

作者结合其多年CUDA教学经验以及与Nvidia公司多年合作积累的工程经验精心撰写  
紧扣深度学习热点，包含cuDNN技术  
通俗易懂、循序渐进，是学习CUDA编程的最佳选择

 CRC Press  
Taylor & Francis Group



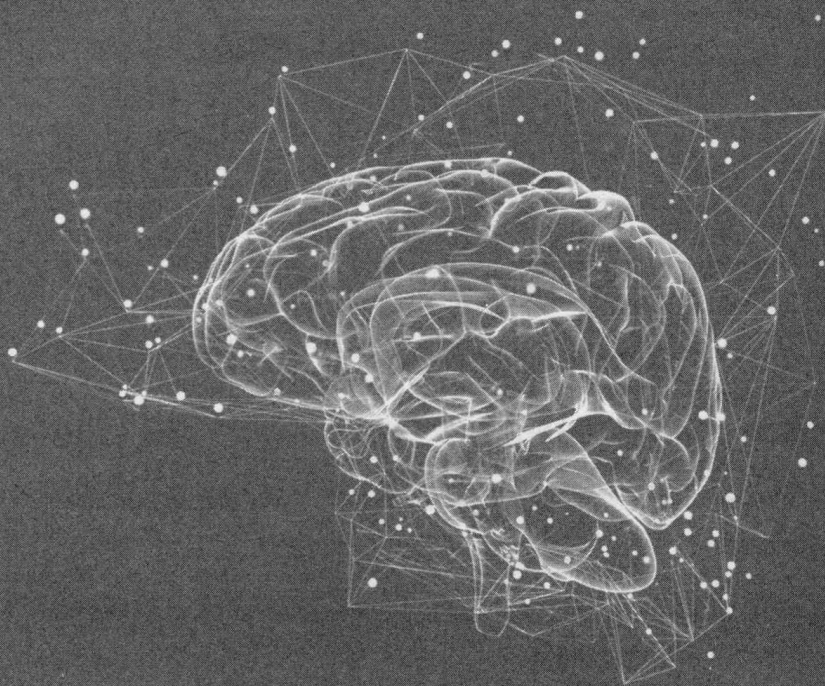
GPU Parallel Program Development Using CUDA

# 基于CUDA的 GPU并行程序开发指南

[美] 托尔加·索亚塔(Tolga Soyata) 著  
唐杰 译



机械工业出版社  
China Machine Press



GPU Parallel Program Development Using CUDA

# 基于CUDA的 GPU并行程序开发指南



【美】托尔加·索亚塔(Tolga Soyata) 著  
唐杰 译



机械工业出版社  
China Machine Press

## 图书在版编目 (CIP) 数据

基于 CUDA 的 GPU 并行程序开发指南 / (美) 托尔加·索亚塔 (Tolga Soyata) 著; 唐杰译. —北京: 机械工业出版社, 2019.6

(高性能计算技术丛书)

书名原文: GPU Parallel Program Development Using CUDA

ISBN 978-7-111-63061-6

I. 基… II. ①托… ②唐… III. 图像处理 - 程序设计 - 指南 IV. TP391.413-62

中国版本图书馆 CIP 数据核字 (2019) 第 126626 号

本书版权登记号: 图字 01-2018-2746

GPU Parallel Program Development Using CUDA by Tolga Soyata (978-1-4987-5075-2).

© 2018 by Taylor & Francis Group, LLC.

Authorized translation from the English language edition published by CRC Press, part of Taylor & Francis Group LLC. All rights reserved.

China Machine Press is authorized to publish and distribute exclusively the Chinese (Simplified Characters) language edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

Copies of this book sold without a Taylor & Francis sticker on the cover are unauthorized and illegal.

本书原版由 Taylor & Francis 出版集团旗下 CRC 出版公司出版, 并授权翻译出版。版权所有, 侵权必究。

本书中文简体字翻译版授权由机械工业出版社独家出版并仅限在中华人民共和国境内 (不包括香港、澳门特别行政区及台湾地区) 销售。未经出版者书面许可, 不得以任何方式复制或抄袭本书的任何内容。

本书封面贴有 Taylor & Francis 公司防伪标签, 无标签者不得销售。

## 基于 CUDA 的 GPU 并行程序开发指南

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: 赵 静

责任校对: 殷 虹

印 刷: 中国电影出版社印刷厂

版 次: 2019 年 7 月第 1 版第 1 次印刷

开 本: 186mm × 240mm 1/16

印 张: 27.75

书 号: ISBN 978-7-111-63061-6

定 价: 179.00 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) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294

读者信箱: hzit@hzbook.com

版权所有·侵权必究

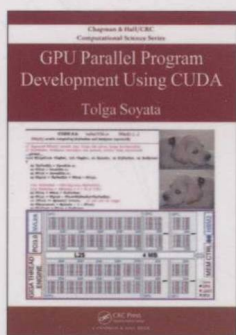
封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

## 内容简介

本书旨在帮助读者了解与基于CUDA的GPU并行编程技术有关的基本概念，并掌握使用C语言进行GPU高性能编程的相关技巧。本书共分为三部分，第一部分通过CPU多线程编程解释了并行计算，使得没有太多并行计算基础的读者也能毫无阻碍地入门CUDA；第二部分重点介绍了基于CUDA的GPU大规模并程序的开发与实现，并通过大量的性能分析帮助读者理解如何开发一个好的GPU并程序以及GPU架构对程序性能的影响；第三部分介绍了一些常用的CUDA库、OpenCL编程语言、其他GPU编程语言和API以及深度学习库cuDNN。

本书内容翔实、实例丰富，可作为高等院校相关专业高年级本科生和研究生课程的教材，也可作为计算机相关技术人员的参考书。



原书封面

## 译者简介

---



**唐杰博士**

南京大学计算机科学与技术系副教授，主要研究高性能计算与并行处理技术，主持和参与国家科技重大专项、国家自然科学基金等十余项课题，在国内外学术会议和期刊上发表了50多篇论文，还参与编写了多部教材。

## The Translator's Words 译者序

近 10 年来，随着大数据、深度学习等相关领域的发展，对计算能力的需求呈几何级数增长。与此同时，大规模集成电路的发展却受到功耗、散热、晶体管尺寸等客观因素的限制，难以继续维持摩尔定律。因此，人们逐渐把目光转向了并行系统。GPU 自诞生之日起就是为计算机的图形图像渲染等大规模并行处理任务而服务的，因而越来越受到研究界和企业界的关注。随着 CUDA 等计算架构模型的出现，这一趋势更加明显。

CUDA (Compute Unified Device Architecture, 统一计算设备架构) 是 Nvidia (英伟达) 提出的并行计算架构，它可以结合 CPU 和 GPU 的优点，处理大规模的计算密集型任务。同时，它采用了基于 C 语言风格的语法，又将 CPU 端和 GPU 端的开发有效地集成到了同一环境中，对于大多数 C 程序员来说，使用十分方便，因而一经推出就迅速占领了 GPU 开发环境的市场。

然而，会写 CUDA 程序与会写好的 CUDA 程序相差甚远！

阻碍 CUDA 程序获得高性能的原因有很多。首先，GPU 属于单指令多数据类型的并行计算，因而任务切分方式非常关键，既要充分挖掘线程级的并行性，也要充分利用流来实现任务级的并行。其次，GPU 的存储类型和访问模式比 CPU 的要丰富得多，一个成功的 CUDA 程序要能充分利用不同类型的存储。再次，Nvidia GPU 的架构还处于高速发展期，新一代 GPU 所推出的新功能也能够有效地提升计算效率。最后，万丈高楼平地起并不是 CUDA 开发的最佳方式，Nvidia 和一些第三方机构都开发了很多基于 CUDA 的支撑库，利用好这些第三方库可以让你的开发过程事半功倍。

Tolga Soyata 结合他 10 多年的 CUDA 教学经验以及与 Nvidia 多年合作的经历精心撰写了本书，针对上述问题进行了详细而生动的阐述。本书最独特的地方是它在第一部分中通过 CPU 多线程解释并行计算，使没有太多并行计算基础的读者也能毫无阻碍地进入 CUDA 天

地。第二部分重点介绍了基于 CUDA 的 GPU 大规模并行程序的开发与实现。与现有的同类书籍相比，本书的特点是在多个 Nvidia GPU 平台（Fermi、Kepler、Maxwell 和 Pascal）上并行化，并进行性能分析，帮助读者理解 GPU 架构对程序性能的影响。第三部分介绍了一些重要的 CUDA 库，比如 cuBLAS、cuFFT、NPP 和 Thrust（第 12 章）；OpenCL 编程语言（第 13 章）；使用其他编程语言和 API 库进行 GPU 编程，包括 Python、Metal、Swift、OpenGL、OpenGL ES、OpenCV 和微软 HLSL（第 14 章）；当下流行的深度学习库 cuDNN（第 15 章）。

本书通过生动的类比、大量的代码和详细的解释向读者循序渐进地介绍了基于 CUDA 编程开发的 GPU 并行计算方法，内容丰富翔实，适合所有具备基本的 C 语言知识的程序员阅读，也适合作为 GPU 并行计算相关课程的教材。

在本书的翻译过程中得到了机械工业出版社华章公司朱捷先生的大力支持，在此表示由衷的感谢！

限于水平，翻译中难免有错误或不妥之处，真诚希望各位读者批评指正。

唐 杰

2019 年 1 月

## Preface 前言

我经历过在 IBM 大型机上编写汇编语言来开发高性能程序的日子。用穿孔卡片编写程序，编译需要一天时间；你要留下在穿孔卡片上编写的程序，第二天再来拿结果。如果出现错误，你需要重复这些操作。在那些日子里，一位优秀的程序员必须理解底层的机器硬件才能编写出好的代码。当我看到现在的计算机科学专业的学生只学习抽象层次较高的内容以及像 Ruby 这样的语言时，我总会感到有些焦虑。尽管抽象是一件好事，因为它可以避免由于不必要的细节而使程序开发陷入困境，但当你尝试开发高性能代码时，抽象就变成了一件坏事。

自第一个 CPU 出现以来，计算机架构师在 CPU 硬件中添加了令人难以置信的功能来“容忍”糟糕的编程技巧。20 年前，你必须手动设置机器指令的执行顺序，而如今在硬件中 CPU 会为你做这些（例如，乱序执行）。在 GPU 世界中也能清晰地看到类似的趋势。由于 GPU 架构师正在改进硬件功能，5 年前我们在 GPU 编程中学习的大多数性能提升技术（例如，线程发散、共享存储体冲突以及减少原子操作的使用）正变得与改进的 GPU 架构越来越不相关，甚至 5 ~ 10 年后，即使是一名非常马虎的程序员，这些因素也会变得无关紧要。当然，这只是一个猜测。GPU 架构师可以做的事取决于晶体管总数及客户需求。当说晶体管总数时，是指 GPU 制造商可以将多少个晶体管封装到集成电路（IC）即“芯片”中。当说客户需求时，是指即使 GPU 架构师能够实现某个功能，但如果客户使用的应用程序不能从中受益，就意味着浪费了部分的晶体管数量。

从编写教科书的角度出发，我考虑了所有的因素，逐渐明确讲授 GPU 编程的最佳方式是说明不同系列 GPU（如 Fermi、Kepler、Maxwell 和 Pascal）之间的不同并指明发展趋势，这可以让读者准备好迎接即将到来的下一代 GPU，再下一代，……我会重点强调那些相对来说会长期存在的概念，同时也关注那些与平台相关的概念。也就是说，GPU 编程完全关乎性能，



如果你了解程序运行的平台架构，编写出了与平台相关的代码，就可以获得更高的性能。所以，提供平台相关的解释与通用的 GPU 概念一样有价值。本书内容的设计方式是，越靠后的章节，内容越具有平台特定性。

我认为本书最独特的地方就是通过第一部分中的 CPU 多线程来解释并行。第二部分介绍了 GPU 的大规模并行（与 CPU 的并行不同）。由于第一部分解释了 CPU 并行的方式，因此读者在第二部分中可以较为容易地理解 GPU 的并行。在过去的 6 年中，我设计了这种方法来讲授 GPU 编程，认识到从未学过并行编程课程的学生并不是很清楚大规模并行的概念。与 GPU 相比，“并行化任务”的概念在 CPU 架构中更容易理解。

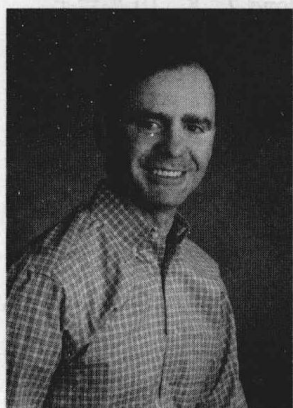
本书的组织如下。第一部分（第 1 章至第 5 章）使用一些简单的程序来演示如何将大任务分成多个并行的子任务并将它们映射到 CPU 线程，分析了同一任务的多种并行实现方式，并根据计算核心和存储单元操作来研究这些方法的优缺点。本书的第二部分（第 6 章至第 11 章）将同一个程序在多个 Nvidia GPU 平台（Fermi、Kepler、Maxwell 和 Pascal）上并行化，并进行性能分析。由于 CPU 和 GPU 的核心和内存结构不同，分析结果的差异有时很有趣，有时与直觉相反。本书指出了这些结果的不同之处，并讨论了如何让 GPU 代码运行得更快。本书的最终目标是让程序员了解所有的做法，这样他们就可以应用好的做法，并避免将不好的做法应用到项目中。

尽管第一部分和第二部分已经完全涵盖了编写一个好的 CUDA 程序需要的所有内容，但总会有更多需要了解的东西。本书的第三部分为希望拓宽视野的读者指明了方向。第三部分并不是相关主题的详细参考文档，只是给出了一些入门介绍，读者可以从中获得学习这些内容的动力。这部分主要介绍了一些流行的 CUDA 库，比如 cuBLAS、cuFFT、Nvidia Performance Primitives 和 Thrust（第 12 章）；OpenCL 编程语言（第 13 章）；使用其他编程语言和 API 库进行 GPU 编程，包括 Python、Metal、Swift、OpenGL、OpenGL ES、OpenCV 和微软 HLSL（第 14 章）；深度学习库 cuDNN（第 15 章）。

书中代码的下载地址为：<https://www.crcpress.com/GPU-Parallel-ProgramDevelopment-Using-CUDA/Soyata/p/book/9781498750752>。

Tolga Soyata

## About the Author 关于作者



Tolga Soyata 于 1988 年在伊斯坦布尔技术大学电子与通信工程系获得学士学位，1992 年在美国马里兰州巴尔的摩的约翰·霍普金斯大学电气与计算机工程系（ECE）获得硕士学位，2000 年在罗切斯特大学电气与计算机工程系获得博士学位。2000 年至 2015 年间，他成立了一家 IT 外包和复印机销售/服务公司。在运营公司的同时，他重返学术界，在罗切斯特大学电气与计算机工程系担任研究员。之后，他成为助理教授，并一直担任电气与计算机工程系教职研究人员至 2016 年。在罗切斯特大学电气与计算

机工程系任职期间，他指导了三名博士研究生。其中两人在他的指导下获得博士学位，另一位在他 2016 年加入纽约州立大学奥尔巴尼分校担任电气与计算机工程系副教授时留在了罗切斯特大学。Soyata 的教学课程包括大规模集成电路、模拟电路以及使用 FPGA 和 GPU 进行并行编程。他的研究兴趣包括信息物理系统、数字健康和高性能医疗移动云计算系统等。

Tolga Soyata 从 2009 年开始从事 GPU 编程的教学，当时他联系 Nvidia 将罗切斯特大学认证为 CUDA 教学中心（CTC）。在 Nvidia 将罗切斯特大学认证为教学中心后，他成为主要负责人。之后，Nvidia 还将罗切斯特大学认证为 CUDA 研究中心（CRC），他也成为项目负责人。Tolga Soyata 在罗切斯特大学担任这些计划的负责人直到他于 2016 年加入纽约州立大学奥尔巴尼分校。这些计划后来被 Nvidia 命名为 GPU 教育中心和 GPU 研究中心。在罗切斯特大学期间，他讲授了 5 年 GPU 编程和高级 GPU 项目开发课程，这些课程同时被列入电气与计算机工程系以及计算机科学与技术系的课程体系。自 2016 年加入纽约州立大学奥尔巴尼分校以来，他一直在讲授类似的课程。本书是他在两所大学讲授 GPU 课程的经验结晶。

# 目 录 Contents

译者序	1.5.2 在 Windows 7、8、10 平台上 开发..... 12
前言	1.5.3 在 Mac 平台上开发..... 14
关于作者	1.5.4 在 Unix 平台上开发..... 14
<b>第一部分 理解 CPU 的并行性</b>	1.6 Unix 速成..... 15
<b>第1章 CPU并行编程概述</b> ..... 2	1.6.1 与目录相关的 Unix 命令..... 15
1.1 并行编程的演化..... 2	1.6.2 与文件相关的 Unix 命令..... 16
1.2 核心越多, 并行性越高..... 3	1.7 调试程序..... 19
1.3 核心与线程..... 4	1.7.1 gdb..... 19
1.3.1 并行化更多的是线程还是 核心..... 5	1.7.2 古典调试方法..... 20
1.3.2 核心资源共享的影响..... 6	1.7.3 valgrind..... 22
1.3.3 内存资源共享的影响..... 6	1.8 第一个串行程序的性能..... 22
1.4 第一个串行程序..... 7	1.8.1 可以估计执行时间吗..... 23
1.4.1 理解数据传输速度..... 8	1.8.2 代码执行时 OS 在做什么..... 23
1.4.2 imflip.c 中的 main() 函数..... 9	1.8.3 如何并行化..... 24
1.4.3 垂直翻转行: FlipImageV()..... 10	1.8.4 关于资源的思考..... 25
1.4.4 水平翻转列: FlipImageH()..... 11	<b>第2章 开发第一个CPU并行程序</b> ..... 26
1.5 程序的编辑、编译、运行..... 12	2.1 第一个并行程序..... 26
1.5.1 选择编辑器和编译器..... 12	2.1.1 imflipP.c 中的 main() 函数..... 27
	2.1.2 运行时间..... 28
	2.1.3 imflipP.c 中 main() 函数代码的 划分..... 28

2.1.4	线程初始化	30	3.4.3	线程状态	58
2.1.5	创建线程	31	3.4.4	将软件线程映射到硬件线程	59
2.1.6	线程启动 / 执行	32	3.4.5	程序性能与启动的线程	60
2.1.7	线程终止 (合并)	33	3.5	改进 imflipP	61
2.1.8	线程任务和数据划分	34	3.5.1	分析 MTFliPH() 中的内存访问模式	62
2.2	位图文件	35	3.5.2	MTFliPH() 的多线程内存访问	63
2.2.1	BMP 是一种无损 / 不压缩的文件格式	35	3.5.3	DRAM 访问的规则	64
2.2.2	BMP 图像文件格式	36	3.6	imflipPM: 遵循 DRAM 的规则	65
2.2.3	头文件 ImageStuff.h	37	3.6.1	imflipP 的混乱内存访问模式	65
2.2.4	ImageStuff.c 中的图像操作函数	38	3.6.2	改进 imflipP 的内存访问模式	65
2.3	执行线程任务	40	3.6.3	MTFliPHM(): 内存友好的 MTFliPH()	66
2.3.1	启动线程	41	3.6.4	MTFliPVM(): 内存友好的 MTFliPV()	69
2.3.2	多线程垂直翻转函数 MTFliPV()	43	3.7	imflipPM.C 的性能	69
2.3.3	FlipImageV() 和 MTFliPV() 的比较	46	3.7.1	imflipP.c 和 imflipPM.c 的性能比较	70
2.3.4	多线程水平翻转函数 MTFliPH()	47	3.7.2	速度提升: MTFliPV() 与 MTFliPVM()	71
2.4	多线程代码的测试 / 计时	49	3.7.3	速度提升: MTFliPH() 与 MTFliPHM()	71
<b>第3章</b>	<b>改进第一个CPU并程序</b>	<b>51</b>	3.7.4	理解加速: MTFliPH() 与 MTFliPHM()	71
3.1	程序员对性能的影响	51	3.8	进程内存映像	72
3.2	CPU 对性能的影响	52	3.9	英特尔 MIC 架构: Xeon Phi	74
3.2.1	按序核心与乱序核心	53	3.10	GPU 是怎样的	75
3.2.2	瘦线程与胖线程	55	3.11	本章小结	76
3.3	imflipP 的性能	55			
3.4	操作系统对性能的影响	56			
3.4.1	创建线程	57			
3.4.2	线程启动和执行	57			

<b>第4章 理解核心和内存</b> .....	77	4.7.3 Rotate5(): 整数除法 / 乘法 有多差 .....	98
4.1 曾经的英特尔 .....	77	4.7.4 Rotate6(): 合并计算 .....	100
4.2 CPU 和内存制造商 .....	78	4.7.5 Rotate7(): 合并更多计算 .....	100
4.3 动态存储器与静态存储器 .....	79	4.7.6 imrotateMC 的总体性能 .....	101
4.3.1 静态随机存取存储器 (SRAM) .....	79	4.8 本章小结 .....	103
4.3.2 动态随机存取存储器 (DRAM) .....	79	<b>第5章 线程管理和同步</b> .....	104
4.3.3 DRAM 接口标准 .....	79	5.1 边缘检测程序: imedge.c .....	104
4.3.4 DRAM 对程序性能的影响 .....	80	5.1.1 imedge.c 的说明 .....	105
4.3.5 SRAM 对程序性能的影响 .....	81	5.1.2 imedge.c: 参数限制和简化 .....	106
4.4 图像旋转程序: imrotate.c .....	81	5.1.3 imedge.c: 实现原理 .....	106
4.4.1 imrotate.c 的说明 .....	82	5.2 imedge.c: 实现 .....	108
4.4.2 imrotate.c: 参数限制和简化 .....	82	5.2.1 初始化和时间戳 .....	109
4.4.3 imrotate.c: 实现原理 .....	83	5.2.2 不同图像表示的初始化 函数 .....	110
4.5 imrotate 的性能 .....	87	5.2.3 启动和终止线程 .....	111
4.5.1 线程效率的定性分析 .....	87	5.2.4 高斯滤波 .....	112
4.5.2 定量分析: 定义线程效率 .....	87	5.2.5 Sobel .....	113
4.6 计算机的体系结构 .....	89	5.2.6 阈值过滤 .....	114
4.6.1 核心、L1\$ 和 L2\$ .....	89	5.3 imedge 的性能 .....	115
4.6.2 核心内部资源 .....	90	5.4 imedgeMC: 让 imedge 更高效 .....	116
4.6.3 共享 L3 高速缓存 (L3 \$) .....	91	5.4.1 利用预计算降低带宽 .....	116
4.6.4 内存控制器 .....	92	5.4.2 存储预计算的像素值 .....	117
4.6.5 主存 .....	92	5.4.3 预计算像素值 .....	118
4.6.6 队列、非核心和 I/O .....	93	5.4.4 读取图像并预计算像素值 .....	119
4.7 imrotateMC: 让 imrotate 更 高效 .....	94	5.4.5 PrGaussianFilter .....	120
4.7.1 Rotate2(): 平方根和浮点 除法有多差 .....	96	5.4.6 PrSobel .....	121
4.7.2 Rotate3() 和 Rotate4(): sin() 和 cos() 有多差 .....	97	5.4.7 PrThreshold .....	121
		5.5 imedgeMC 的性能 .....	122
		5.6 imedgeMCT: 高效的线程同步 .....	123

5.6.1	屏障同步	124
5.6.2	用于数据共享的 MUTEX 结构	125
5.7	imedgeMCT: 实现	127
5.7.1	使用 MUTEX: 读取图像、 预计算	128
5.7.2	一次预计算一行	130
5.8	imedgeMCT 的性能	131

## 第二部分 基于 CUDA 的 GPU 编程

### 第6章 GPU并行性和CUDA概述 134

6.1	曾经的 Nvidia	134
6.1.1	GPU 的诞生	134
6.1.2	早期的 GPU 架构	136
6.1.3	GPGPU 的诞生	137
6.1.4	Nvidia、ATI Technologies 和 Intel	138
6.2	统一计算设备架构	140
6.2.1	CUDA、OpenCL 和其他 GPU 语言	140
6.2.2	设备端与主机端代码	140
6.3	理解 GPU 并行	141
6.3.1	GPU 如何实现高性能	142
6.3.2	CPU 与 GPU 架构的差异	143
6.4	图像翻转的 CUDA 版: imflipG.cu	144
6.4.1	imflipG.cu: 将图像读入 CPU 端数组	146
6.4.2	初始化和查询 GPU	147

6.4.3	GPU 端的时间戳	148
6.4.4	GPU 端内存分配	152
6.4.5	GPU 驱动程序和 Nvidia 运行时 引擎	153
6.4.6	CPU 到 GPU 的数据传输	153
6.4.7	用封装函数进行错误报告	154
6.4.8	GPU 核函数的执行	155
6.4.9	完成 GPU 核函数的执行	157
6.4.10	将 GPU 结果传回 CPU	158
6.4.11	完成时间戳	158
6.4.12	输出结果和清理	158
6.4.13	读取和输出 BMP 文件	159
6.4.14	Vflip(): 垂直翻转的 GPU 核函数	160
6.4.15	什么是线程 ID、块 ID 和块 维度	163
6.4.16	Hflip(): 水平翻转的 GPU 核函数	165
6.4.17	硬件参数: threadIdx.x、 blockIdx.x 和 blockDim.x	165
6.4.18	PixCopy(): 复制图像的 GPU 核函数	165
6.4.19	CUDA 关键字	166
6.5	Windows 中的 CUDA 程序 开发	167
6.5.1	安装 MS Visual Studio 2015 和 CUDA Toolkit 8.0	167
6.5.2	在 Visual Studio 2015 中创建 项目 imflipG.cu	168
6.5.3	在 Visual Studio 2015 中编译 项目 imflipG.cu	170

6.5.4	运行第一个 CUDA 应用程序: imflipG.exe	173	7.3	imflipG.cu: 理解核函数的 细节	189
6.5.5	确保程序的正确性	174	7.3.1	在 main() 中启动核函数并将 参数传递给它们	189
6.6	Mac 平台上的 CUDA 程序 开发	175	7.3.2	线程执行步骤	190
6.6.1	在 Mac 上安装 XCode	175	7.3.3	Vflip() 核函数	191
6.6.2	安装 CUDA 驱动程序和 CUDA 工具包	176	7.3.4	Vflip() 和 MTFliPv() 的 比较	192
6.6.3	在 Mac 上编译和运行 CUDA 应用程序	177	7.3.5	Hflip() 核函数	194
6.7	Unix 平台上的 CUDA 程序 开发	177	7.3.6	PixCopy() 核函数	194
6.7.1	安装 Eclipse 和 CUDA 工具包	177	7.4	PCIe 速度与 CPU 的关系	196
6.7.2	使用 ssh 登录一个集群	178	7.5	PCIe 总线对性能的影响	196
6.7.3	编译和执行 CUDA 代码	179	7.5.1	数据传输时间、速度、延迟、 吞吐量和带宽	196
<b>第7章</b>	<b>CUDA主机/设备编程模型</b>	181	7.5.2	imflipG.cu 的 PCIe 吞吐量	197
7.1	设计程序的并行性	181	7.6	全局内存总线对性能的影响	200
7.1.1	任务的并行化	182	7.7	计算能力对性能的影响	203
7.1.2	什么是 Vflip() 的最佳块 尺寸	183	7.7.1	Fermi、Kepler、Maxwell、 Pascal 和 Volta 系列	203
7.1.3	imflipG.cu: 程序输出的 解释	183	7.7.2	不同系列实现的相对带宽	204
7.1.4	imflipG.cu: 线程块和图像的 大小对性能的影响	184	7.7.3	imflipG2.cu: 计算能力 2.0 版本 的 imflipG.cu	205
7.2	核函数的启动	185	7.7.4	imflipG2.cu: main() 的 修改	206
7.2.1	网格	185	7.7.5	核函数 PxCC20()	208
7.2.2	线程块	187	7.7.6	核函数 VfCC20()	208
7.2.3	线程	187	7.8	imflipG2.cu 的性能	210
7.2.4	线程束和通道	189	7.9	古典的 CUDA 调试方法	212
			7.9.1	常见的 CUDA 错误	212
			7.9.2	return 调试法	214
			7.9.3	基于注释的调试	216

7.9.4	printf() 调试	216	8.4.6	输出核函数性能	236
7.10	软件错误的生物学原因	217	8.5	imedgeG: 核函数	237
7.10.1	大脑如何参与编写 / 调试 代码	218	8.5.1	BWKernel()	237
7.10.2	当我们疲倦时是否会写出 错误代码	219	8.5.2	GaussKernel()	239
<b>第8章</b>	<b>理解GPU的硬件架构</b>	<b>221</b>	8.5.3	SobelKernel()	240
8.1	GPU 硬件架构	222	8.5.4	ThresholdKernel()	242
8.2	GPU 硬件的部件	222	8.6	imedgeG.cu 的性能	243
8.2.1	SM: 流处理器	222	8.6.1	imedgeG.cu: PCIe 总线利 用率	244
8.2.2	GPU 核心	223	8.6.2	imedgeG.cu: 运行时间	245
8.2.3	千兆线程调度器	223	8.6.3	imedgeG.cu: 核函数性能 比较	247
8.2.4	内存控制器	225	8.7	GPU 代码: 编译时间	248
8.2.5	共享高速缓存 (L2\$)	225	8.7.1	设计 CUDA 代码	248
8.2.6	主机接口	225	8.7.2	编译 CUDA 代码	250
8.3	Nvidia GPU 架构	226	8.7.3	GPU 汇编: PTX、CUBIN	250
8.3.1	Fermi 架构	227	8.8	GPU 代码: 启动	250
8.3.2	GT、GTX 和计算加速器	227	8.8.1	操作系统的参与和 CUDA DLL 文件	250
8.3.3	Kepler 架构	228	8.8.2	GPU 图形驱动	251
8.3.4	Maxwell 架构	228	8.8.3	CPU 与 GPU 之间的内存 传输	251
8.3.5	Pascal 架构和 NVLink	229	8.9	GPU 代码: 执行 (运行时间)	252
8.4	CUDA 边缘检测: imedgeG.cu	229	8.9.1	获取数据	252
8.4.1	CPU 和 GPU 内存中存储图像 的变量	229	8.9.2	获取代码和参数	252
8.4.2	为 GPU 变量分配内存	231	8.9.3	启动线程块网格	252
8.4.3	调用核函数并对其进行 计时	233	8.9.4	千兆线程调度器 (GTS)	253
8.4.4	计算核函数的性能	234	8.9.5	线程块调度	254
8.4.5	计算核函数的数据移动量	235	8.9.6	线程块的执行	255
			8.9.7	透明的可扩展性	256



<b>第9章 理解GPU核心</b> .....	257	9.3.2 INT16: 16 位整数 .....	274
<b>9.1 GPU 的架构系列</b> .....	258	9.3.3 24 位整数 .....	275
9.1.1 Fermi 架构 .....	258	9.3.4 INT32: 32 位整数 .....	275
9.1.2 Fermi SM 的结构 .....	259	9.3.5 判定寄存器 (32 位) .....	275
9.1.3 Kepler 架构 .....	260	9.3.6 INT64: 64 位整数 .....	276
9.1.4 Kepler SMX 的结构 .....	260	9.3.7 128 位整数 .....	276
9.1.5 Maxwell 架构 .....	261	9.3.8 FP32: 单精度浮点 (float) .....	277
9.1.6 Maxwell SMM 的结构 .....	262	9.3.9 FP64: 双精度浮点 (double) .....	277
9.1.7 Pascal GP100 架构 .....	264	9.3.10 FP16: 半精度浮点 (half) .....	278
9.1.8 Pascal GP100 SM 的结构 .....	265	9.3.11 什么是 FLOP .....	278
9.1.9 系列比较: 峰值 GFLOPS 和 峰值 DGFLOPS .....	266	9.3.12 融合乘法累加 (FMA) 与 乘加 (MAD) .....	278
9.1.10 GPU 睿频 .....	267	9.3.13 四倍和八倍精度浮点 .....	279
9.1.11 GPU 功耗 .....	268	9.3.14 Pascal GP104 引擎的 SM 结构 .....	279
9.1.12 计算机电源 .....	268	<b>9.4 imflipGC.cu: 核心友好的 imflipG</b> .....	280
<b>9.2 流处理器的构建模块</b> .....	269	9.4.1 Hflip2(): 预计算核函数 参数 .....	282
9.2.1 GPU 核心 .....	269	9.4.2 Vflip2(): 预计算核函数 参数 .....	284
9.2.2 双精度单元 (DPU) .....	270	9.4.3 使用线程计算图像坐标 .....	285
9.2.3 特殊功能单元 (SFU) .....	270	9.4.4 线程块 ID 与图像的行 映射 .....	285
9.2.4 寄存器文件 (RF) .....	270	9.4.5 Hflip3(): 使用二维启动 网格 .....	286
9.2.5 读取 / 存储队列 (LDST) .....	271	9.4.6 Vflip3(): 使用二维启动 网格 .....	287
9.2.6 L1\$ 和纹理高速缓存 .....	272	9.4.7 Hflip4(): 计算 2 个连续的 像素 .....	288
9.2.7 共享内存 .....	272		
9.2.8 常量高速缓存 .....	272		
9.2.9 指令高速缓存 .....	272		
9.2.10 指令缓冲区 .....	272		
9.2.11 线程束调度器 .....	272		
9.2.12 分发单元 .....	273		
<b>9.3 并行线程执行 (PTX) 的数据   类型</b> .....	273		
9.3.1 INT8: 8 位整数 .....	274		