansky, J. Conditional sum addition logic. IRE Trans. Electron. Comput. 9 (2), June 1960, pp. 226–231.



N-体问题

N-体 (N-body) 计算是一类并行计算模型，该模型由一组粒子（每个粒子称为一个个体）组成，在每个个体的计算中，必须考虑其他所有个体对它的影响。N-体计算的应用例子有如下一些（但是 N-体计算并不局限于此）：

- 恒星引力场中的重力模拟。
- 离子静电场中的分子模拟。
- 计算机图形学中用于模拟水和火的粒子系统。
- “类鸟群”，一种用来模拟鸟群行为的计算机动画技术。

通常情况下，个体行为的模拟按时间步长进行，因此，对 N 个个体而言，每个时间步长的计算复杂度为 $O(N^2)$ 。在大多数模型中，个体间的影响随着距离的增加快速减小，因此需要使用（例如）分层算法，以利用一组个体质心中个体的质量和位置来避免复杂度为 $O(N^2)$ 的全部计算。Barnes-Hut 算法通过引入一个空间层次，来近似计算两组对象之间的作用力，将计算的复杂度减小到 $O(N \lg N)$ 。在实际应用中，叶子节点包含 K 个个体，对于给定的叶子节点必须进行复杂度为 $O(K^2)$ 的计算。而复杂度为 $O(K^2)$ 的这部分计算是 GPU 最擅长的。

N-体负载已经被证实为能让 GPU 达到理论处理能力极限的最有效方式。在 Harris 等人的《GPU 精粹》的“基于 CUDA 的快速 N-体模拟”一章[⊖]中，频繁引用这个理论极限来解释为什么进一步提高计算性能是不可能的。所使用的英伟达公司的 GeForce 8800 GTX GPU 在 N-体计算任务上是如此的高效，甚至超过了专门设计用来执行天体运算的 GRAPE-6 硬件。

我们希望读者能够将 GPU 引入自己的计算中，并且找到运行最快速的方法。本章将展

[⊖] http://developer.nvidia.com/GPUGems3/gpugems3_ch31.html。

示使用 CUDA 技术进行 N- 体及其相关计算的几种不同方式。

- 简单的实现方法来说明技术本身，并强调使用高速缓存和循环展开的有效性。
- 采用共享内存的实现方法（对于引力计算速度最快的方法），借鉴 Harris 等人的结果，在最内层循环使用线程块大小的个体集合对计算任务进行分块，以减小访问内存带来的延时。
- 采用常量内存的实现方法，这种设计灵感来自于 Stone 等人的直接库伦求和（Direct Coulomb Summation, DCS）[⊖]计算，使用常量内存存储个体信息，可以释放共享内存供其他用途使用。

因为读者的应用也许不是 N- 体之间引力计算，呈现这些不同的实现方法的目的不是单纯地为了针对特定计算进行优化。它们能够根据目标 SM 构架、问题规模以及核心计算的具体细节改写为不同的实现方法。

既然 N- 体引力计算已经被作为 GPU 理论计算能力的典型代表（它能达到 400 倍的加速比），本章最后一节将提出 CPU 的优化策略。通过使用 SSE 指令集和多线程技术重新编写计算代码，可以得到超过 300 倍的加速比。根据第 14-9 小节报告的结果，一个 GK104 构架的 GPU，性能显著优于两个英特尔的至强 E2670 高端服务器 CPU。CUDA 实现方案相对于优化过的 CPU 应用，具有运行速度更快、代码可读性强和维护性更好等优点。

贯穿于本章的性能评价，使用的是服务器级的机器，它有 2 个至强 E2670 “沙桥” CPU 和多达 4 个 GK104 构架的 GPU，为了减少散热和能耗，这 4 个 GPU 被降低了主频。我们报告的性能结果是根据每秒可以计算的个体之间相互作用的次数，而不是 GFLOPS 结果。

14.1 概述

给定 N 个个体，个体 i ($1 < i < N$) 的初始位置为 \mathbf{x}_i ，速度为 \mathbf{v}_i ，个体 i 受到来自个体 j 作用的力向量 \mathbf{f}_{ij} 为：

$$\mathbf{f}_{ij} = G \frac{\mathbf{m}_i \mathbf{m}_j}{\|\mathbf{d}_{ij}\|^2} \cdot \frac{\mathbf{d}_{ij}}{\|\mathbf{d}_{ij}\|}$$

这里 m_i 和 m_j 分别表示个体 i 和个体 j 的质量； \mathbf{d}_{ij} 表示个体 i 到个体 j 的差向量； G 是引力常数。由于除法可能导致溢出，表达式会因为 \mathbf{d}_{ij} 的幅值过小而发散。通常进行补偿的方法是使用一个软化因子，对两个普拉默质点（plummer mass）间的相互作用进行建模，普拉默质点把个体看成球形星体。假设软化因子为 ϵ ，表达式将变成：

$$\mathbf{f}_{ij} = G \frac{\mathbf{m}_i \mathbf{m}_j \mathbf{d}_{ij}}{(\|\mathbf{d}_{ij}\|^2 + \epsilon^2)^{3/2}}$$

鉴于对 i 的影响来自其他 $N-1$ 个个体，所有对个体 i 的作用为 \mathbf{F}_i ，是所有作用力的和：

$$\mathbf{F}_i = \sum_{j=1}^N \mathbf{f}_{ij} = G m_i \sum_{j=1}^N \frac{\mathbf{m}_i \mathbf{m}_j \mathbf{d}_{ij}}{(\|\mathbf{d}_{ij}\|^2 + \epsilon^2)^{3/2}}$$

[⊖] www.ncbi.nlm.nih.gov/pubmed/17894371

② 原文公式中有笔误，已修正。——译者注

为了更新每个个体的位置和速度，对个体 i 产生的作用力（加速度）为 $\mathbf{a}_i = \mathbf{F}_i/m_i$ ，因此 m_i 可以从表达式分子中删去，如下：

$$\mathbf{F}_i = \sum_{\substack{1 \leq j \leq N \\ j \neq i}}^N \mathbf{f}_{ij} = Gm_i \sum_{j=1}^N \frac{m_j \mathbf{d}_{ij}}{(\|\mathbf{d}_{ij}\|^2 + \varepsilon^2)^{3/2}}$$

就像 Nyland 等人一样，我们使用一个轮替跨越 Verlet 算法（整数步长的时间点），进行一个时间步长的模拟。个体的位置和速度值的改变是按照比对方增长半个时间步长方式进行，因为在我们的例子中，初始位置和速度是随机值，所以这一特点在代码中并不是十分的明显。我们的轮替 Verlet 方法先来更新速度，接下来更新位置：

$$\begin{aligned}\mathbf{V}_i(t + \frac{1}{2} \partial t) &= \mathbf{V}_i(t - \frac{1}{2} \partial t) + \partial t \mathbf{F}_i \\ \mathbf{P}_i(t + \partial t) &= \mathbf{P}_i(t) + \partial t \mathbf{V}_i(t + \frac{1}{2} \partial t)\end{aligned}$$

这种方式只是用于更新模拟系统的众多积分算法的一种，但是进一步的讨论已经超出本书的范畴。

因为积分运算的时间复杂度为 $O(N)$ ，而计算个体相互作用力的时间复杂度为 $O(N^2)$ ，因此移植到 CUDA 平台的最大优势就是优化作用力计算的运行时间。优化这部分的计算时间也是本章的主要任务。

力的矩阵

一个实现 N- 体计算的简单算法，通过双重嵌套循环，计算每一个个体受到其他所有个体作用力的和。复杂度为 $O(N^2)$ 的个体之间相互作用力的计算可以看作是一个 $N \times N$ 的矩阵，矩阵第 i 行的和是个体 i 的所有作用力的和：

$$\mathbf{F}_i = \sum_{j=1}^N \mathbf{f}_{ij}$$

矩阵的对角线值为零，因为个体对自身作用力为零，因此可以忽略。

矩阵中的每一个元素是独立计算得到的，因此存在大量潜在的并行。每一行的和是一个归约，可以使用一个线程计算，也可以使用第 12 章中组合多个线程的结果进行计算。

图 14-1 考察了一个 8- 体矩阵。每一行的和对应的是计算的输出。当使用 CUDA 时，N- 体计算通常使用一个线程来计算每一行的和。

因为矩阵中的每一个元素都是独立的，它们也可以在给定行中进行并行计算：例如计算每隔 4 个元素的和，然后将 4 个部分结果相加得到最终结果。Harris 等人使用过这种方式，由于 N 较小以致没有足够的线程隐藏 GPU 的延时，他们的方法是这样的：启动更多的线程，在每个线程计算部分和，然后在共享内存上进行归约计算最终结果。Harris 等人展示了当 $N \leq 4096$ 时能够收到成效。

由于物理作用力是对称的（例如，个体 i 受到个体 j 的作用和个体 j 受到个体 i 作用是等大反向的），就像引力，矩阵的转置元素有相反的符号：

$$\mathbf{f}_{ij} = -\mathbf{f}_{ji}$$

在这种情况下，矩阵的形式如图 14-2 展示的那样。利用对称性，只需要计算矩阵右上三角的值，进行大约一半的计算。[⊖]

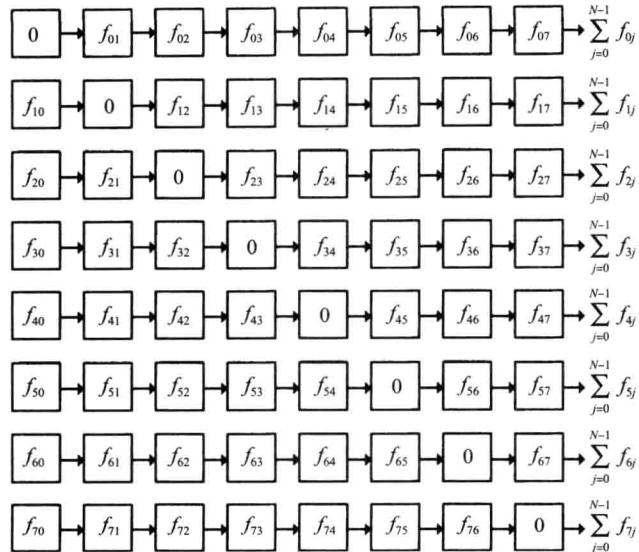


图 14-1 作用力矩阵 (8 个体)

0	f_{01}	f_{02}	f_{03}	f_{04}	f_{05}	f_{06}	f_{07}
$-f_{01}$	0	f_{12}	f_{13}	f_{14}	f_{15}	f_{16}	f_{17}
$-f_{02}$	$-f_{12}$	0	f_{23}	f_{24}	f_{25}	f_{26}	f_{27}
$-f_{03}$	$-f_{13}$	$-f_{23}$	0	f_{34}	f_{35}	f_{36}	f_{37}
$-f_{04}$	$-f_{14}$	$-f_{24}$	$-f_{34}$	0	f_{45}	f_{46}	f_{47}
$-f_{05}$	$-f_{15}$	$-f_{25}$	$-f_{35}$	$-f_{45}$	0	f_{56}	f_{57}
$-f_{06}$	$-f_{16}$	$-f_{26}$	$-f_{36}$	$-f_{46}$	$-f_{56}$	0	f_{67}
$-f_{07}$	$-f_{17}$	$-f_{27}$	$-f_{37}$	$-f_{47}$	$-f_{57}$	$-f_{67}$	0

图 14-2 作用力对称时的矩阵

⊖ 准确来说，要计算 $\frac{N(N-1)}{2}$ 个作用力。

现在的问题并不像图 14-1 那种暴力计算方法，当方法利用作用力的对称性时，不同的线程会对给定的输出和有所贡献。部分和不断被累计并保存到一个临时变量，最后进行归约计算，或者系统使用互斥机制来保护最终结果（使用原子操作或线程同步）。由于个体之间的计算量大约是 20FLOPS（单精度）或者 30FLOPS（双精度），减少总和的计算似乎对于性能的提升有着决定性作用。

不幸的是，多余的开销往往超过了仅执行一半计算带来的好处。例如，一个实现个体间计算中两个浮点数的原子加法比暴力计算方法更慢。

图 14-3 展示了一个对两种极端方法的折衷方案：通过对计算分块，只需要对右上角进行计算。对于一个尺寸为 k 的分块，这种方法需要在总数为 $\frac{N/k(N/k-1)}{2}$ 的每个分块中进行 k^2 次计算，然后再加上 N/k 个对角分块上分别进行 $k(k-1)$ 个个体间计算。对于尺寸很大的 N ，个体间的计算节约时间大致相同，^②但因为在尺寸为 k 的分块上可以进行部分和的计算并加到最终结果，可以降低同步带来的时间开销。图 14-3 展示了尺寸 $k=2$ 的分块，但分块尺寸对应线程束 ($k=32$) 的大小是更实用的。

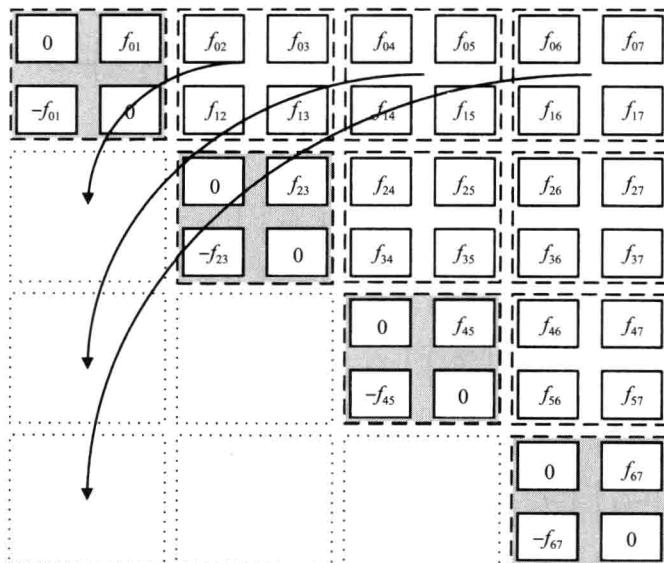


图 14-3 分块的 N- 体问题 ($k=2$)

图 14-4 展示了如何在给定的分块上进行部分和的计算。在行和列计算得到部分和（分别进行加和减），以得到必须被加到对应输出和上的部分和。

流行于分子建模领域的“AMBER 应用”利用了力的对称性，使用的分块大小调优后设

^② 例如，使用 $N=65536$ 和 $k=32$ ，这种方法仅是暴力计算 52.5% 的性能，虽然相对于对称算法提高了 3% 的性能。

置为线程束的大小 32,^①但是在大量的测试中，这种方法并未在此处描述的轻量级计算中显现出优势。

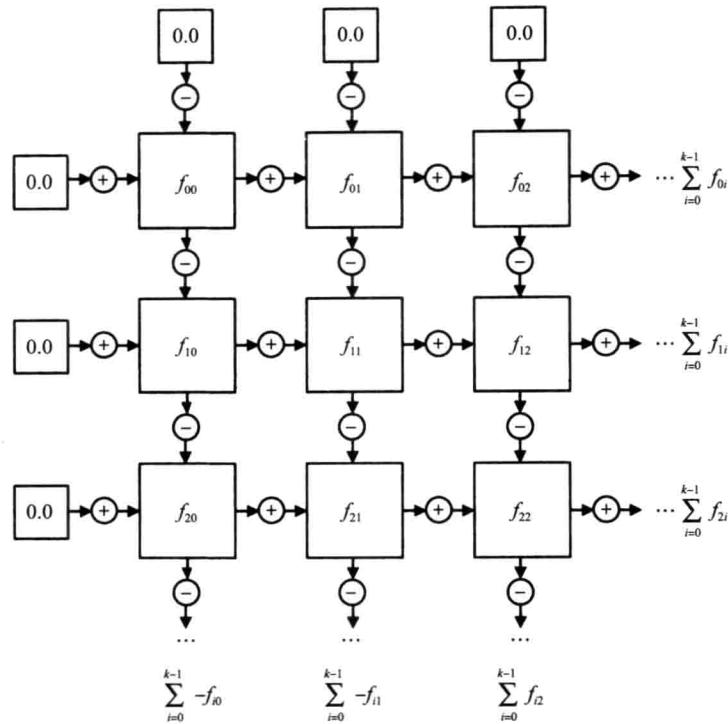


图 14-4 N- 体的一个分块

14.2 简单实现

代码清单 14-1 给出了上一节描述的个体之间相互作用的函数实现，使用了 `_host_` 和 `_device_` 关键词双重修饰，CUDA 编译器就会知道它既可以被 CPU 也可以被 GPU 调用。这个函数带有模板，因此它可以被 `float` 和 `double` 同时使用（在这本书中，只有 `float` 型是完全实现了的）。它返回三维作用力矢量，用三元组 (ax, ay, az) ^② 表示。

代码清单 14-1 bodyBodyInteraction 函数

```
template <typename T>
__host__ __device__ void bodyBodyInteraction(
    T& ax, T& ay, T& az,
```

① Götz, Andreas, Mark J. Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs—Part I: Generalized Born, *J. Chem. Theory Comput.* 8, no. 5 (2012), pp. 1542–1555.

② 原文误作 (fx, fy, fz) 。——译者注

```

    T x0, T y0, T z0,
    T x1, T y1, T z1, T mass1,
    T softeningSquared)
{
    T dx = x1 - x0;
    T dy = y1 - y0;
    T dz = z1 - z0;

    T distSqr = dx*dx + dy*dy + dz*dz;
    distSqr += softeningSquared;

    T invDist = (T)1.0 / (T)sqrt(distSqr);

    T invDistCube = invDist * invDist * invDist;
    T s = mass1 * invDistCube;

    ax = dx * s;
    ay = dy * s;
    az = dz * s;
}

```

代码清单 14-2 给出了计算每个个体受到所有作用力的函数实现。对于每个个体，它把自己的位置加载到 (*myX*, *myY*, *myZ*)，然后调用函数 `bodyBodyInteraction<float>` 来计算和其他每个个体之间的作用力。函数名中的 AOS，表示输入的数据来自一个结构体数组 (array of structures, AOS)。包含四个 float 型数的结构体 (*x*, *y*, *z*, *mass*) 指定了个体的位置和质量。使用 float4 类型的数据是为了方便 GPU 的实现，这样可以得到硬件在加载和存储上的天然支持。而我们优化的 CPU 实现，详见第 14.9 小节，使用包含 *x*、*y*、*z* 和 *mass* 四个 float 型数组成的结构数组 (structure of arrays, SOA)，这种形式能够有利于 SIMD (单指令多数据) 指令集的处理。结构数组不是很适合 GPU 实现，因为其中所需的四个基指针会耗费过多的寄存器。

代码清单 14-2 ComputerGravitation_AOS 函数 (CPU 实现)

```

float
ComputeGravitation_AOS(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N
)
{
    chTimerTimestamp start, end;
    chTimerGetTime( &start );
    for ( size_t i = 0; i < N; i++ )
    {
        float ax = 0.0f;
        float ay = 0.0f;
        float az = 0.0f;
        float myX = posMass[i*4+0];
        float myY = posMass[i*4+1];
        float myZ = posMass[i*4+2];

        for ( size_t j = 0; j < N; j++ ) {
            float acc[3];
            float bodyX = posMass[j*4+0];

```

```

        float bodyY = posMass[j*4+1];
        float bodyZ = posMass[j*4+2];
        float bodyMass = posMass[j*4+3];

        bodyBodyInteraction<float>(
            ax, ay, az,
            myX, myY, myZ,
            bodyX, bodyY, bodyZ, bodyMass,
            softeningSquared );
        ax += acc[0];
        ay += acc[1];
        az += acc[2];
    }

    force[3*i+0] = ax;
    force[3*i+1] = ay;
    force[3*i+2] = az;
}
chTimerGetTime( &end );
return (float) chTimerElapsedTime( &start, &end ) * 1000.0f;
}

```

代码清单 14-3 给出了等价于代码清单 14-2 中函数的 GPU 版本。对于每一个个体，累计来自其他每一个个体的作用力值，然后将这个累计值写到一个作用力数组中。SM 2.x 及以后版本中有一级和二级缓存，可以很好地加速这类负载，因为在内层循环中有大量的数据重用发生。

外层循环和内存循环都会将输入数组变成 float4 形式，这样可以保证编译器能够正确的发出单条 16 位加载指令。循环展开是在 GPU 上进行 N- 体计算最常用的优化方案，不难想象为什么是这样：GPU 上分支结构的开销远大于 CPU，因此在每次循环中减少指令能够带来很大好处，而且循环展开能够得到更多指令级并行 (instruction level parallelism, ILP) 的机会，这样 GPU 可以隐藏指令执行延时和内存访问延时。

表 14-1 简单内核上的循环展开

展开因子	个体间作用力数目 (10 亿个 / 秒)
1	25
2	30
16	34.3

为了在我们的 N- 体计算中利用循环展开的优点，我们需要在内层针对 j 的 for 循环前插入这行代码：

```
#pragma unroll <factor>
```

不幸的是最佳循环展开因子需要依靠经验决定。表 14-1 总结了在这个内核中循环展开的效果。

在这个内核案例中，不使用循环展开时，每秒仅能计算个体间的作用力数目为 250 亿。即使展开因子是 2，也能将计算性能提升到了 300 亿个每秒；将展开因子增加到 16 时，观测

到内核的最佳性能：每秒计算个体间的作用力高达 343 亿个，得到 37% 的性能提升。

代码清单 14-3 ComputeNBodyGravitation_GPU_AOS

```

template<typename T>
__global__ void
ComputeNBodyGravitation_GPU_AOS(
    T *force,
    T *posMass,
    size_t N,
    T softeningSquared )
{
    for ( int i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        T acc[3] = {0};
        float4 me = ((float4 *) posMass)[i];
        T myX = me.x;
        T myY = me.y;
        T myZ = me.z;
        for ( int j = 0; j < N; j++ ) {
            float4 body = ((float4 *) posMass)[j];
            float fx, fy, fz;
            bodyBodyInteraction(
                &fx, &fy, &fz,
                myX, myY, myZ,
                body.x, body.y, body.z, body.w,
                softeningSquared);
            acc[0] += fx;
            acc[1] += fy;
            acc[2] += fz;
        }
        force[3*i+0] = acc[0];
        force[3*i+1] = acc[1];
        force[3*i+2] = acc[2];
    }
}

```

14.3 基于共享内存实现

在 N- 体计算中，最内层循环有足够的局部性和重用性，无须编程人员的介入即可让缓存有效运转，但是在 CUDA 构架上，使用共享内存来作为显式缓存使用能带来很大好处，^②正如代码清单 14-4 展示的。内层的循环采用两层循环进行分块：第一层循环按照每次一个线程块方式遍历 N 个个体并加载到共享内存，第二层循环使用保存在共享内存的个体数据进行迭代计算。当同一线程束的线程访问共享内存中的同一个位置时，将得到广播机制的优化，因此这种使用模式非常适合 GPU 硬件的构架。

这种方法同样在 Harris 等人的报告中被提到。当 N 值很大时，该方法能够得到非常好的性能，几乎达到 GPU 的理论计算极限。

^② 共享内存流处理器簇（SM）1.x 构架的必备资源，它没有缓存。但是它被证明在所有的 CUDA 构架上都很有用，尽管在流处理器簇（SM）2.x 和 3.x 上很轻微。

代码清单 14-4 ComputeNBodyGravitation_shared

```

__global__ void
ComputeNBodyGravitation_Shared(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N )
{
    float4 *posMass4 = posMass;
    extern __shared__ float4 shPosMass[];
    for ( int i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        float acc[3] = {0};
        float4 myPosMass = posMass4[i];
#pragma unroll 32
        for ( int j = 0; j < N; j += blockDim.x ) {
            shPosMass[threadIdx.x] = posMass4[j+threadIdx.x];
            __syncthreads();
            for ( size_t k = 0; k < blockDim.x; k++ ) {
                float fx, fy, fz;
                float4 bodyPosMass = shPosMass[k];
                bodyBodyInteraction(
                    &fx, &fy, &fz,
                    myPosMass.x, myPosMass.y, myPosMass.z,
                    bodyPosMass.x,
                    bodyPosMass.y,
                    bodyPosMass.z,
                    bodyPosMass.w,
                    softeningSquared );
                acc[0] += fx;
                acc[1] += fy;
                acc[2] += fz;
            }
            __syncthreads();
        }
        force[3*i+0] = acc[0];
        force[3*i+1] = acc[1];
        force[3*i+2] = acc[2];
    }
}

```

与之前用到的内核一样，循环展开得到了更好的性能。表 14-2 总结了在使用共享内存实现时循环展开的效果。此时最优展开因子为 4，得到 18% 的性能提升。

表 14-2 使用共享内存实现的内核的循环展开效果

展开因子	个体间作用力数目 (10 亿个 / 秒)
1	38.2
2	44.5
3	42.6
4	45.2

14.4 基于常量内存实现

Stone 等人描述了一种直接库伦求和 (direct coulomb summation, DCS) 的方法，使用

共享内存来存储分子建模应用中的图像格点[⊖]，因此必须使用常量内存储存个体的数据。代码清单 14-5 展示了一个 CUDA 内核，使用与我们的引力计算相同的方式进行处理。对于给定的一个内核，只有 64KB 的常量内存可以供开发人员使用，因此每一次内核调用只能处理大约 4000 个 16 字节长的个体数据。常量 `g_bodiesPerPass` 指定最内层循环可以考虑的个体数。

因为在最内层循环中，每个线程正在读取相同的个体数据，广播机制能够优化同一线程束的所有线程的读操作，使得常量内存能够获得很好的效果。

代码清单 14-5 N- 体内核（常量内存）

```

const int g_bodiesPerPass = 4000;
__constant__ __device__ float4 g_constantBodies[g_bodiesPerPass];

template<typename T>
__global__ void
ComputeNBodyGravitation_GPU_AOS_const(
    T *force,
    T *posMass,
    T softeningSquared,
    size_t n,
    size_t N )
{
    for ( int i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        T acc[3] = {0};
        float4 me = ((float4 *) posMass)[i];
        T myX = me.x;
        T myY = me.y;
        T myZ = me.z;
        for ( int j = 0; j < n; j++ ) {
            float4 body = g_constantBodies[j];
            float fx, fy, fz;
            bodyBodyInteraction(
                &fx, &fy, &fz,
                myX, myY, myZ,
                body.x, body.y, body.z, body.w,
                softeningSquared);
            acc[0] += fx;
            acc[1] += fy;
            acc[2] += fz;
        }
        force[3*i+0] += acc[0];
        force[3*i+1] += acc[1];
        force[3*i+2] += acc[2];
    }
}

```

正如代码清单 14-6 展示的，主机端的代码必须遍历所有的个体，在每一次调用内核前先调用 `cudaMemcpyToSymbolAsync()` 函数复制数据到常量内存。

[⊖] www.ncbi.nlm.nih.gov/pubmed/17894371。

代码清单 14-6 主机端代码（常量内存 N- 体）

```

float
ComputeNBodyGravitation_GPU_AOS_const(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N
)
{
    cudaError_t status;
    cudaEvent_t evStart = 0, evStop = 0;
    float ms = 0.0;
    size_t bodiesLeft = N;

    void *p;
    CUDART_CHECK( cudaGetSymbolAddress( &p, g_constantBodies ) );
    CUDART_CHECK( cudaEventCreate( &evStart ) );
    CUDART_CHECK( cudaEventCreate( &evStop ) );
    CUDART_CHECK( cudaEventRecord( evStart, NULL ) );
    for ( size_t i = 0; i < N; i += g_bodiesPerPass ) {
        // bodiesThisPass = max(bodiesLeft, g_bodiesPerPass);
        size_t bodiesThisPass = bodiesLeft;
        if ( bodiesThisPass > g_bodiesPerPass ) {
            bodiesThisPass = g_bodiesPerPass;
        }
        CUDART_CHECK( cudaMemcpyToSymbolAsync(
            g_constantBodies,
            ((float4 *) posMass)+i,
            bodiesThisPass*sizeof(float4),
            0,
            cudaMemcpyDeviceToDevice,
            NULL ) );
        ComputeNBodyGravitation_GPU_AOS_const<float> <<<300,256>>>(
            force, posMass, softeningSquared, bodiesThisPass, N );
        bodiesLeft -= bodiesThisPass;
    }
    CUDART_CHECK( cudaEventRecord( evStop, NULL ) );
    CUDART_CHECK( cudaDeviceSynchronize() );
    CUDART_CHECK( cudaEventElapsedTime( &ms, evStart, evStop ) );
Error:
    cudaEventDestroy( evStop );
    cudaEventDestroy( evStart );
    return ms;
}

```

14.5 基于线程束洗牌实现

流处理器簇 (SM) 3.x 新增加了线程束洗牌指令 (参见 8.6.1 节的描述), 让线程能够在寄存器之间交换数据而不用通过共享内存中转。`_shfl()` 内置函数能够将某个线程的寄存器数据直接向同线程束中其他所有线程进行广播。就如代码清单 14-4 所展示的, 代替线程块大小的分块和共享内存, 我们可以使用尺寸为 32 的数据分块 (对应线程束大小), 然后在线程束内部通过广播将个体的数据在线程之间进行读取。

有趣的是, 这种策略相对于共享内存实现方式降低了 25% 的性能 (每秒 340 亿个相对于 452 亿个个体相互作用力计算)。线程洗牌指令的开销几乎等同共享内存的访问, 而且计算的

执行是按线程束（32 线程）尺寸分块的而不是线程块的尺寸。看起来使用线程洗牌指令代替读写共享内存获得的效果更好，而不只是代替单纯的读共享内存。线程洗牌应该只被用于当内核需要共享内存做其他事的时候。

代码清单 14-7 ComputeNBodyGravitation_Shuffle 内核

```

__global__ void
ComputeNBodyGravitation_Shuffle(
    float *force,
    float *posMass,
    float softeningSquared,
    size_t N )
{
    const int laneid = threadIdx.x & 31;
    for ( int i = blockIdx.x*blockDim.x + threadIdx.x;
          i < N;
          i += blockDim.x*gridDim.x )
    {
        float acc[3] = {0};
        float4 myPosMass = ((float4 *) posMass)[i];

        for ( int j = 0; j < N; j += 32 ) {
            float4 shufSrcPosMass = ((float4 *) posMass)[j+laneid];
#pragma unroll 32
            for ( int k = 0; k < 32; k++ ) {
                float fx, fy, fz;
                float4 shufDstPosMass;

                shufDstPosMass.x = __shfl( shufSrcPosMass.x, k );
                shufDstPosMass.y = __shfl( shufSrcPosMass.y, k );
                shufDstPosMass.z = __shfl( shufSrcPosMass.z, k );
                shufDstPosMass.w = __shfl( shufSrcPosMass.w, k );

                bodyBodyInteraction(
                    &fx, &fy, &fz,
                    myPosMass.x, myPosMass.y, myPosMass.z,
                    shufDstPosMass.x,
                    shufDstPosMass.y,
                    shufDstPosMass.z,
                    shufDstPosMass.w,
                    softeningSquared);
                acc[0] += fx;
                acc[1] += fy;
                acc[2] += fz;
            }
        }

        force[3*i+0] = acc[0];
        force[3*i+1] = acc[1];
        force[3*i+2] = acc[2];
    }
}

```

14.6 多 GPU 及其扩展性

因为具有高计算密度，使用多个 GPU，N- 体计算将获得很好的扩展性。使用可分享锁页内存来储存个体数据，可以被系统中的多个 GPU 轻松引用。对于一个包含 k 个 GPU 的系统，

每个 GPU 将被分配 N/k 个作用力的计算。[⊖] 我们使用多 GPU 实现 N- 体应用的详情在第 9 章已经描述。计算中，数据行被平均分配到每一个 GPU，输入的数据通过可分享锁页内存广播到所有 GPU 并且每一个 GPU 独立的计算自己的输出。在使用多 GPU 的 CUDA 应用中，CPU 端可以是多线程，也可以是单线程。第 9 章给出了优化的 N- 体实现，使用了这两种优化策略。

对于 N- 体计算，CPU 端使用单线程和多线程性能相当，因为需要 CPU 做的工作很少。表 14-3 总结了采用多达 4 个 GPU，CPU 端使用多线程解决问题规模为 96k 个个体时的扩展性。计算效率是通过单个 GPU 展现的性能与最好性能比较得到的。从这个结果可知，计算性能仍然有提升的空间，因为这里报告的性能结果中包括了所有时间步长中 GPU 对于内存分配与释放的操作时间。

表 14-3 N- 体的可扩展性评价

GPU 数目	性能 (每秒可计算的个体间作用力数目，单位 10 亿个 / 秒)	效 率
1	44.1	100 %
2	85.6	97.0%
3	124.2	93.4%
4	161.5	91.6%

14.7 CPU 的优化

在 CUDA 移植的论文中，会经常和没有进行最优化处理的 CPU 实现相比较。虽然在这些论文中描述的负载上，CUDA 硬件的计算速度比 CPU 快，但是得到的加速比往往比一旦对 CPU 实现予以优化的结果高。

为了得到一些有助于在 GPU 与最先进的 CPU 优化间进行权衡的启示，我们将使用让多核 CPU 达到峰值性能必要的两个关键策略来优化 N- 体计算。

- SIMD 指令可以在一个指令周期执行多个单精度浮点数的计算。
- 当 CPU 有可用的执行核心时，多线程将实现趋近于线性的加速比。从 2006 年开始，多核 CPU 已经被广泛地使用，预计 N- 体计算的加速比和核心数量呈线性关系。

既然 N- 体计算有如此高的计算密度，我们不必关注亲和性（例如，尝试使用 NUMA API 来关联内存缓冲区到特定 CPU）。这个计算存在大量重用数据，CPU 的缓存将使得计算仅需少量内存的流量。

20 世纪 90 年代末期，从奔腾 III（Pentium III）开始，英特尔的 x86 构架增加了 SIMD 流扩展指令集（SSE）。它们增加了一组八个 128 位的 XMM 寄存器，能够操作 4 个被捆绑在一起的 32 位浮点数。[⊖] 例如，ADDPS 指令使用 XMM 寄存器能实现 4 个浮点数的并行相加。

⊖ 我们的实现要求 N 能被 k 整除。

⊖ 英特尔后来又增加了一些指令，能够让 XMM 寄存器捆绑处理多个整型数（多达 16 字节）或者两个双精度的浮点数，但是我们没有使用这些特性。我们也没有使用 AVX（高级向量扩展）指令集。AVX 的特性在于寄存器和指令支持两倍位宽的单指令多数据操作（256 位），因此它可能拥有两倍于当前性能的潜能。

当将 N- 体计算采用 SSE 指令集，我们一直在使用的 AOS（结构体数组）内存结构将成为一个问题。虽然个体数据和 XMM 寄存器位数一样是 16 个字节，指令集却要求我们重新排列数据，将 x 、 y 、 z 和 $mass$ 分量放在独立的寄存器。而不是在计算个体间作用力时执行这一操作，我们重新排列内存结构为“数组结构”：不是使用一个 float4 数组（每个元素包含给定个体的 x 、 y 、 z 和 $mass$ ），我们使用 4 个 float 数组、即 x 坐标数组、 y 坐标数组、 z 坐标数组和 $mass$ 坐标数组。数据重新排列后，仅通过 4 个机器指令即可将 4 个个体的数据载入 XMM 寄存器；4 个个体位置的差向量将通过 3 条 SUBPS 指令计算得到；等等。

为了简化 SSE 指令集代码编写工作，英特尔与编译器厂商合作，使 SSE 指令集获得跨平台的支持。特殊的数据类型 `_m128` 对应于 128 位的寄存器和操作数大小，可以用于内置函数，例如对应于 SUBPS 指令的 `_mm_sub_ps()`。

为了便于 N- 体计算的实现，我们还需要使用全精度的平方根倒数实现。SSE 指令集有一个 RSQRTSS 内置指令来计算近似的平方根倒数，但是它的 12 位的估计值必须使用牛顿-拉夫逊迭代精化到完整浮点精度。^①

$$\begin{aligned}x_0 &= \text{RSQRTSS}(a) \\x_1 &= \frac{x_0(3 - ax_0^2)}{2}\end{aligned}$$

代码清单 14-8 给出了 SSE 指令集在计算个体间作用力的实现，它带有 2 个 `_m128` 类型变量代表 2 个个体数据，并行地计算 4 个个体间的作用力，并返回 3 个最终力向量。代码清单 14-8 描述的功能等同于代码清单 14-1 和 14-2，但可读性明显减弱。注意： x_0 、 y_0 和 z_0 变量包含同一个个体的数据，通过复制 4 次 `_m128` 型变量得到。

为了利用多核的优势，我们必须使用多个线程，并且让每一个线程承担部分计算任务。多核 CPU 使用和多 GPU 相同的策略：^②只需要把输出行平均划分给每一个线程（每个 CPU 核一个线程），在每一时间步长，父线程通过信号指挥工作线程进行工作并等待工作线程结束。因为线程的创建需要时间并且可能失败，我们的应用在初始化时创建一个 CPU 线程池，并使用线程同步让工作线程等待它们的工作，当工作完成时发出信号。

本书的可移植线程库，详见附录 A-2。库中实现了函数 `processorCount()`，返回系统中 CPU 核的数目，并有一个 C++ 类 `wokerThread` 包含了创建和销毁 CPU 线程以及以同步或异步的方法委派工作的功能。在使用 `delegateAsynchronous()` 成员函数异步委派工作后，调用静态函数 `waitForAll()` 等待工作线程的结束。

代码清单 14-9 给出了分配 N- 体计算任务到 CPU 工作线程的代码。`sseDelegation` 结构是用来向每一个工作线程传达委派信息的；`delegateSynchronous` 函数调用需要传递一个函数指针和一个将传递给传入函数的 `void*` 类型的指针（在这个例子中，`void*` 指向对应 CPU 线程的 `sseDelegation` 结构）。

^① 这段代码没有出现在 SSE 编译器支持中而且特别难找。我们的实现来自 http://nume.google.com/svn/trunk/fosh/src/sse_approx.h。

^② 事实上，在第 9 章的多线程多 GPU 中使用了相同的独立于平台的线程库。

代码清单 14-8 个体间相互作用 (SSE 版本)

```

static inline __m128
rcp_sqrt_nr_ps(const __m128 x)
{
    const __m128
        nr      = _mm_rsqrt_ps(x),
        muls   = _mm_mul_ps(_mm_mul_ps(nr, nr), x),
        beta   = _mm_mul_ps(_mm_set_ps1(0.5f), nr),
        gamma  = _mm_sub_ps(_mm_set_ps1(3.0f), muls);
    return _mm_mul_ps(beta, gamma);
}

static inline __m128
horizontal_sum_ps( const __m128 x )
{
    const __m128 t = _mm_add_ps(x, _mm_movehl_ps(x, x));
    return _mm_add_ss(t, _mm_shuffle_ps(t, t, 1));
}

inline void
bodyBodyInteraction(
    __m128& f0,
    __m128& f1,
    __m128& f2,

    const __m128& x0,
    const __m128& y0,
    const __m128& z0,

    const __m128& x1,
    const __m128& y1,
    const __m128& z1,
    const __m128& mass1,

    const __m128& softeningSquared )
{
    __m128 dx = _mm_sub_ps( x1, x0 );
    __m128 dy = _mm_sub_ps( y1, y0 );
    __m128 dz = _mm_sub_ps( z1, z0 );

    __m128 distSq =
        _mm_add_ps(
            _mm_add_ps(
                _mm_mul_ps( dx, dx ),
                _mm_mul_ps( dy, dy )
            ),
            _mm_mul_ps( dz, dz )
        );
    distSq = _mm_add_ps( distSq, softeningSquared );

    __m128 invDist = rcp_sqrt_nr_ps( distSq );
    __m128 invDistCube =
        _mm_mul_ps(
            invDist,
            _mm_mul_ps(
                invDist, invDist )
        );

    __m128 s = _mm_mul_ps( mass1, invDistCube );

    f0 = _mm_add_ps( a0, _mm_mul_ps( dx, s ) );
    f1 = _mm_add_ps( a1, _mm_mul_ps( dy, s ) );
    f2 = _mm_add_ps( a2, _mm_mul_ps( dz, s ) );
}

```

代码清单 14-9 多线程 SSE 指令集（主控线程代码）

```

float
ComputeGravitation_SSE_threaded(
    float *force[3],
    float *pos[4],
    float *mass,
    float softeningSquared,
    size_t N
)
{
    chTimerTimestamp start, end;
    chTimerGetTime( &start );

    {
        sseDelegation *psse = new sseDelegation[g_numCPUCores];
        size_t bodiesPerCore = N / g_numCPUCores;
        if ( N % g_numCPUCores ) {
            return 0.0f;
        }
        for ( size_t i = 0; i < g_numCPUCores; i++ ) {
            psse[i].hostPosSOA[0] = pos[0];
            psse[i].hostPosSOA[1] = pos[1];
            psse[i].hostPosSOA[2] = pos[2];
            psse[i].hostMassSOA = mass;
            psse[i].hostForceSOA[0] = force[0];
            psse[i].hostForceSOA[1] = force[1];
            psse[i].hostForceSOA[2] = force[2];
            psse[i].softeningSquared = softeningSquared;

            psse[i].i = bodiesPerCore*i;
            psse[i].n = bodiesPerCore;
            psse[i].N = N;

            g_CPUThreadPool[i].delegateAsynchronous(
                sseWorkerThread,
                &psse[i] );
        }
        workerThread::waitForAll( g_CPUThreadPool, g_numCPUCores );
        delete[] psse;
    }

    chTimerGetTime( &end );

    return (float) chTimerElapsedTime( &start, &end ) * 1000.0f;
}

```

最后，代码清单 14-10 给出了 sseDelegation 结构和被代码清单 14-9 中 ComputeGravitation_SSE_threaded 调用的委派函数。它实现一次完成 4 个个体间作用力的计算，在存储最后的输出结果前使用 horizontal_sum_ps() 函数累计得到的 4 个部分和。这个函数以及其调用的所有函数，使用 SOA 的内存结构进行所有的输入和输出。

代码清单 14-10 从属线程代码 sseWokerThread

```

struct sseDelegation {
    size_t i; // base offset for this thread to process
    size_t n; // size of this thread's problem
    size_t N; // total number of bodies

    float *hostPosSOA[3];
    float *hostMassSOA;

```

```

float *hostForceSOA[3];
float softeningSquared;

};

void
sseWorkerThread( void *_p )
{
    sseDelegation *p = (sseDelegation *) _p;
    for (int k = 0; k < p->n; k++)
    {
        int i = p->i + k;
        __m128 ax = __mm_setzero_ps();
        __m128 ay = __mm_setzero_ps();
        __m128 az = __mm_setzero_ps();
        __m128 *px = (__m128 *) p->hostPosSOA[0];
        __m128 *py = (__m128 *) p->hostPosSOA[1];
        __m128 *pz = (__m128 *) p->hostPosSOA[2];
        __m128 *pmass = (__m128 *) p->hostMassSOA;
        __m128 x0 = __mm_set_ps1( p->hostPosSOA[0][i] );
        __m128 y0 = __mm_set_ps1( p->hostPosSOA[1][i] );
        __m128 z0 = __mm_set_ps1( p->hostPosSOA[2][i] );

        for ( int j = 0; j < p->N/4; j++ ) {

            bodyBodyInteraction(
                ax, ay, az,
                x0, y0, z0,
                px[j], py[j], pz[j], pmass[j],
                __mm_set_ps1( p->softeningSquared ) );

        }
        // Accumulate sum of four floats in the SSE register
        ax = horizontal_sum_ps( ax );
        ay = horizontal_sum_ps( ay );
        az = horizontal_sum_ps( az );

        __mm_store_ss( (float *) &p->hostForceSOA[0][i], ax );
        __mm_store_ss( (float *) &p->hostForceSOA[1][i], ay );
        __mm_store_ss( (float *) &p->hostForceSOA[2][i], az );
    }
}

```

14.8 小结

由于指令集和架构的不同，性能是以每秒衡量计算个体间作用力数目，而不是用每秒执行浮点运算次数。性能是在一个具有 2 个至强 E5-2670 处理器（类似于亚马逊的 cc2.8xlarge 实例类型）、64GB 的 RAM 和 4 个 GK104 构架 GPU（主频为 800MHz 左右）的沙桥系统上衡量的。该 GK104 构架 GPU 插在有 16 个 PCIe 3.0 插槽的双 GPU 主板上。

表 14-4 来自 CPU 优化的加速比

实现方法	每秒计算的个体间作用力数（10 亿个）	相较标量 CPU 的加速比
标量 CPU	0.017	1 倍
SSE 指令集	0.307	17.8 倍
多线程 SSE 指令集	5.650	332 倍

表 14-4 总结了来自 CPU 优化的加速比。所有的测量均是在拥有双至强 E2670 CPU (2.6GHZ) 的服务器上进行的。在这个系统中，代码清单 14-2 中的传统 CPU 代码执行速度是每秒 17.2M 个个体间作用力，而单线程的 SSE (单指令多数据) 代码执行速度是每秒 307M，是前者的 17.8 倍。正如预期的那样，多线程的 SSE 代码实现了良好的加速比，使用 32 个 CPU 线程获得每秒 5650M 个个体间作用力，是单线程的 18 倍。在结合 SSE 和多线程之后，这个平台上的 CPU 总加速比超过 300 倍。

因为我们从 CPU 优化得到了巨大的性能提升，使用 GPU 提升性能不再那么明显。^① 在我们的测试中性能最高的内核（代码清单 14-4 中的共享内存实现并以 4 为循环展开因子）进行每秒 452 亿个个体间作用力的计算，是使用最快的多线程 SSE 指令集的 8 倍。这个结果低估了 CUDA 在某些方面的性能的优点，因为用于测试的服务器拥有 2 个高端 CPU，并且 GPU 为了降低功耗和散热均采取了降频措施。

此外，未来这 2 种技术都会有相应的改进。对于 CPU，把该类负载移植到 AVX (高级矢量扩展) 将可能有双倍的性能，但是它只能在沙桥及其之后的芯片上运行，而且优化的 CPU 实现没有利用对称性。对于 GPU，英伟达的 GK110 大约是 GK104 的 2 倍大 (大约 2 倍快)。通过比较代码清单 14-1 和代码清单 14-9 的源代码 (即 GPU 和 SSE 分别实现的个体间作用力计算的核心代码)，显而易见，支持 CUDA 的因素并不只是性能上优于 CPU。Vincent Natoli 博士在他 2010 年 6 月的文章 “Kudos for CUDA” 中暗示了这一权衡。^②

类似的，我们发现在许多情况下，CUDA 表达并行算法和 CPU 代码相比更紧凑、简洁和可读。这些领域包括石油和天然气、生物信息学和金融学。在最近的一个项目，我们将一个 3500 行的高度优化的 C 代码精简为大约 800 行的 CUDA 内核。优化的 C 代码塞入了内联汇编、SSE 宏、循环展开和很多特殊情况，使其在未来难以阅读、理解算法含义并扩展。相比之下，CUDA 代码更少并且可读性更好。最终，它将更容易维护。

虽然在这个应用程序中开发 SSE 的实现是可行的 (一个核心的个体间作用力的计算使用了 50 行代码，见代码清单 14-8)，但是很难想象使用 SSE 指令集优化类似“类鸟群”这类应用的源代码会多么庞大。因为这类算法的每个个体必须进行条件评估，但是在 CUDA 硬件上运行时，这些代码将被按分支打包。SSE 指令集支持断定形式 (使用掩码和布尔指令序列，如用 ANDPS/ANDNOTPS/ORPS 来构建结果) 和分支 (通常使用 MOVMSKPS 提取条件进行评估)，但是，要得到的此类负载的理论加速比极限需要巨大的工程投入，除非它们能用向量化编译器进行自动提取。

^① 平心而论，这一结论也适用于这本书中的其他负载，例如第 11 章的 SAXPY 实现和第 15 章的归一化互相关实现。移植这些负载到多线程单指令多数据 (SIMD) 模式，和 CUDA 版本比较，需要把性能与工程投资、可读性以及可维护性进行权衡。

^② www.hpcwire.com/hpcwire/2010-01-06/kudos_for_cuda.html。

14.9 参考文献与延伸阅读

N-体问题及具有类似高计算密度的算法是获取高效加速比的资源，因为它们可以接近GPU计算能力的理论上限。下面只是无数类似于N-体这样的计算密集型方法的部分示例。

重力模拟

Burtscher, Martin, and Keshav Pingali. An efficient CUDA implementation of the tree-based Barnes-Hut n-body algorithm. In GPU Gems Emerald Edition, Wen-Mei Hwu, ed., Morgan-Kaufmann, 2011, Burlington, MA, pp. 75 ~ 92.

<http://cs.txstate.edu/~burtscher/papers/gcg11.pdf>

Harris, Mark, Lars Nyland, and Jan Prins. Fast n-body simulation with CUDA. In GPU Gems 3, Addison-Wesley, Boston, MA, 2007, pp. 677 ~ 695.

http://developer.nvidia.com/GPUGems3/gpugems3_ch31.html

分子模拟

Götz, Andreas, Mark J. Williamson, Dong Xu, Duncan Poole, Scott Le Grand, and Ross C. Walker. Routine microsecond molecular dynamics simulations with AMBER on GPUs—Part I: Generalized Born, *J. Chem. Theory Comput.* 8 (5), 2012, pp. 1542 ~ 1555.

Hwu, Wen-Mei, and David Kirk. Programming Massively Parallel Processors. Morgan-Kaufmann, 2010, pp. 173-188.

Hardy, David J., John E. Stone, Kirby L. Vandivort, David Gohara, Christopher Rodrigues, and Klaus Schulten. Fast molecular electrostatics algorithms on GPUs. In GPU Computing Gems, Elsevier, Burlington, MA, 2011, pp. 43 ~ 58.

Stone, John E., James C. Phillips, Peter L. Freddolino, David J. Hardy, Leonardo G. Trabuco, and Klaus Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* 28 (2007), pp. 2618 ~ 2640.

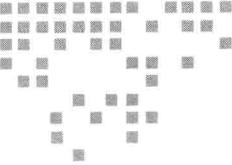
<http://cacs.usc.edu/education/cs653/Stone-MDGPU-JCC07.pdf>

Stone, John E., David J. Hardy, Barry Israelewitz, and Klaus Schulten. GPU algorithms for molecular modeling. In *Scientific Computing with Multicore and Accelerators*, Jack Dongarra, David A. Bader, and Jakob Kurzak, eds. Chapman & Hall/CRC Press, London, UK, 2010, pp. 351 ~ 371.

类鸟群

da Silva, A.R., W.S. Lages, and L. Chaimowicz. Boids that see: Using self-occlusion for simulating large groups on GPUs. *ACM Comput. Entertain.* 7 (4), 2009.

<http://doi.acm.org/10.1145/1658866.1658870>



图像处理的归一化相关系数计算

在图像处理与计算机视觉领域中，归一化互相关（normalized cross-correlation）是一种很流行的模板匹配算法。这里所谓的模板（template）通常是一幅我们感兴趣的图像。通过反复地计算模板图像和输入图像子集相对应的像素之间的统计信息，搜索算法可找到当前输入图像中包含的模板实例。

归一化互相关的流行源于其幅度独立性（amplitude independence）。这也就意味着在图像处理中，当输入图像与模板的亮度发生变化时，其统计信息仍是稳定的。鉴于归一化相关的用途极广泛，并且其计算复杂度也足够密集，已经促使很多企业为其构建定制的硬件。本章针对 8 位灰度图像的归一化互相关进行优化实现，但其中的概念也可以扩展到其他类型的图像处理和计算机视觉算法领域中。

15.1 概述

图像与模板这两幅二维图像通过以下公式来计算相关系数：

$$\gamma(s, t) = \frac{\sum_x \sum_y [I(x, y) - \bar{I}(x, y)][T(x - s, y - t) - \bar{T}]}{\sqrt{\sum_x \sum_y [I(x, y) - \bar{I}(x, y)]^2} \sqrt{\sum_x \sum_y [T(x - s, y - t) - \bar{T}]^2}}$$

其中， I 和 T 分别代表图像和模板， \bar{T} 为模板像素的平均值， \bar{I} 为图像中对应模板范围的像素的平均值。

该系数的值将介于 $[-1.0, 1.0]$ 的范围。值 1.0 意味着一个完美的匹配。归一化互相关的一种优化实现方案，不仅提取出能被预先计算的统计量，而且还使用加和而不是平均值计算，以避免输入数据的多次遍历。若有 N 个像素进行比较，用 $\sum_x \sum_y [T(x, y)]$ 和 $\frac{\sum_x \sum_y [T(x, y)]}{N}$

分别来代替 \bar{T} 和 \bar{I}^\ominus ，并在分子、分母上同时乘以 N ，即会产生一个完全使用加和来表示的系数。略去坐标，公式可以整理成：

$$\frac{N\sum IT - \Sigma I \Sigma T}{\sqrt{(N\sum I^2 - (\Sigma I)^2)(N\sum T^2 - (\Sigma T)^2)} \ominus}$$

假设该模板对多个其他的相关计算来说都是相同的，则可以预先计算出模板的统计量 ΣT 和 ΣT^2 ，同理可以计算分母的子表达 $(N\sum T^2 - (\Sigma T)^2)$ 。下表给出上述公式符号的 C 语言变量表示：

统计信息	C 变量名称
ΣI	SumI
ΣT	SumT
ΣIT	SumIT
ΣI^2	SumSqI
ΣT^2	SumSqT

然后可以使用以下函数来计算一个归一化相关值。

```
float
CorrelationValue( float SumI, float SumISq,
float SumT, float SumTSq, float SumIT,
float N )
{
    float Numerator = N*SumIT - SumI*SumT;
    float Denominator = (N*SumISq - SumI*SumI)*
(N*SumTSq - SumT*SumT);
    return Numerator / sqrtf(Denominator);
}
```

实际应用中使用这个算法时，该模板在许多调用中都保持不变，只是在图像的不同偏移处进行匹配。所以预先计算模板的统计量和分母子表达式很有意义。

```
float fDenomExp = N*SumSqT - SumT*SumT;
```

实际上，建议采用双精度来计算 fDenomExp。

```
float fDenomExp = (float) ((double) N*SumSqT - (double) SumT*SumT);
```



注意 此计算在 CPU 上执行，一个模板需要计算一次。

而乘以平方根的倒数比除以平方根运算更快，所以我们有函数 CorrelationValue() 定义如下：

```
float
CorrelationValue( float SumI, float SumISq, float SumIT,
float N, float fDenomExp )
```

⊕ 此处修正了原书错误。——译者注

⊖ 式中的 Σ 表示 Σ_{x, y_0} ——译者注

```

    {
        float Numerator = cPixels*SumIT - SumI*SumT;
        float Denominator = fDenomExp*(cPixels*SumISq - SumI*SumI);
        return Numerator * rsqrtf(Denominator);
    }
}

```

因此，这个算法的优化实现只需要三个用来计算给定相关系数的像素的统计量： ΣI ， ΣI^2 ， ΣIT 。流处理器簇中包含了专门支持整数乘加运算（multiply-add）的硬件，所以 GPU 能够极快的执行此运算。

CUDA 提供了如下几种将数据复制至流处理器簇的方法：

- 使用全局内存或纹理内存存储图像或者模板
- 使用常量内存存储模板或其他可能的特定模板参数（可达 64KB）
- 使用共享内存来保存图像和 / 或模板值以备重用

本章假定所有图像都为 8 位灰度图像。硬件在更高精度的图像中也具有非常好的表现，但这里是为了简化高效利用全局内存和共享内存的问题。

本章中所有的 CUDA 实现都使用了纹理内存来存储同模板对比的图像，这样做有如下几个原因：

- 纹理单元能够更加优雅、高效的处理边界条件。
- 在进行相邻的相关值计算时，纹理缓存可以在重用性方面聚集外部的带宽。
- 对相关性搜索算法来说，纹理缓存的 2D 局部性能能够很好地适合算法的访存模式。

我们将继续权衡使用纹理内存和常量内存存储模板。

15.2 简单的纹理实现

我们利用纹理单元读取图像和模板的像素值来实现第一个归一化互相关。虽然它并不是一个最优实现，甚至没有预先计算模板的统计量。但这种方法简单易懂，并且是更进一步优化（当然也更复杂）的一个良好的基础。

代码清单 15-1 给出了执行此计算的内核程序，它计算出了 5 个数据的加和信息，然后利用前面给出的函数 CorrelationValue() 将浮点型的相关系数写入到输出数组中。但要注意的是，计算 fDenomExp 的表达式可能会发出警告，这是因为 SM1.3 架构之前不支持双精度浮点型的计算，但只要模板像素数量在允许的范围之内，内核仍可以正常工作。

(xUL, yUL) 表示图像左上角， w 和 h 表示搜索窗口的宽和高，也就是表示系数的输出数组。如果该模板是在纹理内存中，则其纹理图像的左上角可由 $(xTemplate, yTemplate)$ 表示。

最后，偏移量 $(xOffset, yOffset)$ 描述了模板是如何与待比较的图像进行叠加的。当读取图像像素时，这个偏移量被添加到了左上角为 (xUL, yUL) 的搜索矩形框所在的坐标中。

从模板所在位置向周边行进时，我们看到其相关函数递减，这对我们很有启发。示例程序 normalizedCrossCorrelation.cu 记录下的模板附近的相关系数如下所示：

模板周围：

```
Neighborhood around template:
0.71 0.75 0.79 0.81 0.82 0.81 0.81 0.80 0.78
0.72 0.77 0.81 0.84 0.84 0.84 0.83 0.81 0.79
0.74 0.79 0.84 0.88 0.88 0.87 0.85 0.82 0.79
0.75 0.80 0.86 0.93 0.95 0.91 0.86 0.83 0.80
0.75 0.80 0.87 0.95 1.00 0.95 0.88 0.83 0.81
0.75 0.80 0.86 0.91 0.95 0.93 0.87 0.82 0.80
0.75 0.80 0.84 0.87 0.89 0.88 0.85 0.81 0.78
0.73 0.78 0.81 0.83 0.85 0.85 0.82 0.79 0.76
0.71 0.75 0.78 0.81 0.82 0.82 0.80 0.77 0.75
```

图 15-1 中包含了很多硬币，右下角的被高亮显示的 52×52 大小的镍币图像便是默认模板。默认的程序可以有选择的输出一个 PGM 文件，并转换相关像素值的范围为 0 ~ 255。图 15-1 中的高亮部分的相关图像如图 15-2 所示。其他镍币所在的位置的亮度较强，表明匹配的程度较好，而那些硬币的响应更弱。



图 15-1 Coins.pgm (高亮部分为默认的模板)

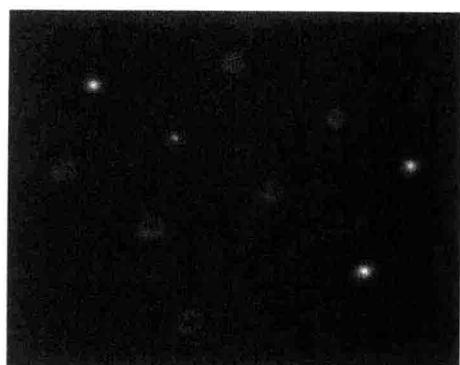


图 15-2 默认模板的相关图像

代码清单 15-1 corrTexTex2D_kernel

```
_global__ void
corrTexTex2D_kernel(
    float *pCorr, size_t CorrPitch,
    float cPixels,
    int xOffset, int yOffset,
    int xTemplate, int yTemplate,
    int wTemplate, int hTemplate,
    float xUL, float yUL, int w, int h )
{
    size_t row = blockIdx.y*blockDim.y + threadIdx.y;
    size_t col = blockIdx.x*blockDim.x + threadIdx.x;

    // adjust pCorr to point to row
    pCorr = (float *) ((char *) pCorr+row*CorrPitch);

    // No __syncthreads in this kernel, so we can early-out
    // without worrying about the effects of divergence.
    if ( col >= w || row >= h )
        return;

    int SumI = 0;
```

```

int SumT = 0;
int SumISq = 0;
int SumTSq = 0;
int SumIT = 0;
for ( int y = 0; y < hTemplate; y++ ) {
    for ( int x = 0; x < wTemplate; x++ ) {
        unsigned char I = tex2D( texImage,
            (float) col+xUL+xOffset+x, (float) row+yUL+yOffset+y );
        unsigned char T = tex2D( texTemplate,
            (float) xTemplate+x, (float) yTemplate+y );
        SumI += I;
        SumT += T;
        SumISq += I*I;
        SumTSq += T*T;
        SumIT += I*T;
    }
    float fDenomExp = (float) ( (double) cPixels*SumTSq -
        (double) SumT*SumT );
    pCorr[col] = CorrelationValue(
        SumI, SumISq, SumIT, SumT, cPixels, fDenomExp );
}
}

```

代码清单 15-2 给出了调用 corrTexTex2D_kernel() 的主机端代码。它是用来测试源文件 normalizedCrossCorrelation.cu 的性能表现的，所以有很多参数。此主机端的函数此时只是用来传递内核函数所需要的参数，但它在后面的实现中，还会检查它所发现的设备的属性，并启动不同的内核。对于一个大小适中的图像来说，做这样的检查的成本相比于 GPU 运行时间是微不足道的。

代码清单 15-2 corrTexTex2D() (主机端代码)

```

void
corrTexTex2D(
    float *dCorr, int CorrPitch,
    int wTile,
    int wTemplate, int hTemplate,
    float cPixels,
    float fDenomExp,
    int sharedPitch,
    int sharedPitch,
    int xOffset, int yOffset,
    int xTemplate, int yTemplate,
    int xUL, int yUL, int w, int h,
    dim3 threads, dim3 blocks,
    int sharedMem )
{
    corrTexTex2D_kernel<<<blocks, threads>>>(
        dCorr, CorrPitch,
        cPixels,
        xOffset, yOffset,
        xTemplate+xOffset, yTemplate+yOffset,
        wTemplate, hTemplate,
        (float) xUL, (float) yUL, w, h );
}

```

在搜索过程中，如果应用程序选择不同的模板和不同的图像时，均采用纹理实现是非常

适宜的一例如，在模板和图像比对时，需要变换模板的数据时。但是在大多数应用中，模板仅被选择一次，并且用来和图像内很多不同偏移的内容作计算和比较。本章的剩余部分将会针对这一情况继续进行优化。

15.3 常量内存中的模板

在大多数模板匹配的应用中，我们都是使用同一个模板与输入图像的不同偏移区域做相关计算。在这种情况下，模板的统计量（*SumT* 和 *fDenomExp*）是可以被预先计算的，并且模板数据可以被放入到特殊的存储器当中。对于 CUDA 来说，显而易见，我们可以将模板数据放入常量内存中，这样模板数据可以以广播的方式传递给线程，并在图像中的不同位置做相关计算。

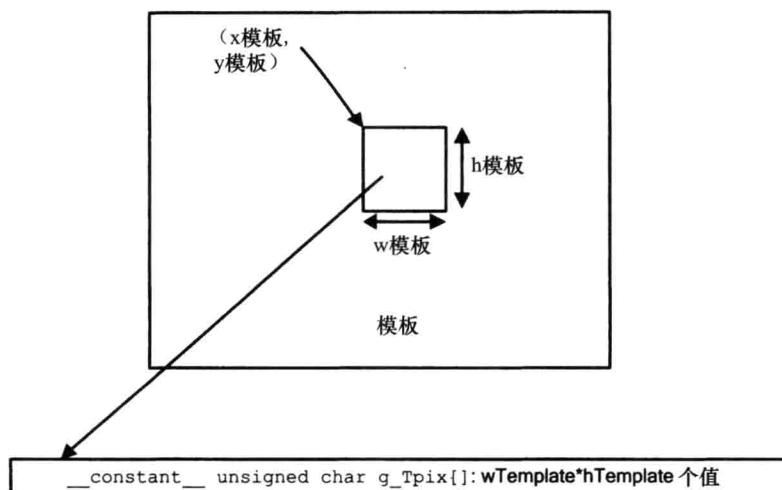


图 15-3 常量内存中的模板

代码清单 15-3 的 `CopyToTemplate` 函数的功能是从输入图像中提取出一个矩形块，计算其统计量，然后将数据和统计量一起复制到常量内存中。

代码清单 15-3 `CopyToTemplate`

```

cudaError_t
CopyToTemplate(
    unsigned char *img, size_t imgPitch,
    int xTemplate, int yTemplate,
    int wTemplate, int hTemplate,
    int OffsetX, int OffsetY
)
{
    cudaError_t status;
    unsigned char pixels[maxTemplatePixels];

    int inx = 0;

```

```

int SumT = 0;
int SumTSq = 0;
int cPixels = wTemplate*hTemplate;
size_t sizeOffsets = cPixels*sizeof(int);
float fSumT, fDenomExp, fcPixels;

cudaMemcpy2D(
    pixels, wTemplate,
    img+yTemplate*imgPitch+xTemplate, imgPitch,
    wTemplate, hTemplate,
    cudaMemcpyDeviceToHost );

cudaMemcpyToSymbol( g_Tpix, pixels, cPixels );

for ( int i = OffsetY; i < OffsetY+hTemplate; i++ ) {
    for ( int j = OffsetX; j < OffsetX+wTemplate; j++ ) {
        SumT += pixels[inx];
        SumTSq += pixels[inx]*pixels[inx];
        poffsetx[inx] = j;
        poffsety[inx] = i;
        inx += 1;
    }
}
g_cpuSumT = SumT;
g_cpuSumTSq = SumTSq;

cudaMemcpyToSymbol(g_xOffset, poffsetx, sizeOffsets);
cudaMemcpyToSymbol(g_yOffset, poffsety, sizeOffsets);

fSumT = (float) SumT;
cudaMemcpyToSymbol(g_SumT, &fSumT, sizeof(float));

fDenomExp = float( (double)cPixels*SumTSq - (double) SumT*SumT );
cudaMemcpyToSymbol(g_fDenomExp, &fDenomExp, sizeof(float));
fcPixels = (float) cPixels;
cudaMemcpyToSymbol(g_cPixels, &fcPixels, sizeof(float));
Error:
    return status;
}

```

代码清单 15-4 给出了内核函数 corrTemplate2D(), 它从常量内存中的 g_Tpix[] 读取模板的像素值。这个函数比 corrTexTex2D() 更简短，甚至都不必计算模板的统计量。

代码清单 15-4 内核函数 corrTemplate2D

```

__global__ void
corrTemplate2D_kernel(
    float *pCorr, size_t CorrPitch,
    float cPixels, float fDenomExp,
    float xUL, float yUL, int w, int h,
    int xOffset, int yOffset,
    int wTemplate, int hTemplate )
{
    size_t row = blockIdx.y*blockDim.y + threadIdx.y;
    size_t col = blockIdx.x*blockDim.x + threadIdx.x;

    // adjust pointers to row
    pCorr = (float *) ((char *) pCorr+row*CorrPitch);

    // No __syncthreads in this kernel, so we can early-out

```

```

// without worrying about the effects of divergence.
if ( col >= w || row >= h )
    return;

int SumI = 0;
int SumISq = 0;
int SumIT = 0;
int inx = 0;

for ( int j = 0; j < hTemplate; j++ ) {
    for ( int i = 0; i < wTemplate; i++ ) {
        unsigned char I = tex2D( texImage,
            (float) col+xUL+xOffset+i,
            (float) row+yUL+yOffset+j );
        unsigned char T = g_Tpix[inx++];
        SumI += I;
        SumISq += I*I;
        SumIT += I*T;
    }
}
pCorr[col] =
    CorrelationValue(
        SumI, SumISq, SumIT, g_SumT, cPixels, fDenomExp );
}

```

15.4 共享内存中的图像

对于那些供我们的示例程序计算相关值的矩阵来说，CUDA 内核程序展示了同模板匹配的图像的像素是如何被大量复用的。目前为止，我们的代码仅依靠纹理缓存而不需要外部存储便可以进行大量的读取操作。然而，对于更小的模板，通过使用共享内存可以使得图像数据得到更低的延迟，从而进一步提高性能。

使用代码清单 15-1 和 15-3 的内核函数时，它会根据线程块大小将输入图像稳式地划成大小相同的图像块。代码清单 15-5 展示了一个在共享内存上的实现，我们使用了线程块的高度 (blockDim.y)，但显式指定了宽度为 wTile。在我们的示例程序中，wTile 的值为 32。图 15-4 展示了内核函数是如何“过读取”(overfetch) 处于图像块之外 $wTemplate \times hTemplate$ 大小的矩形区域的，边界条件是通过纹理寻址模式处理的。一旦将图像数据复制到共享内存完毕后，内核函数通过 `_syncthreads()` 同步线程块内的各个线程，计算并输出当前图像块的相关系数。

代码清单 15-5 corrShared_kernel()

```

__global__ void
corrShared_kernel(
    float *pCorr, size_t CorrPitch,
    int wTile,
    int wTemplate, int hTemplate,
    float xOffset, float yOffset,
    float cPixels, float fDenomExp, int SharedPitch,
    float xUL, float yUL, int w, int h )

```

```

{
    int uTile = blockIdx.x*wTile;
    int vTile = blockIdx.y*blockDim.y;
    int v = vTile + threadIdx.y;

    float *pOut = (float *) (((char *) pCorr)+v*CorrPitch);
    for ( int row = threadIdx.y;
          row < blockDim.y+hTemplate;
          row += blockDim.y ) {
        int SharedIdx = row * SharedPitch;
        for ( int col = threadIdx.x;
              col < wTile+wTemplate;
              col += blockDim.x ) {

            LocalBlock[SharedIdx+col] =
                tex2D( texImage,
                        (float) (uTile+col+xUL+xOffset),
                        (float) (vTile+row+yUL+yOffset) );

        }
    }
    __syncthreads();

    for ( int col = threadIdx.x;
          col < wTile;
          col += blockDim.x ) {

        int SumI = 0;
        int SumISq = 0;
        int SumIT = 0;
        int idx = 0;
        int SharedIdx = threadIdx.y * SharedPitch + col;
        for ( int j = 0; j < hTemplate; j++ ) {
            for ( int i = 0; i < wTemplate; i++ ) {
                unsigned char I = LocalBlock[SharedIdx+i];
                unsigned char T = g_Tpix[idx++];
                SumI += I;
                SumISq += I*I;
                SumIT += I*T;
            }
            SharedIdx += SharedPitch;
        }
        if ( uTile+col < w && v < h ) {
            pOut[uTile+col] =
                CorrelationValue( SumI, SumISq, SumIT, g_SumT,
                                  cPixels, fDenomExp );
        }
    }
    __syncthreads();
}

```

为确保共享内存中相邻行之间不会发生存储片冲突 (bank conflict), 我们填充每行的数据量为 64 的倍数。

```
sharedPitch = ~63&(((wTile+wTemplate)+63));
```

每个线程块所需要的共享内存的总量为步长 (pitch) 乘以行数 (即块的高度加上模板的高度):

```
sharedMem = sharedPitch*(threads.y+hTemplate);
```

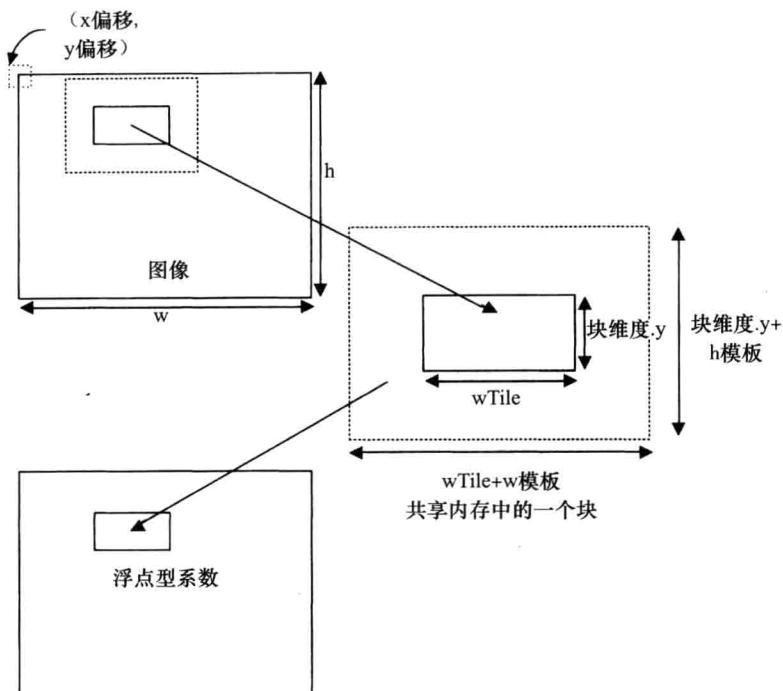


图 15-4 共享内存中的图像

代码清单 15-6 的代码启动内核函数 `corrShared_kernel()` 的同时，检测内核是否需要比当前更多的共享内存。如果需要，它将调用 `corrTexTex2D()`，这个函数适用于任意大小的模板。

代码清单 15-6 `corrShared()` (主机端代码)

```

void
corrShared(
    float *dCorr, int CorrPitch,
    int wTile,
    int wTemplate, int hTemplate,
    float cPixels,
    float fDenomExp,
    int sharedPitch,
    int xOffset, int yOffset,
    int xTemplate, int yTemplate,
    int xUL, int yUL, int w, int h,
    dim3 threads, dim3 blocks,
    int sharedMem )
{
    int device;
    cudaDeviceProp props;
    cudaError_t status;

    CUDART_CHECK( cudaGetDevice( &device ) );
    CUDART_CHECK( cudaGetDeviceProperties( &props, device ) );
    if ( sharedMem > props.sharedMemPerBlock ) {
        dim3 threads88(8, 8, 1);
        dim3 blocks88;
    }
}

```

```

blocks88.x = INTCEIL(w, 8);
blocks88.y = INTCEIL(h, 8);
blocks88.z = 1;
return corrTexTex2D(
    dCorr, CorrPitch,
    wTile,
    wTemplate, hTemplate,
    cPixels,
    fDenomExp,
    sharedPitch,
    xOffset, yOffset,
    xTemplate, yTemplate,
    xUL, yUL, w, h,
    threads88, blocks88,
    sharedMem );
}
corrShared_kernel<<<blocks, threads, sharedMem>>>(
    dCorr, CorrPitch,
    wTile,
    wTemplate, hTemplate,
    (float) xOffset, (float) yOffset,
    cPixels, fDenomExp,
    sharedPitch,
    (float) xUL, (float) yUL, w, h );
Error:
    return;
}

```

15.5 进一步优化

这里介绍 2 种进一步优化实现的代码：基于流处理器簇的内核函数调用（在流处理器簇 1.x 的架构中支持很多不同的乘法指令集，在这个计算的最内层循环中）以及内循环展开的内核程序。

15.5.1 基于流处理器簇的实现代码

流处理器簇 1.x 架构的硬件使用一个 24 位乘法器（对于内层的乘法计算是足够宽的），但 2.x 和 3.x 的硬件中使用的是 32 位乘法器。有时编译器可以检测到参与计算的整数的宽度，如果足够窄的话它便会使用 1.x 架构的 24 位乘法器来计算，但这似乎并不应该是内核函数 corrShared_kernel() 所需要考虑的事情。为了解决这个问题，我们可以在内核里声明一个模板类（C++ 语言特性）。

```

template<bool bSM1>
__global__ void
corrSharedSM_kernel( ... )

```

内核函数的内循环则变为：

```

for ( int j = 0; j < hTemplate; j++ ) {
    for ( int i = 0; i < wTemplate; i++ ) {
        unsigned char I = LocalBlock[SharedIdx+i];
        unsigned char T = g_Tpix[idx++];
        SumI += I;
    }
}

```

```

        if ( bSM1 ) {
            SumISq += __umul24(I, I);
            SumIT += __umul24(I, T);
        }
        else {
            SumISq += I*I;
            SumIT += I*T;
        }
    }
    SharedIdx += SharedPitch;
}

```

主机端在调用内核函数时必须检测设备是否为 1.x 架构，如果是，则调用内核时 bSM1=true。上述实例源代码出自 corrSharedSM.cuh 和 corrSharedSMSums.cuh。

15.5.2 循环展开

由于每个线程都会存取共享内存中相邻字节，1.x 架构的硬件中的内核函数的最内层循环不可避免的会产生四路存储体冲突。

如果我们将如下代码

```

for ( int j = 0; j < hTemplate; j++ ) {
    for ( int i = 0; i < wTemplate; i++) {
        unsigned char I = LocalBlock[SharedIdx+i];
        unsigned char T = g_Tpix[idx++];
        SumI += I;
        SumISq += I*I;
        SumIT += I*T;
    }
    SharedIdx += SharedPitch;
}

```

重写为

```

for ( int j = 0; j < hTemplate; j++ ) {
    for ( int i = 0; i < wTemplate/4; i++) {
        corrSharedAccumulate<bSM1>( ... LocalBlock[SharedIdx+i*4+0], );
        corrSharedAccumulate<bSM1>( ... LocalBlock[SharedIdx+i*4+1], );
        corrSharedAccumulate<bSM1>( ... LocalBlock[SharedIdx+i*4+2], );
        corrSharedAccumulate<bSM1>( ... LocalBlock[SharedIdx+i*4+3], );
    }
    SharedIdx += SharedPitch;
}

```

其中 corrSharedAccumulate() 封装了模板类 (C++ 特性) 参数 bSM1，代码如下

```

template<bool bSM1>
__device__ void
corrSharedAccumulate(
    int& SumI, int& SumISq, int& SumIT,
    unsigned char I, unsigned char T )
{
    SumI += I;
    if ( bSM1 ) {
        SumISq += __umul24(I, I);
        SumIT += __umul24(I, T);
    }
    else {
        SumISq += I*I;
        SumIT += I*T;
    }
}

```

虽然我们主要的目的是为了降低因字节存取而产生的存储体冲突，但也产生了另外一个效果，就是在流处理器簇为 1.x 的 CUDA 设备中内核函数运行的更快。

15.6 源代码

在优化归一化互相关程序时，我们发现这并不容易，并且很易出错。第 15.1 节中转换加和信息为相关系数时，我们必须要注意浮点型与整型数据精度不同（浮点型具有更大的动态范围，但只有 24 位的精度）。有一个很好的做法就是制作一些独立的子程序，这些子程序会报告计算相关系数时出错的原因（加和信息计算出错，还是计算方法出错，导致的不正确的系数）。此外，加和信息可以按位与 CPU 计算的结果作比较，而浮点值系数必须使用 ϵ 容忍值进行模糊比对。

不同的相关性实现方法被分放在不同的头文件 (.cuh) 之中，内核函数计算的加和信息和相关性系数也被分放在不同的地方。

文 件	描 述
corrShared.cuh	加载纹理至共享内存，然后从共享内存中读取图像
corrSharedSums.cuh	
corrShared4.cuh	最内层循环展开 $\times 4$ 执行 corrSharedSM
corrShared4Sums.cuh	
corrSharedSM.cuh	基于流处理器簇的内核启动方式执行 corrShared
corrSharedSMSums.cuh	
corrTexConstant.cuh	从纹理中读取图像，常量内存中读取模板
corrTexConstantSums.cuh	
corrTexTex.cuh	从纹理中读取图像和模板
corrTexTexSums.cuh	
normalizedCrossCorrelation.cu	测试程序

测试程序 normalizedCrossCorrelation.cu 不仅可以测试内核程序的功能，也可以评价其性能。默认情况下，它加载 coins.pgm 并检测到图中右下角的镍币。镍币的坐标为 (210, 148)，占用 52×52 像素。该程序还将性能测量结果写入到 stdout，例如：

```
$ normalizedCrossCorrelation --padWidth 1024 --padHeight 1024
-wTemplate 16 -hTemplate 16
corrTexTex2D: 54.86 Mpix/s 14.05Gtpix/s
corrTemplate2D: 72.87 Mpix/s 18.65Gtpix/s
corrShared: 69.66 Mpix/s 17.83Gtpix/s
corrSharedSM: 78.66 Mpix/s 20.14Gtpix/s
corrShared4: 97.02 Mpix/s 24.84Gtpix/s
```

该程序支持下列命令行选项：

- input <filename>：指定输入文件名（默认：coins.pgm）。
- output <filename>：指定输出文件名（可选）。如果指定，那么程序会将包含灰

--padWidth <width>:	度图（类似图 15-3）的 PGM 文件写入到此文件中补齐的图像宽度。
--padHeight <height>:	补齐的图像高度。
--xTemplate <value>:	指定模板左上角的 X 坐标。
--yTemplate <value>:	指定模板左上角的 Y 坐标。
--wTemplate <value>:	指定模板的宽度。
--hTemplate <value>:	指定模板的高度。

15.7 性能评价

我们的示例程序使用 CUDA 事件来评价连续启动多个内核（默认为 100 个）的性能表现。同时它能够向我们展示系数（随模板的大小变化而变化）和模板像素的输出速率，或单位时间“内循环”迭代的数量。

执行此计算的 GPU 的性能是惊人的。GeForce GTX 280 (GT200) 计算模板像素时可以达到每秒 25 Gtpix/s (250 亿次)，而 GeForce 680 GTX (GK104) 的速率可以超过 100 Gtpix/s。

程序的默认参数并不能提供最理想的性能，它检测到右下角的镍币并有选择的将其写进图 15-3 中。特别指出，该图像很小，并不能使 GPU 达到充分忙碌的状态。该图像为 300×246 像素（大小为 74KB），所以通过共享内存来实现的话仅需要 310 个线程块来执行此计算。`--padWidth` 和 `--padHeight` 命令行选项可用于在示例程序中补齐图像的大小，所以同时执行的相关系数的计算也增加了（代码中的数据没有依赖关系，所以可以填充任意数据）。对几乎所有的 GPU 测试来说， 1024×1024 大小的图像能够使 GPU 得到最充分的利用。

图 15-5 总结了我们之前的 5 种实现的性能表现：

- `corrTexTex`: 模板和图像都在纹理内存中。
- `corrTexConstant`: 模板在常量内存中。
- `corrShared`: 模板在常量内存中，图像在共享内存中。
- `corrSharedSM`: 基于流处理器簇的内核启动执行 `corrShared`。
- `corrShared4`: 内循环展开 $\times 4$ 执行 `corrSharedSM`。

图 15-6 展示了不同显卡上的不同方法所带来的性能的对比。将模板放入常量内存中对 GK104 的影响最大，最大性能提高 80%。将图像放入共享内存中对 GF100 的影响最大，性能可提高 70%。而基于流处理器簇启动内核函数的方式影响最小，能够对 GT200 的性能提高 14%（它并不会影响其他架构上的性能，因为使用内置的乘法运算也可达到最快）。

使用 GT200 时，`corrShared` 在共享内存中会受到存储片冲突的影响，以至于运行速率比 `corrTexConstant` 还慢，`corrShared4` 能够避免存储片冲突，性能提高 23%。

模板的大小也会对该算法的效率有影响。模板越大，对于每个模板像素的计算就越高

效。图 15-6 展示了模板尺寸是如何影响 corrShared4 的性能的。

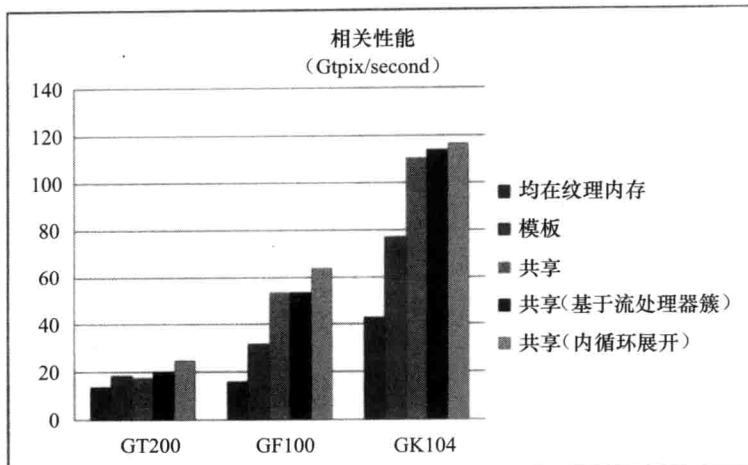


图 15-5 性能比较

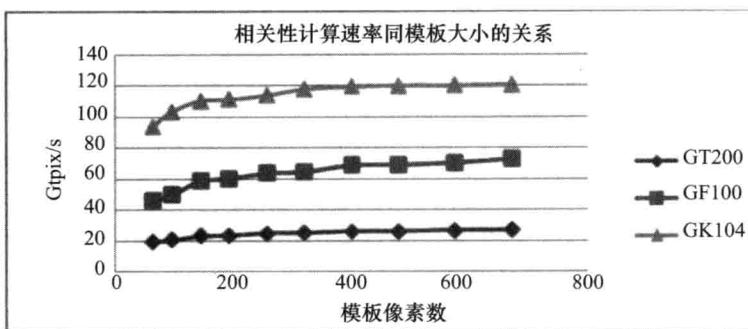


图 15-6 相关系数计算速率同模板大小的关系

模板大小从 8×8 增长至 28×28 时, GT200 的性能表现提高 36% (19.6 Gtpix/s 提高至 26.8 Gtpix/s)、GF100 提高了 57% (46.5 Gtpix/s 提高至 72.9 Gtpix/s)、GK104 提高了 30% (93.9 Gtpix/s 提高至 120.72 Gtpix/s)。

对于小模板来说, 若在编译时就已知模板的大小, 则编译器能够生成更快的代码。将 wTemplate 和 hTemplate 作为参数并专门为 8×8 的模板做改进后的性能如下所示:

显卡	速率 (GTPIX/s)		性能的提高比
	CORRSHARED4	CORRSHARED4 (改进的)	
GT200	19.63	24.37	24%
GF100	46.49	65.08	40%
GK104	93.88	97.95	4 %

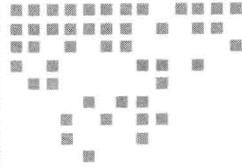
15.8 延伸阅读

《数字图像处理》一书中对归一化相关 (pp. 583 ~ 586) 和对数变换 (pp. 168 ~ 169) 有相关讨论，其中对数变换部分在我们的示例程序中被用来计算输出像素：

Gonzalez, Rafael C., and Richard E. Woods. Digital image processing. Addison-Wesley, Reading, MA, 1992. www.imageprocessingplace.com/root_files_V3/publications.htm

我们的示例程序需要模板去匹配输入图像的每一个像素，而 J. P. Lewis 曾经讨论过一个更加渐近有效的方式来加速这种程序中各种相关操作类型的实现。他通过快速傅里叶变换计算分子，并用区域求和表来计算系数的分母：

Lewis, J. P. Fast template matching. Vision Interface 10, 1995, pp. 120 ~ 123。其扩充版名为“Fast Normalized Correlation”，可以在 <http://bit.ly/NJnZPI> 找到。



CUDA 专家手册库

正如第 1 章所述，本书附带的源代码遵循（简化的）2 句版 BSD 开源许可证。源代码可在 www.cudahandbook.com 下载，开发人员可以在 <https://github.com/ArchaeaSoftware/cudahandbook> 找到对应的 Git 存储库。

本附录简要介绍 CUDA 专家手册库（chLib）。该库包含一套位于源代码项目的 chLib/ 子目录的可移植头文件。chLib 不可重用于软件产品。它提供了最小功能，使用尽可能少量的源代码来说明本书涵盖的概念。chLib 可以移植到任何目标操作系统的 CUDA 环境，所以它往往必须支持这些操作系统的共有特性。

A.1 计时操作

CUDA 专家手册库包括一个使用 QueryPerformanceCounter()（在 Windows 系统）和 gettimeofday()（非 Windows 系统）的可移植计时库。一个示例用法如下：

```
float
TimeNULLKernelLaunches(int cIterations = 1000000 )
{
    chTimerTimestamp start, stop;
    chTimerGetTime( &start );
    for ( int i = 0; i < cIterations; i++ ) {
        NullKernel<<<1,1>>>();
    }
    cudaThreadSynchronize();
    chTimerGetTime( &stop );
    return 1e6*chTimerElapsed Time( &start, &stop ) /
        (float) cIterations;
}
```

这个函数测定启动指定次数的内核所需的时间，并返回每次启动消耗的微秒值。chTimerTimestamp 是一个高分辨率的时间戳。通常它是一个 64 位的计数器，它随时间单调递增，所以需要两个时间戳来计算时间间隔。

chTimerGetTime() 函数取得当前时间的快照。chTimerElapsedTime() 函数返回两个时间戳间隔的秒数。这些计时器的分辨率是非常精细的（也许是微秒），所以 chTimerElapsedTime() 返回双精度浮点值。

```
#ifdef _WIN32
#include <windows.h>
typedef LARGE_INTEGER chTimerTimestamp;
#else
typedef struct timeval chTimerTimestamp;
#endif

void chTimerGetTime(chTimerTimestamp *p);
double chTimerElapsedTime( chTimerTimestamp *pStart, chTimerTimestamp
*pEnd );
double chTimerBandwidth( chTimerTimestamp *pStart, chTimerTimestamp
*pEnd, double cBytes );
```

在支持 CUDA 的 GPU 上对性能进行隔离测量时，我们可以使用 CUDA 事件，例如，测量一个内核的设备内存带宽。使用 CUDA 事件进行计时是一把双刃剑：它们较少受系统级事件影响，如网络流量，但有时会导致过于乐观的计时结果。

A.2 线程操作

chLib 包括最低限度的线程操作库，支持创建“工作”CPU 线程，并拥有允许一个父线程把工作委派给工作线程的工具。线程操作是一个特别难以进行抽象的功能，因为不同的操作系统需要启用不同的工具。有些操作系统甚至有“线程池”，使线程容易被回收，因此应用程序不必让线程挂起以等待一个同步事件（当一些工作出现，即会发出信号）。

代码清单 A-1 给出了来自 chLib/chThread.h 的抽象线程操作。它包括一个 processorCount() 函数和一个 C++ 类 WorkerThread。前者返回可用的 CPU 核的数量（很多应用程序使用多线程以充分利用多个 CPU 核，例如第 14 章的多线程 N- 体实现，意在让每个核上运行一个线程），后者支持一些简单的线程操作。

创建和销毁

- delegateSynchronous()：父线程指定一个工作线程要执行的函数指针，而且该函数直到工作线程完成才返回。
- delegateAsynchronous()：父线程指定一个工作线程要异步执行的函数指针；workerThread::waitForAll 必须被调用以便该父线程与它的子线程同步。
- 成员函数 waitAll() 进入等待，直到所有指定的工作线程完成其被委派的工作。

代码清单 A-1 workerThread 类

```

// Return the number of execution cores on the platform.
// unsigned int processorCount();

// workerThread class - includes a thread ID (specified to constructor)
class workerThread
{
public:
    workerThread( int cpuThreadId = 0 );
    virtual ~workerThread();
    bool initialize( );

    // thread routine (platform specific)
    static void threadRoutine( LPVOID );

    //
    // call this from your app thread to delegate to the worker.
    // it will not return until your pointer-to-function has been
    // called with the given parameter.
    //
    bool delegateSynchronous( void (*pfn)(void *), void *parameter );

    //
    // call this from your app thread to delegate to the worker
    // asynchronously. Since it returns immediately, you must call
    // waitAll later
    //

    bool delegateAsynchronous( void (*pfn)(void *), void *parameter );
    static bool waitAll( workerThread *p, size_t N );
};


```

A.3 驱动程序 API 工具

chDrv.h 包含为驱动程序 API 开发者提供的一些有用工具：chCUDADevice 类，如代码清单 A-2 所示，它简化了设备和上下文的管理。其 loadModuleFromFile 方法简化了从一个 .cubin 或 .ptx 文件创建一个模块。

此外，chGetErrorString() 函数传回一个对应于错误值的只读字符串。不仅针对驱动程序 API 的 CUresult 类型实现了这一函数，chGetErrorString() 的一个特化也包装了 CUDA 运行时的 cudaGetErrorString() 函数。

代码清单 A-2 chCUDADevice 类

```

class chCUDADevice
{
public:
    chCUDADevice();
    virtual ~chCUDADevice();

    CUresult Initialize(

```

```

int ordinal,
list<string>& moduleList,
unsigned int Flags = 0,
unsigned int numOptions = 0,
CUjit_option *options = NULL,
void **optionValues = NULL );
CUresult loadModuleFromFile(
CUmodule *pModule,
string fileName,
unsigned int numOptions = 0,
CUjit_option *options = NULL,
void **optionValues = NULL );
CUdevice device() const { return m_device; }
CUcontext context() const { return m_context; }
CUmodule module( string s ) const { return (*m_modules.find(s)).second; }

private:
CUdevice m_device;
CUcontext m_context;
map<string, CUmodule> m_modules;
};

```

A.4 Shmoo 工具

“shmoo 图”是测试电路图性能的可视化曲线，它随两个输入的变化而变化（例如，电压和时钟频率）。在编写代码以确定不同内核的最佳线程块配置参数时，它有类似的用处：改变如线程块大小和循环展开因子等输入，观察输出性能。代码清单 A-3 显示了 chShmooRange 类和 chShmooIterator 类。前者封装了参数范围，后者使 for 循环可以方便地在给定范围内迭代。

代码清单 A-3 chShmooRange 类和 chShmooIterator 类

```

class chShmooRange {
public:
    chShmooRange( ) { }
    void Initialize( int value );
    bool Initialize( int min, int max, int step );
    bool isstatic() const { return m_min==m_max; }

    friend class chShmooIterator;

    int min() const { return m_min; }
    int max() const { return m_max; }

private:
    bool m_initialized;
    int m_min, m_max, m_step;
};

class chShmooIterator
{
public:
    chShmooIterator( const chShmooRange& range );

```

```

int operator *() const { return m_i; }
operator bool() const { return m_i <= m_max; }
void operator++(int) { m_i += m_step; }
private:
    int m_i;
    int m_max;
    int m_step;
};

```

命令行分析器还包括一个特化，用来基于命令行参数创建 chShmooRange：在关键词之前追加“min”、“max”和“step”，而返回的结果为对应的范围。如果这三项均未提供，函数返回 false。例如，concurrencyKernelKerne 样例（在本书附带源代码的 concurrency/ 子目录下）对流的数量和时钟周期数这两个参数在一定范围内进行测量。在命令行提取这些值的代码如下所示：

```

chShmooRange streamsRange;
const int numStreams = 8;
if ( ! chCommandLineGet(&streamsRange, "Streams", argc, argv) ) {
    streamsRange.Initialize( numStreams );
}
chShmooRange cyclesRange;
{
    const int minCycles = 8;
    const int maxCycles = 512;
    const int stepCycles = 8;
    cyclesRange.Initialize( minCycles, maxCycles, stepCycles );
    chCommandLineGet( &cyclesRange, "Cycles", argc, argv );
}

```

并且用户可以按照如下方式为应用程序指定参数。

```
concurrencyKernelKernel -- minStreams 2 --maxStreams 16 stepStreams 2
```

A.5 命令行分析工具

chCommandLine.h 给出了一个可移植的命令行分析库（只有 100 行的 C++ 代码）。它包括模板函数 chCommandLineGet() 和 chCommandLineGetBool()。前者传回给定类型的变量，而后者返回命令行中是否包括给定关键字的判断。

```
template<typename T> T
chCommandLineGet( T *p, const char *keyword, int argc, char *argv[] );
```

正如上一节所述，chCommandLineGet() 的一个特化将传回 chShmooRange 的一个实例。为了保证这一特化被编译，chShmoo.h 必须在 chCommandLine.h 之前被包含。

A.6 错误处理

chError.h 实现了一组基于 goto 语句的错误处理宏，这一错误处理机制已在本书 1.2.3 小节提到过。这些宏的执行步骤如下：

- 把返回值赋给一个名为 status 的变量；
- 检查 status 是否成功，如果在调试模式下，错误报告输出到 stderr；
- 如果 status 包含一个错误，则 goto 到名为 Error 的语句标签。

CUDA 运行时版本的错误处理如下：

```
#ifdef DEBUG
#define CUDART_CHECK( fn ) do { \
    (status) = (fn); \
    if ( cudaSuccess != (status) ) { \
        fprintf( stderr, "CUDA Runtime Failure (line %d of file %s):\n\t" \
        "%s returned 0x%x (%s)\n", \
        __LINE__, __FILE__, #fn, status, cudaGetErrorString(status) ); \
        goto Error; \
    } \
} while (0);
#else

#define CUDART_CHECK( fn ) do { \
    status = (fn); \
    if ( cudaSuccess != (status) ) { \
        goto Error; \
    } \
} while (0);
#endif
```

do…while 语句是一个 C 语言编程里的惯用组合，经常用在宏中。该语句让每次宏调用执行一条语句。如果变量 status 和标签 Error：中的任何一个没有定义，这些宏将产生编译错误。

使用 goto 语句的一个隐含条件是所有的变量必须在代码块的顶部进行声明。否则，一些编译器会产生错误，因为 goto 语句可以绕过初始化。出现这种情况时，待初始化的变量必须移到第一个 goto 语句之前，或者移到一个基本代码块以使 goto 语句超出其范围。

代码清单 A-4 给出了一个遵循这一惯例的示例函数。返回值和中间资源被初始化为能够由清理代码进行处理的值。在这种情况下，所有由该函数分配的资源也由该函数释放，所以清理代码和错误处理代码是相同的。那些只会释放它们分配的一些资源的函数，必须在不同的代码块中实现成功和失败两种情形。

代码清单 A-4 goto 风格的错误处理示例

```
double
TimedReduction(
    int *answer, const int *deviceIn, size_t N,
    int cBlocks, int cThreads,
    pfnReduction hostReduction
)
{
    double ret = 0.0;
    int *deviceAnswer = 0;
    int *partialSums = 0;
    cudaEvent_t start = 0;
    cudaEvent_t stop = 0;
    cudaError_t status;

    CUDART_CHECK( cudaMalloc( &deviceAnswer, sizeof(int) ) );
    CUDART_CHECK( cudaMalloc( &partialSums, cBlocks*sizeof(int) ) );
    hostReduction( deviceIn, deviceAnswer, partialSums, N, cBlocks, cThreads );
    cudaEventRecord( start );
    hostReduction( partialSums, deviceAnswer, partialSums, N, cBlocks, cThreads );
    cudaEventRecord( stop );
    cudaEventSynchronize( stop );
    cudaEventElapsedTime( &ret, start, stop );
    hostReduction( partialSums, deviceAnswer, partialSums, N, cBlocks, cThreads );
    hostReduction( deviceAnswer, answer, partialSums, N, cBlocks, cThreads );
    hostReduction( partialSums, 0, partialSums, N, cBlocks, cThreads );
    cudaFree( deviceAnswer );
    cudaFree( partialSums );
}
```

```
CUDART_CHECK( cudaEventCreate( &start ) );
CUDART_CHECK( cudaEventCreate( &stop ) );
CUDART_CHECK( cudaThreadSynchronize() );

CUDART_CHECK( cudaEventRecord( start, 0 ) );
hostReduction(
    deviceAnswer,
    partialSums,
    deviceIn,
    N,
    cBlocks,
    cThreads );
CUDART_CHECK( cudaEventRecord( stop, 0 ) );
CUDART_CHECK( cudaMemcpy(
    answer,
    deviceAnswer,
    sizeof(int),
    cudaMemcpyDeviceToHost ) );

ret = chEventBandwidth( start, stop, N*sizeof(int) ) /
    powf(2.0f,30.0f);
// fall through to free resources before returning
Error:
cudaFree( deviceAnswer );
cudaFree( partialSums );
cudaEventDestroy( start );
cudaEventDestroy( stop );
return ret;
}
```

术语表

赋别名 (aliasing) 针对同一个内存指针，建立多个访问方式。例如，CUDA 中一个映射锁页缓冲会在主机指针和设备指针之间互为别名；一个纹理引用绑定到设备内存，成为编程设备内存的别名。

AOS array of structures 的缩写，参见结构体数组。

应用程序编程接口 (API) application programming interface 的缩写。

结构体数组 (array of structures) 一种内存结构，描述对象的元素在内存上是连续存放的（就好像在一个结构体上声明）。与数组结构相对。

异步的 (asynchronous) 函数调用在所请求的操作执行完毕之前返回。为保证得到正确的结果，基于异步操作的 CUDA 应用程序必须使用 CUDA 流或事件进行 CPU/GPU 同步。

计算密度 (computational density) 计算开销相对于外部内存传输开销的比例。

常量内存 (constant memory) 只读内存，当执行同一内存位置的读操作时，为广播机制作了优化。

中央处理器 (CPU) central processing unit 的缩写。当代计算机（无论是 x86、x86-64，还是 ARM）的大脑。

CUDA 数组 (CUDA array) 结构细节对开发者

未公开的一维、二维或三维数组。应用程序可用内存复制函数读写 CUDA 数组。CUDA 内核可通过纹理读取 CUDA 数组，或者利用表面加载 / 存储内置函数对它进行读 / 写。

CUDA 运行时 (CUDART) CUDA runtime 的缩写。带有语言集成特性的高级 API。

设备驱动程序接口 (DDI) device driver interface 的缩写。设备驱动程序接口的例子包括 XPDDM 和 WDDM。

请求式换页 (demand paging) 操作系统可把页面标记为“非驻留”，当应用程序试图访问一个非驻留页面时，硬件发出中断信号。操作系统可以使用该机制，依据一些启发式规则，对“一段时间”没有被访问的页面标记为非驻留，使这些页面的内容换回到磁盘以释放更多的物理内存供更活跃的虚拟页面使用。[⊖]如果应用程序再次访问某换出的页面，该页面会根据“请求”重新加载到内存（可能在与原来不同的物理页面上）。迄今为止，GPU 实现了一个比较强大的虚拟内存系统，可以让虚拟地址和物理地址分离，但未实现硬件上的请求式换页。

设备内存 (device memory) 适宜 GPU 访问的内存。CUDA 数组、全局内存、常量内存和本地内存都是设备内存的不同形式。

[⊖] 请求式换页硬件可以用来实现很多其他功能，例如，写时复制，映射文件 z/o 等。可以查阅操作系统方面的教科书获取更详细的内容。

直接内存访问 (DMA) direct memory access 的缩写。外设异步于、独立于 CPU 进行 CPU 内存的读或写。

驱动程序 (driver) 使用操作系统的工具对外设的硬件功能予以暴露的软件。

驱动程序 API (driver API) 低级 API，允许对 CUDA 工具进行全面访问。

动态指令总数 (dynamic instruction count) 一个程序实际执行的机器指令数目。与静态指令总数相对。

错误纠正码 (ECC) error correction code 的缩写。一些 CUDA 硬件保护 GPU 外部内存接口的方式，它预留 12.5% 的设备内存（可用内存每 8 位外配 1 位纠错码），并用它来检测和（有时）纠正内存事务的错误。可用 nvidia-smi 或英伟达管理库查询是发生了可纠正（1 位）的错误，还是无法纠正（2 位）的错误。

前端总线 (FSB) 在非 NUMA 系统配置下，芯片组与内存的接口。

全局内存 (global memory) CUDA 内核使用指针进行读写的设备内存。

图形处理器 (GPU) graphics processing unit 的缩写。

GPU 时间 (GPU time) 通过 CUDA 事件测定的时间，与系统计数器时间不同。这一时间可以指导优化，但它无法提供系统整体性能的准确图景。与系统时钟时间相对。

高性能计算 (HPC) high performance computing 的缩写。

ILP instruction level parallelism 的缩写。参见指令级并行。

指令级并行 (instruction level parallelism) 程序执行过程中，不同操作之间的细粒度并行。

内置函数 (intrinsic function) 直接对应于一个低级机器指令的函数。

即时编译 (JIT) just-in-time compilation 场合下的缩写。另请参阅在线编译。

内核模式 (kernel mode) 可以执行如编辑页表等敏感操作的特权执行模式。

内核转换 (kernel thunk) 从用户模式到内核模

式的转换。该操作需要几千个时钟周期，所以操作系统上运行的驱动程序在需要内核转换操作以提交命令给硬件时，必须在执行内核转换之前在用户模式下排队等候硬件指令。

束内线程 (lane) 一个线程束内的线程。束内线程编号可以使用 threadIdx.x & 31 计算。

内存管理单元 (MMU) memory management unit 的缩写。负责把虚拟地址转换到物理地址，并在所指定地址无效时发出错误信号的 CPU 或 GPU 硬件。

节点 (node) NUMA 系统上产生内存带宽的一个单元。在廉价的 NUMA 系统中，节点通常对应于物理 CPU。

非一致内存访问 (NUMA) nonuniform memory access 的缩写。指的是如 AMD 霍龙或者英特尔 Nehalem 处理器的内存架构，其中的内存控制器集成到 CPU 以得到更低的延迟和更高的性能。

占用率 (occupancy) 一个 SM 上执行的线程束数量与理论最大值的比值。

在线编译 (online compilation) 在运行之际（而不是在开发人员生成应用程序时）进行编译。

事前允诺 (opt-in) 一个开发者必须在接口级别请求行为改变的 API 机制。例如，创建一个阻塞事件是一个“事前允诺”，因为开发者必须为创建事件 API 传入一个特殊的标志。由于现有应用程序依赖于旧行为，事前允诺是一种公开新功能而无须冒让步风险的途径。

事后拒绝 (opt-out) 一个禁止合法行为的 API 机制，例如，创建一个禁用计时的事件。

可换页内存 (pageable memory) 可以被 VMM 换出的内存。操作系统的设计师更喜欢可换页内存，因为它能使操作系统把页面“换出”到磁盘并把可用物理内存用于其他目的。

页面故障 (page fault) 当应用程序访问到被操作系统标记为非驻留的虚拟内存时发生的执行故障。如果访问有效，操作系统更新其数据结构（可能是把页面读入物理内存并更新指向该内存的物理地址）并恢复执行。如果访问无效，操作系统发送一个异常信号给应用程序。

页面锁定内存 (page-locked memory) 已经由操作系统在物理上予以分配并被标记为非分页的内存。通常这是为了支持硬件通过 DMA 访问内存。

PCIe PCI Express 总线, CUDA 用来在主机和设备内存之间交换数据。

锁页内存 (pinned memory) 参见页面锁定内存 (page-locked memory)。

等步长内存分配 (pitched memory allocation) 一种内存分配方式, 其中每行字节数需要另外指定, 不是由每行元素数乘以每个元素大小得到。用于满足对齐约束, 数组的每一行都必须对齐在同一位置。

等步长线性结构 (pitch-linear layout) 用于等步长内存分配的内存结构, 由一个“基地址和每行字节数”(即“步长”)构成的“元组”指定。

断定 (predicate) 布尔型的 1 个位, 值为“真”或“假”。在 C 语言中, 整数可以通过评估它是否非零 (真) 或零 (假) 转换为一个断定。

进程 (process) 多任务操作系统的执行单位, 它拥有自己的地址空间和资源生命周期的管理权 (如文件句柄)。当进程退出时, 与它相关联的所有资源均由操作系统“清理”。

页表项 (PTE) page table entry 的缩写。

并行线程执行 (PTX) parallel thread eXecution 的缩写, 一种中间汇编语言和字节码, 它会被作为驱动程序 JIT 执行的输入, 编译成针对目标 GPU 的二进制代码。

SASS 支持 CUDA 的 GPU 的汇编语言级原生指令集。该缩写的确切全称已不可考, 但着色器汇编 (shader ASSEMBLY) 语言似乎是一个合理的猜测!

SBIOS 系统 BIOS (基本输入 / 输出系统)。控制计算机系统最基本的 I/O 子系统 (例如, 是否启用可能不被某些操作系统支持的 CPU 或芯片组功能) 的固件。SBIOS 是较操作系统更低层级的。

共享内存 (shared memory) CUDA 内核可访问的快速板载 GPU 内存, 用于保存临时结果。

单指令多数据 (SIMD) single instruction multiple

data 的缩写, 一个并行编程原语, 涉及针对不同数据并行执行一个统一操作。在 CUDA 硬件中, 流处理器簇以 SIMD 方式运行 32 个线程。x86 硬件上的 SSE 指令也以 SIMD 方式处理通过宽寄存器操作包装的数据。

流处理器簇 (SM) streaming multiprocessor 的缩写, GPU 的核心执行单元之一。一个 GPU 包含的 SM 数目范围可以从 2 到几十个。此外, 一个 GPU 的指令集可以被指定一个版本号, 例如, SM 2.0。

扩展的流处理器簇 (SMX) SM 在开普勒架构 (SM 3.x) 硬件上的扩展实现。

单指令多数据流扩展 (SSE) 在 20 世纪 90 年代末期添加到 x86 的指令集扩展, 它可以使用单条指令执行 4 个单精度浮点运算。后来增加的特性包括支持整型 SIMD 运算, 并把操作的字宽从 128 位拓展到 256 位。

静态指令总数 (static instruction count) 程序的机器指令数目; 程序占用空间与静态指令总数成正比。与动态指令总数相对。

数组结构 (SOA) 使用一个数组来描述对象的每个元素的内存结构。与结构体数组 (AOS) 相对。

同步的 (synchronous) 该形容词用来描述那些直到所请求的操作完成才返回的函数。

特斯拉计算集群驱动程序 (TCC) Tesla compute cluster driver 的缩写。它是可以在 Windows Vista 和更高版本上运行的 XPDDM 类驱动程序。它没有 WDDM 的优势 (Windows 桌面管理器加速、图形互操作性、模拟分页), 但可以无须执行内核转换就可提交命令到硬件, 并实现 64 位统一地址空间。

Thrust 为提高 CUDA 生产效率开发的 C++ 库, 参考了 STL 的部分特性。

三字母缩写 (TLA) 为 three-letter acronym 的缩略语。

线程本地存储 (TLS) 为 thread local storage 的缩略语。

最小精度单位 (ulp) unit of last precision 的缩写, 意指浮点数尾数的最低有效数字。

用户模式（user mode） 非特权执行模式，在此模式下，内存一般是分页的，而硬件资源只能通过 API 与操作系统的内核模式软件进行交互。

统一虚拟寻址（UVA） 为 unified virtual addressing 的缩略语。（参见 2.4.5 小节）

虚拟内存管理器（VMM） 为 virtual memory manager 的缩略语。操作系统中用于管理内存的部分：分配、锁页、管理缺页，等等。

系统时钟时间（wall clock time） 依据系统时钟测定一组操作执行前和执行后的时间。系统时钟时间包括了系统的所有影响并提供整体性能的最精确测量。与 GPU 时间相对。

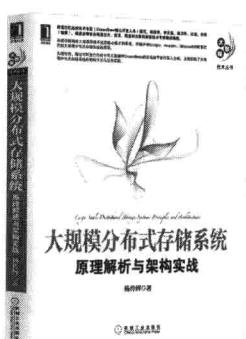
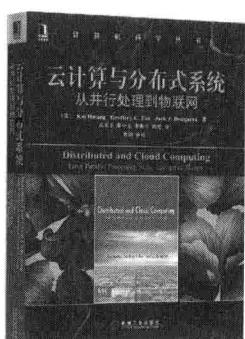
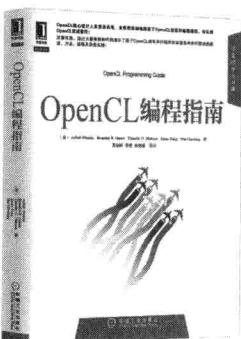
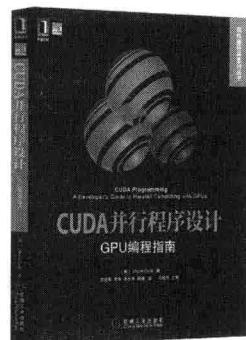
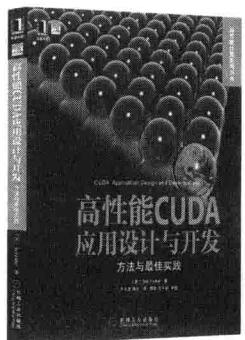
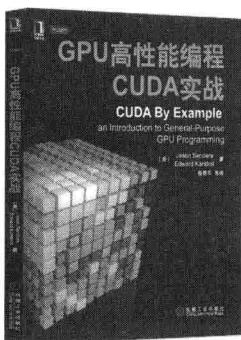
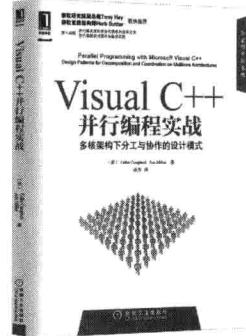
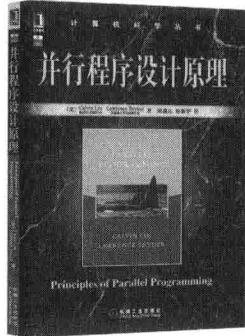
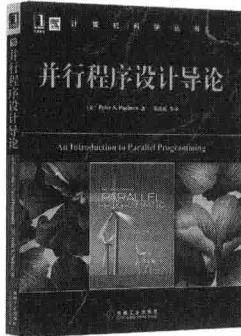
线程束（warp） 流处理器簇的基本执行单位。

对于前三代的 CUDA 硬件，线程束正好有 32 个线程，所以在一维线程块的线程束 ID 可以使用 `threadIdx.x>>5` 计算。另请参阅束内线程编号（lane）。

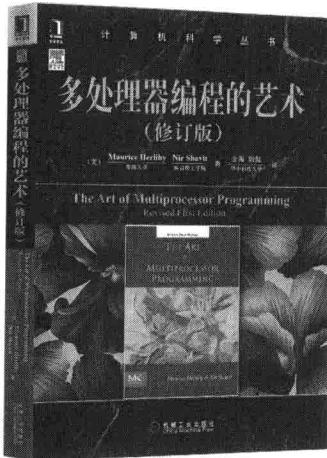
Windows 显示驱动程序模型（WDDM） 为 Windows display driver model 的缩略语。此驱动程序模型，是新加入 Windows Vista 的，它把多数显示驱动程序逻辑从内核模式移入用户模式。

Windows XP 显示驱动程序模型（XPDDM） 为 Windows XP display driver model 的缩略语。从架构上来说，这个驱动程序模型可以追溯到 Windows NT 4.0（即 1996 年）。这个缩写是为了与“WDDM”对照而一起发明的。

推荐阅读

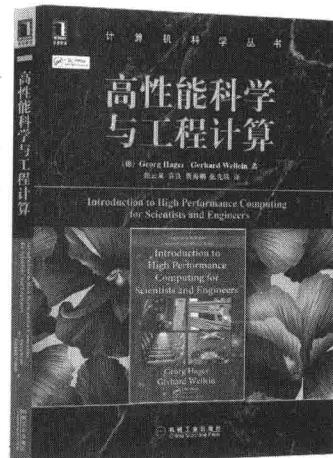


推荐阅读



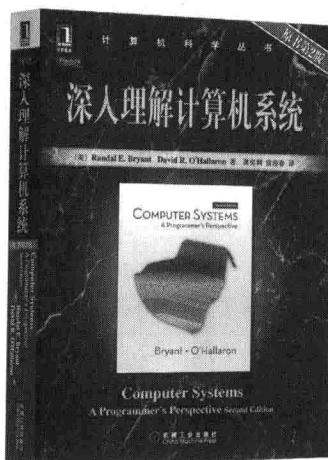
多处理器编程的艺术（修订版）

作者: Maurice Herlihy 等 ISBN: 978-7-111-41858-0 定价: 69.00元



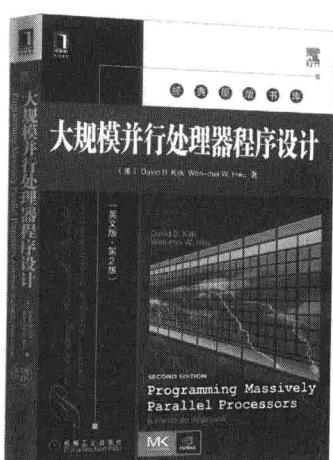
高性能科学与工程计算

作者: Georg Hager 等 ISBN: 978-7-111-46652-9 定价: 69.00元



深入理解计算机系统（原书第2版）

作者: Randal E. Bryant 等 ISBN: 978-7-111-32133-0 定价: 99.00元



大规模并行处理器程序设计（英文版·第2版）

作者: David B. Kirk 等 ISBN: 978-7-111-41629-6 定价: 79.00元