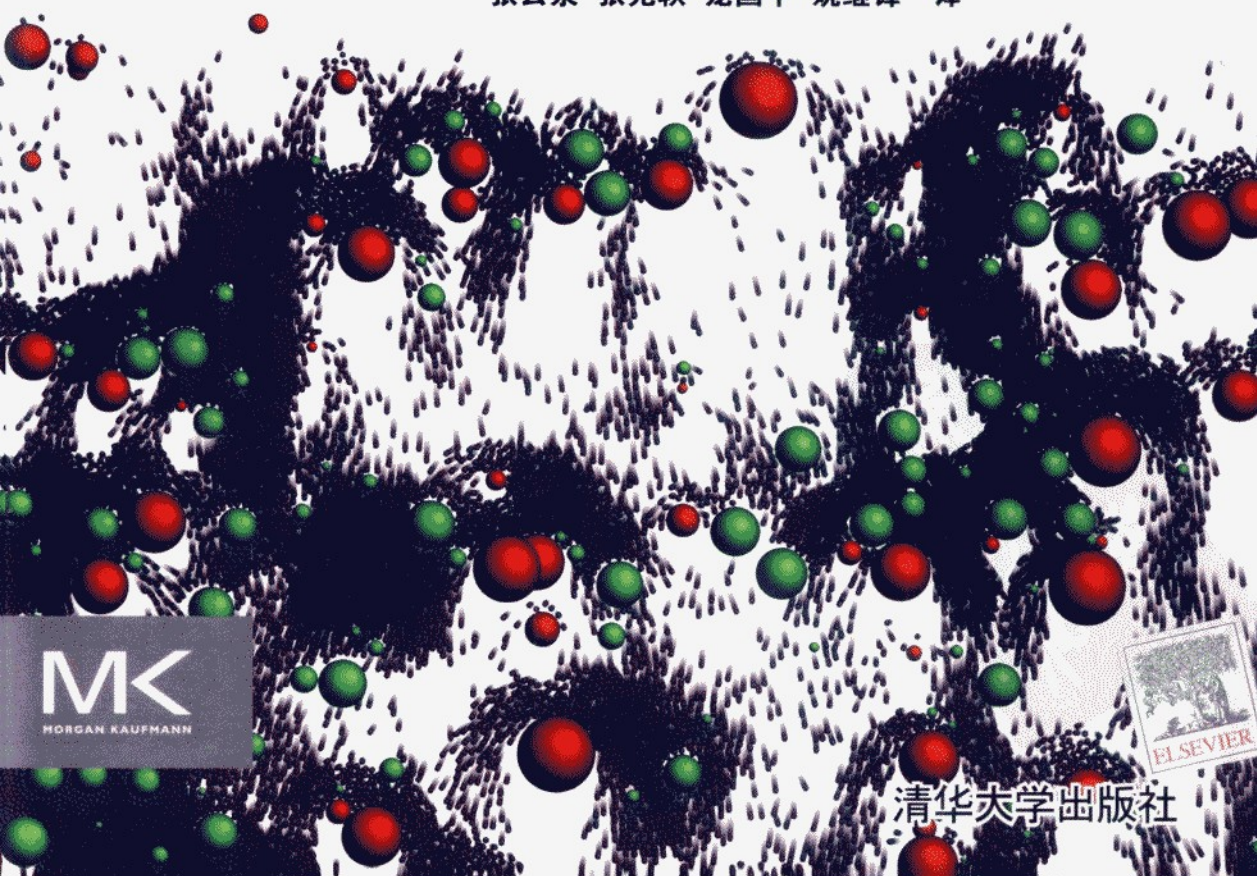


Heterogeneous Computing with OpenCL

OpenCL

异构计算

Benedict R. Gaster Lee Howes David R. Kaeli 著
Perhaad Mistry Dana Schaa
张云泉 张先轶 龙国平 姚继锋 译



MK

MORGAN KAUFMANN



清华大学出版社

Heterogeneous Computing with OpenCL

OpenCL 异构计算

来自OpenCL社区的CPU, GPU和APU并行编程权威指南

专家点评

并行计算现在已经成为主流。针对CPU-GPU系统加速应用的最新流行技术, 本书提供了非常出色的参考。

——David A. Bader, 乔治亚技术学院

《OpenCL异构计算》讲解了OpenCL和如何针对复杂的异构系统(多核处理器、GPU和APU)进行并行编程。OpenCL是为支持多平台设计的, 受行业广泛支持, 有助于帮助读者为异构环境编写高性能应用程序。

本书由并行计算和OpenCL社区中的领头人撰写, 可帮助读者体验OpenCL从而了解大量基础的并行算法。书中探索了内存空间、优化技术、图形的互操作、性能扩展, 调试和性能剖析等。

本书适合软件工程师、程序员、硬件工程师、高年级本科生/研究生阅读和参考, 也可供并行编程课程使用。

配套网站: <http://www.heterogeneouscompute.org>

本书特色

- 解释OpenCL并行编程的原理和策略, 从理解四大抽象模型开始, 一直介绍到测试和调试完整的应用
- 覆盖图像处理、Web插件、粒子模拟、视频编辑和性能优化等主题
- 展示OpenCL如何映射到目标体系结构, 并解释一些映射到各种体系结构的折衷方案
- 提出一系列基础编程技术, 并通过若干个实例和案例分析来展示针对不同硬件平台的OpenCL扩展

作者简介

Benedict R. Gaster AMD OpenCL架构师

Lee Howes AMD技术骨干

David R. Kaeli 美国东北大学计算机架构研究实验室(NUCAR)主管、信息安全保障研究院(IIA)联合主管、工程学院副院长

Perhaad Mistry 美国东北大学电气与计算机工程系博士研究生

Dana Schaa 美国东北大学电气与计算机工程系博士研究生

清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

ISBN 978-7-302-28685-1



9 787302 286851 >

定价: 76.00元

OpenCL 异构计算

Benedict R. Gaster Lee Howes David R. Kaeli 著
Perhaad Mistry Dana Schaa

张云泉 张先轶 龙国平 姚继锋 译

清华大学出版社
北 京

内 容 简 介

《OpenCL 异构计算》提供 OpenCL 的第一手资料, 详尽阐述了如何在异构环境下进行并行编程。第 1 章首先介绍如何在并行系统下编程, 定义异构编程需要理解的概念。第 2~4 章循序渐进地介绍 OpenCL 的基本架构。第 5 章和第 6 章对这些概念加以扩展, 旨在帮助读者更好地理解。第 7~10 章提供 4 个更复杂的案例学习, 让读者理解到 OpenCL 具有广泛的应用。第 11~13 章锁定高级主题展开讨论。

本书可帮助学生和研究人员更好地理解通用异构计算(尤其是 OpenCL 提供的解决方案), 尤其是适合不同经验水平的学生, 可以作为 OpenCL 课程的教材或其他课程的参考, 例如并行编程课程和高级课程。

Heterogeneous Computing with OpenCL, 1st Edition

Benedict R. Gaster, Lee Howes, David R. Kaeli, Perhaad Mistry, Dana Schaa

ISBN: 978-0123877666

Copyright©2012 Advanced Micro Devices, Inc. Published by Elsevier(Inc). All rights reserved.

Authorized Simplified Chinese translation edition published by Elsevier (Singapore) Pte Ltd Press and Tsinghua University.

Copyright©2012 by Elsevier (Singapore) Pte Ltd and Tsinghua University Press. All rights reserved.

Published in China by Tsinghua University Press under special arrangement with Elsevier (Singapore) Pte Ltd.. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由 Elsevier (Singapore) Pte Ltd. 授予清华大学出版社在中国大陆地区(不包括香港、澳门特别行政区以及台湾地区)出版与发行。未经许可之出口, 视为违反著作权法, 将受法律之制裁。

本书封底贴有 Elsevier 防伪标签, 无标签者不得销售。

版权所有, 侵权必究。侵权举报电话: 010-62782989 13701121933

北京市版权局著作权合同登记号 图字: 01-2012-2719

图书在版编目(CIP)数据

OpenCL 异构计算/贾斯特(Gaster,R. B.)等著; 张云泉等译. --北京: 清华大学出版社, 2012.6

书名原文: Heterogeneous Computing with OpenCL

ISBN 978-7-302-28685-1

I. ①O… II. ①贾… ②张… III. ①图形软件 IV. ①TP391.41

中国版本图书馆 CIP 数据核字(2012)第 083719 号

责任编辑: 文开琪

装帧设计: 杨玉兰

责任校对: 周剑云

责任印制: 李红英

出版发行: 清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址: 北京清华大学学研大厦 A 座 邮 编: 100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

课 件 下 载: <http://www.tup.com.cn>, 010-62791865

印 刷 者: 清华大学印刷厂

装 订 者: 三河市新茂装订有限公司

经 销: 全国新华书店

开 本: 185mm×230mm

印 次: 2012 年 6 月第 1 版

印 数: 1~4000

定 价: 76.00 元

印 张: 18.75

字 数: 390 千字

印 次: 2012 年 6 月第 1 次印刷

产品编号: 045681-01

推荐序 1

陈国良

中国科学院院士

我国超级计算机研制近几年奋起直追。2009年，天河一号(TH-1)的发布，使得我国成为继美国之后世界上第二个能够独立研制千万亿次峰值超级计算机的国家。2010年9月发布的国产天河一号A千万亿次超级计算机通过采用先进的CPU+GPGPU的异构混合加速体系架构，以2.56Pflops的Linpack性能夺取2010年11月国际HPC TOP500排行榜的第一名，创下中国国产超级计算机历史上首次夺得世界冠军的佳绩。同时，我国安装的超级计算机的上榜数量也首次超越日本和欧盟，排名世界第二，仅次于美国。目前，中国HPC TOP100排行榜中机器的平均Linpack性能为120Tflops，是2010年63Tflops的1.9倍，而国际上平均性能接近120Tflops的时间为2011年6月(117.76Tflops)，我国与世界平均性能差距从原来保持了很久的一年半差距，快速缩短为半年的差距！这标志着我国超级计算机系统的整机集成水平步入国际领先行列，且在一些核心硬件技术上取得了关键性的突破。2011年底的国际HPC TOP500排行榜中，前十名中的第二、四和第五名都采用了GPGPU异构加速体系架构——500台系统里有39台系统采用了GPGPU异构加速，由此可见，异构计算作为一种新的计算机体系架构逐渐成为主流。

每一次超级计算机体系结构的变化，都会伴随着并行程序设计环境和语言的进化。为了高效地为分布式存储和共享存储的并行计算机编写并行程序，工业界在总结众多各类并行语言的优点的基础上，分别提出了目前已经被广泛接受和使用的并行程序开发环境MPI和OpenMP工业界标准。这两个标准并行程序设计环境和语言的提出和推广普及，极大地提升了大规模并行应用软件的开发效率，降低了开发成本，确保了软件在不同厂家机器之间的可移植性和高性能。随着当前国际上主流体系架构逐渐向异构混合并行的转变，如何为这一新体系架构设计高效的且被广为接受的工业界标准并行程序开发环境和语言，成为一个迫切需要解决的问题。

OpenCL正是在这一背景下由Khronos国际组织提出，并得到众多硬件和软件厂商的支持，成为工业界异构计算的标准语言。虽然目前比较流行的异构计算的語言是CUDA语言，但是，我们相信，随着时间的推移，正如当年早期最流行的PVM分布式存储并

目 录

第 1 章 并行编程入门.....1

引言.....1

OpenCL.....1

本书目标.....2

并行思维.....2

并发编程模型和并行编程模型.....6

线程和共享内存.....9

消息传递通信.....9

不同的并行粒度.....10

数据共享和同步.....11

本书结构.....11

参考文献.....12

扩展阅读和相关网站.....13

第 2 章 OpenCL 简介.....15

引言.....15

OpenCL 标准.....15

OpenCL 规范.....15

kernel 和 OpenCL 执行模型.....16

平台和设备.....19

主机-设备交互.....19

执行环境.....21

上下文.....22

命令队列.....22

事件.....23

内存对象.....23

flush 命令和 finish 命令.....26

新建一个 OpenCL 程序对象.....26

OpenCL 的 kernel.....27

内存模型.....29

编写 kernel.....31

向量相加实例的完整代码.....32

小结.....39

参考文献.....39

第 3 章 OpenCL 设备架构.....41

引言.....41

硬件权衡.....41

性能随频率的提升及其限制.....43

超标量执行.....44

VLW.....44

SIMD 和向量处理.....47

硬件多线程.....48

多核架构.....51

集成：片上系统和 APU.....53

高速缓存层次和内存系统.....54

架构设计空间.....55

CPU 设计.....56

GPU 体系结构.....60

APU 和类 APU 的设计.....63

小结.....64

参考文献.....65

第 4 章 OpenCL 基本实例.....67

引言.....67

应用实例	67	OpenCL 在 AMD RADEON HD6970	
简单的矩阵相乘	67	GPU 上的实现	128
图像卷积实例	77	多线程和内存系统	130
小结	85	基于 clause 的 SIMD 执行	132
第 5 章 OpenCL 的并发与执行模型	87	资源分配	137
引言	87	OpenCL 的内存性能	139
kernel, work_item, workgroup 和		OpenCL 全局内存	139
执行域	87	本地内存——软件管理的 cache	142
OpenCL 同步: kernel, fence 和 barrier	90	小结	148
队列与全局同步	94	参考文献	149
OpenCL 内存一致性	96	第 7 章 OpenCL 案例学习 1: 卷积 ...	151
事件	96	引言	151
命令 barrier 与 marker	108	计算卷积的 kernel	151
主机端内存模型	109	选择合适的 workgroup 大小	151
buffer 对象	110	将数据缓存到本地内存	154
image 对象	113	执行卷积	160
设备端内存模型	115	小结	161
设备端宽松的内存一致性	116	代码清单	162
全局内存	117	主机端代码	162
本地内存	119	kernel 代码	166
常量内存	121	参考文献	171
私有内存	122	第 8 章 OpenCL 案例学习 2:	
小结	122	视频处理	173
第 6 章 OpenCL 在 CPU/GPU		引言	173
平台上的实现	123	获得视频帧	173
引言	123	CPU 上的解码	174
OpenCL 在 AMD PHENOM II X6 上的		在 GPU 上解码视频	175
实现	123	在 OpenCL 中处理一个视频	179
		在多个视频上处理多个不同 effect	180

事件链	180	粒子模拟的 kernel 代码	206
最终输出显示到屏幕	181	第 11 章 OpenCL 扩展	211
OpenCL/OpenGL 协同工作能力	181	引言	211
小结	184	扩展机制概览	211
第 9 章 OpenCL 案例学习 3:		设备拆分	214
直方图	185	双精度	225
引言	185	参考文献	233
选择适量的 work-group	185	第 12 章 OpenCL 的性能剖析和	
选择最优的 work-group 大小	186	调试	235
全局内存访问优化	187	引言	235
使用原子操作计算局部直方图	189	基于事件的剖析	236
本地内存访问优化	190	AMD APP Profiler	238
局部直方图的规约	192	收集 OpenCL 程序轨迹	239
全局规约	193	收集 OpenCL GPU Kernel 性能	
完整的 kernel 代码	193	计数器	242
性能和小结	196	AMD APP KernelAnalyzer	243
第 10 章 OpenCL 案例学习 4:		演示 AMD APP Profiler	245
混合粒子模拟	197	启动 AMD APP Profiler	245
引言	197	使用应用程序的轨迹数据	
计算概览	197	以发现性能瓶颈	245
GPU 实现	200	使用 GPU 性能计数器发现 kernel 的	
创建 buffer	200	性能瓶颈	247
构造加速结构	201	调试 OpenCL 应用程序	248
计算碰撞	201	gDEBgger 概览	249
合成	202	使用 gDEBgger 调试并行 OpenCL	
CPU 实现	202	应用程序	249
负载均衡	203	AMD printf 扩展	251
性能和小结	204	小结	253
生成均匀网格的 kernel 代码	205		

第 13 章 WebCL	255	Web 照片编辑器	264
引言	255	讨论	266
框架设计	256	小结	268
WebCL 实验性实现	257	参考文献	268
Firefox 扩展	257	扩展阅读和相关网站	269
连接 JavaScript 和 OpenCL	258		
WebCL 动手练习	260	索引	271

引言

当前计算环境变得越来越多样化，需要充分利用包括多核微处理器、CPU、数字信号处理器(DSP)、可重构硬件(FPGA)和图形处理单元(GPU)的能力。面对如此大的异构性，在众多硬件体系架构上开发高效的软件，给编程社区带来了诸多挑战。

通常，应用包含了大量工作负载行为，从控制密集型(搜索、排序和分析等)到数据密集型(图像处理、模拟和建模，数据挖掘等)。此外，还包括计算密集型应用(迭代方法、数值方法和金融建模等)，这类应用的吞吐量严重依赖于底层硬件的计算效率。每一类工作负载一般都能在某种特定类型的硬件体系架构上获得最高的运行效率，但是没有一种硬件架构能够让所有类别的工作负载都取得最优的运行效率。而且，大部分应用都包含多种工作负载的特征。例如，控制密集型应用在超标量 CPU 上运行得更快，因为 CPU 使用大量的芯片面积来进行分支预测。数据密集型应用在向量架构上运行更快，同样的操作可以在多个数据元素上并行执行。

OpenCL

开放计算语言(Open Computing Language, OpenCL)是一个由非盈利性技术联盟 Khronos Group 进行管理的异构编程框架。OpenCL 框架用于开发可以在各种设备上运行的应用程序。OpenCL 支持多种层次的并行，可以高效映射到同构或异构的体系结构上，比如单个或者多个 CPU、GPU 和其他类型的设备等。OpenCL 提供了一个系统中设备端语言和主机端控制层两方面的定义。设备端语言可以高效映射到众多的内存系统架构上。主机端语言的目标是以较低开销来高效管理复杂的并发程序。两者共同为开发人员提供了一种从算法设计高效过渡到实现的途径。

OpenCL 提供了基于任务和基于数据这两种并行计算方式。当前支持的 CPU 包括 x86, ARM 和 PowerPC, 同时支持 AMD (Accelerated Parallel Processing SDK)和 NVIDIA 的显卡。支持 OpenCL 的硬件平台在迅速增长, 众多硬件平台厂商已经采用和支持或者计划在其平台上支持 OpenCL。从嵌入式厂商(ARM 和 Imagination Technologies), 到 HPC 厂商(AMD, Intel, NVIDIA 和 IBM), 其市场范围相当广阔。OpenCL 支持的体系架构众多, 包括多核 CPU, GPU 之类的吞吐量和向量处理器, FPGA 之类的细粒度并行设备。

最重要的是, OpenCL 是跨平台的, 得到工业界广泛的支持, 所以它是适合开发人员学习和使用的极佳编程模型。而且, 可以确信 OpenCL 在很多年内会继续持续发展, 不断提高可用性和扩大适用范围。

本书目标

本书是第一本适合课堂教学的介绍 OpenCL 编程的书籍。本书的内容组织符合在当前系统架构上教授 OpenCL 并行编程的要求, 包括 CPU 和 GPU 应用示例, 以及集成两者的加速处理器(Accelerated Processing Unit, APU)。本书的另一个主要目标为程序员提供一个指南, 用于在并行系统上设计开发良好的 OpenCL 程序。本书引导程序员熟悉 OpenCL 编程环境提供的多种抽象和功能。应用实例给读者提供了许多简单的背景介绍和复杂的优化技术, 提出了进一步开发的建议和目标。此外, 本书也介绍了性能剖析和调试等不可缺少的工具, 使读者不至于在开发过程中迷失方向。

本书有一套用于教学的幻灯片和编程范例等补充材料, 以方便 OpenCL 教师使用。详情请访问 <http://heterogeneouscomputingwithopencl.org/>。

并行思维

大部分应用最初编写程序时都是为了在单处理器上运行。在高性能计算领域, 有许多计算资源时, 采用了很多经典的方法以加速计算。通常, 标准的方法包括“分治法”和“散列-收集法”, 程序员可以通过这些策略有效地利用高性能计算系统上的并行资源。

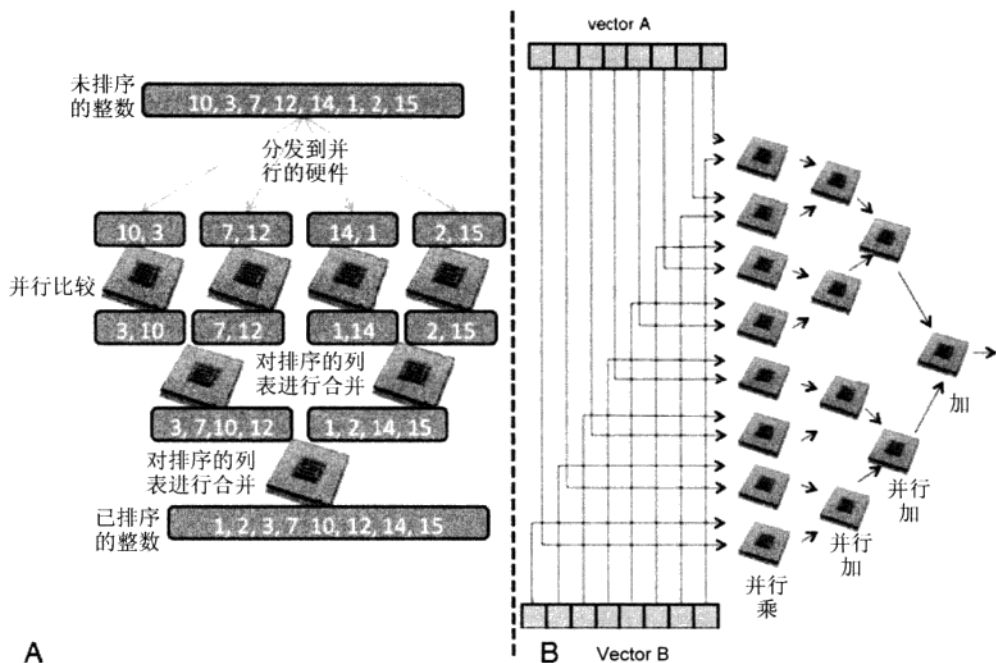


图 1.1

(A)简单排序；(B)向量点乘

“分治法”是将一个问题迭代的划分成子问题，直到子问题可以在当前计算资源上较好地求解。“散列-收集法”是将一些输入数据的子集分配给每一个并行计算资源，之后再收集它们的计算结果，生成最终的结果数据。任务的划分像从前一样基于各个并行资源的计算能力，图 1.1 展示了排序和向量标量相乘等应用如何有效映射到并行计算资源上。

随着现代并行处理器上并行度的增长和异构性的出现，编程越来越成为一项挑战。在当前 CMOS 技术的功耗和散热限制下，微处理器厂商发现很难通过提高主频来提升设备的性能。他们转而采用在一个芯片上放置多个处理器的方法。由此，如何从一个应用程序中提取并行性的问题留给程序员解决。程序员必须分解应用的底层算法，将它们有效地映射到多样化的目标硬件平台上。

在过去的 5 年中，并行计算设备在数量和处理能力上都取得了长足进展。GPU 也开

始用来进行通用计算，以非常低的成本提供了较高的处理能力。受实时三维图像渲染这种高度数据并行问题的驱动，GPU 发展成为一种非常强劲的、可编程的、任务和数据并行的体系架构。现在，硬件厂商将 CPU 和 GPU 结合在一块硅片上，开启了新一代的异构计算。应用中的计算密集型和数据密集型部分，我们称为 kernel(内核)，此部分由 GPU 执行，可以显著提高每瓦特的性能和原始的性能收益，同时主机端的 CPU 也可以继续执行其他的任务。

自然世界和人工环境中的许多系统和现象存在着不同类别的并行性和并发性：

- 分子动力学
- 天气和海洋模式
- 多媒体系统
- 构造板块漂移
- 细胞生长
- 汽车流水线
- 声音和光波传播

针对并行计算，Almasi and Gottlieb(1989)如此定义：“并行计算是许多运算同时进行的一种计算形式，其原理是将大的问题分解成小的问题，之后同时求解。”可达到的并行程度，除了依赖于问题本身的内在并行性，也依赖于算法和软件设计人员识别和表达问题中并行性的技巧。我们将通过向量相乘和文本搜索这两个简单的例子来展示和讲解并行计算。

我们首先考察两个数组的元素相乘的例子。如图 1.2 所示，包含 N 个元素的数组 A 和 B，将相乘后的结果对应的保存在数组 C 中。串行的 C++代码如下所示：

```
for (i=0; i<N; i++)  
    C[i] = A[i] * B[i];
```

这段代码的并行度非常高，但是运算量很小。数组 C 中每个元素的计算都是相互独立的。如果我们并行化这段代码，会选择使用为计算数组 C 中每一个元素生成一个单独的计算实例。这段代码是一个典型的数据并行示例，因为数组 A 和 B 的每个元素生成数组 C 的元素时都执行了相同的操作。我们也可以将它视为任务并行的一种简化形式，每个任务都操作同样数据的不同子集。但是，任务并行一般表现为数据的流水线执行或者一些更复杂的并行交互。图 1.3 是任务并行的例子，采用流水线方式使用 FFT 在频域空间上进行图像滤波。

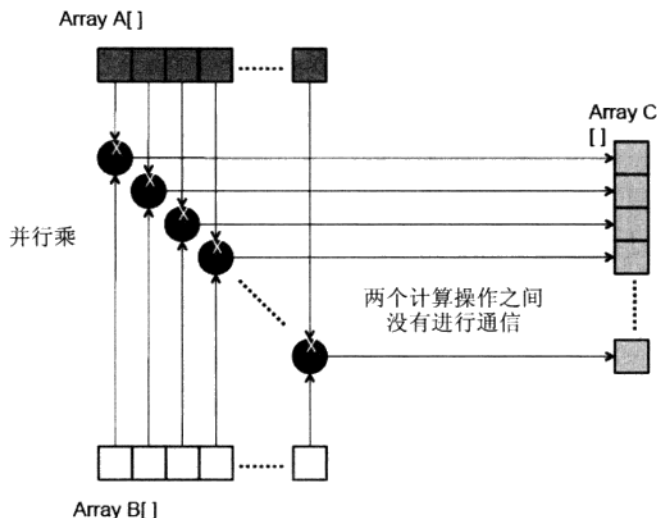


图 1.2

两个数组相乘：本例不需要互相通信

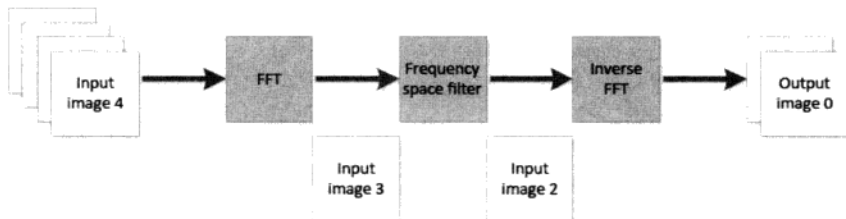


图 1.3

使用 FFT 对一系列图像进行滤波，清楚展示了任务并行，即使用流水线的方式进行一系列操作以计算出最终结果

接下来是第二个例子。在本例中，我们需要统计某个字符串在一段文本中出现的次数(图 1.4)。假设这段文本已经变为 N 个单词的集合。我们将任务分解为 N 个字符串的比较(即任务)，即每个单词与此字符串进行匹配。这种方法虽然在搜索效率上非常低，但具有很高的并行性。在每个比较任务中，我们可以进一步挖掘并行性，比如每个字符和字符间的比较隐含着一种细粒度的并行。这个例子同时包含数据并行(在多个数据元素上执行同样的操作)和任务并行(将字符串与所有的单词同时比较)。

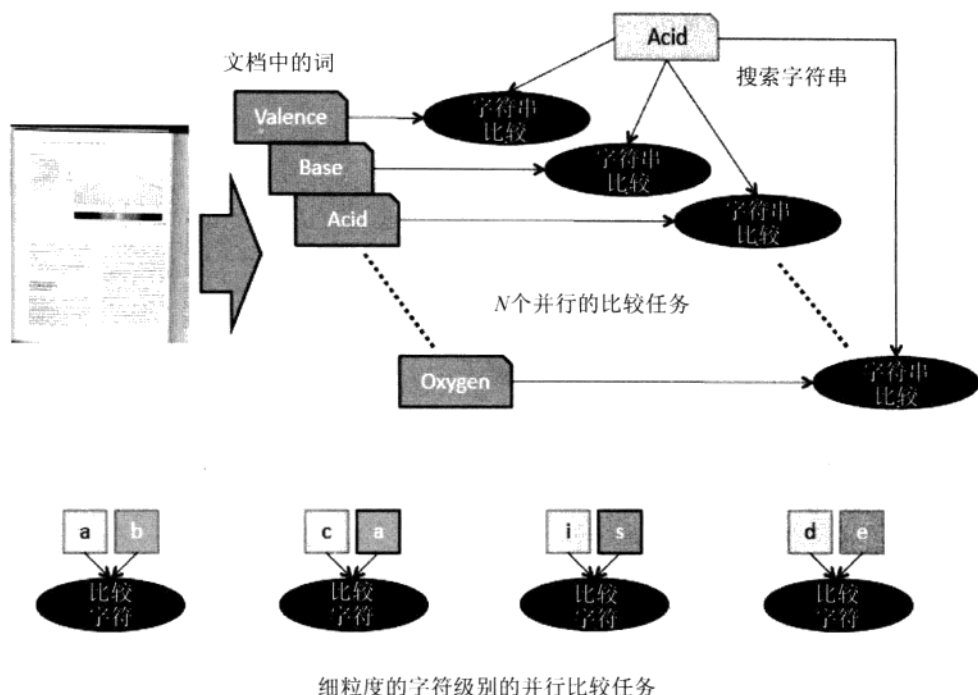


图 1.4

一个同时具有任务级并行和数据级并行的例子。我们可以以任务并行的方式计算字符串在文档中出现的次数，图中下面的部分展示了字符串比较可以分解成细粒度并行的方式来处理

一旦确定了匹配的数量，我们就需要将它们累加起来提供总的出现次数。另一方面，求和过程也可以并行。在这一步，我们引入归约的概念，用一个非常有效的方式来利用可用的并行资源求部分和。图 1.5 展示了一棵归约树，它描述了在 $\log N$ 步内完成这个求和过程。

并发编程模型和并行编程模型

在此，我们将讨论并发和并行处理模型，以便日后在将 OpenCL 处理模型应用映射到具体的并行平台上能够选定正确的模型。虽然 OpenCL 支持下面的所有模型，但是具体用哪个模型在实际使用的时候可能会受限于底层硬件的一些限制。

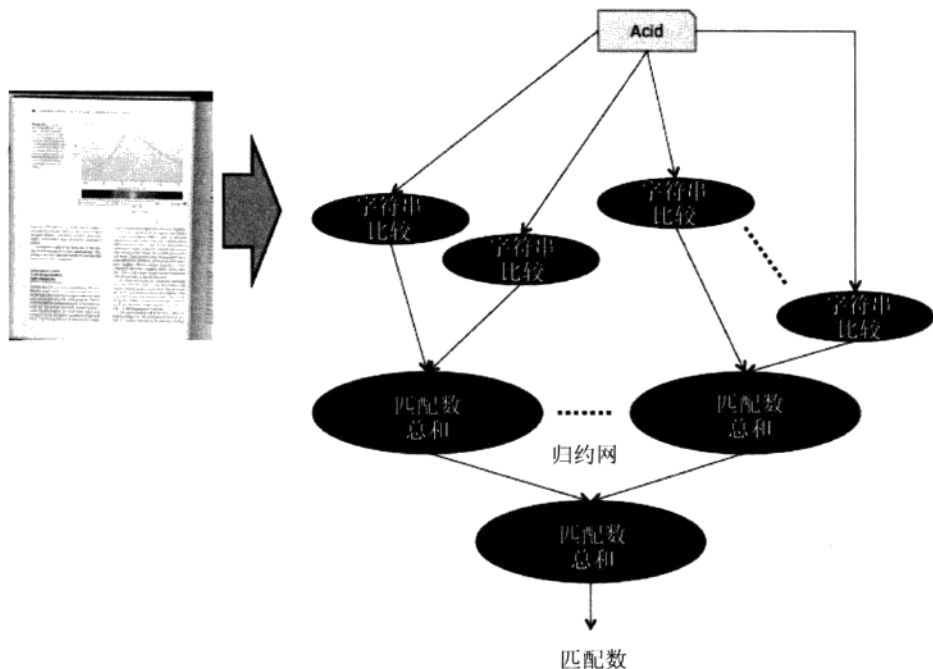


图 1.5

字符串比较完成之后，我们可以在一个组合网中求出匹配数总和

并发性(Concurrency)考虑的是同时发生的两个或两个以上的活动。在现实世界中，我们经常发现并发性——例如，用手臂抱着一个小孩横过马路，或更普遍的，一个人脑中在想一件事情，而手在做另外一件事情。

从计算机编程的角度来讨论并发性，意味着一个系统独立处理多个任务。尽管并发进行的任务也许会被同时执行，但并不要求一定要这样。例如，一个简单的画图应用程序，它或者接收用户通过鼠标和键盘的输入，或者将当前的图像输出到显示屏上。从概念上来说，接收和处理输入与更新显示屏是两个不同的操作(即任务)。这些任务可以认为是并发的，但它们不需要并行执行。实际上，它们是在单核 CPU 上运行，是不能并行执行的。在这种情况下，某个应用或者操作系统会进行任务切换，使两者都有机会在单核上并发执行。

并行性(Parallelism)指的是以提高总体性能为明确目标，并行进行两个或多个任务。例如，考虑以下几个操作：

```
step 1)  A = B + C  
step 2)  D = E + G  
step 3)  R = A + D
```

在步骤 1 和 2 中的 A 和 D 的赋值(各自的)被认为是相互独立的, 因为这两个步骤之间没有数据依赖。(即步骤 2 中右边的 E 和 G 没有出现在步骤 1 的左边, 步骤 1 中右边的 B 和 C 没有出现在步骤 2 的左边)。而且步骤 1 的左边部分(A)与步骤 2 的左边部分(D)不相同。这意味着步骤 1 和 2 可以并行执行(即同时执行)。步骤 3 依赖于步骤 1 和 2, 所以既不能和步骤 1 也不能和步骤 2 并行执行。

并行程序必须具有并发性, 但是并发程序并不一定需要具有并行性。虽然很多并发程序可以并行执行, 但是并发任务之间的相互依赖性会使一些任务并不能并行。例如, 一个交错执行仍然满足并发的定义, 但并不是并行执行的。因此, 只有一部分并发程序是并行的, 同时并发程序也是所有程序的一个子集。图 1.6 展示了这种关系。

在本小节的剩余部分, 将介绍一些常见的并发系统和并行系统编程方法, 为第 2 章介绍 OpenCL 打下基础。

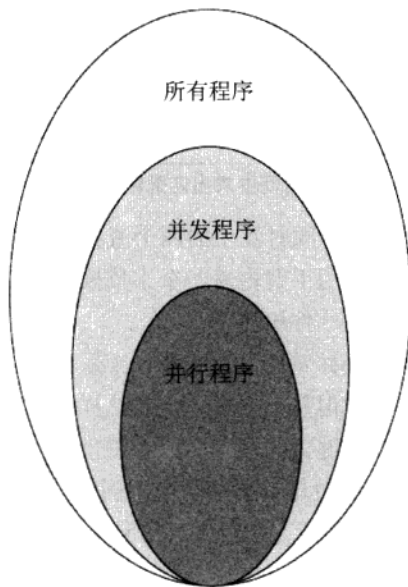


图 1.6

并行程序和并发程序都是程序的一个子集

线程和共享内存

一个正在运行中的程序可以由多个子程序组成，它们分别维护着自己独立的控制流，可以并发执行。这些子程序被定义为线程(thread)。线程之间的通信通过读写同一个地址空间的内存来完成。每一个线程拥有它们的局部内存池——即变量——但所有线程可以看到同样的全局变量集。可以用一个简单的类比来描述线程的用途，即一个拥有很多子程序的主程序。主程序由操作系统调度执行，并加载相应的系统和用户资源。主程序一开始做一些串行工作，然后利用线程通过创建一定数量可以由操作系统调度执行的任务继续并发执行。

内存一致性模型被用来管理内存读取和写入的顺序。所有进程看到的是同样的地址空间，在其他进程的帮助下直接访问这些地址。如 lock/semaphore 这样的机制常用来控制多个任务对同一段共享内存的访问。由于依赖于硬件或者是运行时系统中的一致性模型的实现，一些级别的内存一致性可能需要人工强制保证，但共享内存模型的一个主要特征是程序员不需要考虑数据移动。这使任务之间的数据通信变得很容易，并行代码的开发因而也往往可以比较简单。

支持一个完全的共享内存一致性模型，在硬件上会有很大的开销。对多处理器系统来说，硬件结构要支持这样的模型很困难。共享总线是设计上的瓶颈。由于共享内存模型的复杂性，所以当我们想要增加额外的处理器时，需要的硬件呈指数级增长。在低端市场，这已经阻碍了多核和多处理器系统的推广。共享总线和一致性协议的负担成为瓶颈，已经限制了在共享内存一致性系统中同时工作的核数。虽然更松散的共享内存系统的规模可以更大，但是大规模的共享存储系统总离不开复杂和昂贵的互连开销。

大部分多核 CPU 平台支持一种或多种形式的共享内存，OpenCL 支持在共享内存的设备上运行。

消息传递通信

消息传递通信模型使我们能详细指定计算时可能会使用内存的一系列任务之间如何进行通信。若干个任务可以位于同一个物理设备或任意数量的设备。任务通过通信发送

和接收具体的消息来实现数据交换。数据传输通常需要各个进程执行协同性操作。例如，一个发送操作必须有一个匹配的接收操作。

从编程的角度来看，消息传递往往由一个独立于硬件的通用库程序来实现发送和接收消息。程序员负责明确管理各个任务之间的通信。历史上，从上世纪 80 年代开始，出现了各种各样的消息传递库的实现。MPI 是目前最流行的消息传递中间件。这些实现在本质上差异很大，使程序员很难开发出可移植的应用程序。

不同的并行粒度

在并行计算中，粒度定义为计算与通信之比。通信和计算交错出现，并通过同步事件明确区分开。并行粒度受限于应用程序算法的内在特性。由于选择正确的并行粒度有助于程序暴露更多的可并行性，因此为了充分发挥基础平台的性能，选择正确的并行粒度非常重要。有时候，这种选择被称为“分块”(chunking)，决定分给每一个任务的数据量。选择正确的“分块”大小有助于更好地提升程序在并行硬件上的加速比。接下来，我们考虑一些与获得正确并行粒度大小相关的折中因素：

- 细粒度的并行
 - 计算强度低
 - 没有足够的任务来隐藏长时间的异步通信
 - 容易通过提供大量可管理的(即更小的)工作单元来实现负载均衡
 - 如果粒度太细，则可能使任务之间的通信和同步开销过大，这样的并行实现有可能比原始的串行算法执行速度更慢
- 粗粒度的并行
 - 计算强度高
 - 完整的应用可以作为并行的粒度
 - 难以有效实现负载均衡

给出这些折中，就能找到最佳实现的粒度吗？选择多大的并行粒度更多取决于算法本身和其运行的硬件环境。在大多数情况下，如果额外的通信和同步开销和当前计算时间之比越高，那么采取粗粒度并行对工作越有利。细粒度并行有助于减少由于负载不均衡和访存延迟所带来的开销(这个尤其在 GPU 上更明显，因为为了隐藏访存延迟，它的

细粒度线程切换是零开销的)。程序甚至可以实现指令级的细粒度并行(这种方法主要用于超长指令字(VLIW)和超标量体系结构来实现)

数据共享和同步

考虑两个应用程序在运行时不共享任何数据。只要运行时系统或操作系统获得充足的执行资源,它们可以并发运行甚至并行运行。如果第一个应用程序执行的结果是第二个应用程序后来所需要的,我们就得在系统中引入某种形式的同步,使线程不能跨越同步点并行执行。

编写并发软件时,数据共享和同步所扮演的角色至关重要。并发程序中需要数据共享的情况包括如下:

- 某个任务的输出依赖于另一个任务产生的结果,例如生产者/消费者或者管道执行模型;
- 需要汇集中间计算结果(例如,归约计算中某部分,图 1.4 显示的单词搜索实例)。

理想的情况下,我们仅尝试并行应用程序中没有数据依赖的部分,但这并不总是可行的。显示同步原语,如 **barrier** (栅栏)或锁在必要时可用于同步。我们在这里只提出了这个问题,后面的章节中将对此再次讨论,包括支持主机和设备端程序之间通信或任务之间所需的同步。

本书结构

本书其余各章简介如下。

第 2 章主要介绍 OpenCL,主题包括:主要的概念,如 **kernel**,平台和设备;四个不同的抽象模型;编写简单的 OpenCL **kernel**。了解这些不同的模型对充分掌握 OpenCL 编程模型是至关重要的。

第 3 章介绍一些面向 OpenCL 的架构,包括 X86 CPU, GPU 和 APU。讨论不同架构风格,包括单指令多数据和 VLIW。本章涵盖的概念包括多核和面向吞吐量的系统,以及异构架构中的高级概念。

第 4 章介绍基本的矩阵相乘、图像旋转和卷积实现,以帮助读者通过实例来了解 OpenCL。

第 5 章讨论 OpenCL 编程模型的并发性和执行。在本章中,我们讨论 kernel、work-item 以及 OpenCL 的执行和内存层次结构。我们还要展示如何在 OpenCL 中将执行命令添入队列和同步操作,以便于读者了解如何编写 OpenGL 程序使之能与内存正确交互。

第 6 章演示如何将 OpenCL 映射到硬件架构上。针对这方面的学习,我们采用的平台是由 AMD Phenom II CPU 和 AMD Radeon HD6970 GPU 组成的系统。本章描述如何将 OpenCL 编程模型映射到串行的 CPU 架构和面向向量/吞吐量的 GPU 架构,提供一些如何针对具体架构特点进行优化的方法。

第 7 章呈现的是一个对卷积算法进行加速的案例分析。考虑的问题牵涉到内存空间的利用和效率,以及 work-item 的调度, wavefront 利用率和整体效率。这些技术为开发高性能的 OpenCL 代码提供了必要的基础。

第 8 章呈现的是一个针对视频处理的案例分析,利用 OpenCL 构建高性能图像处理效果,可应用于视频流。

第 9 章呈现的是另一个案例分析,研究如何优化直方图的性能。具体而言,它着重于如何精心设计 workgroup 的大小和内存访问模式(这在内存受限的应用中能带来巨大的性能差异,如直方图)。

第 10 章讨论如何利用 CPU - GPU 异构环境,使用的应用程序实例是混合粒子模拟(本书封面所示),横跨 CPU 和 GPU 的任务分配取决于系统中粒子的粒度。

第 11 章借助实例来展示如何使用 OpenCL 中的设备拆分(device fission) 和双精度扩展。

第 12 章向读者介绍 OpenCL 程序的调试和分析。正确的调试工具可以节省大量的时间,使编程人员关注于学习具体的计算机语言和如何解决手头的问题。

第 13 章综述 WebCL 及其性能权衡。虽然目前 WebCL 还不是一个产品,但是想象一下,如果 OpenCL 推动 Web 发展,足以使 WebCL 变为产品成为可能。本章介绍 OpenCL 绑定的 JavaScript 实例,要讨论并实现一个 Firefox 插件,使 Web 应用充分利用现代 CPU 和 GPU 强大的并行计算能力。

参考文献

Almasi, G. S., & Gottlieb, A. (1989). *Highly Parallel Computing*. Redwood City, CA: Benjamin Cummings.

扩展阅读和相关网站

Chapman, B., Jost, G., van der Pas, R., & Kuck, D. J. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press.

Duffy, J. (2008). *Concurrent Programming on Windows*. Upper Saddle River, NJ: Addison-Wesley.

Gropp, W., Lusk, E., & Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press Scientific and Engineering Computation Series. Cambridge, MA: MIT Press.

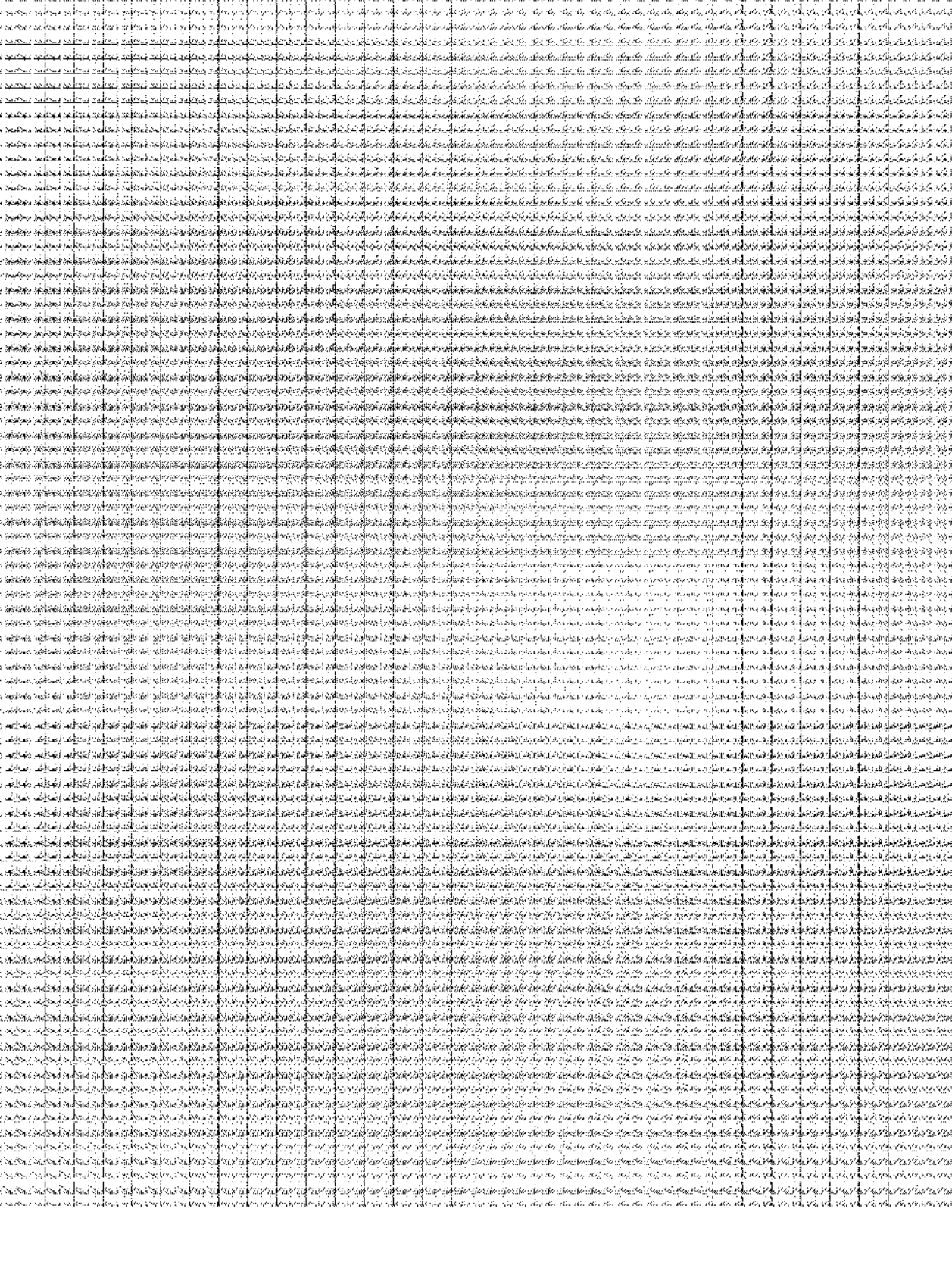
Herlihy, M., & Shavit, N. (2008). *The Art of Multiprocessor Programming*. Burlington, MA: Morgan Kaufmann.

Khronos Group. *OpenCL*. www.khronos.org/opencl.

Mattson, T. G., Sanders, B. A., & Massingill, B. L. (2004). *Patterns for Parallel Programming*. Upper Saddle River, NJ: Addison-Wesley.

NVIDIA. CUDA Zone. http://www.nvidia.com/object/cuda_home_new.html.

AMD. OpenCL Zone. <http://developer.amd.com/openclzone>.



引言

本章介绍 OpenCL，其编程要素允许我们快捷地编写并发运行的应用程序。熟悉 C 和 C++ 的程序员应该不难理解 OpenCL 的语法。我们首先回顾一下 OpenCL 标准。

OpenCL 标准

开放式编程标准的设计者面临着一个巨大的挑战：制定一套能够满足一系列竞争性需求的公共编程标准。负责管理 OpenCL 标准的 Khronos 联盟已经做了卓有成效的工作来满足这些要求。该联盟开发了一套能够适用于大量各种不同硬件架构同时又足以获得高性能的通用 API。只要使用 OpenCL 语言并正确遵循编程规范，为特定厂商平台开发的任何程序都能够其他厂商的硬件上运行。使用 OpenCL 编程模型能够开发出可移植的、独立于厂商和设备的且能够在众多不同硬件平台上加速运行的程序。

OpenCL API 是按照 C API 定义的，由 C 和 C++ 封装而成，现在针对很多编程语言的第三方绑定，包括 Java，Python 以及 .NET。使用 OpenCL C 语言编写的代码可在支持 OpenCL 的设备上运行，这些设备一般不同于主机 CPU。OpenCL C 是 C99 语言的子集，并适当地扩展了在众多异构设备上执行数据并行代码的能力。

OpenCL 规范

OpenCL 规范由四个模型组成，总结如下。

1. 平台模型：描述了协同执行的单个处理器(宿主机)及一个或多个能执行 OpenCL 代码的处理器(设备)。它定义了一个抽象的硬件模型，供编程人员用于编写能够在这些设备上执行的 OpenCL C 函数(称作 kernel)。
2. 执行模型：定义了在主机上如何配置 OpenCL 环境以及如何在设备上执行 kernel。这包括在主机端建立 OpenCL 上下文，提供主机-设备之间的交互机制，定义一个并发模型供在设备上执行 kernel 所用。
3. 内存模型：定义被 kernel 所用的抽象内存层次(memory hierarchy)，无需考虑实际的底层内存架构。尽管内存模型十分接近当前的 GPU 内存层次，但同样也适用于其他硬件加速器。
4. 编程模型：定义了如何将并发模型映射到物理硬件上。

在通常情况下，我们可能看到使用 GPU 设备作为加速器的 X86 CPU 主机上执行的一个 OpenCL 实例。OpenCL 平台模型定义了主机和设备之间的这种关系。以指定的并行度来实例化它，这就是执行模型。kernel 内部的数据由编程人员分配到抽象内存层级中的指定位置。OpenCL 运行时(runtime)和驱动程序将这些抽象内存空间映射到物理层。最后，借助于编程模型，必定是新建执行 kernel 的硬件线程上下文并映射到实际的 GPU 硬件单元。在整个本章中，我们将对此进行更详细的讨论。

本章首先介绍如何编写 OpenCL kernel，介绍它们所使用的并行执行模型。紧接着，我们借助于向量相加实例来描述和演示 OpenCL 主机端 API。向量相加实例的完整代码清单在本章结尾处给出。

kernel 和 OpenCL 执行模型

kernel 是 OpenCL 程序在设备上实际运行的那部分代码。OpenCL API 允许应用程序新建上下文来管理 OpenCL 命令的执行，包括描述主机端和 OpenCL 内存结构之间的数据传输以及 kernel 代码的执行(即对这些数据进行处理以完成特定的任务)。

像许多 CPU 并发模型一样，OpenCL kernel 的语法类似于标准的 C 函数，关键区别在于它额外有一个关键字集和 OpenCL kernel 实现的执行模型。使用 OS 线程 API 或 OpenMP 在 CPU 上开发并发程序时，编程人员需要考虑可用的计算资源(例如 CPU 核数)

以及线程数超过资源限制时所带来的新建和切换开销。使用 OpenCL, 编程人员的目标是尽可能细粒度地表示程序中的并行性。OpenCL 接口的通用性和底层的 kernel 语言使应用程序能够高效地映射到大多数硬件。以下将讨论向量相加函数的三个实现版本: 串行 C 版本、多线程 C 版本和 OpenCL 版本。

向量相加的串行 C 实现代码执行的是一个循环, 其迭代次数与需要计算的数组元素一样多。每次循环迭代都将输入数组中循环下标相应的元素相加, 并将结果保存到输出数组中:

```
// Perform an element-wise addition of A and B and store in C.
// There are N elements per array.
void vecadd(int *C, int *A, int *B, int N) {
    for(int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
    }
}
```

对于一个简单的多核设备, 我们可以使用低层次的粗粒度线程 API(如 Win32 或 POSIX)或使用数据并行模型(如 OpenMP)。编写一个具有相同功能的粗粒度多线程版本需要在线程之间划分任务(即循环迭代)。因为可能有大量的循环迭代, 而每次迭代的工作量很小, 所以我们需要将循环迭代按块划分成更大的粒度(称为“循环切分”(strip mining), Cooper and Torczon [2011])。多线程版本的代码可能像下面这样:

```
// Perform an element-wise addition of A and B and store in C.
// There are N elements per array and NP CPU cores.
void vecadd(int *C, int *A, int *B, int N, int NP, int tid) {
    int ept = N/NP; // elements per thread
    for(int i = tid*ept; i < (tid+1)*ept; i++) {
        C[i] = A[i] + B[i];
    }
}
```

相比 Win32 和 POSIX 线程 API, OpenCL 更接近于 OpenMP, 支持数据并行执行, 但不足之处在于不太容易控制。OpenCL C 并发执行单位是一个 work-item^①。如同前两个例子, 每个 work-item 执行 kernel 函数体。不用手动切分循环, 我们往往将单层循环迭代映射到一个 work-item, 使 OpenCL 运行时产生的 work-item 数目和输入输出元素数目保持一致, 并允许在运行时将这些 work-item 映射到底层硬件, 因此 CPU 或 GPU 核,

① 译者注: 更通俗地讲, 在计算单元上运行的 kernel 程序的每个实例都称为一个 work-item, 多个 work-item 则构成一个 workgroup。

以它们认为适当的任何方式执行程序。从概念上讲,这是非常类似 `map` 操作函数^①固有的并行性或针对循环的数据并行模型,如 `OpenMP`。当 `OpenCL` 设备开始执行 `kernel` 时,会提供内置(`intrinsic`)函数让 `work-item` 表明它自身。在以下代码中,编程人员通过调用 `get_global_id(0)` 来获取当前 `work-item` 所在的位置,在这个简单的例子中用来重新获取循环计数器。

```
// Perform an element-wise addition of A and B and store in C
// N work-items will be created to execute this kernel.
__kernel
void vecadd(__global int *C, __global int *A, __global int *B) {
    int tid = get_global_id(0); // OpenCL intrinsic function
    C[tid] = A[tid] + B[tid];
}
```

由于 `OpenCL` 能够以细粒度 `work-item` 的方式来描述执行,并能够将大量的 `work-item` 分发到硬件上支持细粒度线程的架构,因此很容易引起大家对 `OpenCL` 程序可扩展能力的担忧。`OpenCL` 的层次并发模型确保了在支持大量 `work-item` 执行的同时也能取得良好的可扩展性。在 `kernel` 执行期间,编程人员通过指定一个 n 维索引空间(`NDRange`)来声明需要新建多少个 `work-item`。一个 `NDRange` 是 `work-item` 的 1 维、2 维或 3 维索引空间,往往被映射到输入或输出数据的索引空间上。`NDRange` 的维度被定义为一个数据类型为 `size_t`、长度为 N 的数组,其中 N 代表用来描述所新建的 `work-item` 的维度。

在向量相加实例中,我们的数据是一维的,并假设有 1024 个元素,大小可以被定义为一个 1 维,2 维或者 3 维的向量。用于为 1024 个元素指定一个 `NDRange` 的主机端代码如下:

```
size_t indexSpaceSize[3] = {1024, 1, 1};
```

可扩展性的获得来自于将 `NDRange` 的 `work-item` 划分成更小的部分,等同于改变 `workgroup` 的大小(图 2.1)。 N 维度的索引空间需要指定同样具有 N 维度的 `workgroup`。因此,一个 3 维的索引空间需要有 3 维的 `workgroup`。

一个 `workgroup` 中各个 `work-item` 之间具有特定的关联:它们可以执行 `barrier` 操作进行同步,可以访问同一段共享的内存地址空间。由于 `workgroup` 的大小是固定的,并且通信量不大,所以并不影响大规模并发调度的可扩展性。

对于向量相加实例, `workgroup` 的大小可能设置如下:

```
size_t workGroupSize[3] = {64, 1, 1};
```

如果每个数组的 `work-item` 总数为 1024,结果便是新建 16 个 `workgroup`($1024 \text{ 个 } \text{work-item} / (\text{每个 } \text{workgroup} \text{ 中 } 64 \text{ 个 } \text{work-item}) = 16 \text{ 个 } \text{workgroup}$)。注意, `OpenCL` 要求

^① 译者注:譬如 `MapReduce`,更多相关介绍可参考 Hadoop 项目负责人 Tom White 所著的《Hadoop 权威指南》。

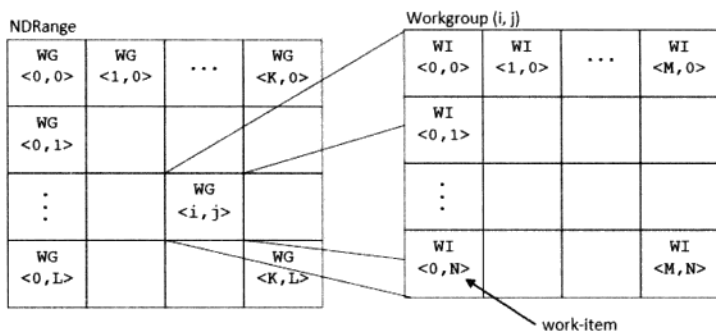


图 2.1

work-item 以 NDRange 的形式新建并被划分为多个 workgroup

索引空间中每个维度的长度能被 workgroup 的大小整除。考虑到硬件效率, workgroup 的规模通常固定为一个合适的大小, 我们通过对每个维度索引空间的长度取整来满足这种不可分要求。在 kernel 代码中, 可以在每维中指定额外的 work-item 使函数立即返回而不输出任何数据。

该向量相加程序中的 work-item 是独立运行的(甚至在同一个 workgroup 内), OpenCL 允许编程人员将传递的参数设置为 NULL 来忽略本地 workgroup 的大小而靠实现来自动生成。

平台和设备

OpenCL 平台模型定义主机和设备的角色, 并为设备提供一个抽象的硬件模型。

主机-设备交互

在平台模型中, 一个主机协调在一个或多个 OpenCL 设备上的程序执行。平台可以被看作是厂商特定的 OpenCL API 实现。因此, 平台上的设备只局限于厂商知晓如何进行交互的设备。例如, 如果选择 A 公司的平台, 它就无法与 B 公司的 GPU 进行通信。

平台模型还带来了一个抽象的设备架构, 编程人员在编写 OpenCL 代码时以它为目标。厂商将这个抽象架构映射到具体的硬件。考虑到可扩展性, 平台模型将一个设备定义为一组计算单元(compute unit), 每个计算单元功能独立。计算单元又进一步划分为处理部件(processing element)。图 2.2 说明了这个层次化的平台模型。

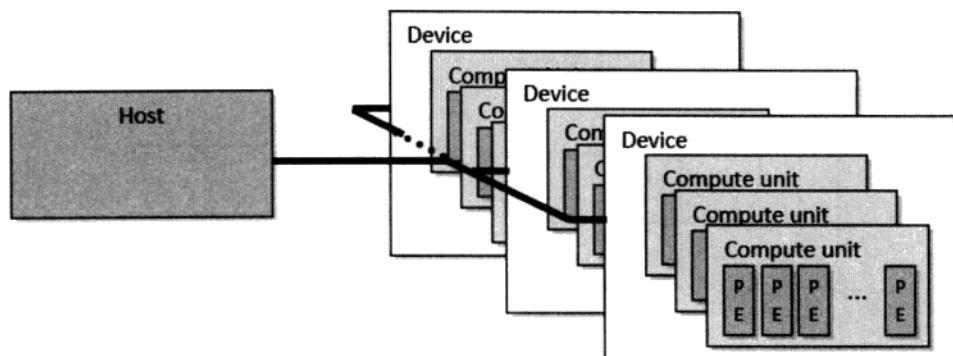


图 2.2

平台模型为设备定义一个抽象的架构

平台设备模型相当接近于某些 GPU 的硬件模型。例如, AMD 的 Radeon 6970 图形卡包括 24 个 SIMD 核(计算单元), 每个 SIMD 核包含 16 个通路(处理部件)。由于在 6970 上每个 SIMD 通路执行一条四路超长指令字(VLIW)指令, 该 OpenCL 设备使我们能够一次调度高达 1536 指令去执行。

API 函数 `clGetPlatformIDs()` 用来获取指定系统上可用的计算平台:

```
cl_int
clGetPlatformIDs(cl_uint num_entries,
                 cl_platform_id *platforms,
                 cl_uint *num_platforms)
```

`clGetPlatformIDs()` 通常由应用程序调用两次。首次调用时, 将 `unsigned int` 指针和 `NULL` 分别传递给 `num_platforms` 和 `platforms` 参数, 编程人员可以分配空间来容纳该平台的信息。第二次调用时, 将 `cl_platform_id` 指针传递给已为 `num_entries` 平台分配足够空间的具体实现。平台被发现之后, 调用 `clGetPlatformInfo()` 来确定具体使用哪一个已定义的实现(厂商)平台。本章结尾处所提供的完整源代码清单中, 步骤 1 演示的便是这一过程。

函数 `clGetDeviceIDs()` 的调用过程类似于 `clGetPlatformIDs()`。它也需要 3 步, 区别在于额外还有平台和设备类型参数。`device_type` 参数将设备限制为仅有 GPU (`CL_DEVICE_TYPE_GPU`)、仅 CPU (`CL_DEVICE_TYPE_CPU`)、所有设备 (`CL_DEVICE_TYPE_ALL`) 和其他选项。至于平台, 通过调 `clGetDeviceInfo()` 重新获取设

备名称、类型、设备厂商相关信息。在本章结尾处提供的完整源代码清单中，步骤 2 阐述的便是如何发现可用设备。

```
cl_int
clGetDeviceIDs(cl_platform_id platform,
               cl_device_type device_type,
               cl_uint num_entries,
               cl_device_id *devices,
               cl_uint *num_devices)
```

AMD APP SDK 中的 CLInfo 程序使用 clGetPlatformInfo()和 clGetDeviceInfo()命令来打印详细的系统中支持 OpenCL 的平台和设备详细信息。硬件详细信息(如内存大小和总线宽度)可以通过这两个命令获得。cLInfo 程序的输出片段如下所示：

```
$ ./CLInfo
Number of platforms: 1
Platform Profile: FULL_PROFILE
Platform Version: OpenCL 1.1 AMD-APP-SDK-v2.4
Platform Name: AMD Accelerated Parallel Processing
Platform Vendor: Advanced Micro Devices, Inc.
Number of devices: 2
Device Type: CL_DEVICE_TYPE_GPU
Name: Cypress
Max compute units: 20
Address bits: 32
Max memory allocation: 268435456
Global memory size: 1073741824
Constant buffer size: 65536
Local memory type: Scratchpad
Local memory size: 32768
Device endianness: Little

Device Type: CL_DEVICE_TYPE_CPU
Max compute units: 16
Name: AMD Phenom(tm) II X4 945 Processor
...
```

执行环境

主机端请求在设备上执行 **kernel** 之前，必须先配置好用于向设备传递命令和数据的上下文。

上下文

OpenCL 中, 上下文(Context)是一个抽象容器并存在于主机端。它能协调主机-设备之间的交互机制, 管理设备上可用的内存对象, 跟踪针对每个设备新建的 kernel 和程序。

新建上下文的 API 函数是 `clCreateContext()`。参数 `properties` 用来限定上文的范围。它可提供能够指定具体的硬件平台, 开启 OpenGL/OpenCL 的互操作性或者开启其他着眼于未来的参数。将上下文局限于某个特定的平台使编程人员为多个平台提供上下文, 并且充分利用由多个厂商资源组成的系统。随后, 编程人员必须提供希望与上下文关联的设备数量和设备 ID。OpenCL 允许在新建上下文时提供用户回调函数, 这些函数用于报告可能在其整个生命周期中产生额外的错误信息。本章结尾所提供的完整源代码清单中, 步骤 3 演示的是如何新建上下文。

```
cl_context
clCreateContext (const cl_context_properties *properties,
                cl_uint num_devices,
                const cl_device_id *devices,
                void (CL_CALLBACK *pfn_notify)(
                    const char *errinfo,
                    const void *private_info,
                    size_t cb,
                    void *user_data),
                void *user_data,
                cl_int *errcode_ret)
```

OpenCL 规范还提供了不需要新建整个设备列表的 API。 `clCreateContextFromType()` 允许编程人员自动新建所有指定设备类型(例如 CPU、GPU 以及所有设备)的上下文。新建完上下文之后, `clGetContextInfo()` 函数可用来查询现有设备数量和设备结构的信息。

在 OpenCL 中, 发现设备平台和新建上下文的过程都是冗长乏味的。尽管如此, 完成这些步骤的代码只需要编写一次, 几乎就可以重复用于任何一个项目中。

命令队列

通过提交命令到命令队列(command queue)来开始与设备进行通信。命令队列是主机端用于向设备端发送请求的行为机制。一旦主机端指定运行 kernel 的设备并且上下文已新建, 那么每个设备必须新建一个命令队列(即每个命令队列只关联一个设备)。只要主机需

要对设备进行操作,就会把命令提交到正确的命令队列。API 函数 `clCreateCommandQueue()` 用来新建命令队列并且将其关联到某个设备。

```
cl_command_queue
clCreateCommandQueue(
    cl_context context,
    cl_device_id device,
    cl_command_queue_properties properties,
    cl_int* errcode_ret)
```

`clCreateCommandQueue` 属性参数是一个位域,用于开启程序剖析(profiling)命令 (`CL_QUEUE_PROFILING_ENABLE`)和/或允许乱序执行命令(`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`)。在第 12 章中,我们将讨论程序剖析。在顺序执行命令队列中(默认),命令按照其接收的顺序压入命令队列。乱序队列允许 OpenCL 在实现时重排命令以便高效地执行。如果使用乱序队列,用户必须指定依赖关系以确保正确的执行顺序。在本章结尾处提供的完整源代码清单中,步骤 4 演示的是如何新建命令队列。

任何一个指定主机与设备间交互的 API,总是以 `clEnqueue` 字样开头并且需要一个命令队列作为参数。例如 `clEnqueueReadBuffer()` 命令,请求设备将数据发送到主机端,而 `clEnqueueNDRangeKernel()` 请求在设备上执行 kernel。这些 API 调用稍后在本章中进行讨论。

事件

任何操作被作为一个命令入队到一个命令队列中,即任何以 `clEnqueue` 字样开头的 API 调用——都会产生一个事件(event)。事件在 OpenCL 中有两大作用:

1. 表示依赖
2. 提供程序剖析机制

除了产生事件对象之外,以 `clEnqueue` 字样开头的 API 调用也使用“等待列表”(wait list)作为参数。OpenCL 生成一个 API 调用事件,并将其作为多次连续调用的参数来表示依赖关系。每次 `clEnqueue` 调用都将阻塞直到其等待列表中的所有事件完成。第 5 章将提供几个使用事件来表示依赖关系的例子。

通过使用关联计时器和命令队列,OpenCL 事件也可以用来剖析应用程序。第 12 章将介绍如何使用 OpenCL 事件进行程序剖析。

内存对象

OpenCL 应用程序经常要处理大型数组或多维矩阵。在能够开始执行之前,需要这

些数据实际存在于设备之上。为了将数据传输到设备上，首先必须将其封装成内存对象。OpenCL 定义了两种不同的内存对象：buffer 和 image。buffer 类似于 C 语言中的数组，由 malloc() 新建，buffer 中的数据元素在内存中是连续存储的。另一方面，image 被设计成不透明的对象，以利于进行数据填充和其他可能改善设备性能的优化。

任何时候，只要是新建的内存对象，它都只在一个上下文中有有效。OpenCL 运行时根据数据依赖关系的需要来管理特定设备的数据传输。

buffer 对象

从概念上讲，它也许有助于将一个内存对象抽象为设备上的一个有效的指针。这类似于 C 中的 malloc 函数调用或者 C++ 中的 new 操作符。API 函数 clCreateBuffer() 分配缓冲区，并返回一个内存对象：

```
cl_mem clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *errcode_ret)
```

新建一个 buffer，需要提供 buffer 的大小和与其关联的上下文，它对所有与该上下文相关联的设备是可见的。调用时可以选择性地提供标志，用于指定数据是只读、只写或读写，其他标志用于提供新建和初始化 buffer 时的额外选项。一个简单的选项就是初始化 buffer 数据时提供主机端指针。在本章结尾处提供的完整源代码清单中，步骤 5 演示的是如何新建两个输入 buffer 和一个输出 buffer。

将主机端内存中的数据传入到 OpenCL buffer 或者反之，对应的命令分别为 clEnqueueWriteBuffer() 和 clEnqueueReadBuffer()。如果在一个独立的加速器设备(如 GPU)上执行的 kernel 依赖于某个 buffer，那么该 buffer 必须要传输到设备上。这个 buffer 链接的是上下文，而非设备，因此，决定数据移动准确时间的是 OpenCL 运行时。

读写 buffer 的 API 调用非常类似，clEnqueueWriteBuffer() 的函数签名如下：

```
cl_int
clEnqueueWriteBuffer(
    cl_command_queue command_queue,
    cl_mem buffer,
    cl_bool blocking_write,
    size_t offset,
    size_t cb,
    const void *ptr,
```

```
cl_uint num_events_in_wait_list,
const cl_event *event_wait_list,
cl_event *event)
```

与其他入队操作类似，读或写 **buffer** 需要一个命令队列来管理指令的执行和调度。此外，入队函数需要 **buffer**、数据传输的字节数以及在 **buffer** 内偏移量。如果数据传输(从主机端到 OpenCL **buffer**)完成之后函数才返回，**blocking_write** 选项就应该设置为 **CL_TRUE**，也就是说，它会阻塞直到操作完成。**blocking_write** 设置为 **CL_FALSE** 时允许 **clEnqueueWriteBuffer()** 函数返回先于写操作完成。在本章结尾处提供的完整源代码清单中，步骤 6 使用入队命令将输入数据写入设备的 **buffer**，步骤 12 将数据写回到主机端。

image 对象

image 是考虑到具体设备优化而对实际数据存储进行抽象的 OpenCL 内存对象。不要求所有 OpenCL 设备都支持它们。应用程序使用 **clGetDeviceInfo()** 可以查看该设备是否支持该特性。否则，行为未定义。与 **buffer** 不一样，**image** 不能像数组一样直接进行引用。此外，相邻的元素不能保证被放在连续的内存中。使用 **image** 的目的是允许硬件利用空间局部性来进行硬件加速。**image** 架构设计和权衡将在第 5 章中进行详细的讨论。

在 OpenCL 标准中，**image** 依赖于底层的硬件设备，例如 GPU 其他。**image** 中元素使用格式描述符(**cl_image_format**)来表示。格式描述符指定了图像元素基于通道(channel)的形式如何存储在内存中。通道顺序描述了组成一个图像点需要的元素个数(传统使用 **RGBA** 像素，最多需要 4 个元素)，通道类型描述了每个元素的大小，不同格式(例如整型或者浮点)中的元素大小从 1 个字节到 4 个字节之间变化。

利用 **clCreateImage2D()** 或者 **clCreateImage3D()** 命令新建 OpenCL **image**。与新建 **buffer** 相比，新建 **image** 对象需要额外的参数。首先，必须指定图像的高度和宽度(对于 3 维图像需要指定深度)。此外，**image pitch**^①要在数据初始化的时候提供。

① **pitch** 指一幅图像数据中两行之间差距所占的字节数。

```

cl_mem
clCreateImage2D(
    cl_context context,
    cl_mem_flags flags,
    const cl_image_format *image_format,
    size_t image_width,
    size_t image_height,
    size_t image_row_pitch,
    void *host_ptr,
    cl_int *errcode_ret)

```

读/写 image 时也需要提供额外的参数。读或写操作需要的额外参数包括：用于定义图像传输起始位置的 `origin[3]` 数组(类似 `buffer` 偏移量)和定义数据传输范围的 `region[3]` 数组。

在 kernel 中使用指定数据类型的内建函数访问 image。例如，函数 `read_imagef()` 用来读取浮点而 `read_imageui()` 用来读取整数。当数据是从 image 中读取时，需要 `sampler` 对象。`sampler` 指定了 image 越界访问时如何处理，在坐标规格化前提下是否使用插值。将数据写入 image 某个位置时需要手工转换成对应的数据格式(即存储为合适的通道和大小)。第 4 章将提供使用 image 的 OpenCL 程序实例。

flush 命令和 finish 命令

在命令队列中是两种不同类型的 `barrier` 操作。`clFinish()` 函数阻塞直到命令队列中的所有命令完成，它的功能类似于 `barrier`。`clFlush()` 阻塞直到命令队列中的所有命令被移出队列，这意味这些命令已经准备就绪但无法保证执行完毕。

```

cl_int clFlush(cl_command_queue command_queue);
cl_int clFinish(cl_command_queue command_queue);

```

新建一个 OpenCL 程序对象

OpenCL C 代码(编写的代码在 OpenCL 设备上执行)称为程序(program)，是称为 kernel 的函数的集合，kernel 是被调度安排到设备上运行的执行单位。

OpenCL 程序在运行时通过调用一系列 API 进行编译，编译系统针对具体的设备进行优化。OpenCL 应用程序能同时移植到 AMD、NVIDIA 以及 Intel 的设备上，则不需要对这些平台分别进行预编译。OpenCL 软件仅链接到一个公共运行层(称 ICD)，所有平台

特定的 SDK 通过一个动态库接口委托给某个厂商的运行时。

新建 kernel 的步骤如下。

1. OpenCL 源代码以字符串形式存储。如果代码保存在文件中，则必须读到内存并存储为字符串数组。
2. 源代码通过调用 `clCreateProgramWithSource()` 转化为一个 `cl_program` 对象。
3. 然后，使用 `clBuildProgram()` 在多个支持 OpenCL 的设备上编译程序对象，如果出现编译错误，则报告错误信息。

程序准确的二进制表示因具体设备而异。在 AMD 运行时中，主要有两类设备：x86 CPU 和 GPU。对于 x86 CPU，`clBuildProgram()` 会产生 x86 指令并且能够直接在该设备上执行。对于 GPU，首先新建 AMD GPU 中间语言(intermediate language, IL)以及表示单个 work-item 的高级中间语言，随之针对该架构进行实时编译，产生众所周知的 ISA(即具体指令集架构上代码)。NVIDIA 使用类似的方法，调用中间代码表示 PTX。使用 IL 的优势在于允许 GPU ISA 能够适应硬件架构的更新步伐。

构建过程的另外一个特点是有能力生成最终的二进制格式和多种中间表示并且序列化它们(例如，将它们写回到磁盘中)。和大多数对象一样，OpenCL 提供返回程序对象信息的 `clGetProgramInfo()` 函数。带有 `CL_PROGRAM_BINARIES` 标志的该函数能够返回 `clBuildProgram()` 针对厂商特定设备所产生的二进制代码。

除了 `clCreateProgramWithSource()` 之外，OpenCL 提供 `clCreateProgramWithBinary()` 匹配设备列表的二进制列表。这个二进制列表是以前使用 `clGetProgramInfo()` 新建的。

OpenCL 的 kernel

最终阶段是通过从 `cl_program` 中抽取出 kernel，获取在设备上执行的 `cl_kernel` 对象。从程序中抽取 kernel 类似于从动态库中导出函数。程序导出的 kernel 名称要求在程序对象中编译。kernel 名称和程序对象传递给 `clCreateKernel()`，如果程序对象有效并且指定的 kernel 已被发现，则会有 kernel 对象返回。

与常规 C 程序中函数调用不同，kernel 在被执行前还需要一些步骤。我们不能简单地提供一些参数就可以调用 kernel。

kernel 的执行需要通过一个入队函数来进行分发。由于 C 语言的语法和 kernel 参数是持久的(因此我们不必重复设置它们来构建调度的参数列表),我们必须分别使用函数 `clSetKernelArg()` 指定每个 kernel 参数,该函数接受一个 kernel 对象,参数索引、大小和指针。当 kernel 执行时,这些信息作为参数传输给设备。然后,运行时将 kernel 参数列表中的类型信息拆箱成(类似于类型转换)适当的类型。设置 kernel 参数的过程,将在本章结尾处提供的源代码清单中步骤 9 进行详细描述。

所需要的内存对象全部都传输到设备且 kernel 参数设置之后,通过调用 `clEnqueueNDRangeKernel()` 来开始执行 kernel:

```
cl_int
clEnqueueNDRangeKernel(
    cl_command_queue command_queue,
    cl_kernel kernel,
    cl_uint work_dim,
    const size_t *global_work_offset,
    const size_t *global_work_size,
    const size_t *local_work_size,
    cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,
    cl_event *event)
```

看看这个函数的签名。必须先指定一个命令队列,以便发现目标设备。

类似地, kernel 对象识别出将要执行的代码。有四个字段与 work-item 新建相关, `work_dim` 参数指定新建 work-item 的维度(1,2,3)。`global_work_size` 参数指定 NDRange 中每维 work-item 的数量, `local_work_size` 指定 workgroup 中每维 work-item 的数量。可以使用参数 `global_work_offset` 为 work-item 提供全局的 ID,该参数不从 0 开始。和所有 `clEnqueue` 命令一样,需要提供 `event_wait_list`。非 NULL 值时,运行时将保证所有相应的事件在 kernel 开始执行之前完成。`clEnqueueNDRangeKernel()` 调用是异步的:当命令进入命令队列中后立即返回,甚至可能在 kernel 已经开始执行之前。`clWaitForEvents()` 或 `clFinish()` 可以阻塞直到 kernel 完成。在本章结尾处提供的源代码清单中,步骤 10 显示了 work-item 的配置,向量相加 kernel 入队的命令则出现在步骤 11 中。

至此,我们已经介绍了运行一个完整的 OpenCL 程序所需要的所有主机端 API 命令。

内存模型

在一般情况下，计算平台之间的内存子系统差异巨大。例如，所有现代 CPU 都支持自动缓存，不过许多 GPU 并不支持。为了支持代码的可移植性，OpenCL 采用的方法是定义一个抽象的内存模型，编程人员可以根据该模型编写代码，厂商可以映射到他们自己实际的内存硬件。这里讨论的是 OpenCL 定义的内存空间，如图 2.3 所示。

这些内存空间与 OpenCL 程序密切相关。与每个内存空间关联的关键字能够用于新建变量和新建变量指向的数据。

全局内存对设备上所有计算单位都是可见的(类似于 CPU 上的主存)。每次从主机端到设备端传输的数据都将驻留在全局内存中，从设备端传输到主机端的任何数据，也是如此。在指针声明中添加关键字 `__global` 用于指定该指针所引用的数据驻留于全局内存。例如，在本章结尾处 OpenCL 代码中的 `__global float* A`，指针 A 所指向的数据驻留在全局内存内(A 实际上驻留在私有内存中)。

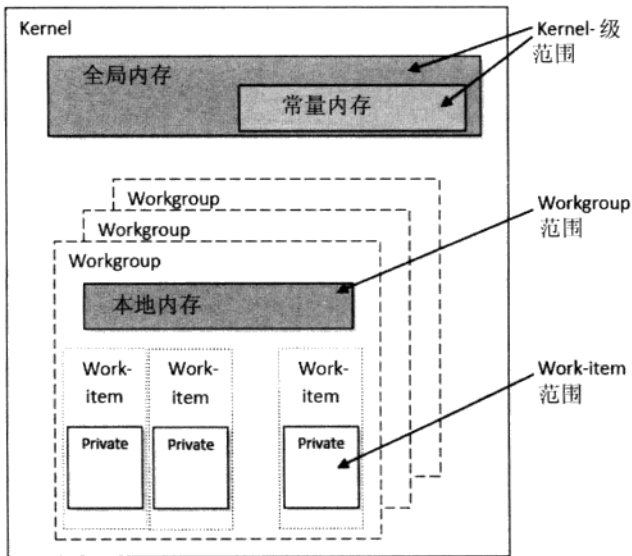


图 2.3

OpenCL 所定义的抽象内存模型

常量内存不是专门为每种只读类型数据而设计的，而是便于所有 work-item 同时访问数据中的每个元素。常量变量(例如值始终为 p 的数据变量)也属于这一类。常量内存为全局内存的一部分，因此，传输到全局内存中的内存对象可以指定为常量。使用 `__constant` 关键字将数据映射到常量内存。

本地内存是高速暂存存储器(scratchpad memory)，其地址空间对于计算设备是独一无二的，通常当作片上存储器，但是 OpenCL 标准没有这样要求。本地内存供 workgroup 共享。因此，相比于全局内存，访问延迟更短，带宽更高。调用只有大小而无参数的 `clSetKernelArg` 函数，允许在运行时将 kernel 中参数定义为 `__local` 的指针(例如，`__local float*sharedData`)分配本地内存。另外，数组通过附加关键字 `__local`(例如，`__local float[64]sharedData`)可以静态声明在本地内存中，但是这需要在编译时指定数组的大小。

私有内存只对于单个 work-item 可见，局部变量和非指针类型的 kernel 参数默认情况下是私有的。在实践中，这些变量通常映射到寄存器，但是私有数组和溢出的寄存器通常映射到一个片外(即长延迟)的内存中。

OpenCL 的内存空间和现代 GPU 类似，图 2.4 显示了 OpenCL 的内存空间和 AMD 6970 GPU 内存之间的关系。

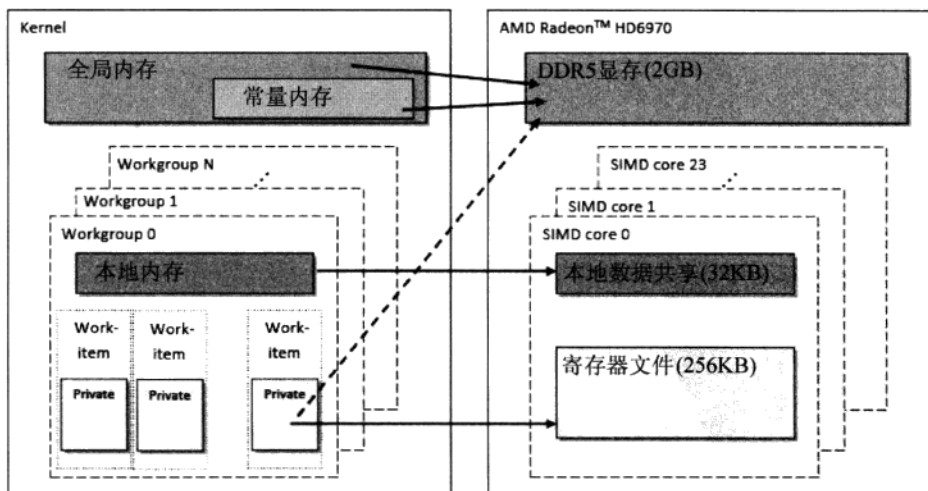


图 2.4

将 OpenCL 定义的内存模型映射到 AMD Radeon 6970 GPU 上，一般私有内存存储于寄存器中，复杂的取址操作或者过分使用参数将使数据存储在 DRAM 中

编写 kernel

如前所述, OpenCL C kernel 与 C 函数类似, 可以看作并行映射操作的实例。函数体就像映射函数一样, 将针对新建的每一个 work-item 执行。我们借助 OpenCL kernel 代码中的 cache kernel 来说明如何完成这种映射。

kernel 以关键字 `__kernel` 开始, 返回类型必须为 `void`。另外像 C 函数一样要求必须指定指针所指向的地址空间。`buffer` 可以声明为全局内存(`__global`)或常量内存(`__constant`)中。`image` 被分配到全局内存, 可以选择性地访问限定符(`__read_only`, `__write_only` 以及 `__read_write`)可以随意指定, 因为它们与编译器和硬件优化相关。

`__local` 限定符用来声明一段内存, 供 workgroup 中所有 work-item 共享。OpenCL 新手程序员往往搞不清楚这个概念。当 kernel 参数声明为局部指针时, 如 `__local float* sharedData`, 它所指向的数组将由整个 workgroup 共享。换句话说, 每个 workgroup 将新建一个具有 64 个元素的数组, workgroup 中所有 work-item 都可以访问它。

声明本地内存分配的另一种方法是在 kernel 范围级别声明一个变量:

```
__kernel void aKernel(...) {
    // Shared by all work-items in the group
    __local float sharedData[32];
    ...
}
```

这似乎具有 kernel 文本的范围, 但在整个 workgroup 中所有 work-item 共享相同的命名实体, 类似于 `__local` 参数, 方法是等价的。重要的是, 要注意 `__local` 参数可以为每次调度设置不同的大小, 在 kernel 中每个 `__local` 声明在编译时是固定的。

针对 OpenCL 设备(尤其是 GPU)进行编程时, 将单个 work-item 或同一个 workgroup 内多个 work-item 中多次使用的数据缓存到本地内存(具有时间局部性的数据)可能会提高性能。开发某个 kernel 时, 我们通过显示的将全局内存指针赋给本地内存指针来实现, 如下面的代码示例所示:

```
__kernel void cache(
    __global float* data,
    __local float* sharedData) {
    int globalId = get_global_id(0);
```

```

    int localId = get_local_id(0);
    // Cache data to local memory
    sharedData[localId] = data[globalId];
    ...
}

```

一旦 **work-item** 执行完成，其状态信息和使用的本地内存是临时的，如果需要保存计算结果，必须将结果回传到全局内部。

向量相加实例的完整代码

以下是本章所讨论的向量相加实例的具体实现代码^①：

```

// This program implements a vector addition using OpenCL

// System includes
#include <stdio.h>
#include <stdlib.h>

// OpenCL includes
#include <CL/cl.h>

// OpenCL kernel to perform an element-wise
// add of two arrays
const char* programSource =
    "__kernel\n"
    "void vecadd(__global int *A,\n"
    "             __global int *B,\n"
    "             __global int *C)\n"
    "{\n"
    "\n"
    "    // Get the work-item's unique ID\n"
    "    int idx = get_global_id(0);\n"
    "\n"
    "    // Add the corresponding locations of\n"
    "    // 'A' and 'B', and store the result in 'C'.  

    "    C[idx] = A[idx] + B[idx];\n"
    "}\n"
    ;

int main() {
    // This code executes on the OpenCL host

    // Host data

```

① 为了简洁起见，我们忽略了代码中的错误检测。考虑到正确性，每次函数调用后需要处理返回值是否为 **CL_SUCCESS**。

```

int *A = NULL; // Input array
int *B = NULL; // Input array
int *C = NULL; // Output array

// Elements in each array
const int elements = 2048;

// Compute the size of the data
size_t datasize = sizeof(int)*elements;

// Allocate space for input/output data
A = (int*)malloc(datasize);
B = (int*)malloc(datasize);
C = (int*)malloc(datasize);
// Initialize the input data
for(int i = 0; i < elements; i++) {
    A[i] = i;
    B[i] = i;
}

// Use this to check the output of each API call
cl_int status;

//-----
// STEP 1: Discover and initialize the platforms
//-----

cl_uint numPlatforms = 0;
cl_platform_id *platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of
// platforms
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
platforms =
    (cl_platform_id*)malloc(
        numPlatforms*sizeof(cl_platform_id));

// Fill in platforms with clGetPlatformIDs()
status = clGetPlatformIDs(numPlatforms, platforms,
    NULL);

//-----
// STEP 2: Discover and initialize the devices
//-----

cl_uint numDevices = 0;
cl_device_id *devices = NULL;
// Use clGetDeviceIDs() to retrieve the number of
// devices present
status = clGetDeviceIDs(

```

```

        platforms[0],
        CL_DEVICE_TYPE_ALL,
        0,
        NULL,
        &numDevices);

// Allocate enough space for each device
devices =
    (cl_device_id*)malloc(
        numDevices*sizeof(cl_device_id));

// Fill in devices with clGetDeviceIDs()
status = clGetDeviceIDs(
    platforms[0],
    CL_DEVICE_TYPE_ALL,
    numDevices,
    devices,
    NULL);

//-----
// STEP 3: Create a context
//-----

cl_context context = NULL;

// Create a context using clCreateContext() and
// associate it with the devices
context = clCreateContext(
    NULL,
    numDevices,
    devices,
    NULL,
    NULL,
    &status);

//-----
// STEP 4: Create a command queue
//-----

cl_command_queue cmdQueue;
// Create a command queue using clCreateCommandQueue(),
// and associate it with the device you want to execute
// on
cmdQueue = clCreateCommandQueue(
    context,
    devices[0],
    0,
    &status);

//-----
// STEP 5: Create device buffers
//-----

```

```

cl_mem bufferA; // Input array on the device
cl_mem bufferB; // Input array on the device
cl_mem bufferC; // Output array on the device

// Use clCreateBuffer() to create a buffer object (d_A)
// that will contain the data from the host array A
bufferA = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    datasize,
    NULL,
    &status);

// Use clCreateBuffer() to create a buffer object (d_B)
// that will contain the data from the host array B
bufferB = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    datasize,
    NULL,
    &status);

// Use clCreateBuffer() to create a buffer object (d_C)
// with enough space to hold the output data
bufferC = clCreateBuffer(
    context,
    CL_MEM_WRITE_ONLY,
    datasize,
    NULL,
    &status);

//-----
// STEP 6: Write host data to device buffers
//-----

// Use clEnqueueWriteBuffer() to write input array A to
// the device buffer bufferA
status = clEnqueueWriteBuffer(
    cmdQueue,
    bufferA,
    CL_FALSE,
    0,
    datasize,
    A,
    0,
    NULL,
    NULL);

// Use clEnqueueWriteBuffer() to write input array B to
// the device buffer bufferB
status = clEnqueueWriteBuffer(
    cmdQueue,

```



```

        bufferB,
        CL_FALSE,
        0,
        datasize,
        B,
        0,
        NULL,
        NULL);

//-----
// SILP 7: Create and compile the program
//-----

// Create a program using clCreateProgramWithSource()
cl_program program = clCreateProgramWithSource(
    context,
    1,
    (const char**)&programSource,
    NULL,
    &status);

// Build (compile) the program for the devices with
// clBuildProgram()
status = clBuildProgram(
    program,
    numDevices,
    devices,
    NULL,
    NULL,
    NULL);

//-----
// SILP 8: Create the kernel
//-----

cl_kernel kernel = NULL;

// Use clCreateKernel() to create a kernel from the
// vector addition function (named "vecadd")
kernel = clCreateKernel(program, "vecadd", &status);

//-----
// SILP 9: Set the kernel arguments
//-----

// Associate the input and output buffers with the
// kernel
// using clSetKernelArg()
status = clSetKernelArg(
    kernel,
    0,

```

```

        sizeof(cl_mem),
        &bufferA);
status |= clSetKernelArg(
    kernel,
    1,
    sizeof(cl_mem),
    &bufferB);
status |= clSetKernelArg(
    kernel,
    2,
    sizeof(cl_mem),
    &bufferC);

//-----
// STEP 10: Configure the work-item structure
//-----

// Define an index space (global work size) of work
// items for
// execution. A workgroup size (local work size) is not
// required.
// but can be used.
size_t globalWorkSize[1];
// There are 'elements' work-items
globalWorkSize[0] = elements;

//-----
// STEP 11: Enqueue the kernel for execution
//-----

// Execute the kernel by using
// clEnqueueNDRangeKernel().
// 'globalWorkSize' is the 1D dimension of the
// work-items
status = clEnqueueNDRangeKernel(
    cmdQueue,
    kernel,
    1,
    NULL,
    globalWorkSize,
    NULL,
    0,
    NULL,
    NULL);

//-----
// STEP 12: Read the output buffer back to the host
//-----

// Use clEnqueueReadBuffer() to read the OpenCL output

```

```

// buffer (bufferC)
// to the host output array (C)
clEnqueueReadBuffer(
    cmdQueue,
    bufferC,
    CL_TRUE,
    0,
    datasize,
    C,
    0,
    NULL,
    NULL);

// Verify the output
bool result = true;
for(int i = 0; i < elements; i++) {
    if(C[i] != i+i) {
        result = false;
        break;
    }
}
if(result) {
    printf("Output is correct\n");
} else {
    printf("Output is incorrect\n");
}

//-----
// STEP 13: Release OpenCL resources
//-----

// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufferA);
clReleaseMemObject(bufferB);
clReleaseMemObject(bufferC);
clReleaseContext(context);

// Free host resources
free(A);
free(B);
free(C);
free(platforms);
free(devices);
}

```

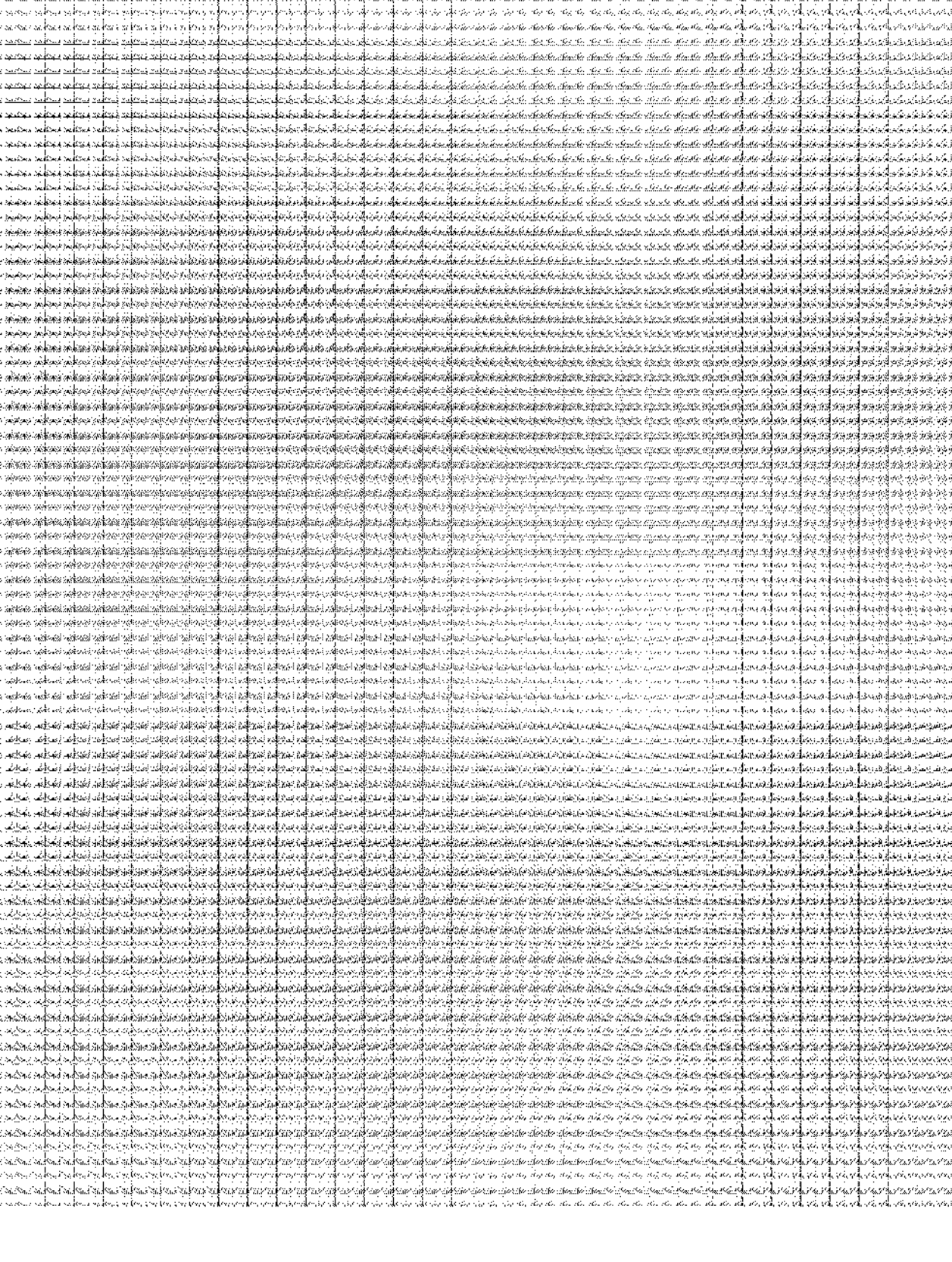
小结

在本章中，我们简单介绍了如何使用 OpenCL 标准开发并行程序，描述了标准中定义的四种不同的抽象模型，并使用 OpenCL 范例进行了展示。

在第 3 章，我们将讨论 OpenCL 设备架构，主题包括各种指令集、线程以及内存。

参考文献

Cooper, K., & Torczon, L. (2011). *Engineering a Compiler*. Burlington, MA: Morgan Kaufmann.



OpenCL 设备架构

引言

OpenCL 由众多行业组织广泛参与开发，旨在满足标准化编程模型的需求，使其能在市面上诸多设备上取得出色或优秀的性能。对于 OpenCL 的设计目标和在特定体系结构上需要具备哪些特点才能确保正确运行，每个厂商都有自己的想法。最后，OpenCL 包含了一系列的特性，以期进一步探索独有的硬件功能。比如，OpenCL 支持基于块并行的宽松的一致性模型，可以在串行、对称处理器(symmetric multiprocessing, SMP)、多线程和单指令多数据(single instruction multiple data, SIMD)或者向量设备上相对高效地运行。在本章，我们将讨论一些这样的设备及其整体设计空间。

尽管 OpenCL 被设计成与平台无关的应用编程接口(API)，但在算法设计以及 kernel 实现层面，从性能的角度来看，真正实现平台无关仍然只是一个目标(与实际相比)。作为开发者，我们需要理解不同硬件特性的潜在优势、这些设备的关键运行特征以及如何在计算机系统结构中对这些设备进行分类。一旦读者具备了对目标硬件的深刻了解，就可以在设计并行算法和软件的时候做出明智的选择。读者也会从编程、内存和运行时模型的角度更好地理解 OpenCL 的设计思路。

硬件权衡

考虑到 OpenCL 的发展历史及其早期在图形 API 和像素着色器方面的应用，我们很容易理解 OpenCL 为何能够发展成为一种面向 GPU 编程的主流语言。在高性能计算市场，OpenCL 已成为一种流行的编程 API。随着支持 OpenCL 的设备种类增多(特别是在嵌入式系统领域)，OpenCL 的整体影响力也会大大增加。

在本章的讨论中,我们没有必要一定清楚说明 GPU 到底是什么以及它与“其他设备”有何区别之类的问题。当我们为 GPU 开发通用代码的时候,是什么使其仍然是一个“图形处理器”呢?该设备上除了专门用于图形的电路,是否还具备其他的功能?

当我们试图仔细思考这个问题的时候,会产生更多的问题。一个 GPU 上有多少个核呢?为了回答这个问题,我们必须定义什么是“核”。什么是“众核”设备?它与“多核”设备是否有明显的区别?一般来说,不同的体系结构在给定的功耗/晶体管数限制下,会选择不同方法来提升性能。除了进行计算能力、功耗和面积之间的简单权衡,硬件开发人员往往还必须考虑编程所付出的工作。上述因素之间的权衡会在设计上产生巨大的差异。

多核 CPU 可以让我们在频率和控制复杂度不变的情况下,随着晶体管尺寸的减小而增加更多的核。在精心设计之下,功耗还可以保持在合理范围内。SIMD 和超长指令字 (very long instruction word, VLIW) 架构尝试通过提高算术操作逻辑电路的比例,进一步提高有效任务的执行量。在这种情况下,如何产生足够多的任务负载使算术逻辑部件 (arithmetic logic units, ALU) 保持饱和,会变得比较困难。多线程从另外一个角度达到提高性能的目的。与提高计算逻辑部件相对控制逻辑部件比例的方法不同,多线程方法通过增加可用的有效任务数量来占用计算逻辑部件,以隐藏其他使用非计算部件的时间,比如存储访问流水线。因此,多线程提高了已有部件的使用率。多线程方法可以用在多核芯片的设计上,也可以用在单核芯片的设计上,从软件方面看,都可以看作单核管理多个软件线程。依据不同应用以权衡晶体管数的情形下,高速缓存和内存系统的权衡允许不同的体系结构面向不同的数据访问模式进行优化。

在前述情形下,根据核的不同定义,我们可以把这些权衡应用到单独一个核或者一组核上。然而,我们不需要在整个设备使用同样的权衡。异构性可以允许我们针对多种不同类型同时运行的算法进行硬件优化,从而在两类算法甚至整体上得到更好的性能。目前一个比较普遍的例子是在当代 PC 上所见到的在系统层次的 CPU+GPU 的组合(与其他一些低性能的处理芯片散布在整个系统里)。最新一代高性能处理器将两者融合成一颗芯片,AMD 称之为加速处理单元 (accelerated processing unit, APU)。

在现实中,我们可以看到面向不同目标市场、应用和零售价格的不同设计组合。本节我们将介绍其中一些体系结构特性,并讨论它们在常见的不同体系结构下的应用。

性能随频率的提升及其限制

作为开发人员,在考虑如何写代码的时候,最简单的方式是创建一个以线性方式执行的软件:执行一个任务,完成这个任务,再执行另外一个任务。对于开发人员来说,编写并行代码是相对困难的。即使在具有一定的内在 SIMD 或向量并行性的图形领域中,也是如此。而对于其他一些应用,没有了图形领域中多通道像素的便利条件, SIMD 向量的抽取操作变得更加困难。因此,在迁移到并行计算之前,计算机体系结构过去只着眼于提升单线程的执行性能,而把并行计算交给特定市场的高性能专业计算机去完成。

相当长一段时间以来, CMOS 电路的缩小使晶体管之间的距离成比例地缩小。距离的缩减和电容电量的减小使得硬件体系结构师可以将电路运行在更高的频率上。因而导致戈登·摩尔(Gordon Moore)提出著名的关于晶体管密度的已被验证的预言,并被错误应用于解释运行频率和整体性能。确实,在相当一段时间里,提高频率使非并行代码的性能获得持续的提升,以至于一直到 21 世纪早期都被软件开发人员看做理所应当的。

然而,在过去的十多年里,主要由于功耗和散热的限制,继续提高 CPU 的主频已经明显变得不现实。其主要原因是功耗以非线性的方式依赖于主频。CMOS 的动态功耗可以近似为动态功率和静态功率之和:

$$P=ACV^2F+VI_{\text{leak}}$$

其中:

A 是活动因子,即在电路中正在切换的晶体管数量

C 是电路的电容量

V 是电路的电压

F 是切换频率

I_{leak} 是估计的晶体管漏电电流

从公式可以看出,功耗与频率成线性关系。在实际中,为了增加频率,必须增加电路中电容的充放电频率。而要做到这一点,需要相应地提升电压,从而导致公式中的动态部分和静态部分都上升。在很长一段时间内,电压可以通过每代制程工艺的改进而减小,从而使得频率的提升不会导致功耗不可控的增加。但是,随着制程工艺达到当今如此小的尺度,我们已经不可能在不增加晶体管切换错误率的情况下继续降低电压。因此,频率的提升必须相应提升电压。进而,频率的任何提升所带来的功耗提升便变得不可忽视。

由此带来的第二个问题是，提升时钟频率要求要么增加片外主存带宽，要么增加系统高速缓存容量，以便能够快速提供数据，防止处理器上运行的串行任务阻塞。

如果不能通过持续提高主频来获得更高的性能，则需要其他的解决方案。任何此类方案的核心都是在给定时钟周期下增加执行操作的数量。

超标量执行

超标量甚至更广义的乱序执行已经在 CPU 中使用了很长时间；从奔腾时代开始就出现在 x86 的设计中。在这些设计中，CPU 负责维护指令流中指令间的依赖关系并在条件具备的时候，调度指令在空闲的功能单元上执行。示例如图 3.1 所示。

乱序执行的主要优势是方便软件开发人员。硬件自动从程序员编写的代码中抽取并行性，串行代码不需要程序员额外的努力就可以更快地执行。实际上，超标量作为一种可超线性提升整体性能的设计手段，早于通过频率提升性能的设计手段十年甚至更长时间，便已经应用在大规模生产的产品中了。但是，不能说它没有一点缺点。

为了完全通过硬件处理动态调度问题，乱序执行的调度逻辑需要 CPU 上预留大量面积以维护指令的依赖关系和队列。此外，投机指令执行进一步扩大了乱序执行可并行调度的指令窗口的大小，但由此也导致一些被丢弃任务低效执行。结果，CPU 通过乱序执行获得的性能收益越来越少，随着晶体管体积的减小，工业界还需采取其他办法以提高性能。这种情况即使在超标量设计以前可行的高性能设备上，也是如此。在嵌入式和其他专用设备上，从串行代码中提取并行性从来就不是主要目标，因此此类设计在历史上就很少采用。

超标量处理器的成功案例数不胜数，从西蒙-克雷(Seymour Cray)的 CDC 6600 到二十世纪九十年代设计的很多 RISC 处理器。当前，高端 CPU 基本上都是超标量的。很多 GPU 也具有超标量的能力。

VLIW

VLIW 是一种严重依赖编译器的提高处理器指令并行性的方法。不同于之前我们讨论的完全依赖复杂乱序执行硬件来维护依赖关系的超标量执行，VLIW 将依赖关系分析的工作转嫁给编译器。与提供标量指令流不同，VLIW 处理器发射的每条指令都是一条由多条可并行执行的指令组合而成的长指令字。该指令可以直接映射到处理器的执行流水线上。

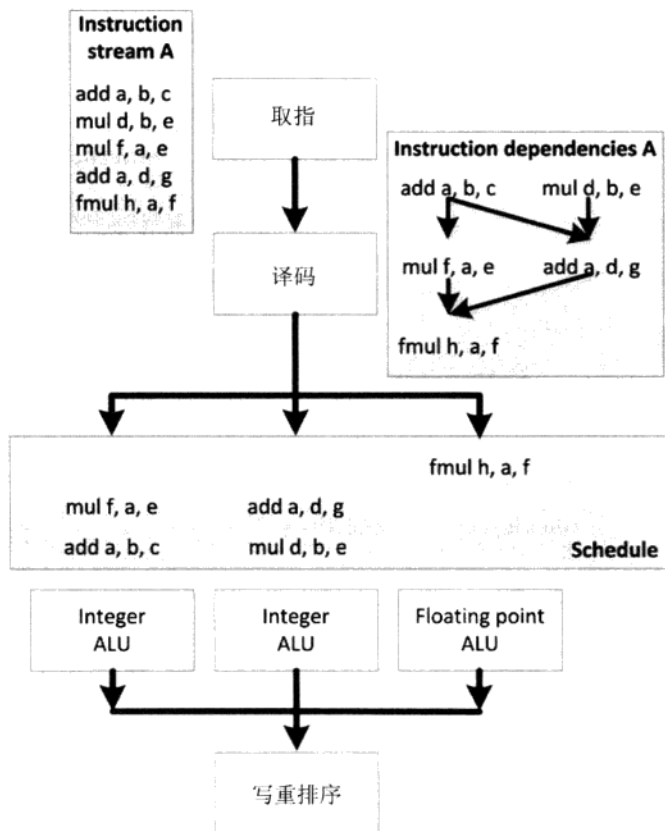


图 3.1

一段简单的类似汇编指令流的乱序执行。注意在该语法中，目标寄存器在前，比如，Add a, b, c 是 $a=b+c$

图 3.2 展示了一个 VLIW 执行的例子。在示例中，使用了与图 3.1 中相同的指令集，但不是串行读取，而是打包成三个水平的最多含 3 条指令的长指令。我们现在可以注意到指令流中的依赖关系是线性的，处理器可以直接以此顺序处理而不需要提取和记录更复杂的依赖关系图。VLIW 指令包被解码后，每个独立的指令流会被映射到处理器上特定的计算部件上执行。在一些 VLIW 设计中，比如这个例子，由于处理器单元是异构的，所以一些指令只能被调度到 VLIW 指令流某个特定通道上。而在其他一些更接近同构的体系结构上，任意指令可以在任意位置发射，满足依赖关系的限制即可。

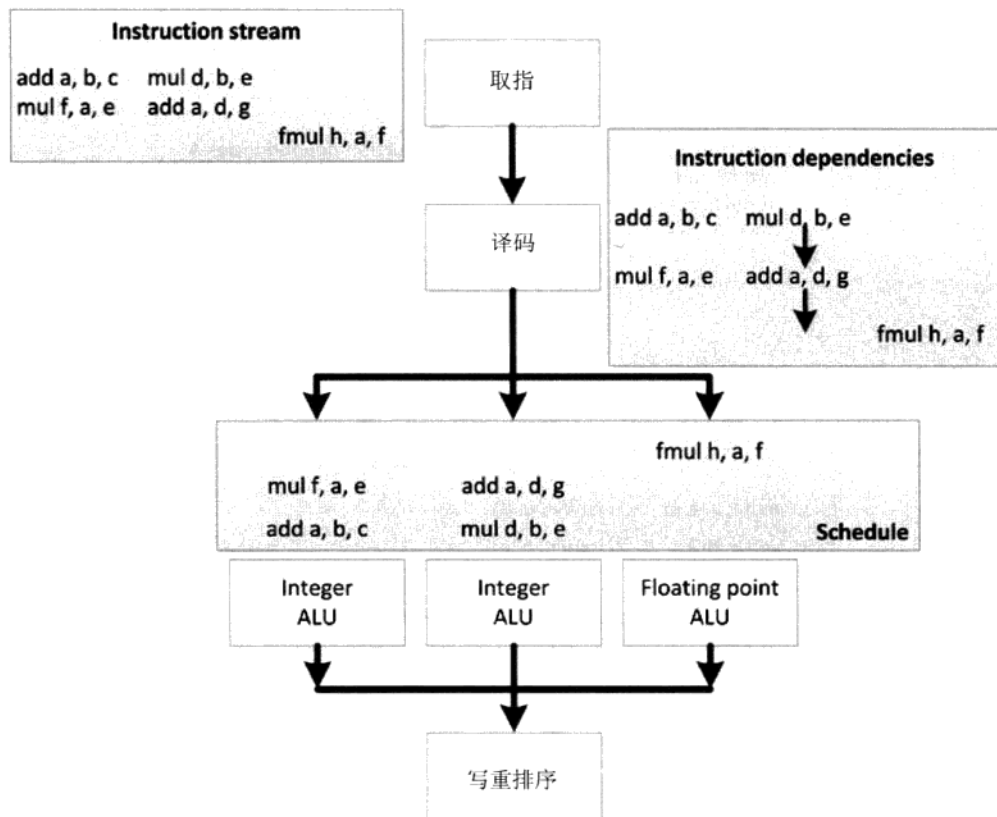


图 3.2

基于图 3.1 的乱序执行图的 VLIW 执行

在图 3.2 的例子中，我们看到指令调度中存在间隙：前两条 VLIW 指令包没有第 3 项，第 3 条没有前两项。很明显，这个示例非常简单，只有很少指令可以打包，但这是 VLIW 架构的通病，效率会因为编译器不能够填满包而损失。这可以归结为编译器能力局限或者指令流缺乏内在并行性。对于后一种情况，VLIW 不会比乱序执行更糟，反而会因为调度部件复杂性的降低而有更好的效率。对于前一种情况，最终会变成权衡未填满执行指令槽而造成的效率损失与由降低硬件控制开销而带来的收益。此外，对 VLIW 执行和硬件调度超标量执行之间进行开销与收益分析时，还要考虑到开发编译器的额外开销。

VLIW 经常出现在 DSP 芯片的设计中。在高端消费类设备中采用 VLIW 设计的有 Intel 的安腾 CPU 产品线和 AMD 的 R600 GPU 系列。

SIMD 和向量处理

SIMD 及其广义向量并行方法采用与之前介绍的方法稍微不同的角度来提高效率。相对于 VLIW 和硬件管理的超标量在指令流中提取不相关指令来获得并行性，SIMD 和向量并行则是直接允许硬件指令参与数据并行执行。

与定义标量操作不同，一个 SIMD 指令表示将同样的操作在多个数据元素上同时执行——在概念上与一个短向量操作类似。向量计算进一步将这一概念推广到长向量上，通常采用数据流水的形式进行计算而不是同时对所有数据单元进行操作，广泛支持对内存的采集(gather)读和散列(scatter)写操作。

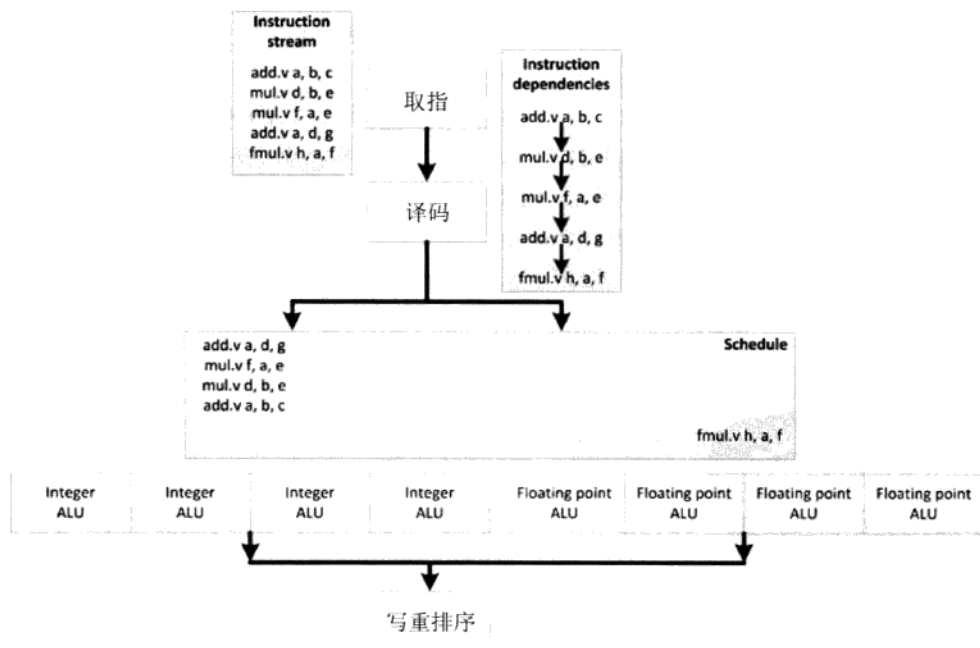


图 3.3

每条 SIMD 指令顺序调度但在多个 ALU 上同时执行

在图 3.3 中, 我们再次看见了之前同样例子的变化版本, 可以看到指令流现在是线性而不是乱序发射的, 但现在每一条指令同时在 4 个 ALU 组成的向量部件上运行。整数指令一条一条地运行在左边的 4 路整数向量 ALU 上; 浮点指令采用类似的发射方式运行在右边的 4 路浮点 ALU 上。值得一提的是, 虽然在这个例子中我们线性地发射指令, 但并不意味我们不能将这些操作在超标量或 VLIW 指令流水线上执行。在后续的讨论中, 我们会看到有些体系结构就是这么做的。

SIMD 执行的优势是相对于 ALU 来说, 可以同时减少调度和指令译码部件逻辑电路的数量。我们现在是在单条指令和单个依赖调度点上执行 4 个操作。

当然, 像之前的方案一样, 这里也有权衡。大量的代码不是数据并行的, 因而不可能转换成向量指令执行。在其他情况下, 用编译器从代码中提取数据并行相当难。比如, 循环向量化依旧是一个挑战, 除了最简单的情况, 鲜有成功。在这些情况下, 我们最终由于没能充分利用 ALU 而造成了晶体管的浪费。

向量处理器最早发源于超级计算机市场, 但 SIMD 设计在很多领域都十分常见。CPU 一般可以通过显式在标量指令流中包含 SIMD 指令实现 SIMD 流水, 其实际例子包括诸如 x86 芯片的流 SIMD 扩展 (SSE) 和 AVX 指令, PowerPC 的 AltiVec 扩展指令; ARM 的 NEON 扩展指令; 等等。历史上, GPU 体系结构就包含显式 SIMD 操作以支持像素向量。许多现代 GPU 也执行隐式的宽 SIMD 向量, 其中的标量指令流代表一条执行通路。实际上, 由于向量在很多情况下是逻辑概念, 这类机器可以视为向量机。比如, AMD 的 Radeon HD6970 架构就可以执行 64 位宽的 SIMD 向量操作, 其中每一路含有一个 128 位的向量寄存器文件。这些宽向量指令以流水线的方式在 16 路 SIMD 单元上经过多个周期执行完毕。

硬件多线程

在指令和数据并行之后的第三种常见并行形式是线程并行, 也就是多个独立指令流的执行。很显然, 此种方式大量用于大规模并行计算机, 但在单个 CPU 核中也有用处。如前所述, 从一个指令流中提取独立指令是很困难的, 从硬件和编译器两个角度来说, 有时候甚至是不可能的。但是, 从两个独立的线程提取这种指令级并行是非常简单的, 因为这些线程已经确保了独立性。硬件上的难度是需要管理额外的指令流和维护第二条指令流的状态所需要的寄存器和缓存。

有两种实现片上多线程的方法:

- 同时多线程
- 分时多线程

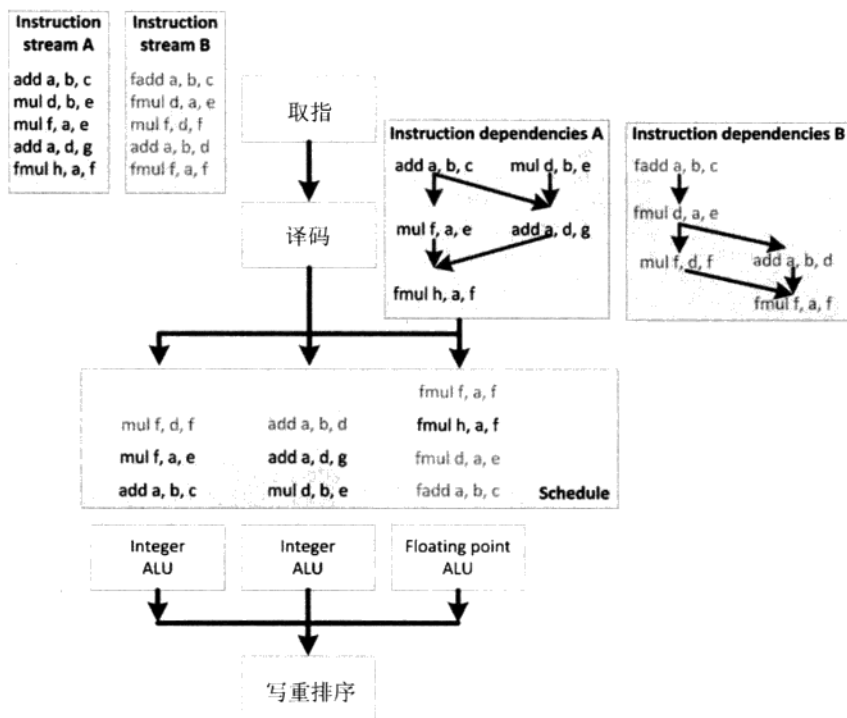


图 3.4

图 3.1 所示的乱序调度在结合第二个线程之后同时执行

同时多线程(Simultaneous multithreading, SMT)如图 3.4 所示。在这种设计中,通过扩展超标量调度逻辑电路,来记录指令间依赖关系和源线程,从而使得多线程指令在资源上交叉执行。正如图中所展示的那样,其目标是更有效利用硬件资源,提高有效任务在执行槽中所占的比例。由于需要管理两个不同集合的依赖关系、资源和执行序列,该方法的代价便是使指令依赖和调度逻辑电路变得更加复杂。

图 3.5 展示了更简单的基于时间片芯片多线程。在这种情况下,采用轮转法使每个线程在连续执行槽位上运行。为了简化,在图中我们只画出了一个共享的 ALU 单元。

该方法的优势如下:

- 调度逻辑电路简单;
- 可以通过调度更多的线程转发逻辑电路的数量来隐藏流水线的延迟;

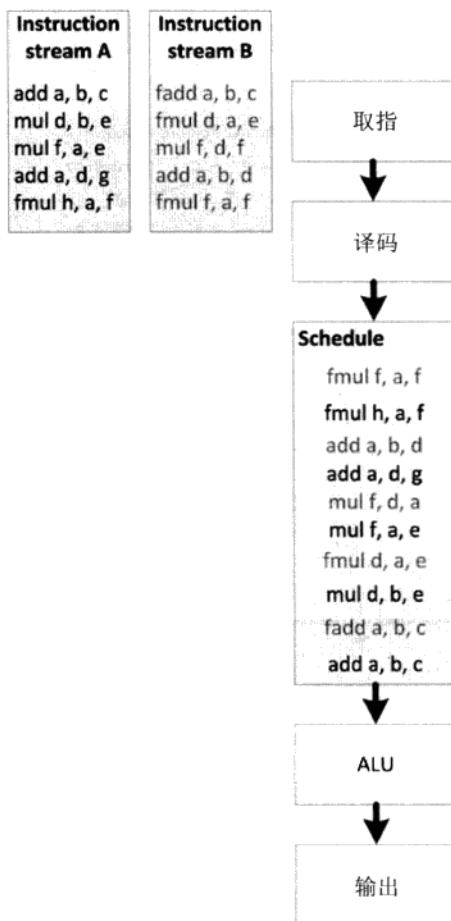


图 3.5

两个线程通过时间片轮转方式调度执行

- 缓存缺失，等待分支判断计算完成或类似事件导致的单一线程挂起，可以采用改变线程执行顺序和运行比隐藏流水线延迟所需的更多线程来隐藏。

最后一个情况在扩展到复杂问题时最有用。很多体系结构能够执行比必需线程数更多的线程。当某个线程因为某种原因被挂起时，它会从就绪队列中移除，这样一来，只有在就绪队列上的线程可以调度执行。一旦挂起结束，该线程可以再次放回到就绪队列。在这种方式下，尽管单个线程的时间可能比乱序执行机器运行更长，但整个机器的吞吐

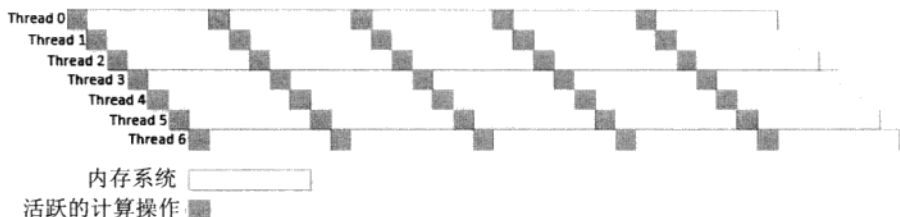


图 3.6

将分时多线程发挥到极致的吞吐量计算：大量线程交叉执行使设备保持忙碌，其中每个独立线程需要的运行时间比理论最短时间更长

量却可以保持在较高水平，且在不需要过于复杂控制逻辑的情况下更高效地利用资源。在极端情况下，这种重多线程可看成吞吐量计算：通过牺牲延迟来最大化吞吐量。图 3.6 展示了设计原则。

两种芯片多线程形式都是比较常见的。来自 Tera 的 MTA 设计是一个经典的时间片多线程超级计算机。MTA 的设计比较难于制造；随后，Cray 公司实现了 MTA-2 的设计，通过利用 CPU 上的 128 组寄存器在线程间快速切换状态，跳过挂起的线程。XMT 设计进一步扩展了该思想以适应基于标准 AMD Opteron 的 Cray 系统内的多线程处理器。Sun 的 Niagara 系列芯片实现了一个多核多线程设计(每个核 8 线程)以达到在数据中心应用负载中低功耗和高吞吐量的目标。Intel 的奔腾 4 和 Nehalem/Sandy Bridge 设计实现了一种被称为“超线程(hyperthreading)”的 SMT。现代的 GPU 设计中在每个核上采用分时方式运行大量的线程，其中的线程数量受限于硬件资源。在当前的 AMD GPU 产品中，通常需要每个核运行 8~10 个线程以隐藏延迟和挂起。

多核架构

至少在概念上，提高每时钟周期执行任务量最简单的方法是简单地将单核在芯片上克隆多次。最简单的情况是在缓存一致性协议的支持下通过内存共享数据，这些核的每一个都可以最大程度地独立运行。这种设计是简化的传统多插槽服务器 SMP 系统的最简单情况，而 SMP 系统用来提高性能已经有几十年历史了。

然而，多核系统有很多不同的外观，准确定义什么是核可能非常困难。比如，一个高端主流 CPU 通常包含大量的功能部件，如阻塞接口逻辑和内存控制器等，在芯片上与其他核是独立的，是不大可能算作核的。但是，如图 3.7 所示，在 AMD 简化版的“Bobcat”（低功耗核）旁边的“推土机(Bulldozer)”（高功耗核）设计中，同一个可复制单

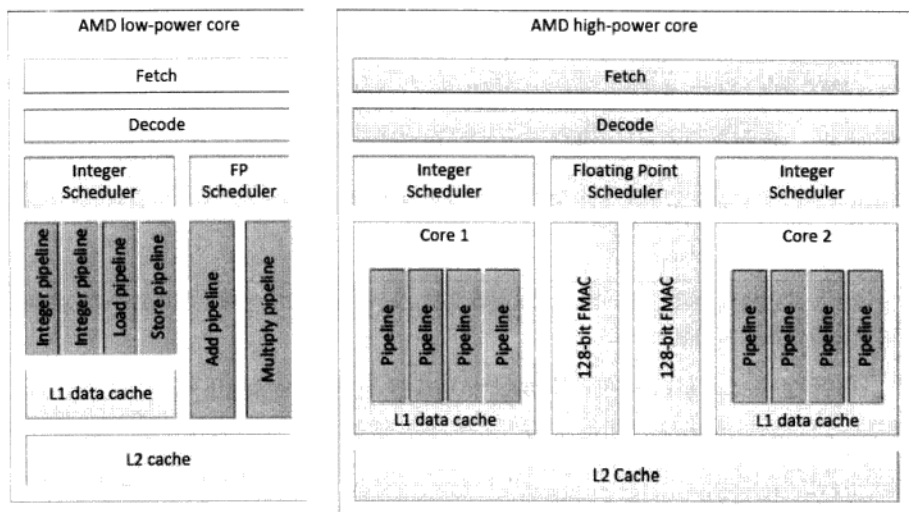


图 3.7

AMD Bobcat 和 Bulldozer 的高层设计(没有展示共享尺度)。Bobcat(左)在低功耗设计中遵循传统的功能部件映射到核的方法。Bulldozer(右)在一个模块内整合 2 个核,提供一些功能部件的共享。Bulldozer 的示意图采用两种阴影分别表示核间共享的功能部件与非共享的功能部件

元(称之为模块)内的两个核共享了一些功能部件。其中,每个线程仍然以传统方式运行在每个核上,但是其中的浮点计算指令将在共享的浮点流水线上交叉执行。这种设计的目标是通过改进功能部件的使用率来提高效率。

GPU 的设计以类似方式展示了对核的不同定义。现代 GPU 的当前高端产品拥有几十个核,根据复杂性的不同在 16~32 个核之间变化。可是,如果我们观察图 3.8 所展示的 AMD Radeon HD 6970 的概要示意图,可以看到 Bulldozer 将类似方法发挥到了极致。公平的与传统 CPU 的执行单元相比,该设备有 24 个 SIMD 核,这些核只能执行浮点和整数 ALU 操作。指令调度,译码和分发是由 wave(波)调度单元执行的。因为被调度单元是称为 wavefront(波前)的宽 SIMD 线程上下文,所以调度部件称为 wave 调度器。实际上,6970 采用两个调度器以防止过高的复杂性,相应地,在低端系列产品,则只使用一个调度器,但对 SIMD 核数进行缩减。这是另外一个通过划分成多核以提高效率的例子。6970 上的内存操作直接从 wave 调度器发送到内存系统,而不需要在 SIMD 核上执行,使内存操作如 Bulldozer 中共享的浮点处理单元一样在多个核间共享。

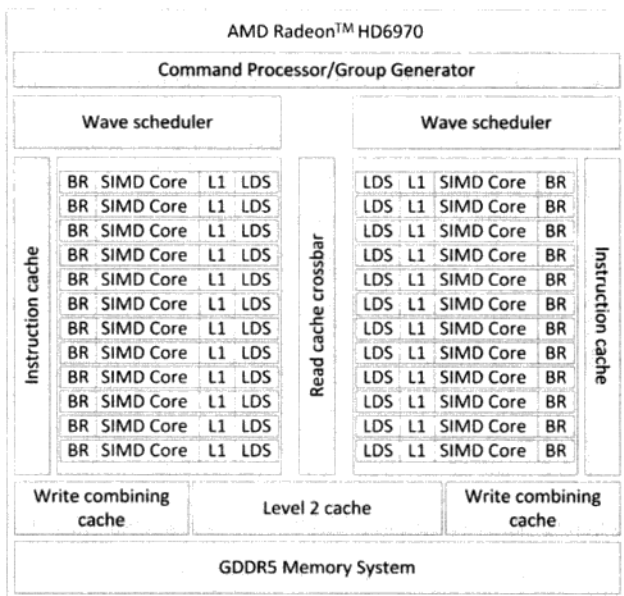


图 3.8

AMD Radeon HD6970 GPU 的架构。设备对分成两部分，分别由各自的 wave 调度器对每一部分的指令进行调度和分发执行。24 个 16 路 SIMD 核，每一路都可以运行 4 路 VLIW 指令，并包含私有的 L1 高速缓存和本地数据共享(local data share, LDS)

集成：片上系统和 APU

在嵌入式领域，通常采用异构程度更高的多核设计方法。为了实现低功耗，嵌入式研发人员往往集成多种不同组件来构建复杂的片上系统(SoC)，达到紧凑和经济的设计。这种集成专用部件的办法可以允许针对设备的应用场景进行优化，这一点对于手机等市场尤为重要。

SoC 的优点如下。

- 通过将多种组件集成到一个设备中，只需要面对一种生产制程和一种产品，从而降低制造成本。
- 更少的模块在终端设备上占用更小的空间。允许更低的设备成本和更小的设备体积，这些对移动手机这类市场尤为重要。
- 更短的距离意味着更少的通信能耗，更容易通过单一内存系统共享数据。

- 传输延迟低，可缩短将任务分派到协处理器的时间。

在手机市场领域有很多出色的例子，比如高通(Qualcomm)的 Snapdragon SoC 芯片和德州仪器(Texas Instruments)的 OMAP 系列芯片。这些设计一般包含一个 ARM ISA 的核、一个移动 GPU、内存控制器和各类无线和媒体处理组件。

在高性能市场，索尼(Sony)，东芝(Toshiba)和 IBM 开发了 Cell Broadband 处理器，在传统的全功能核上集成了一些高性能但简单的小核，以提高每瓦特的性能。AMD 和 Intel 都开发了集成 CPU 和 GPU 的 SoC 芯片，AMD 称之为 APU，在更高效的单一芯片中集成高性能图形和 CPU 的功能。

高速缓存层次和内存系统

早期超级计算机的内存带宽和延迟，使 CPU 总是可以随时访问需要的数据，这一情形持续了很长时间。但是，现在从 CPU 访问内存到数据返回的延迟是几百甚至上千个 CPU 周期。在单线程 CPU 上，为了隐藏如此巨大的延迟，乱序执行逻辑会变得超乎想象的复杂。

幸运的是，大多数应用程序不会进行完全独立的内存访问。通常来说，内存访问模式表现出一定程度的局部性，具体有如下两种。

- 空间：两个或更多个内存访问的读/写地址或多或少在内存中相邻。
- 时间：两个或更多的内存访问的读/写地址相同(比如在不同的时间执行相同的读操作)。

从两种形式的局部性，我们可以得出结论：如果我们保留内存中读到的值及其邻居，那么后续的读取就可以重用这些数据。为此，CPU 设计人员在中间增加了结构复杂的内存缓冲层以支持这种优化。

高速缓存有很多不同的设计，但是我们可以根据工作负载的不同将它们大体分成两大类。CPU 上高速缓存的设计目标是最小化延迟。为此，CPU 的高速缓存一般比较大，具有复杂的层次，目的是让尽可能多的数据尽可能靠近 CPU 核。由于乱序执行只能隐藏有限的延迟，因此访问数据的延迟越小性能越好。此外，使数据尽量靠近计算单元还可以降低能耗开销：远距离的数据移动是 CPU 能耗的一个重要组成部分。

通过采用多线程隐藏数据请求和返回的时间，高吞吐处理器能够更加容忍延迟。在

这些设计中，高速缓存的目标不是最小化访存延迟，而更多的是在有限的内存总线上降低传输流量，所以大容量的多层高速缓存架构并不常见。小容量的可以捕捉到邻近内存访问但不太关注长时间间隔重用的高速缓存，在这种情况下比较常见，作用更像一个空间滤波器。宽 SIMD 单元和编程模型的目标是高效的合并内存访问，以增加所发射的内存事务的大小。结果是可以将芯片中更多的晶体管用作算术运算部件。此外，尽管还存在一些系统级的(诸如可用管脚数和是否允许内存芯片交换等同等重要的)问题需要考虑，但高延迟和高带宽的内存互连使得这种设计可以更高效的工作。我们经常在 GPU 设计中看到偏向空间局部性的扩展，对内存进行布局以便更高效的支持高速缓存的二维访问。

在某些 GPU 和 Cell 处理器设计中，与高速缓存并列或者在高速缓存层次内还包含软件控制的 scratchpad 存储空间。这类缓冲区在给定功率和面积预算的前提下可以获得更好的性能，但需要更复杂的编程。

所有特定设计的真谛都是基于处理器上的预期工作负载，平衡高速缓存的层次和功能。但是不幸的是到目前为止还不存在能够回答所有处理器设计和工作负载组合的标准答案。

架构设计空间

在现实世界中，能够简洁的完全符合之前提及的体系结构类别的架构不是很多。实际上，计算机系统结构在各个方向上都有丰富变化的巨大设计空间。现代的体系结构一般都落在设计空间里的不同点上。

这可以帮助我们意识到当前一些关于体系结构的众所周知的观点可能过于简化。例如在 GPU 领域，我们经常会遇到如下表述：

- CPU 是串行的，GPU 是并行的
- CPU 只有少量的核，GPU 有上百个核
- GPU 可以运行上千个线程，CPU 只运行 1 个(或 2 个)

实际上任何设计都很复杂，需要考虑诸如内部缓冲区、流水线数量、流水线类型等广泛的变化因素。本小节的主旨是说明 GPU 和 CPU 之间，甚至大部分的当代体系结构之间并不存在本质上的差别。今天，我们所看到的大部分显而易见的体系结构区别，仅仅是在基本设计基础上应用了一组不同参数。这些差异是普通程序员需要理解的，而只有专家才需要关注缓冲区的大小比例和如何安排指令来协助硬件发射。

在本节，我们将讨论几种真实的体系结构，它们符合我们前面讲到的设计空间权衡问题的某些特点。希望这能给读者一个关于体系结构权衡方面的更细致入微的感受，并

帮助读者培养判断算法是否可以在实际的体系结构上良好运行的能力。图 3.9 给出了一些体系结构权衡的图形化的示例。在做图时,为了一定程度的简化而忽略了高速缓存。在设计中,我们将自己限定于极端简单的情况。该图的作用是展示 GPU 设计中的宽 SIMD 和状态存储设计是在简单 CPU 的基础上通过考虑处理器面积的使用情况长期发展而来的。这些设计选择可以实现性能最大化和降低获取良好性能的难度。

CPU 设计

大部分用于开发的设备可以广义称为“处理器(CPU)”。甚至仅仅在该空间中,如何利用不同形式的并行性存在着巨大差异。

低功耗 CPU

在功耗的最低端, CPU 核一般是非常简单且顺序执行的。在这个层面,功耗是设计的首要目标,而性能退居为次要目标。此类设计经常不支持浮点操作,且没有并行性的需求。

当前,最广泛使用的低功耗 CPU ISA 是由 ARM 公司开发的以 IP 核方式提供的 ARM ISA。ARM 体系结构最早作为一种桌面机体系结构,来源于 Acorn 计算机公司的 Acorn RISC 机。但是,近来对该体系结构的简化使得 ARM 在移动和嵌入式市场逐渐居于主导地位。1996~1998 年,DEC 公司尝试进入 Acorn 公司的桌面项目,制造了 StrongARM 芯片。由于 ARM 公司将 ISA IP 授权给各厂商,而厂商可以自由设计自己的核,ARM 的芯片变种非常广泛。一般来说,在 SoC 内,ARM 核会与其他功能部件如蜂窝、调适解调器、嵌入式图形处理器、视频加速器和类似的设备集成在一起。

大部分的 ARM ISA 是具有 3~7 个流水级的顺序执行核。基于 ARMv7 ISA 的 Cortex-A8, -A9 和-A15 是支持超标量和多达四个核的对称多核体系结构。基于 ARMv7 的核还支持 Neon SIMD 指令,每个核可以进行 64 位和 128 位的 SIMD 计算。

图 3.7 展示的 AMD Bobcat CPU 核是 AMD 产品线中为 1-10W 功耗范围设计的低功耗核。为了达到预定的低功耗数, Bobcat 核以低峰值性能为代价,使用了比高端产品更慢的时钟频率,并且通过精心的设计来降低开销。Bobcat 设计成 64 位,并支持 2 路乱序发射,包含一个 64 位的可运行多周期 SSE 操作的 SIMD 单元。

Intel 的凌动(Atom)采用了与 AMD 的 Bobcat 稍微不同的设计路线。凌动(Atom)不支持乱序执行从而影响了单线程的性能。然而,凌动(Atom)却支持某种形式的分时多线程,即可以同时运行两个线程。这使得 Atom 在某些情形下运行多线程工作负载的性能富有竞争力。

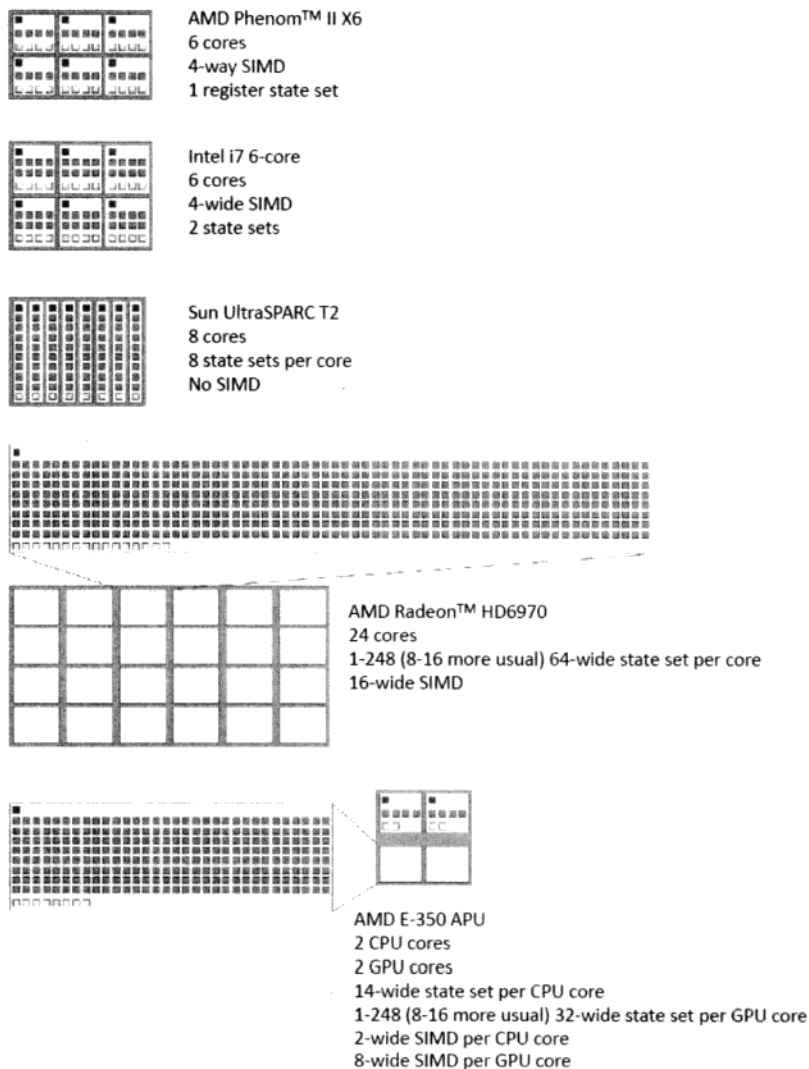


图 3.9

以图形化方式展示本节中讨论的体系结构在硅片面积使用上的权衡。特别需注意的是，CPU 和 GPU 在状态存储方面的不同。在示意图中 ALU 的定义比较灵活。我们统计整数 ALU 和浮点 ALU 的总数。但没必要区分乘法和加法的 ALU，即使它们是能够双发射的。图中的重点是展示大致的比例，而不是该设备的精确数字

通常来说, 这些低功耗 CPU 支持顺序或基于相对窄 SIMD 单元的有限乱序执行。核数量的变化可以用来在多线程应用下满足不同功耗/性能点的要求。在所有情况下, 相对于桌面级 CPU 来说, 这些功能都尽量保持简单, 以降低能耗。

主流台式 CPU

AMD 和 Intel 的主流台式 CPU 设计与 Bobcat 的设计没有大的不同, 只是在每种情况下略微增加每个元素的复杂度。

AMD Phenom II 代表了 AMD 当前的主流 CPU, 也是我们后续将要介绍的 Llano APU 的组成部分。在售的 Phenom II 拥有 6 个核(双硅片(die)的 Magny-Cours 设计有 12 个核), 允许高度的并行执行。该 CPU 中每核拥有高达 512 KB 的 L2 缓存和 6 MB 的共享 L3 大容量高速缓存。

每个核拥有一个全功能的 128 位 SSE 单元, 可以同时发射加法、乘法和其他指令。具有一个宽 L1 高速缓存接口(每周期 128 位), 可以并行的译码、取指和发射 6 条整数操作。因此, 该超标量设计与 Bobcat 核相比具有更广泛的能力, 伴随着增加的缓存容量和分支预测电路以及更高的时钟频率, 使得该 CPU 适合更高性能和更大能耗的市场。

Intel 的 Sandy Bridge 微架构也是抢占类似的市场。与 Phenom 类似, Sandy Bridge 通过多流水线支持全 128 位 SSE 操作, 且可并行发射 6 个混合类型的操作。此外, Sandy Bridge 支持 256 位的 AVX 操作, 允许每时钟周期计算 16 个单精度浮点数。与 Atom 类似, Intel 在 Sandy Bridge 的前一个版本, Nehalem 中就增加了硬件多线程的支持, 并在后续的 Sandy Bridge 保留了这个功能。这是一种真正 SMT: 每个核可以在功能单元混合执行一对线程的操作。这种方法在提高调度的复杂性与提升功能单元利用率间进行了权衡。

如图 3.7 所示, AMD 的推土机(Bulldozer)选择了增加核数和增加每核线程数之间的中间路线提升并行线程执行的速度。不同于之前 AMD 增加核数而导致每核开销过大的设计, 也不同于 Sandy Bridge 使用的造成高调度复杂性的真 SMT 技术, Bulldozer 采取了中间路线。

Bulldozer 的方法是增加第二个独立的整数核, 它拥有自己的私有 ALU、状态和调度器。但是, 浮点 ALU 仍然在一对核间共享, 此外取值、译码模块以及 2 级高速缓存也是共享的。该设计的目标是只共享那些在实际工作负载下不会被过度争用的功能部件。

每个核通过 4 条 ALU 流水线支持乱序执行。共享的浮点 ALU 是一对 128 位(SSE)

的 SIMD 单元,可以联合起来执行 AVX 指令。Bulldozer 依靠多核执行来增加线程数量。由于每个核的面积都相对小,与 Phenom 的设计相比,更高的核密度成为可能。

在主流 CPU 中,我们可以看到较宽的多发射乱序执行,高的时钟频率,大的缓存容量,所有这些特点都是为了保持较高的单线程性能,当然也会造成合理范围内的高功耗。核内的多线程仍然很少或者不存在。同时, SIMD 单元设置在一定宽度,防止在不使用的情况下浪费过多的芯片面积。

Intel 安腾 2

Intel 的安腾架构和后续的安腾 2,代表了一次生产一款基于 VLIW 技术的主流服务器芯片的有趣尝试。安腾架构包含非常多的寄存器(128 个整数和 128 个浮点)。它采用了被称作 EPIC 的 VLIW 方法,指令被捆绑成 128 位的包含 3 条指令的指令束。CPU 每时钟周期从 L1 读取 2 个指令束,因此每周期可执行 6 条指令。每个核有 2 个 64 位的 SIMD 单元,同时,该处理器被设计为可高效集成到多核和多插槽服务器中。

EPIC 的目标是把挖掘并行性的难题从运行时转移到编译时。这一点可以通过将运行时的轨迹信息反馈给编译器来实现。由编译器负责将指令打包成 VLIW/EPIC 格式,因此在该体系结构上的性能严重依赖编译器的能力。为了辅助大量的掩码任务,该体系结构内含了指令束间的依赖标志,预取指令,投机读取和旋转寄存器堆。为了提高处理器的吞吐量,安腾 2 实现了对 2 路分时多线程的支持,对于像访存等延迟可能比较长的事件,线程可以进行切换。

Niagara

Niagara 的设计(图 3.10)最初来自 Sun 公司,现在由 Oracle 公司继续开发,该处理器针对服务器的工作负载,采用了一种面向高吞吐计算的多线程方法。在许多服务器上的工作负载,特别是事务和网页的负载,往往是重度多线程的,有大量的轻量级整数计算线程访问内存系统。Niagara 或者 UltraSPARC T 系列(T3 现在是 SPARC T3),设计成以最低的功耗来高效率的执行大量线程,最大化整个吞吐量。每个核都设计为没有复杂乱序执行电路的简单且高效的核,可以在 2 个执行单元上交叉运行 8 个线程的操作。图 3.9 展示了其中的解码电路或 ALU 部件的状态数对比,与主流 x86 相比,可以清楚显示出隐藏延迟和简化逻辑电路的特点。

为了支持这些线程,该架构需要多组寄存器,但是作为权衡,相比超标量架构只需要更少的投机寄存器存储。此外,其协处理器可以加速密码操作,还有一个片上以太网

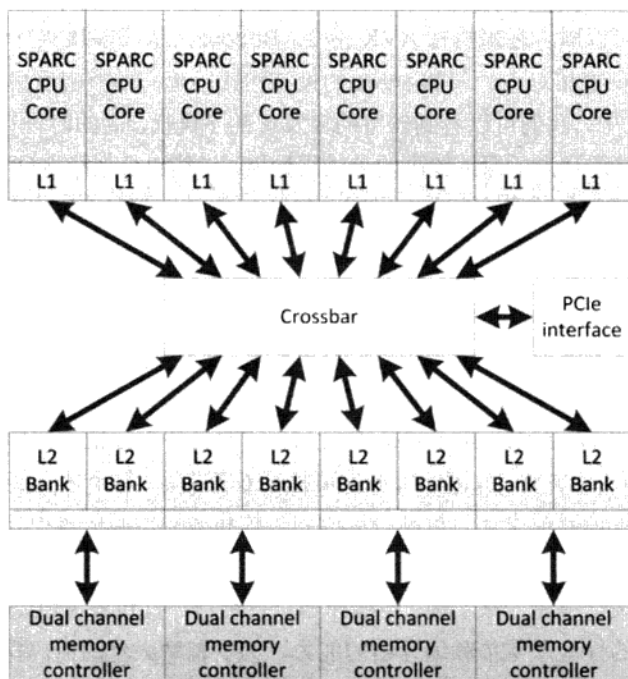


图 3.10

Sun/Oracle 公司的 Niagara 2 CPU 示意图。其设计目的是将多线程的效率提高到一个新水平：注意其与图 3.8 所示的 GPU 设计的类似之处。假设有足够多的线程，我们可以通过有效计算来隐藏所有的内存访问延迟，而不需要复杂的硬件技术进行 ILP 的提取

控制器可提高网络吞吐量。UltraSPARC T2 (Grohoski, 2006) 中有 8 个核，每个核可以执行 8 个线程。SPARC T3 扩展到 16 个核，每个核 8 线程。

GPU 体系结构

与 CPU 类似，GPU 体系结构也面临着多种选择。深入讨论 AMD 体系结构的 OpenCL 编程之前，我们将先简要地讨论其中一些架构。GPU 倾向于通过复杂硬件管理大量的多线程，原因是图形图像的负载主要是复杂顶点，几何图形和像素处理任务图。这些任务和所需处理的像素都是高度并行的，也就是说具有大量的独立任务，适合多核和高延迟的多线程。所以，GPU 不采用复杂的机制去管理任务队列或者由硬件管理来隐藏 SIMD

执行，而是一种简单的多线程处理器，其设计目标是高效的处理大量像素点。

手持 GPU

手持 GPU 刚刚在近期开始具有通用计算的能力，特别的，ARM 和 Imagination Technologies 公司宣布在不远的将来发布完全兼容 OpenCL 的 IP。在这个级别上，GPU 包含少数量的核，典型的只有 1 到 4 个。每个核有像素宽度的一个小的 SIMD 单元，执行大量的独立线程，这与 SSE 向量流水线有些类似。比如，ARM 的 Mali-T604 架构在每个核上使用了三种计算流水线，最高可支持 4 个核。核间任务管理支持跨核的任务调度：通常来说，每个 GPU 线程是由硬件进行控制而不是暴露给操作系统。类似 Mali-T604 的嵌入式设计，采用与嵌入式 CPU 共享同一全局内存的方式，可以减小在不同内存空间传输数据的需求；在 ARM 设计中，数据是完全缓存的。

高端：AMD Radeon HD6970 和 NVIDIA GTX580

高端桌面 GPU 和它们的衍生型号更加专注于性能，而不是最大化能耗利用率。为了达到高内存带宽，芯片上大量的引脚用来进行数据传输，同时使用了 GDDR5 的内存以增加每引脚的内存带宽，相应的开销带来了更高的延迟。这种设备综合使用了多种方法以提高计算吞吐量。这包括使用宽 SIMD 阵列以在一定的指令发射数下最大化计算吞吐量。如图 3.8 所示，AMD Radeon HD6970 架构在硬件上具有 16 路 SIMD，使用向量流水线，在 4 个周期内执行长度为 64 的向量。NVIDIA GTX580 架构(图 3.11)也使用了宽度为 16 的 SIMD 单元，在 2 个周期内执行一个长度为 32 的向量。这两种设备都是多线程的，在每个核上支持大量的宽 SIMD 线程。在 AMD 架构上，每个 SIMD 核最多保存 32 个活跃的 wavefront 的运行状态(每 wave 调度器支持 248 个)，可以每周选择其中的 wavefront 进行执行。但是，活跃的 wavefront 数量还会受到所占用的寄存器数量和本地内存数量的限制。所以，有效的活跃数量会比之前所说的略低，比如 4-8 个 wavefront。

在 AMD 和 NVIDIA 两种体系结构下，编程上都采用 lane-wide SIMD 模型的中间语言，即一条指令流代表了 SIMD 单元上的一条通路，这样每个 SIMD 单元都是程序员可见的，这与 x86 SSE 或 AVX 等向量宽度的指令不同。程序计数器由每个 SIMD 向量进行管理，所以硬件线程实际上是一个宽向量。通过使用一些硬件向量化的技术，将单路的程序组合到一起，可以更有效的管理分支，支持预测执行等。为了支持指令级并行性，AMD 架构通过底层中间语言的 shader 编译器，在每一路上执行 VLIW 指令流。在 HD5xxx

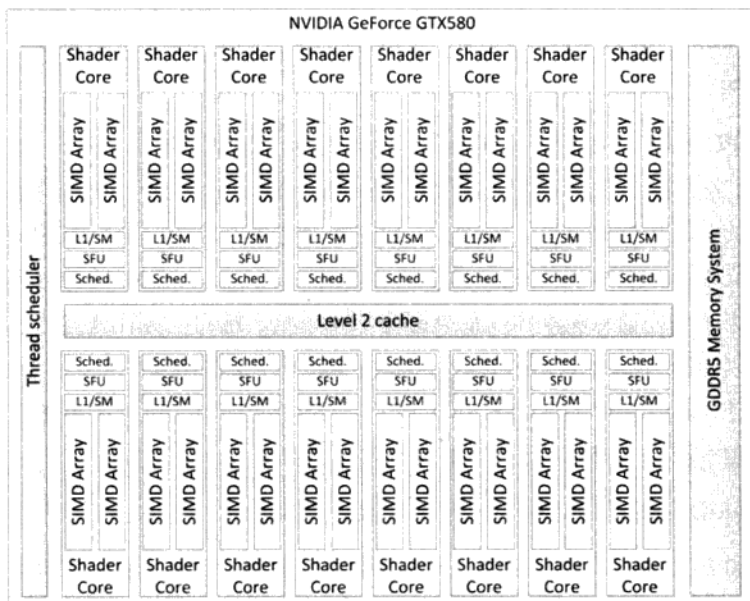


图 3.11

NVIDIA GTX580 架构。此设备有 16 个核，每个核有 2 个 16 路的 SIMD 阵列。每个核包含一个共享内存/L1 Cache，一组 SPU 阵列以执行一些复杂操作。细粒度调度器选择硬件线程(或称为“warp”)，在它们就绪的时候，映射到每个 SIMD 阵列上

设计和大多数 HD6xxx 设计上，是一个宽度为 5 的 VLIW 指令，其中所有路都可以执行大部分的算术指令，但是只有第 5 个可以执行特殊指令，比如高效的 sin 和 cos 指令。在 Radeon HD6970 架构中，VLIW 宽度为 4，VLIW 中的每路都是一样的。NVIDIA 的 GTX580 架构设计，通过在 2 条执行流水线上同时发射 2 个线程以达到指令级并行。这两家公司的设计在执行资源上都是超标量的，核上的运行线程(不需要是同一个线程)可以同时发射访存，算术和其他操作。

与市场上的移动 GPU 类似，高端的 AMD 和 NVIDIA GPU 也会包含多个核。如果采用与 CPU 架构的核类似的定义，那么 AMD 的设计包含 24 个核，NVIDIA 含有 16 个核。

每个核都含有一个 scratchpad memory buffer，这在许多手持 GPU 的设计中是没有的。这个 scratchpad 存储，在 AMD 架构中被称为 LDS (local data share)，在 OpenCL 中用来保存本地内存结构。每个 workgroup 可以申请分配本地内存，如果每个 workgroup 都申

请大量的本地内存，那么硬件容量的限制必然会影响它们的并发执行。这个内存是直接映射到每个执行的组上，非常的快速，而且与硬件管理的缓存相比，消耗能量更小。

图 3.9 展示了 Radeon HD6970 GPU 和 CPU 寄存器状态数量的对比：有多个线程(图中是典型的 8 个)，在 AMD 架构上每个线程的宽度为 64。

APU 和类 APU 的设计

SoC 在嵌入式市场已经很常见了。当前，有一种趋势是将 SoC 使用到更高性能的系统和应用中。最明显的趋势是，结合 CPU 和 GPU 的混合处理器已经开始出现在上网本，笔记本和低端台式机领域。可以想象，这种设计会扩展到高端台式机。我们可能会看到这种集成带来的能耗节约，以及与独立 GPU 类似的计算能力。

当前，市场上有三种主要的架构，分别是 AMD 的基于 Bobcat 和 Phenom II 的 Fusion 产品(Advanced Micro Devices, 2011)，Intel 的 Sandy Bridge。

AMD 的名为 Ontario 或者 Zacate 的设计，是面向低端上网本和笔记本，其功耗只有 9-18W，基于低功耗的 Bobcat CPU 和一个低端 GPU。这些组件使用 40nm 的工艺集成到一个硅片，如图 3.12 所示。在这个例子中，GPU 是一个宽度为 8 的 SIMD 单元，使用了派生自 5xxx GPU 的 5 路 VLIW。2 个 Bobcat 核有 2 路 SIMD 单元，允许每个核每两周期发射一条 SSE 指令。AMD 更高性能的 Llano APU 是基于 Phenom II 核和更高性能的 GPU。

Intel 的 Sandy bridge APU 设计(图 3.13)包含 4 个之前讨论过的 Sandy bridge CPU 核。GPU 是来自 Intel 内部的嵌入式 GPU，当前还不支持 OpenCL，但是可以期待这种融合的处理架构能有效的支持图形处理。

APU 架构可以在 GPU 和 CPU 间共享数据，这会缓解很多 GPU 计算中重要的数据传输瓶颈，减小向 GPU 派发任务的延迟。相比之前由于 PCI express 总线的延迟限制，APU 架构使得设计 GPU 和 CPU 紧密耦合的算法变为可能。APU 架构带来的开销是 CPU 的内存带宽需要在两个设备间共享，而且失去了独立 GPU 上非常高带宽的内存接口。看

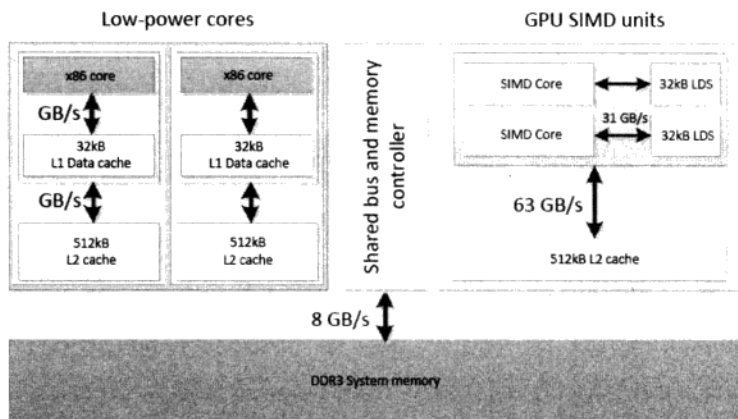


图 3.12

E350 “Zacate” AMD APU 由 2 个 8 宽度的 5 路 VLIW 的 SIMD 核和 2 个“Bobcat”低功耗 x86 核组成，它们通过一个共享总线相连，具有一个到 DRAM 的接口

起来，这种权衡对于大量的纯 GPU 实现无效率的算法是有优势的，原因是 GPU 这种基于吞吐量的设计对于串行代码不是最优的，而且这些算法的数据传输是一个重要性能瓶颈。

小结

在本章，我们讨论了可能支持 OpenCL 运行的多种体系结构类型，以及这些体系结构在设计中的权衡。在更详细介绍 OpenCL 之后，我们将在第 6 章以 AMD Phenom II CPU 和 HD6970 GPU 为例，介绍 OpenCL 的模型是如何映射到体系结构上的。

下面的扩展阅读会对于理解本章内容有益，但是，对于一些专门的硬件设备，很难找到简明的参考资料。Computer Organization and Design (Patterson and Hennessy, 2008) 第 4 版深入讨论了很多体系结构的问题，包含 AMD Opteron, Intel Nehalem (Sandy Bridge 的上一代，共享很多特性), UltraSPARC T2 和很多其他的设计。而且它还包含一段 NVIDIA GPU 架构的内容。Computer Architecture (Hennessy and Patterson, 2011) 第 5 版更加拓展了这些理念。在 2009 年，NVIDIA 发布了 Fermi 架构的白皮书(NVIDIA, 2009)。安腾 2 处理器手册(Intel, 2002)的第 2 章概要介绍了 EPIC 实现和处理器的流水线。

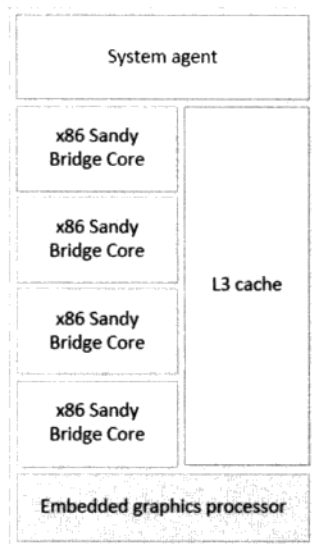


图 3.13

Intel Sandy Bridge 设计。尽管 Intel 没有称之为 APU，但是它的理念与 AMD 的类似。Intel 集成了 4 个 Sandy Bridge x86 核和一个 Intel 的嵌入式图像处理改进版

参考文献

Advanced Micro Devices, Incorporated. (2011). *AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience*. Sunnyvale, CA: Advanced Micro Devices, Incorporated.

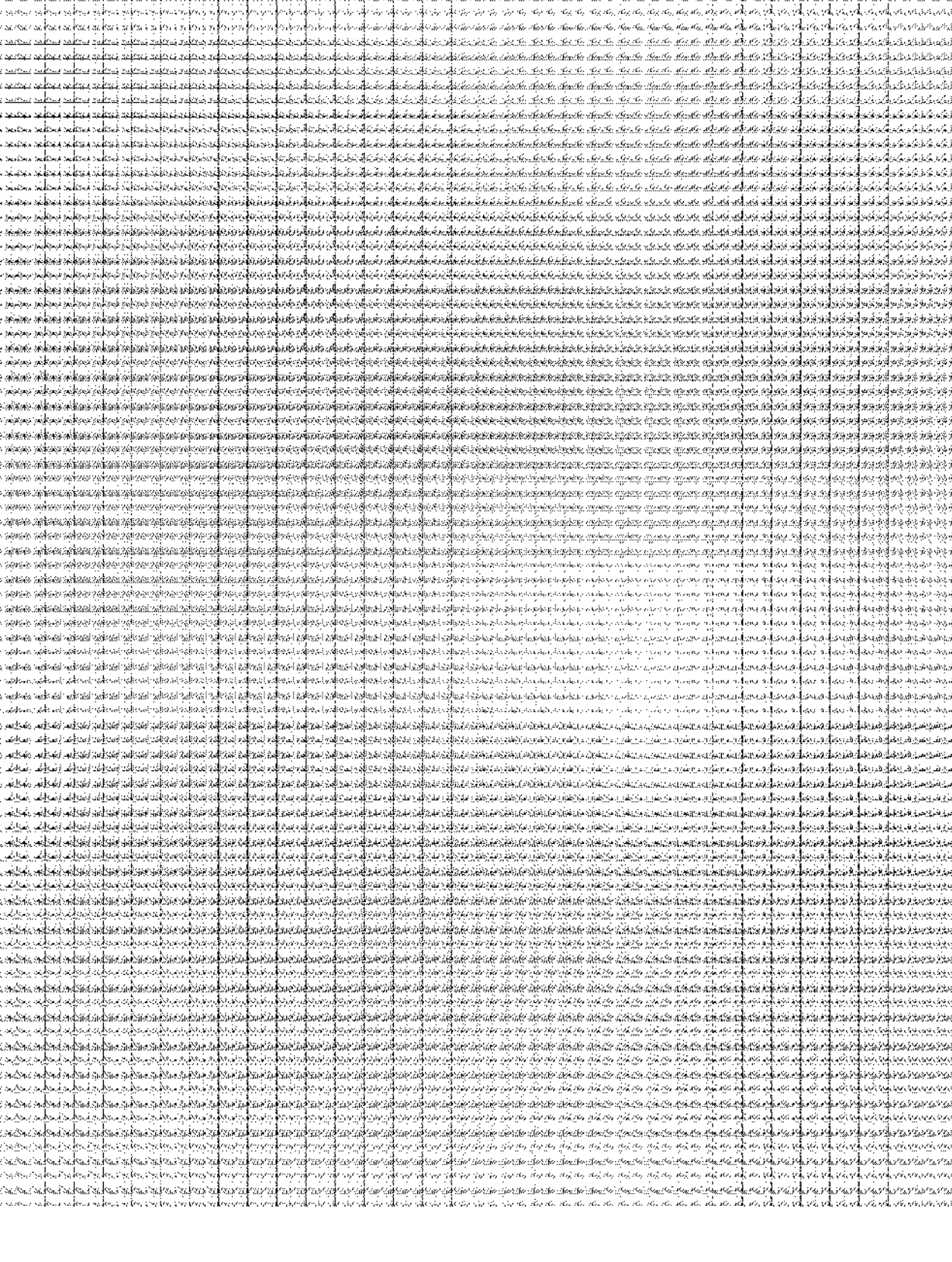
Grohoski, G. (2006). *Niagara-2: A Highly Threaded Server-on-a-Chip*. 18th Hot Chips Symposium, August.

Hennessy, J. L., & Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach* (5th ed.). Burlington, MA: Morgan Kaufmann.

Intel Corporation. (2002). *Intel Itanium 2 Processor: Hardware Developer's Manual*. Santa Clara, CA: Intel Corporation.

NVIDIA Corporation. (2009). *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Santa Clara, CA: NVIDIA Corporation.

Patterson, D. A., & Hennessy, J. L. (2008). *Computer Organization and Design* (4th ed.). Burlington, MA: Morgan Kaufmann.



OpenCL 基本实例

引言

已审阅

Administrator, 14:30:28, 2017/10/24

在第 2 章中，我们介绍了 OpenCL 规范以及如何针对异构平台编写 OpenCL 程序。第 3 章介绍了一些可能适用 OpenCL 的架构。在第 2 章所介绍的诸如向量相加这些简单实例的基础上，本章将介绍几个较为复杂的例子。我们将按步骤详细介绍主机和设备端代码的实现方法。

本章的目的是让读者更直观地了解如何使用 OpenCL 编写数据级并行程序。本章实例均为完整的 OpenCL 实现，但没有针对特定的设备进行性能优化。目标在于给用户提供一些 OpenCL 程序的实现方法，为以后章节中介绍针对特定架构的性能优化打下基础。

应用实例

本节，我们介绍 OpenCL 示例的实现过程。这里的示例包括图像旋转、矩阵相乘和图像卷积。

简单的矩阵相乘

矩阵相乘串行 C 代码如下(OpenCL 主机端程序可以使用 C 语言，也可以使用 OpenCL 提供的 C++封装 API)。矩阵 A 和矩阵 B 相乘，然后将结果存储在矩阵 C 中。代码中有三层 for 循环，外层的两个循环迭代逐个产生输出矩阵的每一个元素。最内层循环对输入矩阵相应元素逐个进行计算，产生输出矩阵相应位置的元素值。


```

// Iterate over the rows of Matrix A
for(int i = 0; i < heightA; i++) {
    // Iterate over the columns of Matrix B
    for(int j = 0; j < widthB; j++) {
        C[i][j] = 0;
        // Multiply and accumulate the values in the current row
        // of A and column of B
        for(int k = 0; k < widthA; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}

```

由于外层两个 `for` 循环相互独立，因此可以直观地将串行实现映射到 OpenCL 上。也就是说，我们可以对输出矩阵的每个元素创建一个单独的 `work-item`。kernel 中，两个外层循环映射到一个二维的 `work-item` 集合上。

如图 4.1 所示，矩阵相乘中，各输出元素之间相互独立。每个 `work-item` 读取矩阵 A 中它需要的那行数据和矩阵 B 中它所需要的那列数据。将读取到的数据相乘之后，结果写入矩阵 C 中相应的位置上。

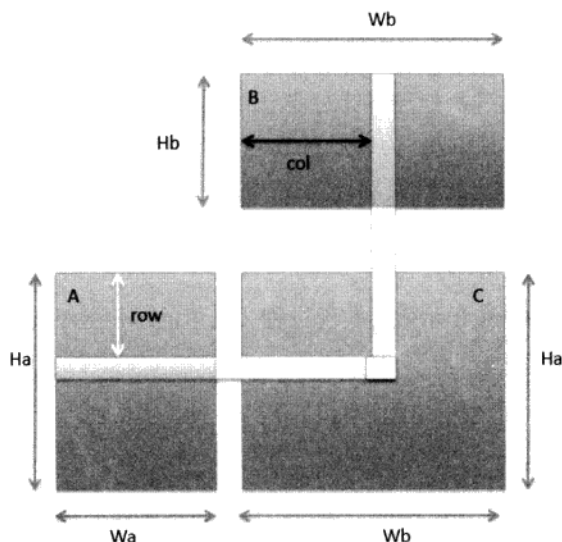


图 4.1

矩阵相乘中各输出元素之间都是相互独立的

```

// widthA = heightB for valid matrix multiplication
__kernel void simpleMultiply(
    __global float* outputC,
    int widthA,
    int heightA,
    int widthB,
    int heightB,
    __global float* inputA,
    __global float* inputB) {

    //Get global position in Y direction
    int row = get_global_id(1);
    //Get global position in X direction
    int col = get_global_id(0);

    float sum = 0.0f;

    //Calculate result of one element of Matrix C
    for (int i = 0; i < widthA; i++) {
        sum += inputA[row*widthA+i] * inputB[i*widthB+col];
    }

    outputC[row*widthB+col] = sum;
}

```

理解数据并行 kernel 的实现过程之后，需要调用 OpenCL API 将数据传输到设备。图 4.2 总结了矩阵相乘其剩余的实现步骤。我们需要为设备创建一个上下文。利用它来创建命令队列，以将命令发送到设备。一旦创建命令队列，就可以将输入数据传输到设备，运行这个并行 kernel，然后从设备端读回输出结果。

步骤 1：建立环境

在这一步，我们声明一个上下文，选择一个设备类型，并创建上下文和一个命令队列。在整个例子中，每次都要检查变量 ciErrNum 确认 API 调用是否返回错误。

```

cl_int ciErrNum;

// Use the first platform
cl_platform_id platform;
ciErrNum = clGetPlatformIDs(1, &platform, NULL);

// Use the first device
cl_device_id device;
ciErrNum = clGetDeviceIDs(
    platform,
    CL_DEVICE_TYPE_ALL,
    1,
    &device,
    NULL);

```

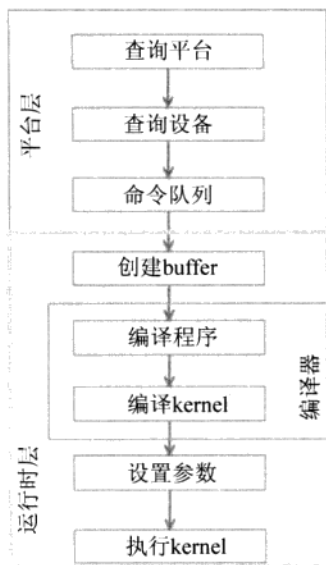


图 4.2

一个完整的 OpenCL 应用程序的编程步骤

```

cl_context_properties cps[3] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0};

// Create the context
cl_context ctx = clCreateContext(
    cps,
    1,
    &device,
    NULL,
    NULL,
    &ciErrNum);

// Create the command queue
cl_command_queue myqueue = clCreateCommandQueue(
    ctx,
    device,
    0,
    &ciErrNum);

```

步骤 2: 声明 buffer, 传输数据

声明设备端的 buffer, 把向设备端拷贝输入矩阵的命令加入命令队列。同时, 声明输出 buffer。

```
// We assume that A, B, C are float arrays which
// have been declared and initialized

// Allocate space for Matrix A on the device
cl_mem bufferA = clCreateBuffer(
    ctx,
    CL_MEM_READ_ONLY,
    wA*hA*sizeof(float),
    NULL,
    &ciErrNum);

// Copy Matrix A to the device
ciErrNum = clEnqueueWriteBuffer(
    myqueue,
    bufferA,
    CL_TRUE,
    0,
    wA*hA*sizeof(float),
    (void *)A,
    0,
    NULL,
    NULL);

// Allocate space for Matrix B on the device
cl_mem bufferB = clCreateBuffer(
    ctx,
    CL_MEM_READ_ONLY,
    wB*hB*sizeof(float),
    NULL,
    &ciErrNum);

// Copy Matrix B to the device
ciErrNum = clEnqueueWriteBuffer(
    myqueue,
    bufferB,
    CL_TRUE,
    0,
    wB*hB*sizeof(float),
    (void *)B,
    0,
    NULL,
    NULL);

// Allocate space for Matrix C on the device
cl_mem bufferC = clCreateBuffer(
    ctx,
    CL_MEM_READ_ONLY,
    hA*wB*sizeof(float),
    NULL,
    &ciErrNum);
```

步骤 3: 运行时 kernel 编译

将 kernel 源代码字符串编译成 program, 创建 program 对象, 并定义 kernel 对象。

```
// We assume that the program source is stored in the variable
// 'source' and is NULL terminated
cl_program myprog = clCreateProgramWithSource (
    ctx,
    1,
    (const char**)&source,
    NULL,
    &clErrNum);

// Compile the program. Passing NULL for the 'device_list'
// argument targets all devices in the context
clErrNum = clBuildProgram(myprog, 0, NULL, NULL, NULL, NULL);

// Create the kernel
cl_kernel mykernel = clCreateKernel(
    myprog,
    "simpleMultiply",
    &clErrNum);
```

步骤 4: 运行程序

设置 kernel 参数和 workgroup(工作组)大小, 然后我们可以将 kernel 入队, 以便在设备端执行它。

```
// Set the kernel arguments
clSetKernelArg(mykernel, 0, sizeof(cl_mem), (void *)&c_C);
clSetKernelArg(mykernel, 1, sizeof(cl_int), (void *)&wA);
clSetKernelArg(mykernel, 2, sizeof(cl_int), (void *)&hA);
clSetKernelArg(mykernel, 3, sizeof(cl_int), (void *)&wB);
clSetKernelArg(mykernel, 4, sizeof(cl_int), (void *)&hB);
clSetKernelArg(mykernel, 5, sizeof(cl_mem), (void *)&d_A);
clSetKernelArg(mykernel, 6, sizeof(cl_mem), (void *)&d_B);

// Set local and global workgroup sizes
// We assume the matrix dimensions are divisible by 16
size_t localws[2] = {16, 16};
size_t globalws[2] = {wC, hC};

// Execute the kernel
clErrNum = clEnqueueNDRangeKernel(
    myqueue,
    mykernel,
    2,
    NULL,
    globalws,
    localws,
    0,
    NULL,
    NULL);
```

步骤 5: 将结果拷贝到主机端

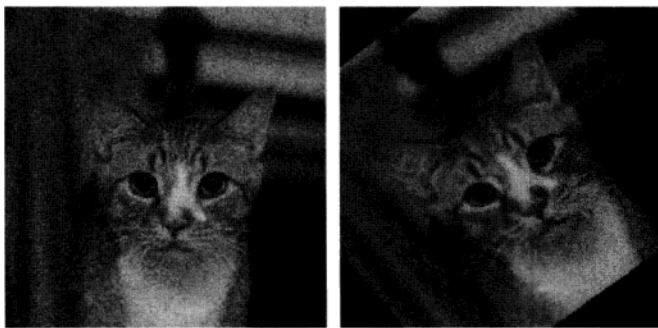
kernel 程序执行完之后, 我们将一个读回结果的命令入队到命令队列中, 从而将结果矩阵从设备端 buffer 读到主机内存中。

```
// Read the output data back to the host
clEnqueueReadBuffer(
    myqueue,
    c_C,
    CL_FALSE,
    0,
    wC*hC*sizeof(float),
    (void *)C,
    0,
    NULL,
    NULL);
```

以上列出的几个步骤展示了用 OpenCL 实现矩阵相乘的过程, 我们可以将其作为一个基本实现过程。接下来的章节中, 我们可以根据数据并行架构的特点对这些基本的数据并行算法进行优化, 从而获得更高的性能。

图像旋转实例

图像旋转是一个在匹配、对齐和其他基于图像的算法中非常常用的处理过程。它的输入是一幅图像、一个旋转角度 y 和一个做旋转的基准点。它的目的是获得图 4.3 那样的效果。在图像旋转例子中, 我们采用 OpenCL 的 C++ 封装 API。



原图

旋转 45° 之后的输出图

图 4.3

一幅图像旋转 45° , 输出图像和输入图像大小相同, 超出边界的值被丢弃

一个坐标为 (x_1, y_1) 的点, 以 (x_0, y_0) 为基准点, 旋转 y 角度转换成一个坐标为 (x_2, y_2) 的点的过程, 如下面的公式所示:

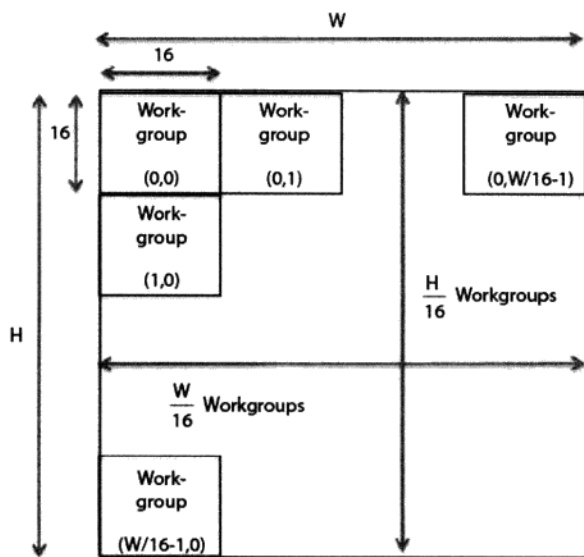
$$\begin{aligned}x_2 &= \cos(\theta) * (x_1 - x_0) + \sin(\theta) * (y_1 - y_0) \\y_2 &= -\sin(\theta) * (x_1 - x_0) + \cos(\theta) * (y_1 - y_0)\end{aligned}$$

如果基准点为原点 $(0, 0)$, 则有如下等式成立:

$$\begin{aligned}x_2 &= \cos(\theta) * (x_1) + \sin(\theta) * (y_1) \\y_2 &= -\sin(\theta) * (x_1) + \cos(\theta) * (y_1)\end{aligned}$$

为了采用 OpenCL 实现图像旋转过程, 我们可以看到, 计算输入矩阵中每个像素点的新坐标 (x, y) 的过程是相互独立的, 每个 work-item 将计算单个像素点的新坐标。类似矩阵相乘的实现过程, 每个 work-item 可以通过全局 ID 获得它所计算的输入像素点的位置, 如图 4.4 所示。

图像旋转是一个很好的输入分解的例子, 它意味着输入(在本例中, 输入是一幅图像)的每一个元素分解到一个 work-item 中。在图像旋转过程中, 如果输入图像和输出图像大小相等, 一些像素点对应的新位置可能会超出输出图像的边界 (如图 4.3 所示, 输入图像的一些顶点落在输出图像的外面)。



输入图像 workgroup 的配置情况

图 4.4

输入图像的每一个元素由一个 work-item 处理, 每一个 work-item 计算其像素的新坐标并输出结果

因此，我们需要检查计算结果是否越界。

```
__kernel void img_rotate(
    __global float* dest_data, __global float* src_data,
    int W, int H,                //Image Dimensions
    float sinItheta, float cosItheta ) //Rotation Parameters
{
    //Work-item gets its index within index space
    const int ix = get_global_id(0);
    const int iy = get_global_id(1);

    //Calculate location of data to move into (ix,iy)
    //Output decomposition as mentioned
    float xpos = ((float)ix)*cosItheta + ((float)iy)*sinItheta;
    float ypos = -1.0*((float)ix)*sinItheta + ((float)iy)*cosItheta;

    //Bound Checking
    if(((int)xpos>=0) && ((int)xpos< W) &&
        ((int)ypos>=0) && ((int)ypos< H))
    {
        // Read (ix,iy) src_data and store at (xpos,ypos) in
        // dest_data
        // In this case, because we rotating about the origin
        // and there is no translation, we know that (xpos,ypos)
        // will be unique for each input (ix,iy) and so each
        // work-item can write its results independently
        dest_data[(int)ypos*W+(int)xpos] = src_data[iy*W+ix];
    }
}
```

如前面的 kernel 代码所示，图像旋转是一个非常容易并行的问题，每个像素点都可以独立计算。主机端主要的步骤类似图 4.2 所示。因此，主机端我们可以大量重用前面矩阵相乘那个例子的代码，包括创建上下文和命令队列的代码。

为了使开发者全面了解 OpenCL，我们采用 OpenCL 1.1 中的 C++封装来编写图像旋转的主机端代码。OpenCL C++封装提供了访问原始 OpenCL C API 底层特性的接口。OpenCL 的 C++绑定兼容标准的 C++编译器，不支持任何内存分配的操作，可以访问 OpenCL 的所有特性，没有不必要的功能屏蔽。

关于 OpenCL 1.1 规范 C++封装 API 的更多细节，可以参见 www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf。

OpenCL 的 C++ 头文件可以通过包含 `cl.hpp` 来获得。为了与 OpenCL 的 C API 相比较, 我们要更简洁地介绍 OpenCL 的 C++ 绑定。接下来采用与矩阵相乘类似的方式来描述使用 OpenCL C++ 绑定 API 的方法。

步骤 1: 建立环境

```
// Discover platforms
cl::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

// Create a context with the first platform
cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[0])(), 0};

// Create a context using this platform for a GPU type device
cl::Context context(CL_DEVICE_TYPE_ALL, cps);

// Get device list from the context
cl::vector<cl::Device> devices =
    context.getInfo<CL_CONTEXT_DEVICES>();

// Create a command queue on the first device
cl::CommandQueue queue = cl::CommandQueue(context,
    devices[0], 0);
```

步骤 2: 声明 buffer, 传输数据

```
// Create buffers for the input and output data ("W" and "H"
// are the width and height of the image, respectively)
cl::Buffer d_ip = cl::Buffer(context, CL_MEM_READ_ONLY,
    W*H*sizeof(float));
cl::Buffer d_op = cl::Buffer(context, CL_MEM_WRITE_ONLY,
    W*H*sizeof(float));

// Copy the input data to the device (assume that the input
// image is the array "ip")
queue.enqueueWriteBuffer(d_ip, CL_TRUE, 0, W*H*
    sizeof(float), ip);
```

步骤 3: 运行时 kernel 编译

```
// Read in the program source
std::ifstream sourceFileName("img_rotate_kernel.cl");

std::string sourceFile{
    std::istreambuf_iterator<char>(sourceFileName),
    (std::istreambuf_iterator<char>())};

cl::Program::Sources rotate_source{1,
    std::make_pair(sourceFile.c_str(),
        sourceFile.length()+1)};
```

```
// Create the program
cl::Program rotn_program(context, rotn_source);

// Build the program
rotn_program.build(devices);

// Create the kernel
cl::Kernel rotn_kernel(rotn_program, "img_rotate");
```

步骤 4: 运行程序

```
// The angle of rotation is theta
float cos_theta = cos(theta);
float sin_theta = sin(theta);

// Set the kernel arguments
rotn_kernel.setArg(0, d_op);
rotn_kernel.setArg(1, d_ip);
rotn_kernel.setArg(2, W);
rotn_kernel.setArg(3, H);
rotn_kernel.setArg(4, cos_theta);
rotn_kernel.setArg(5, sin_theta);

// Set the size of the NDRange and workgroups
cl::NDRange globalws(W,H);
cl::NDRange localws(16,16);

// Run the kernel
queue.enqueueNDRangeKernel(rotn_kernel, cl::NullRange,
    globalws, localws);
```

步骤 5: 将结果读回主机端

```
// Read the output buffer back to the host
queue.enqueueReadBuffer(d_op, CL_TRUE, 0, W*H*sizeof(float), op);
```

从前面的代码可以看出，OpenCL 的 C++ 绑定和 C API 保持着惊人的一致。

图像卷积实例

在图像处理中，卷积是一个很常用的算法。每个像素点通过利用相邻像素点的信息，修改自身的值。一个卷积 **kernel** 或者滤波器描述了每个像素点利用它相邻点的像素值来改变自身值的方法。例如，一个图像模糊化 **kernel** 的每个像素点被设置为其相邻点像素值的加权平均，从而减小像素之间的差值。对相同的源图像，仅仅通过改变滤波器，就可以得到锐化、模糊、边缘增强、浮雕等效果。

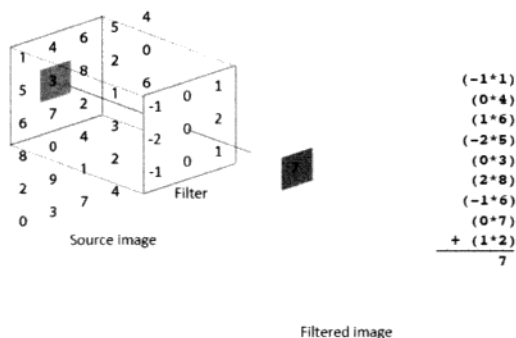


图 4.5

将一个卷积滤波器应用在源图像上

卷积 kernel 在源图像的每个像素点上依次进行计算。对每个源像素点，滤波器以当前像素点为中心覆盖在原始图像上，将自身的值与图像上对应的像素值相乘，再将结果累加起来，产生一个新的值作为当前像素点的值。图 4.5 提供了这个算法的直观图解。源图像如图 4.6A 所示，图 4.6B 显示了使用模糊化滤波器之后的效果。图 4.6C 显示了使用边缘检测滤波器之后的效果。

接下来的代码是一个卷积程序的 C 语言实现。外层的两个循环对源图像进行遍历，选择下一个源像素点。对每一个源像素点及其相邻的像素点使用滤波器的算法。

```
// Iterate over the rows of the source image
for(int i = halfFilterWidth; i < rows - halfFilterWidth; i++) {
    // Iterate over the columns of the source image
    for(int j = halfFilterWidth; j < cols - halfFilterWidth; j++) {
        sum = 0; // Reset sum for new source pixel
        // Apply the filter to the neighborhood
        for(int k = - halfFilterWidth; k <= halfFilterWidth; k++) {
            for(int l = - halfFilterWidth; l <= halfFilterWidth; l++) {
                sum += Image[i+k][j+l] *
                    Filter[k+ halfFilterWidth][l+ halfFilterWidth];
            }
        }
        outputImage[i][j] = sum;
    }
}
```

步骤 1: 创建 image 对象和 buffer 对象

这个例子对于输入图像和输出图像的存储均采用 OpenCL 的 image 类型。采用 image 类型来存储数据有很多优势。对于卷积，计算边界像素值的 work-item 可能会在读数据的时候发生越界。image 类型提供一种机制可以自动的处理这些越界访问并返回有效值。

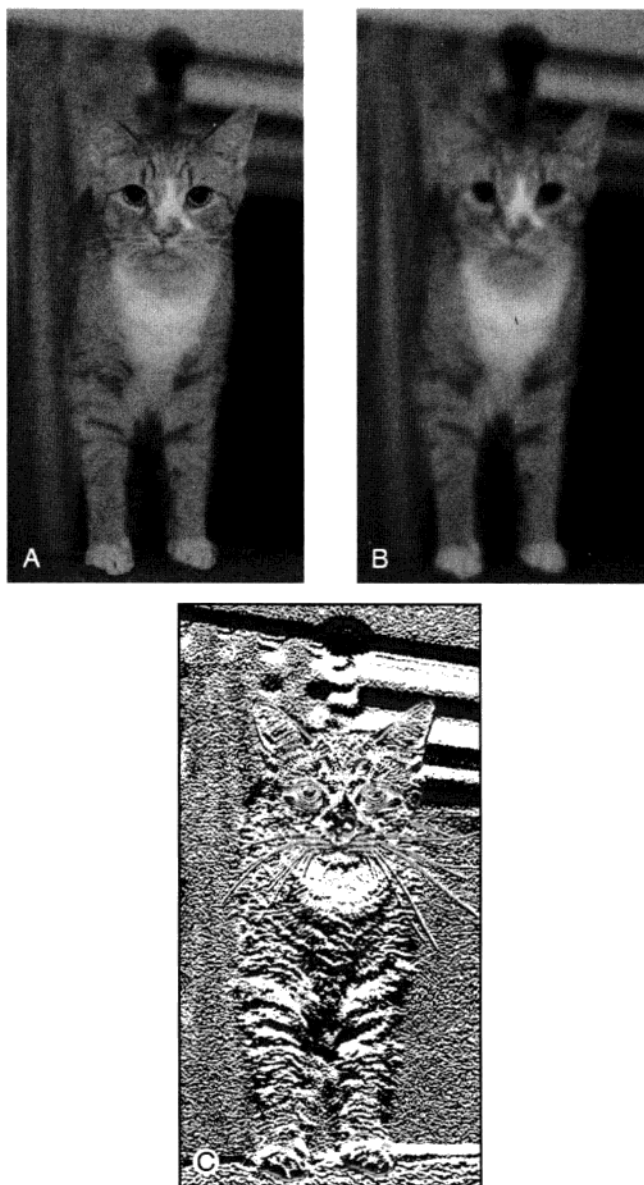


图 4.6

一个模糊化滤波器和一个垂直边界检测滤波器应用于同一个源图像上所产生的效果。(A)源图像；(B)使用模糊化滤波器；(C)使用垂直边界检测滤波器

代码首先假定上下文(context)和命令队列(queue)已经创建好。源图像(sourceImage)、输出图像(outputImage)和滤波器(filter)均已在主机端初始化。每个图像都有 width 和 height 的值。

第一个任务是给源图像、输出图像和滤波器在设备端分配空间。OpenCL 的 image 对象需要一个格式描述符 cl_image_format，用来定义每一个像素数据的大小、类型和通道等。描述符中的 image_channel_order，表示通道的顺序。回想第 2 章中，图像中每个元素存储在四个通道中，每个通道分别代表 RGBA 中的一个。一幅每个像素有四个值的图像可以使用 CL_RGBA 作为通道的顺序。如果每个 work-item 仅仅访问一个值(比如灰度图或者矩阵的元素)这些数据只需要一个通道就可以了，可以采用 CL_R 表示。本例中假设采用灰度图，因此只需要一个通道。数据的类型存储在 cl_image_format 类型描述符的 image_channel_data_type 中。整数采用数据符号和数据大小的组合来表示。例如 CL_SIGNED_INT32 表示一个 32 位有符号整数。CL_UNSIGNED_INT8 等同于 C 语言中的无符号字符(unsigned char)类型。浮点数采用 CL_FLOAT 表示，本例中数据表示就是采用这种类型。

创建 image 的格式描述符之后，通过 clCreateImage2D()函数，我们创建了一个代表 image 的内存对象。此外还创建一个滤波器所需要的 buffer 对象，这个 buffer 对象最终将采用常量内存。

```
// The convolution filter is 7x7
int filterWidth = 7;
int filterSize = filterWidth*filterWidth; // Assume a square kernel

// The image format describes how the data will be stored in memory
cl_image_format format;
format.image_channel_order = CL_R; // single channel
format.image_channel_data_type = CL_FLOAT; // float data type

// Create space for the source image on the device
cl_mem bufferSourceImage = clCreateImage2D(
    context,
    0,
    &format,
    width,
    height,
    0,
    NULL,
    NULL);

// Create space for the output image on the device
cl_mem bufferOutputImage = clCreateImage2D(
```

```
context,
0,
&format,
width,
height,
0,
NULL,
NULL);

// Create space for the /x/ filter on the device
cl_mem bufferFilter = clCreateBuffer(
context,
0,
filterSize*sizeof(float),
NULL,
NULL);
```

步骤 2: 写输入数据

调用函数 `clEnqueueWriteImage()` 将一个 `image` 拷贝到设备端。与 `buffer` 不同的是，拷贝一个 `image` 需要提供一个三维的偏移和范围，用于在拷贝的时候确定每一个维度从何处开始拷贝以及拷贝的范围。

和前面的一些例子一样，滤波器采用 `clEnqueueWriteBuffer()` 函数来拷贝。

```
// Copy the source image to the device
size_t origin[3] = {0, 0, 0}; // Offset within the image to copy from
size_t region[3] = {width, height, 1}; // Elements to per dimension
clEnqueueWriteImage(
queue,
bufferSourceImage,
CL_FALSE, origin,
region,
0,
0,
sourceImage,
0,
NULL,
NULL);

// Copy the /x/ filter to the device
clEnqueueWriteBuffer(
queue,
bufferFilter,
CL_FALSE,
0,
filterSize*sizeof(float),
filter,
```

```
0,
NULL,
NULL);
```

步骤 3: 创建 sampler 对象

在 OpenCL 中, sampler(采样器)是描述如何访问 image 的对象。sampler 规定了图像坐标系的类型和访问越界时的处理方法, 或访问落在多个坐标点之间时是否进行插值等。clCreateSampler()函数的调用格式如下:

```
cl_sampler clCreateSampler(
    cl_context context,
    cl_bool normalized_coords,
    cl_addressing_mode addressing_mode,
    cl_filter_mode filter_mode,
    cl_int *errcode_ret)
```

坐标系可以使用归一化(如: 将坐标值归一化到 0-1 之间)或者标准的坐标系。将第二个参数设置为 CL_TRUE 可以将坐标系归一化。卷积通常不用归一化的坐标系, 因此这个参数设置为 FALSE。

OpenCL 也支持多种处理越界访问的方式。在这个卷积例子中, 我们采用 CL_ADDRESS_CLAMP_TO_EDGE 作为处理越界访问的方式, 访问越界的时候返回图像的边界值。如果采用 CL_ADDRESS_CLAMP 的方式, 当访问越界的时候, 通道为 RG 或 B 返回 0 值, 通道为 A 时返回 0 或者 1(与图像格式有关)。如果采用归一化的坐标系, 还会有更多其他的越界处理方式。

当访问位置处在多个坐标值之间的时候, 滤波器模式可以访问最近的一个坐标值, 也可以利用周围的坐标值进行插值。

```
// Create the image sampler
cl_sampler sampler = clCreateSampler(
    context,
    CL_FALSE,
    CL_ADDRESS_CLAMP_TO_EDGE,
    CL_FILTER_NEAREST,
    NULL);
```

步骤 4: 编译并执行 kernel

这一步创建并编译 program, 创建 kernel, 设置 kernel 参数, 并与前面的两个例子相似, 将 kernel 加入命令队列去执行。不同于前面提到的 C 语言版本, 这里的 OpenCL 代码采用 image 类型, 因此需要创建和图像像素相同数量的 work-item。基于 sampler 对象, 任何由于滤波器大小而导致的越界访问的处理都是自动进行的。

步骤 5: 读取结果

除了需要提供一个存储结果的指向主机端内存的指针以外, 将结果读回主机端与写入 image 的过程类似。

```
// Read the output image back to the host
clEnqueueReadImage(
    queue,
    bufferOutputImage,
    CL_TRUE,
    origin,
    region,
    0,
    0,
    outputImage,
    0,
    NULL,
    NULL);
```

卷积 kernel

如果理解了前面引用的 C 代码, 那么下面的 kernel 就很好理解。每一个 work-item 都会执行最内层的两个循环。根据数据类型使用相应的 OpenCL 函数, 从原始 image 读取数据。在这个例子中, 使用的是 read_imagef() 函数, f 表示读取的数据是单精度浮点数。访问 image 时, 一般会返回一个长度为 4 的向量(向量中每个元素为一个通道), 因此 pixel(访问 image 返回的值) 和 sum(写入到输出 image 的数据)都必须声明为 float4 类型。写入输出 image 也使用一个相应的函数, write_imagef(), 要求写入的数据要按照一定的格式(如 float4)。写入不支持越界访问。在 NDRange 的任何一个维度, 如果 work-item 数量可能超出像素的数量, 则必须在写入输出数据之前进行边界检查。

由于所有 work-item 在每一次迭代中都访问滤波器的同一个元素, 因此本例中的滤波器数组非常适合使用常量内存。常量内存的使用非常简单, 只需在函数的 filter 参数前面加上 __constant 关键词, 滤波器可以放在常量内存中。

```
__kernel
void convolution(
    __read_only image2d_t sourceImage,
    __write_only image2d_t outputImage,
    int rows,
    int cols,
    __constant float* filter,
    int filterWidth,
    sampler_t sampler)
{
    // Store each work-item's unique row and column
```



```

int column = get_global_id(0);
int row = get_global_id(1);

// Half the width of the filter is needed for indexing
// memory later
int halfWidth = (int)(filterWidth/2);

// All accesses to images return data as four-element vector
// (i.e., float4), although only the 'x' component will contain
// meaningful data in this code
float4 sum = {0.0f, 0.0f, 0.0f, 0.0f};

// Iterator for the filter
int filterIdx = 0;

// Each work-item iterates around its local area based on the
// size of the filter
int2 coords; // Coordinates for accessing the image
// Iterate the filter rows
for(int i = -halfWidth; i <= halfWidth; i++) {
    coords.y = row + i;

    // Iterate over the filter columns
    for(int j = -halfWidth; j <= halfWidth; j++) {
        coords.x = column + j;

        float4 pixel;
        // Read a pixel from the image. A single channel image
        // stores the pixel in the 'x' coordinate of the returned
        // vector.
        pixel = read_imagef(sourceImage, sampler, coords);
        sum.x += pixel.x * filter[filterIdx++];
    }
}

// Copy the data to the output image if the
// work-item is in bounds
if(myRow < rows && myCol < cols) {
    coords.x = column;
    coords.y = row;
    write_imagef(outputImage, coords, sum);
}
}

```

编译 OpenCL 主机端应用

在GPU上运行OpenCL程序，需要有支持OpenCL的显卡驱动程序。按照AMD规范编写的OpenCL程序只需要安装一个运行时环境而不需要任何驱动即可在X86 CPU上运行。

编译 OpenCL 程序与编译其他需要动态链接库的程序类似。在编译和链接生成 OpenCL 可执行程序的时候，需要链接上各个厂商发布的 OpenCL 库。在编译 OpenCL 程序的时候，需要设置 include 路径来找到所需要的 OpenCL 头文件(cl.h 或 cl.hpp)。链接器需要知道如何找到 OpenCL 库函数(如果是 Windows，是 OpenCL.lib；如果是 Linux，是 libOpenCL.a)。

假设 OpenCL 安装在\$(AMDAPPSDKROOT)，在 Linux 上的编译命令示例如下：

```
$ g++ -o prog -I$(AMDAPPSDKROOT)/include -L$(AMDAPPSDKROOT)/lib/  
x86_64 -lOpenCL prog.cpp
```

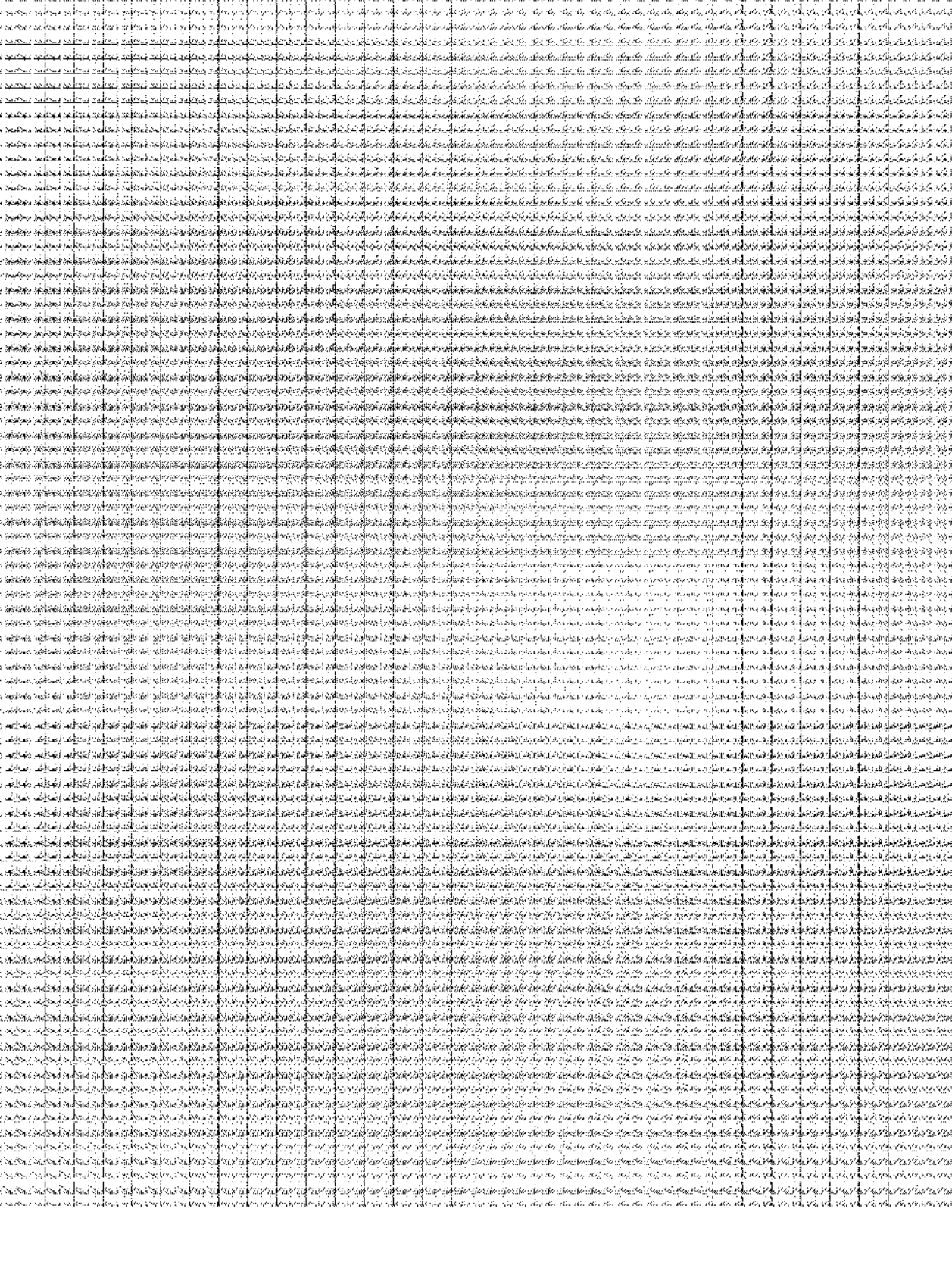
鉴于 OpenCL 应用程序中大部分的步骤都是类似的，因此我们可以重用很多代码。采用 C++封装 API 的程序，编译方法和前面相同。OpenCL 的 C++头文件一般与 C 头文件放在同一个目录下。

小结

在这一章中，我们主要介绍了几个常见的数据并行算法的实现过程。我们学习了 OpenCL buffer 和 image 对象的使用，也使用 OpenCL C++封装 API 实现了图像旋转算法。

在每个例子中，虽然对输入数据的要求不同，但每个 work-item 仅计算一个输出元素。图像旋转的例子每产生一个结果只需要一个输入元素。在矩阵相乘的例子中，每个 work-item 的计算使用输入矩阵 A 的一整行和 B 的一整列来得到输出矩阵的一个元素。卷积需要输入像素的相邻值来计算一个输出像素值。

本章所介绍的例子都适合数据并行的 OpenCL 程序，但是它们的性能还可以针对特定硬件平台进行优化，这也是后续章节的目标。



OpenCL 的并发与执行模型

引言

正如第 3 章讨论过的那样，OpenCL 支持的设备种类非常多。为实现对如此多设备的支持，OpenCL 内存模型和执行模型的定义至为重要，即在不需繁重编程负担的情况下，在不同的硬件架构上都能实现较高的性能。本章将深入研究这些模型，并在第 6 章展示它们如何映射到一个支持 OpenCL 的特定硬件架构上。

kernel, work_item, workgroup 和执行域

Kernel 这个概念是 OpenCL 执行的核心。kernel 是指一个用 OpenCL C 语言编写的、代表一个单一执行实例的代码单元。在 OpenCL C 语言中，kernel 看起来与 C 语言非常相像，都有一个参数列表“局部”变量(定义在私有地址空间，正如后文所示)和标准控制流结构。OpenCL 术语中将这种 kernel 实例称为 work-item(工作项)。

OpenCL kernel 与 C 语言函数的区别在于其并行语义。任何给定的 kernel 实例或者 work-item 是定义在一个很大的并行执行空间中的一小部分。当运行时系统处理执行队列中的事件时会调度 kernel 执行。队列中的事件是在队列对象上调用 enqueueNDRange 函数而创建。kernel 启动后会创建大量线程(work-item)同时执行，以完成 kernel 函数定义的操作。就像入队调用所指出的那样，这个调度创建了一个由 work-item 组成的 NDRange(n-dimensional range, n 维空间)。NDRange 定义了一个一维、二维或三维的 work-item 网格，为并行执行提供了一个简单直观的结构。当映射到 OpenCL 硬件模型上时，每一个 work-item 运行在一个被为处理基元(processing element)的抽象硬件单元上，其中每个处理基元可以处理多个 work-item。

在 kernel 调度运行中, 每一个 work-item 都是相互独立的, OpenCL 没有定义 work-item 之间的同步。这种宽松的执行模型使得 OpenCL 可以扩展到拥有大量处理核心的设备上。不过, 这些硬件实际上都有一系列层次化的执行设备, 特别是一个层次化的存储结构。

为能灵活支持各种设备, OpenCL 将全局执行空间划分为大量大小相等的, 一维、二维或者三维的 work-item 集合, 称为 work-group。在 work-group 内部, 各个 work-item 之间允许一定程度的通信。根据 OpenCL 标准的定义, 一个 work-group 可并发运行在设备层次结构中一个称为计算单元(compute unit)的硬件单元上。这种形式的并发执行对 work-item 之间的同步是至关重要的。work-group 通过保证并发执行来允许其内部 work-item 之间的本地同步, 但会限制相互通信以提高扩展性。在其执行空间中存在全局通信的应用程序很难在基于 OpenCL 的多核设备上并行化, 为满足全局通信, 计算单元将被映射到一个核上。

通过允许定义比实际可并行运行的 work-item 数目更多的线程, OpenCL kernel 可扩展到支持更多线程的设备上, 从而使得更多的 work-group 以及更多的 work-item 都能并行执行。然而出于性能方面的考虑(如同 OpenMP 与 MPI 一样), 明智的选择是只启动足够能并行运行的任务, 因为这样可以对程序优化进行更直接的控制。

已审阅
Reviewed by: 1402.21.1017400

正如在第 2 章中讨论的那样, OpenCL work-item 试图表达能够被 Win32 或 POSIX 线程或 OpenMP 所能表达的并行性。OpenCL 在设计上更进一步: work-group 内的 work-item 可以映射到数量更少的硬件线程的上下文中。这可以看作是在单指令多数据(single instruction multiple data, SIMD)或者流水线的向量执行中, 使用多个时钟周期来运行一条长向量指令。与之不同的是, 在 OpenCL 中, 子向量可以保持自己的程序计数器直到同步点。最好的例子是在 GPU 上, 多达 64 个 work-item 组织为一个硬件线程运行在 SIMD 单元上: 在 AMD 硬件架构上, 称为一个 wavefront, 在 NVIDIA 硬件架构上被称为一个 warp。这种设计使得通过 lane-wise(类多道)编程的 SIMD 执行模型, 与在 x86 处理器上直接使用 SIMD 指令进行 SSE 编程相比, 更为简单。需要指出的是, 由于这种 SIMD 执行模式, 对于一个给定的设备, OpenCL 调度宽度应该是 SIMD 宽度的整数倍。这个值可通过向 clGetKernelWorkGroupInfo 函数传递参数 CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE 来得到。

OpenCL 定义了能在 kernel 内部调用的函数, 用来获取 work-item 在整个执行空间中的位置。这些函数以维度值作为参数, 即下面列出的“uint dimension”参数。这指的是多维 NDRange 中维度的索引值。

- `uint get_work_dim()`: 返回线程调度的维度数。
- `uint get_global_size(uint dimension)`: 返回所请求维度上 work-item 的总数。
- `uint get_global_id(uint dimension)`: 返回在所请求的维度上当前 work-item 在全局空间中的索引。
- `uint get_local_size(uint dimension)`: 返回在所请求维度上 work-group 的大小。
- `uint get_local_id(uint dimension)`: 返回在所请求的维度上, 当前 work-item 在 work-group 中的索引。
- `uint get_number_groups(uint dimension)`: 返回在所请求维度上 work-group 的数目。这个值等于 `get_global_size` 除以 `get_local_size`。
- `uint get_group_id(uint dimension)`: 返回在所请求的维度上当前 work-group 在全局空间中的索引。

举一个简单的 kernel 执行的例子。该 kernel 运行在二维索引空间上, 其作用是将输入数组的元素值乘以 2, 然后赋给输出数组。图 5.1 显示了这个 kernel 的实际执行过程。可以看出, 每一个 work-item 调用 `get_global_id` 和 `get_global_size` 函数返回不同的值, 这些值反映了 work-item 在迭代空间上的位置。在这个简单的例子中, 我们将索引空间中的位置直接映射到一个二维数据结构上。在实际例子中, 根据输入/输出数据的结构和算法处理数据方式的不同, 可能会有更为复杂的映射方式。

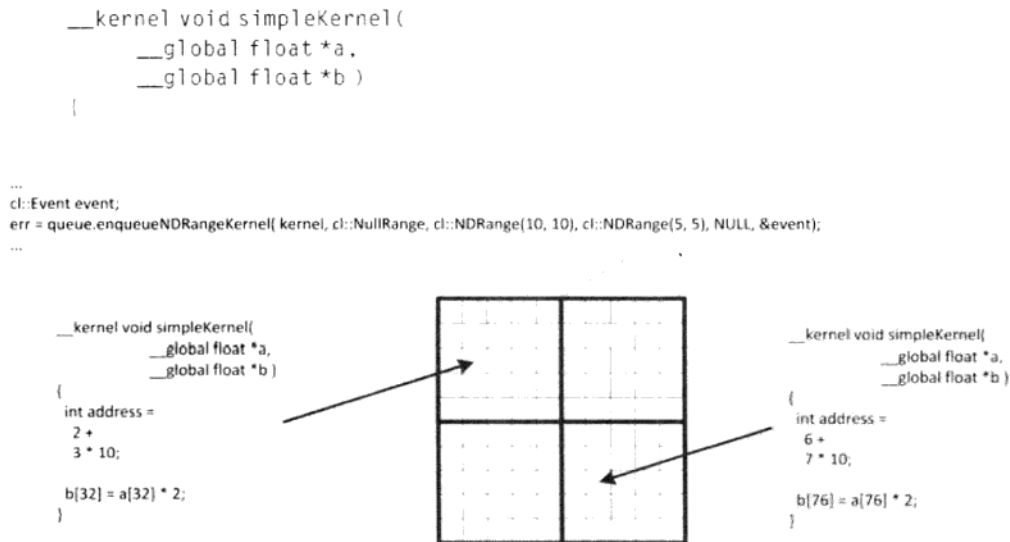


图 5.1

执行一个简单的 kernel, 显示了网格中的 kernel 实例

```
int address =  
    get_global_id(0) +  
    get_global_id(1) * get_global_size(0);  
b[address] = a[address] * 2;  
}
```

OpenCL 同步: kernel, fence 和 barrier

在 OpenCL 最简单的模型中, 每个 work-item 的执行是相互独立的。一个 work-item 的写操作和另一个 work-item 的读操作是没有顺序保证的。OpenCL 采用宽松的同步模型和内存一致性模型。尽管实际硬件实现能保证一些顺序语义, 但作为一个跨平台的 API, 并不保证满足这些顺序语义。OpenCL 的解决办法是在程序员需要确切知道当前系统状态的地方定义同步点, 从而可以依据这些信息获取预期的行为。

因为 OpenCL 不仅运行在由操作系统管理线程的设备上(例如主流的 x86 CPU), 还会运行在由硬件管理线程的设备上(例如 GPU), 所以我们在保证线程的全并发执行时, 要特别小心。一个 x86 线程在信号量不可用的情况下, 有可能降低信号量并阻塞。操作系统可以终止并随时调度线程运行, 几乎不考虑资源限制。在 GPU 上, 如果采用相同的方式调度线程(在 AMD 硬件上称为 wavefront)是有问题的, 因为线程占用的资源是固定的, 并且是有限制的。停止一个 wavefront 的运行并不会释放它所占用的资源。因此, 在 GPU 上有可能出现这样一种状况: 一个已经调度在设备上的 wavefront 需要得到另外一个还没有调度到设备上的 wavefront 释放信号量才能开始运行。wavefront 在得到这个信号量前是不会执行的, 造成系统死锁。

为避免这种情况的发生, 全局同步只定义在 kernel 执行的边界处。也就是说, 位于不同 work-group 的两个 work-item, 并没有定义方法来确保它们的执行顺序。为实现数据共享, 主要是实现同一个 work-group 内的 work-item 在本地内存上的数据共享, OpenCL 定义了 work-group 内部的同步操作。当一个 work-item 调用 barrier 函数时, 在同一个 work-group 内的所有 work-item 都到达这个 barrier 函数之前, 是不会继续执行的。这是程序计数器层次的限制, 表明代码中的每一个 barrier 操作都将作为不同的同步点。因此, 如果在 kernel 代码的条件分支中调用 barrier 操作, 就要确保 work-group 内所有的 work-item 都执行这个分支。当存在执行其他分支的 work-item 时, barrier 行为是不

确定的，在很多设备上，这会导致死锁。

图 5.2 显示了 OpenCL 同步的一个简单例子。在图中，我们可以看到一个 kernel 队列，它包含 4 个 work-group，每个 work-group 有 8 个 work-item。按照 OpenCL 规范的宽松解释(例如，不考虑硬件的具体实现)，每一个 work-group 内每一个 work-item 的执行速率是不一样的。当发出同步指令时，运行较快的 work-item 只有在其他 work-item 都达到这个同步点之后才能继续执行。在整个 kernel 运行完成之前，不同 work-group 与在其他 work-group 内 work-item 的调度运行完全无关。在前后两个 kernel 调度之间，当前 kernel 的所有工作必须完成而且内存数据处于一致状态时，才能启动下一个 kernel，并遵循相同执行语义。

如果我们假设插入队列的 kernel 0 和 1 都来自于同一个 kernel 对象，那么下面的 kernel 代码和 API 调用能产生如图 5.2 所示的行为。在这里，我们看到的 work-item 的行

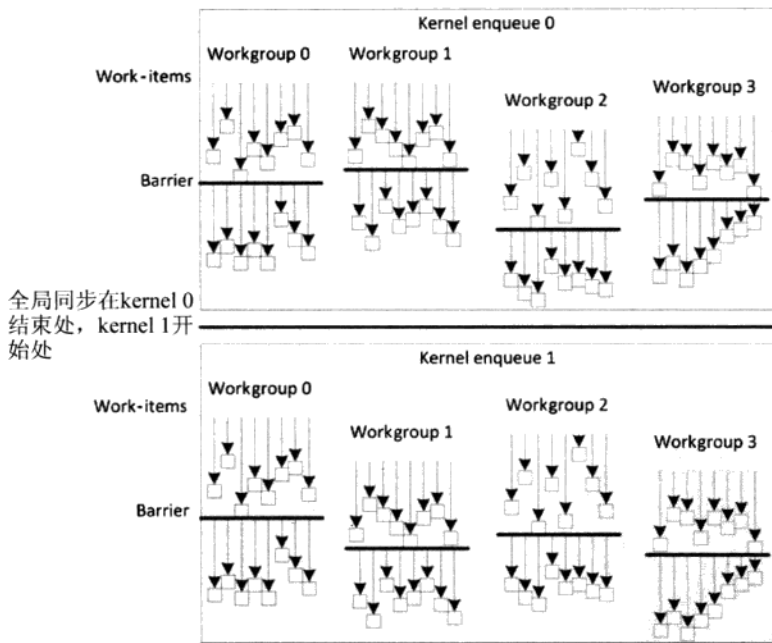


图 5.2

OpenCL 的同步操作。在单个 kernel 调度运行的过程中，仅可以通过 barrier 来保证 workgroup 内部线程间的同步。全局同步通过 kernel 的完成来维护，kernel 完成要保证当前 kernel 工作的全部完成以及内存一致性

为是将存储在本地内存中的周围元素相加，并在其他 **work-item** 读取之前，保证这些数据可用。注意在这个例子中，分配到 **kernel** 对象的参数是持久的，因此不需要重复设置。这与 C 和 C++ API 是一样的。

```
// Host code
...
cl_mem input = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    10*sizeof(float),
    0,
    0);
cl_mem intermediate = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY,
    10*sizeof(float),
    0,
    0);
cl_mem output = clCreateBuffer(
    context,
    CL_MEM_WRITE_ONLY,
    10*sizeof(float),
    0,
    0);
clEnqueueWriteBuffer(
    queue,
    input,
    CL_TRUE,
    0,
    10*sizeof(int),
    (void *)hostInput,
    0,
    NULL,
    NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&input);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&intermediate);
clSetKernelArg(kernel, 2, 2*sizeof(float), 0);

size_t localws[1] = {2};
size_t globalws[1] = {10};

clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
```

```
    globalws,
    localws,
    0,
    NULL,
    NULL);

clSetKernelArg(kernel, 0, sizeof(cl_mem), (void*)&intermediate);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void*)&output);

clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    globalws,
    localws,
    0,
    NULL,
    NULL);

clEnqueueReadBuffer(
    queue,
    output,
    CL_TRUE,
    0,
    10*sizeof(float),
    (void*)&hostOutput,
    0,
    NULL,
    NULL);

...

// Kernel
__kernel void simpleKernel(
    __global float *a,
    __global float *b,
    __local float *l )
{
    l[get_local_id(0)] = a[get_global_id(0)];

    barrier(CLK_LOCAL_MEM_FENCE);

    unsigned int otherAddress =
        (get_local_id(0) + 1) % get_local_size(0);

    b[get_local_id(0)] = l[get_local_id(0)] + l[otherAddress];
}
```

队列与全局同步

OpenCL 是基于任务并行、主机端控制的模型，其中每一项任务都是数据并行的。具体通过使用和设备关联的线程安全的命令队列来实现。算法内核(kernel)、数据传输以及其他操作并不是简单地通过用户调用运行时函数实现的，而是首先使用异步入队操作加入到一个指定的队列中，然后在将来的某个时间点执行。

能够加入 OpenCL 命令队列中的命令如下：

- kernel 执行命令
- 内存操作命令
- 同步命令

所有的 kernel 执行命令和同步命令都是异步入队的。在主机端程序来看，一个命令的完成只能在同步点得到保证，主要的同步点有下面几个。

- 调用 `clFinish` 函数，该函数将阻塞程序的执行直到命令队列中所有的命令都完成执行。
- 等待一个特定事件的完成。
- 执行一个阻塞访存操作。

最后一个同步方式是最简单的，经常用在简单的 OpenCL 例子中。下面的程序片段将一系列命令异步入队，并启动一个阻塞访存操作来执行与主机端的同步。

```
// Perform setup of platform, context and create buffers
...
// Create queue leaving parameters as default so queue is in-order
queue = clCreateCommandQueue( context, devices[0], 0, 0);
...

clEnqueueWriteBuffer(
    queue,
    bufferA,
    CL_FALSE,
    0,
    10 * sizeof(int),
    a,
    0,
    NULL,
    NULL);
clEnqueueWriteBuffer(
    queue,
    bufferB,
    CL_TRUE,
    0,
    10 * sizeof(int),
    b,
```

```
0,
NULL,
NULL);

// Set kernel arguments
...

size_t localws[1] = {2}; size_t globalws[1] = {10};
clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    globalws,
    localws,
    0,
    NULL,
    NULL);

// Perform blocking read-back to synchronize
clEnqueueReadBuffer(
    queue,
    bufferOut,
    CL_TRUE,
    0,
    10 * sizeof(int),
    out,
    0,
    0,
    0);
```

enqueueReadBuffer 函数的第二个参数为 **CL_TRUE**，指明读内存为同步操作，在数据拷贝完成之前，该函数将一直阻塞。为了能正确读回数据，在拷贝操作之前的所有命令都必须正确完成。如果我们将这个参数设置为 **CL_FALSE**，就需要进一步的同步操作，最简单的方式就是在这个队列上调用 **clFinish** 函数：

```
clEnqueueReadBuffer(
    queue,
    bufferOut,
    CL_FALSE,
    0,
    10 * sizeof(int),
    out,
    0,
    0,
    0);
clFinish(queue);
```

OpenCL 内存一致性

OpenCL 同步操作不仅应用于工作完成时的同步，而且还用来保证内存数据的正确视图。为能更加有效地映射到各种设备上，OpenCL 采用了宽松的内存一致性模型。在 OpenCL 模型中，任何被多个入队命令共享的内存对象只在同步点保证一致性。这就意味着命令之间的一致性，甚至通信的正确性，在一个有序队列的命令间或者在产生事件和等待该事件的两个命令之间是有保证的。

即使在这种情况下，也只是在运行时保持内存对象的一致性，而对于主机端的 API 来说，则是不可见的。要保证主机端 API 的正确性，用户必须使用前面所讨论的阻塞操作。例如，`clFinish` 函数将在指定队列中所有操作完成之前一直阻塞，以此保证队列中的所有操作所使用内存的一致性。

设备之间也存在同样的一致性问题。由于内存对象是与上下文相关而不是与设备相关，所以在数据共享和发生相应的事件时，OpenCL 运行时要保证这些对象在不同设备间的一致性。数据在不同设备间的移动是这样的：如果 kernel 在第二个设备上运行，那么在第一个设备上产生的任何结果数据在第二个设备上都是随时可用的。第一个数据结构上事件的完成标志着数据已经可以移动，不需要进行单独的缓冲区拷贝操作。

事件

注意，在构建命令队列时所忽略的最后两个参数将采用默认值。该位域的属性之一是让队列中的任务乱序执行(`CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE`)。因此，虽然前面的例子中队列是有序的，但也可以让队列中的任务乱序执行。

乱序队列并没有定义默认的任务运行顺序。如果运行时存在乱序队列，如一个 DMA 引擎可以在计算单元上并行执行，或者这个设备能同时运行多个 kernel，它将自由地以并行方式安排这些操作的执行而不用保证一个操作必须在另一个操作完成后才能运行。类似的，如果运行时系统有多个队列，不管是在单个设备上还是在多个设备上，这些队列操作的执行都是没有默认执行顺序的。

无论哪种情况，要正确执行一个这样的结构都需要构建一个任务图。在 OpenCL 中，任务图通过事件对象构建。OpenCL 的事件模型允许构建复杂的任务图，这个任务图将连接进入与给定的 OpenCL 上下文相关队列中的任何任务。入队函数的最后一个参数可以指定一个封装了入队命令状态的事件。最重要的是，这个事件不仅注册任务的完成，

而且也保证被此任务访问的所有内存数据的一致性。队列列表可以作为一个依赖性列表传递给入队函数。这意味着, 在所有的事件执行完成之前, 命令将不会执行。下面的代码包含一个无序队列并显式定义了所有的依赖关系。图 5.3 是同一个队列的图解。正如在例子中看到的那样, 直接使用事件是与主机同步的第三种方法, 在这种情况下, 我们在读事件上调用了 `wait()` 函数。

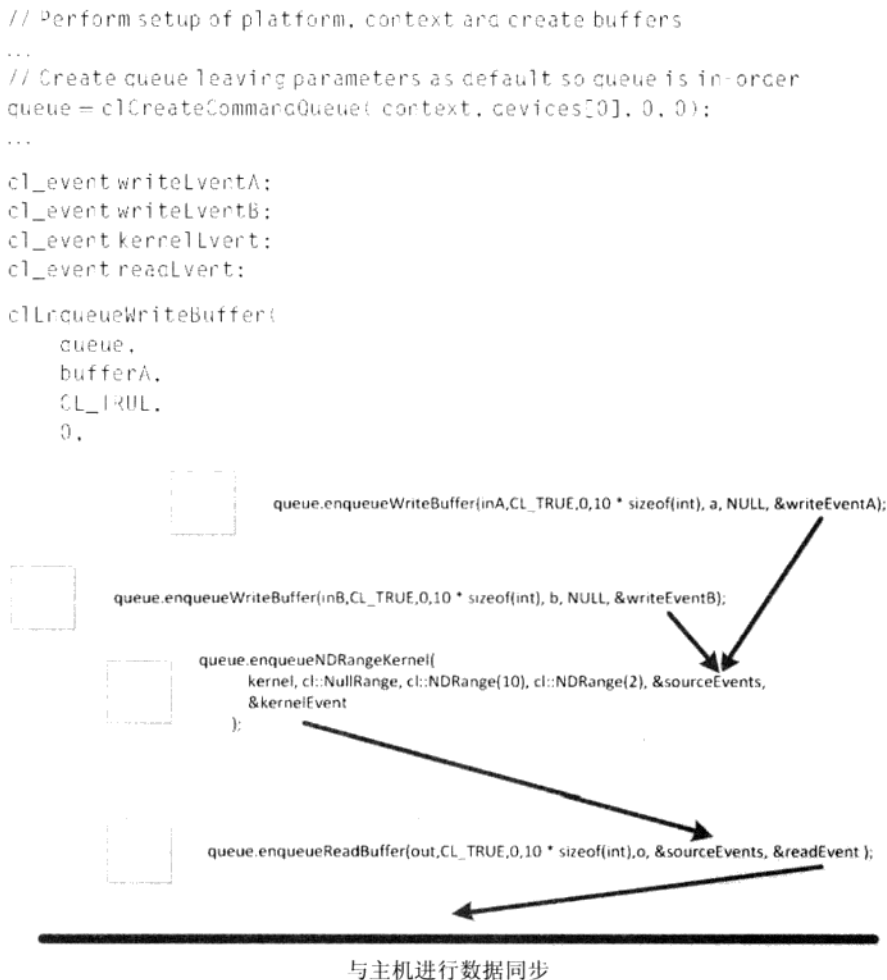


图 5.3

通过向相关入队函数传递事件参数, 一个入队命令可依赖于一组入队命令

```

        10 * sizeof(int),
        a,
        0,
        NULL,
        &writeEventA);
clEnqueueWriteBuffer(
    queue,
    bufferB,
    CL_IRUL,
    0,
    10 * sizeof(int),
    b,
    0,
    NULL,
    &writeEventB);

// Set kernel arguments
...
size_t localws[1] = {2}; size_t globalws[1] = {10};

// Wait on both writes before executing the kernel
cl_event eventList[2];
eventList[0] = writeEventA;
eventList[1] = writeEventB;
clEnqueueNDRangeKernel(
    queue,
    kernel,
    1,
    NULL,
    globalws,
    localws,
    2,
    eventList,
    &kernelEvent);
// Decrease reference count on events
clReleaseEvent(writeEventA);
clReleaseEvent(writeEventB);

// Read will wait on kernel completion to run
clEnqueueReadBuffer(
    queue,
    bufferOut,
    CL_IRUL,
    0,
    10 * sizeof(int),
    out,
    1,
    &kernelEvent,
    &readEvent);

```

```
clReleaseEvent(kernelEvent);  
  
// Block until the read has completed  
clWaitForEvents(1, &readEvent);  
clReleaseEvent(readEvent);
```

多个设备上的队列

了解 OpenCL 的同步功能以及主机端内存模型对于管理多个命令队列是非常有必要的。可以将多个队列映射到相同的设备上, 实现命令间或者命令和主机-设备通信的重叠执行。如果我们在一个系统中有多个设备(例如, 一个 CPU 和一个 GPU 或者多个 GPU), 每个设备都需要自己的命令队列。

图 5.4 显示了一个关联了两个设备的 OpenCL 上下文。每一个设备都有自己单独的命令队列。下面的代码说明了如何在同一个上下文中创建两个命令队列。在这里特别值得注意的是, 使用 OpenCL 事件只能实现在同一个上下文中的命令间的同步。如果为不同的设备创建了单独的上下文, 则不可能使用事件进行同步, 此时设备间共享数据的唯一方式是调用 `clFinish` 函数, 然后在缓存对象间显式地拷贝数据。

```
cl_uint num_devices;  
cl_device_id devices[2];  
cl_context ctx;  
  
//Obtain devices of both CPU and GPU types
```

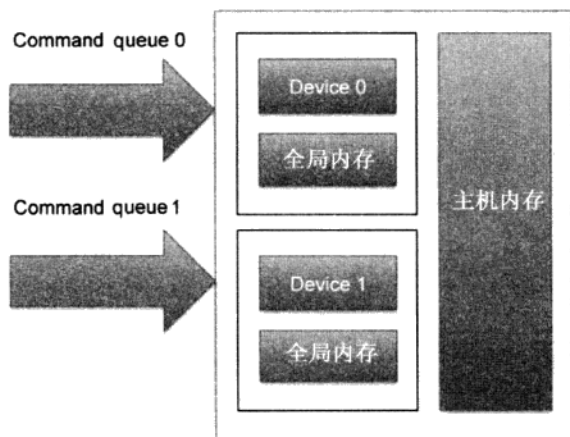


图 5.4

在同一上下文中为不同设备创建多个命令队列


```

err_code = clGetDeviceIDs(
    NULL,
    CL_DEVICE_TYPE_CPU,
    1,
    &devices[0],
    &num_devices);
err_code = clGetDeviceIDs(
    NULL,
    CL_DEVICE_TYPE_GPU,
    1,
    &devices[1],
    &num_devices);
//Create a context including two devices
ctx = clCreateContext(0, 2, devices, NULL, NULL, &err);
cl_command_queue queue_cpu, queue_gpu;

//Create queues to each device
queue_cpu = clCreateCommandQueue(context, devices[0], 0, &err);
queue_gpu = clCreateCommandQueue(context, devices[1], 0, &err);

```

使用 OpenCL 进行多设备编程，在异构并行编程框架下可归结为两种执行模型。

- 两个或多个设备以流水线的形式工作，一个设备等待另一个设备的运行结果。

如图 5.5 所示。

- 多个设备相互独立运行的任务模型。如图 5.6 所示。

在下面代码中，等待列表的有序执行使得 GPU 队列中的 kernel 全部完成后才开始执行 CPU 队列中的 kernel。

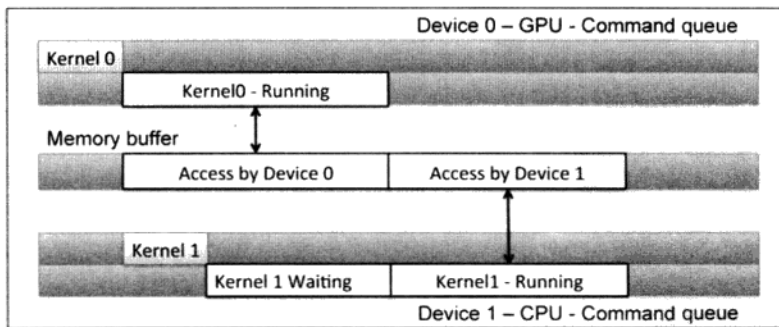


图 5.5

多个设备以相互协作的方式操作相同数据。CPU 队列一直等待直到 GPU kernel 完成执行

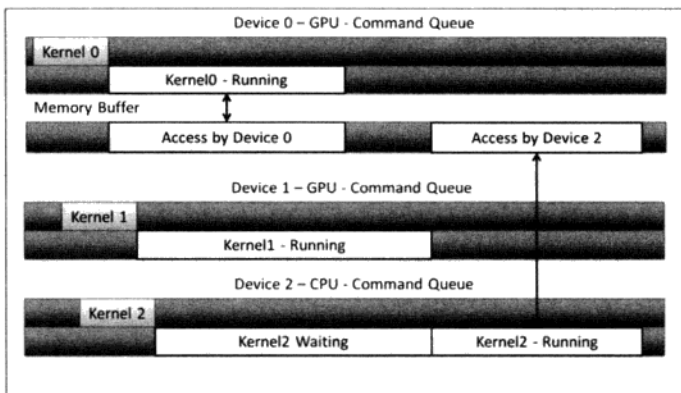


图 5.6

多个设备以并行方式工作。在这种情况下，两个 GPU 使用不同的缓冲区独立执行。CPU 队列一直等待直到两个 GPU 设备上的 kernel 完成执行

```

/// A collaborative - pipelined model of multidevice execution
/// The enqueued kernel on the GPU command queue waits for the kernel on
the CPU
/// command queue to finish executing
cl_event event0_cpu, event1_gpu;

// Starts as soon as enqueued
err = clEnqueueNDRangeKernel(
    queue_gpu,
    kernel1_gpu,
    2,
    NULL,
    global,
    local,
    0,
    NULL,
    &event_gpu);
// Starts after event_gpu is on CL_COMPLETE
err = clEnqueueNDRangeKernel(
    queue_cpu,
    kernel2_cpu,
    2,
    NULL,
    global,
    local,
    1,
    NULL,
    &event0_cpu);

```

```

        &event_gpu,
        &event_cpu);
    clFlush(queue_cpu);
    clFlush(queue_gpu);

```

下面的代码说明了在不同设备上并行执行 **kernel** 的执行模型。一个多设备并行执行的例子如图 5.6 所示，**kernel** 在两个 GPU 上独立运行。这两个设备需要独立的缓冲区，因为只有这样，它们才能并行执行。

```

// Parallel multidevice execution
// We would need to create 3 command queues in this case
// 2 queues for 2 GPUs and 1 queue for the CPU
// The enqueued kernel on the CPU command queue waits
// for the kernels on the GPU command queues to finish

cl_event event_gpu[2];

// Both the GPU devices can execute concurrently as soon as they have
// their respective data since they have no events in their waitlist
err = clEnqueueNDRangeKernel(
    queue_gpu_0,
    kernel_gpu,
    2,
    NULL,
    global,
    local,
    0,
    NULL,
    &event_gpu[0]);
err = clEnqueueNDRangeKernel(
    queue_gpu_1,
    kernel_gpu,
    2,
    NULL,
    global,
    local,
    0,
    NULL,
    &event_gpu[1]);

// The CPU will wait till both GPUs are done executing their kernels
// Two events in the CPU's waitlist
err = clEnqueueNDRangeKernel(
    queue_cpu,
    kernel_cpu,
    2,
    NULL,
    global,

```

```
local,  
2,  
event_gpu,  
NULL);  
  
clFlush(queue_gpu_0);  
clFlush(queue_gpu_1);  
clFlush(queue_cpu);
```

事件的其他应用

由于 OpenCL API 的异步性, 除调用入队函数创建任务队列以外, 获得与 OpenCL 命令执行相关的错误状态或者程序剖析数据是不能通过调用单个 API 实现的。如果能通过 API 调用得到与参数相关的错误状态信息, 就可以通过查询 OpenCL 命令执行返回的事件信息得到具体的错误信息。事实上, 这两种方法是有相似性的。

事件信息查询可通过在事件上调用 `getInfo` 函数来实现, 如下所示。

- **CL_EVENT_COMMAND_QUEUE**: 返回与事件关联的命令队列(当事件在一个复杂的程序中传递时, 这个信息非常有用)。
- **CL_EVENT_CONTEXT**: 返回与事件关联的上下文。
- **CL_EVENT_COMMAND_TYPE**: 返回与事件关联的命令。这个可以是一个类型列表, 包括 **CL_COMMAND_NDRANGE_KERNEL** 和 **CL_COMMAND_READ_BUFFER**。
- **CL_EVENT_COMMAND_EXECUTION_STATUS**: 返回与事件关联的命令执行状态。除了 **CL_COMPLETE** 这个完成状态以外, 命令还可以有入队、提交和执行三个状态。命令执行时返回的错误代码是一个负整数。

如果在创建上下文时开启程序剖析, 那么 `event.getProfilingInfo()` 函数允许开发人员通过命令来得到时间相关信息。使用事件来进行程序剖析将在第 12 章讨论。

用户事件

在 OpenCL 中, 用户事件是用户将依赖于其他任意任务的命令入队的方法。用户事件可以像其他事件一样传递给入队函数, 但必须明确设定事件的执行状态。

例如, 为保证缓冲区不被一个异步 OpenCL 读操作重写, 我们可以使用如下操作:

```
cl_event userEvent = clCreateUserEvent( context 0 );  
clEnqueueReadBuffer(
```

```

    queue,
    bufferOut,
    CL_1RUL,
    0,
    10 * sizeof(int),
    out,
    1,
    &userEvent,
    0);

// Do other things
...

// Make sure that the host pointer bufferOut is safe to overwrite
// at this point because it has been used on the host side
clSetUserEventStatus( userEvent, CL_COMPLETE );

// Now the read buffer operation can continue because
// its dependencies are satisfied

```

事件回调

OpenCL 允许用户定义回调函数，在事件到达指定状态时触发。回调函数将在队列中的命令执行到特定状态时调用。事件回调可用于将新的命令入队，还可用于调用主机端函数，例如特定 CPU 库。`clSetEventCallback` 函数可为事件设定回调函数：

```

// Function call to set an event callback
cl_int clSetEventCallback (
    //OpenCL event
    cl_event event,
    //Event Status which invokes callback
    cl_int command_exec_callback_type,
    //Function pointer - parameter type shown
    void (CL_CALLBACK *pfn_event_notify)
        (cl_event event,
         cl_int event_command_exec_status,
         void *user_data),
    //Pointer to user data which is used by callback
    void *user_data )

```

OpenCL 回调函数的一个应用场景(图 5.7)是主机端 CPU 与设备(如 GPU)紧密交互。在这样一些应用中，主机在设备运行时必须等待，这样会降低系统效率。另一种解决方案就是将一个主机程序设定为回调函数，这样主机在等待 GPU 运行时就可以调度其他任务运行，这样可以提高系统效率。

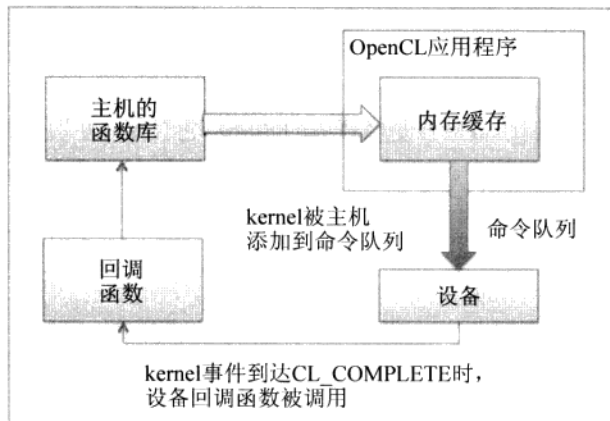


图 5.7

使用回调函数将新命令加入到命令队列中

在下面的代码段中, `clSetEventCallback` 函数的位置非常重要。因为 `clSetEventCallback` 函数需要一个由运行时产生结果来填充的有效事件, 回调函数须在 `clEnqueueNDRangeKernel` 函数之后才调用。

下面的代码将返回一个无效的错误代码, 因为在定义 OpenCL 回调的时候, 运行时还没有产生一个包含有效信息的事件。

```

cl_event completionEvent;
// Wrong location to set callback in
errcode = clSetEventCallback(
    completionEvent,
    CL_COMPLETE,
    myCallback,
    (void *)&ipargs);
// clSetEventCallback will return an invalid event error code
errcode = clEnqueueNDRangeKernel(command_queue,
    kernel,
    2,
    0,
    global_worksize,
    local_worksize,
    &completionEvent);
  
```

如 OpenCL 规范所强调的, 基于以下因素, 回调函数的使用要非常小心。

- 为同一个事件多个执行状态注册的回调函数, 在命令状态改变时, 不保证这些回调函数按照正确的顺序调用执行。

- 回调函数是线程安全的，会被异步调用。
- 当调用昂贵的系统例程或者阻塞 OpenCL API(例如 `clFinish` 函数)时，回调函数的行为是未定义的。

因为这些原因，回调函数应该是轻量级的 **Stub**，它会调用更复杂的函数。下面是一个设置回调函数的例子：

```
// The callback can only get a single void* user_data pointer.
// As a work around, a programmer can pass multiple
// arguments by wrapping them within a structure as shown
struct arg_block{
    data_type arg0;
    data_type arg1;
};
cl_event completionEvent;

//! Simple example showing declaration of a callback
//! The callback function can only have the signature shown below
void CL_CALLBACK
callbackFunction(
    cl_event event,
    cl_int cmd_exec_status,
    void *user_data) {
    //Use this function to invoke a host Library
    arg_block * ipargs = (arg_block *)user_data;
    //Call host function
    host_function(arg_block.arg0, arg_block.arg1);
}

//! Start Device computation
errcode = clEnqueueNDRangeKernel(
    command_queue,
    kernel,
    2,
    0,
    global_worksize,
    local_worksize,
    0,
    NULL
    &completionEvent);

// Set the callback such that callbackFunction is called when
// completionEvent indicates that the kernel
// has completed (CL_COMPLETE)
errcode = clSetEventCallback(
    completionEvent,
    CL_COMPLETE,
    callbackFunction,
    (void *)&ipargs);
```

使用 OpenCL 事件处理的主要好处之一就是应用级行为和同步操作可在不同厂商提供的 CPU 和 GPU 上用统一的方式进行处理。这样就将设备相关的调优局限于计算 kernel 中。

native kernel

除回调函数之外，另一种更好的和 OpenCL 执行模型整合方法是使用 native kernel。native kernel 是允许传统编译器编译而不是由 OpenCL 编译工具链编译的标准 C 函数。在 OpenCL 任务图中由事件触发执行，并触发进一步的事件。

不同于 kernel 入队时使用 `cl_kernel` 对象作为参数，将 native kernel 入队的函数 `clEnqueueNativeKernel` 使用一个指向标准 C 函数的指针。其参数列表不仅包括参数本身，还包括参数的大小。OpenCL 使用 `buffer` 对象与 `image` 对象作为 kernel 参数，而将它们传递给 native 函数是非常有用的。这个过程称为解封装(unboxing)，通过传递一个内存对象列表，参数 `mem_list` 一个指针列表，参数 `args_mem_loc` 是到 `args` 的映射，解封装后的内存指针就放在这个位置上。

为说明这一点，请参考下面的例子。在该例子中，一个 native 函数用到一个包含 5 个值的参数列表，其中第 0 个和第 2 个参数是整数类型，值分别为 5 和 8，第 1, 3, 4 个参数分别是两个 `buffer` 对象和一个 `image` 对象。如图 5.8 所示，对应的代码如下：

```
cl_command_queue queue = clCreateCommandQueue(...);
cl_mem buffer1         = clCreateBuffer(...);
cl_mem buffer2         = clCreateBuffer(...);
cl_mem image           = clCreateImage2D(...);

// initialize buffers, images, and so on

size_t cb_args = 5;
num_mem_objects = 3;
void *args[5] = { (void *)5, NULL, (void *)8, NULL, NULL };

cl_mem mem_list[3] = { buffer1, buffer2, image };
void * args_mem_loc[3] = { &args[1], &args[3], &args[4] };
```

最后，对于给定的 native 函数 `void foo(void * args)`，我们可调用：

```
clEnqueueNativeKernel(
    queue,
    foo,
    args,
    cb_args,
```

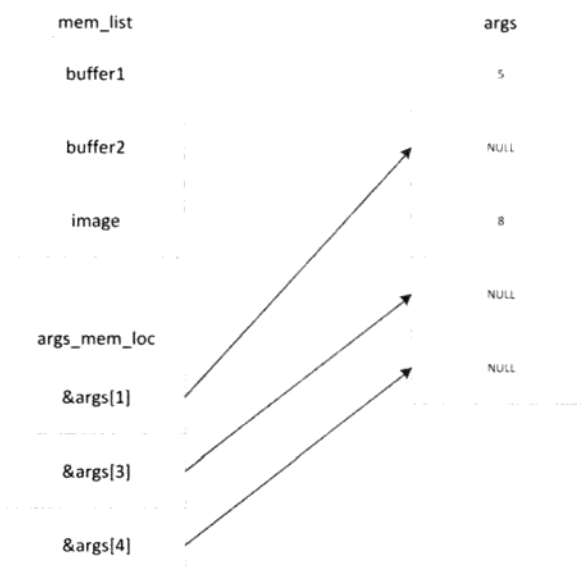



图 5.8

OpenCL 内存对象到 clEnqueue 参数的映射

```
num_mem_objects,
mem_list,
args_mem_loc,
0,
NULL,
NULL);
```

命令 barrier 与 marker

乱序队列同步的另一种方法与 work-group 同步的方法非常类似。在这两种情况下，barrier 操作使队列中其后的任何操作必须在之前的所有操作全部完成之后才能继续执行。barrier 没有任何状态，也不支持事件本身，仅仅存在于队列中保证命令的执行顺序。从概念上讲，这表明 barrier 有一个隐式事件列表(包含队列中同步之前的命令执行所返回事件)和隐式输出事件(包含同步之后每个命令输入事件列表)。barrier 类似于异步的 clFinish 函数。

marker 和 **barrier** 类似但不阻塞执行, 通过 `enqueueMarker` 函数插入任务队列。**marker** 不但 not 阻塞执行, 而且只有当在它之前的命令都执行完成以后才执行完成。它有一个隐式事件列表(包含队列中 **marker** 之前命令执行所返回事件), 并且其输出事件是显式定义的, 可以用于触发进一步的行为。当有事件在等待命令队列中的一些命令完成时, **marker** 非常有用, 但是不会影响到其他命令的执行。

最后一个同步原语是 `waitForEvents`, 其行为与 **marker** 相反。`waitForEvents` 并不等待所有任务完成, 而是只等待指定任务的完成。

通过这些同步命令和广泛应用的事件, OpenCL 提供的功能可用于产生精巧而复杂的任务图以表现出高度复杂的程序行为。

主机端内存模型

诸如 GPU 和其他加速器的 OpenCL 设备的内存与计算机的 CPU 内存是相互分离的如 GPU 和其他加速。除此之外, OpenCL 并发模型支持宽松的内存一致性模型, 只有在事件完成时才定义内存全局同步, 本地同步则使用 **barrier** 操作。为支持这些特性, OpenCL 的内存对象定义在 CPU 内存之外的其他内存空间中。所有从 CPU 内存到 OpenCL 内存对象的数据操作都必须使用 API 函数实现。值得注意的是, OpenCL 内存对象定义在上文中而不是设备上。也就是说, 通常情况下, **buffer** 上的数据操作是没有必要涉及到具体硬件设备的。保证数据在正确的时间到达正确的位置是运行时的工作。

OpenCL 的内存对象被分为两类, 而这两类内存对象的具体位置、布局以及格式是由参数来定义的。OpenCL 规范中定义的两类内存对象是 **buffer** 和 **image**。

以传统 CPU 的意义看, **buffer** 对象是一维数组, 类似于 C 程序中的 `malloc` 函数。**buffer** 对象支持的数据类型可以是任意标量、向量以及用户自定义的数据类型。**buffer** 对象中存储的数据是连续的, 这就使 OpenCL kernel 可以用 C 程序员熟悉的使用指针随机访问的方式来访问它。

image 对象采用的是不同的方法。因为 GPU 是为处理图像任务而设计的, 所以对图像数据的访问进行了深入优化。这主要体现在以下三个方面。

- GPU 的 **cache** 层次结构以及数据流结构都是为了优化图像类型数据的访问而设计的。

- GPU 驱动程序优化了数据布局以提供对硬件高效访存的支持，特别是使用二维访存模式时。
- `image` 访存支持复杂的数据类型转换以允许数据以广泛的压缩格式进行存储。

为优化图像数据访存而进行的数据布局格式的变换，使得通过定义指针来访问数据非常困难，因为不同内存对象之间的位置关系对开发者来说是不可见的。最后的结果是图像数据结构不仅对开发人员不可见，对 `kernel` 代码来说也是完全不可见的，只能通过专门的访问函数来存取。

buffer 对象

`buffer` 对象很容易映射到人们在主机端 C 程序中见到的标准数组。考虑下面的主机端代码：

```
float a[10], b[10];
for( int i = 0; i < 10; ++i ){
    *(a+i) = b[i];
}
```

这个例子表明，我们可以通过指针或使用数组访问语法来访问数组 `a` 和 `b`。这是非常重要的，因为它说明数据是连续分配的，数组 `a` 的第 `i` 个元素 `a[i]` 的存储位置是 `a+i`。

我们可以通过 `sizeof` 操作，计算数组元素在数组中的偏移。在底层代码中，这些特征非常有用。而对 C 派生语言来说，这些是应该支持的特性。例如，下面从 `Bullet Physics SDK` 中取出的 `OpenCL kernel` 代码，就允许我们通过设定基址指针和步长灵活的访问顶点缓冲区，变量 `position` 和数组 `normal` 的数据类型是 `float4`，而输出是一个包含 `position`、`normal` 以及其他信息的数据结构。

```
__kernel void OutputIoVertexArray(
    const int startNode,
    const int numNodes,
    __global float *g_vertexBuffer,
    const int positionOffset,
    const int positionStride,
    const __global float4* g_vertexPositions,
    const int normalOffset,
    const int normalStride,
    const __global float4* g_vertexNormals ){

    int nodeID = get_global_id(0);
    float4 position = g_vertexPositions[nodeID + startNode];
    float4 normal = g_vertexNormals[nodeID + startNode];
```

```
int positionDestination =
    nodeID * positionStride + positionOffset;
g_vertexBuffer[positionDestination] = position.x;
g_vertexBuffer[positionDestination+1] = position.y;
g_vertexBuffer[positionDestination+2] = position.z;

int normalDestination = nodeID * normalStride + normalOffset;
g_vertexBuffer[normalDestination] = normal.x;
g_vertexBuffer[normalDestination+1] = normal.y;
g_vertexBuffer[normalDestination+2] = normal.z;
}
```

操作 buffer 对象

buffer 对象与使用 `malloc` 函数创建的数组类似，因此它们的创建也比较简单。在最简单的情况下，创建 **buffer** 对象需要三个参数：大小、上下文和一组创建时需要的标识：

```
cl_mem clCreateBuffer(
    cl_context context,
    cl_mem_flags flags,
    size_t size,
    void *host_ptr,
    cl_int *err)
```

该函数返回一个 **buffer** 对象，并通过最后一个参数返回错误代码。`flags` 允许只读/只写和分配方案的各种组合。例如下面的代码，我们创建了一个只读的 **buffer**，其数据将直接存储在与 **buffer** 相同大小的数组 `a` 中。注意，OpenCL 只保证队列中所有操作的事件都完成之后的内存一致性，因此，使用 `CL_MEM_USE_HOST_PTR` 时，运行时依然能够将数据复制到设备上执行，并返回结果数据。任何错误代码将通过 `err` 参数返回，错误代码可以是 OpenCL 规范中定义的错误代码范围中的任何一个值。所有 OpenCL 函数成功完成后，都返回 `CL_SUCCESS`。

```
cl_int err;
int a[16];
cl_mem newBuffer = clCreateBuffer(
    context,
    CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
    16*sizeof(int),
    a,
    &err);
if( err != CL_SUCCESS ) {
    // Do whatever error test is necessary
}
```

当 buffer 对象创建完成之后, 其访问可通过访存函数进行。这些函数与其他 OpenCL API 一样, 都是异步调用的。也就是说, 如果我们调用 `clEnqueueReadBuffer` 函数, 在确知从目标主机端数组读取数据完成前不能访问 buffer 对象中的数据, 查询读取完成具体可通过事件机制, 即调用 `clFinish` 函数, 或者通过传递 `CL_TRUE` 参数, 使 `clEnqueueReadBuffer` 成为阻塞操作。因此, 在下面的主机代码例子中, 即使 `outputBuffer` 的内容发生变化, 我们也不能保证两个 `printf` 调用 A 和 B 会打印不同的值。只有 `printf` 调用 C 操作的结果才是正确的。

```
int returnedArray[16];
cl_buffer outputBuffer;
cl_event readEvent;

// Some code that fills returned Array with 0s and invokes kernels
// that generates a result in outputBuffer

printf( "A: %d\n", returnedArray[3] );
clEnqueueReadBuffer(
    commandQueue,
    outputBuffer,
    CL_FALSE,
    0,
    sizeof(int)*16,
    returnedArray,
    0,
    0,
    &readEvent );
printf( "B: %d\n", returnedArray[3] );
clWaitForEvents(1, &readEvent);
printf( "C: %d\n", returnedArray[3] );
```

这是 OpenCL 内存模型一个非常关键的地方。在操作没有完成之前, 对于内存的修改是不可见的, 因此也就不能保证内存的一致性。这样, 在主机 buffer 与设备 buffer 之间的数据传输中, 在确定将数据拷贝到设备 buffer 的异步复制操作完成之前, 主机 buffer 是不能重用的。仔细阅读 OpenCL 规范, 会发现 buffer 是和上下文相关而不是和设备相关的, 一个 `clEnqueueWriteBuffer` 队列, 即使已经完成, 也不能保证数据已经移到设备上, 只能说明主机指针指向的数据已经移出:

If blocking_write is CL_TRUE, the OpenCL implementation copies the data referred to by ptr and enqueues the write operation in the command-queue. The memory pointed to by ptr can be reused by the application after the clEnqueueWriteBuffer call returns

然而，不同于 OpenCL 中其他的 API 调用，buffer 的读写操作允许我们将其设定为同步操作。我们可以用下面的代码替换之前的代码：

```
clEnqueueReadBuffer(  
    commandQueue,  
    outputBuffer,  
    CL_1RUL,  
    0,  
    sizeof(int)*16,  
    returnedArray,  
    0,  
    0,  
    &readEvent );
```

主机线程的执行将在 clEnqueueReadBuffer 函数处挂起，直到所有操作和数据的复制都已经正确完成。

OpenCL 还定义了 sub-buffer 对象上下文，允许我们将一个单独的 buffer 切分成许多可以重叠的较小的 buffer，而这些 sub-buffer 的读、写以及复制操作方式与 buffer 的操作方式是相同的。注意，sub-buffer 的重叠、sub-buffer 和其父 buffer 对象的合并可能会造成别名问题，在这些情况下，其行为是未定义的。

image 对象

image 与 buffer 主要有以下三方面的不同：

- 对设备代码是不可见的，不能通过指针直接访问
- 多维结构
- 仅限于图像数据相关数据类型，而不能自由实现任意数据结构

OpenCL 对 image 对象的访问，是通过为高效获取图像数据而设计的专用功能硬件实现的，这些特殊功能单元并不总能提供访问缓存所需要的全部访存模式，但可以提供额外的访存功能，如高效的硬件滤波。滤波操作能够实现基于像素的图像数据高效转换，这些操作包含进行多次读操作的长指令序列，并可以通过专用硬件单元高效完成。

在 kernel 代码中，image 数据通过专门的访存函数来访问，这个将在后面进行讨论。在主机端访问 image 数据，除所有访存函数都需要扩展支持多维地址空间外，与访问 buffer 并没有明显的不同。例如，clEnqueueReadImage 更像 clEnqueueReadBufferRect 而不是 clEnqueueReadBuffer。

从主机端访问 image 和 buffer 的主要区别是对数据格式的支持。buffer 支持 OpenCL 所有的基本数据类型以及由它们组成的结构体。image 支持的数据类型要少得多。

`image` 的数据格式由通道序列和通道类型组成。通道序列定义了通道数以及它们的组织顺序,例如 `CL_RGB`, `CL_R` 或 `CL_ARGB`。通道类型的选择非常多,从 `CL_FLOAR` 到非常少用的在内存中组织成 16 位字的 `CL_UNORM_SHORT_565`。从 `kernel` 代码中进行数据访问时,读取任何格式的数据都会转换为标准的 OpenCL C 类型, `image` 支持的数据格式可通过调用 `clGetSupportedImageFormats` 函数来查询。

`image` 提供一个额外的功能,支持 `buffer` 所不支持的运行时系统和硬件优化。`image` 数据可通过调用 `clEnqueueMapImage` 函数实现到主机的映射。因此,从主机端映射指针的角度看, `image` 数据必须要有特定的格式,而映射操作支持这种格式转换。OpenCL 数据结构的这个特点,使得运行时系统能够对自己控制的数据进行数据转换。

`image` 对象在设备端不能通过指针进行直接访问,也不能在同一个 `kernel` 中进行读操作和写操作。因此,运行时系统的数据变换对运行在设备端的 `kernel` 来说是完全不可见的,并能在不影响正确性的前提下显著提高 `kernel` 的性能。`image` 的这一特征也消除了锯齿数据(aliaised data)的可能性,如此一来,不像 `buffer` 无法被安全缓存, `image` 对象是可以进行安全缓存的。

举一个常见的优化例子:任何给定的多维数据结构,这里以 `image` 为例,都必须在程序运行的某个阶段映射到一个一维地址空间中。最明显的映射方法,也是绝大多数编程语言对多维数组使用的方法,就是以行模式或者列模式的字典顺序映射。也就是说, (x, y) 在 $(x+1, y)$ 的前面,而离 $(x, y+1)$ 却很远,以此类推。 (x, y) 与 $(x, y+1)$ 在内存中的较长跨度意味着 y 方向上的连续地址访问会访问大量的 `cache line`^①,是非常低效的;相反,由于 (x, y) 与 $(x+1, y)$ 相邻,在 x 方向的连续地址访问是高效的。

Z 序或 Morton 序内存布局所使用的映射方式保留着数据访问的空间局部性。图 5.9 显示数据的存储顺序是 $(0, 0)$, $(1, 0)$, $(0, 1)$, $(1, 1)$, $(2, 0)$, 等等。数据按照 Z 序映射进行存储,进行垂直数据读取时,可能会重复命中同一 `cache line`。如果我们的计算任务也是二维布局(就像在图形流水线上创建四边形一样),会进一步改进数据的局部性。如果我们要求 `kernel` 程序员不必保证内存元素的相对位置,这是唯一可能的优化方法(在不同的架构上采用不同的优化方法)。

该方法还可以更进一步。如果硬件设备不支持向量寄存器,也不支持内存的向量读取,我们希望 `float4a = read_imagef( sourceImage, imageSampler, location)` 编译成四条标量

① 译者注:一次访存操作所访问的数据。

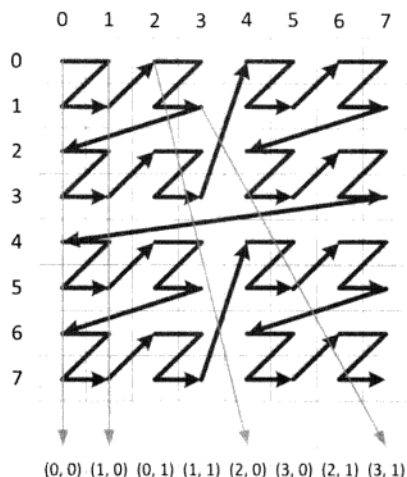


图 5.9

二维地址空间的 Z 序映射

指令，而非一次向量读取操作。在这些情况下，从四个相互分离的数组的相同位置读取数据比从同一个数组中读取四次数据更有效，因为每次单独读取每一个分离数组中的数据时，会表现出更好的数据局部性。

设备端内存模型

在 OpenCL 设备上，内存空间分为以下四大类别：

- 全局内存
- 本地内存
- 常量内存
- 私有内存

这些内存空间的示意图如图 5.10 所示。正如第 3 章所讨论的那样，OpenCL 是跨平台的。如此组织内存层次结构的目的是使 OpenCL 能够有效运行在各种不同的平台上。每个内存空间的实际内容，即和硬件结构的映射关系，很大程度上也与具体实现相关。然而，不管这些内存空间和硬件实现之间是如何映射的，作为一种编程抽象，这些内存

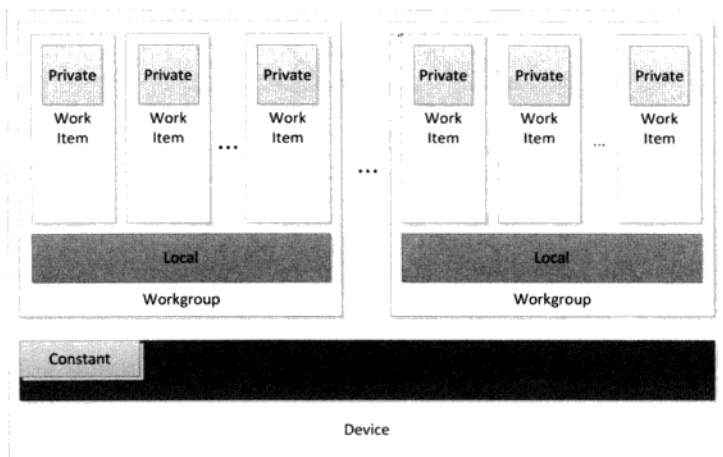


图 5.10

OpenCL 设备端内存模型

空间是相互分离的。如图 5.10 所示，本地内存与私有内存被划分为分离的两部分。通过定义这种相互分离的内存层次关系，可以实现从宽松一致性模型(如 GPU 可编程控制的 scratchpad[可擦除]缓存)到完全一致性内存系统(如 x86 架构)的硬件映射。

函数参数以及函数或代码块中局部变量的默认地址空间是私有内存。函数的指针参数所指向的数据可根据其来源和用途放在其他任意的地址空间中。注意，指针本身会一直放在私有地址空间中。

使用指针时，地址空间是严格不相交的。从一个地址空间到另一个地址空间的转换是不允许的。因为这种地址空间的转换意味着数据位于可被全局访问的地址空间中或者编译器不得不产生一个复制操作来完成这种地址空间的转换，这在实际应用中是不可行的。函数的 `image` 参数始终位于全局地址空间中，因此我们在讨论 `image` 时只限于全局地址空间。

设备端宽松的内存一致性

OpenCL 的宽松内存一致性模型不仅体现在 kernel 内部，还体现在 kernel 的调度之间。内存的写入操作在 kernel 执行完成之前并不保证其可见性，除非使用 `fence` 操作。因此，我们定义了一个一致性层次结构。

- **work-item** 内部内存操作有可预测的顺序：即任意两个对同一地址的读和写操作是不会被硬件或者编译器重新排序的。
- 在同一个 **work-group** 的不同 **work-item** 之间，只在 **barrier** 操作处保证内存一致性。
- 在 **work-group** 之间，在 **kernel** 执行完成前，不保证内存一致性。

给定前面的层次结构，除非是给定地址的最后一次写操作，编译器不需要保证对该地址的其他操作对给定 **work-item** 之外的其他 **work-item** 可见。为了使位于同一个或者不同 **work-group** 中的 **work-item** 实现某一层次的通信，OpenCL 定义了一组 **fence** 操作。然而，即使使用这些 **fence** 操作，**work-item** 的执行顺序也得不到保证。

下面是 **fence** 操作的读、写以及读/写函数：

- `read_mem_fence( cl_mem_fence_flags flags )`
- `write_mem_fence( cl_mem_fence_flags flags )`
- `mem_fence( cl_mem_fence_flags flags )`

在每种情况下，**fence** 函数都有一个标志参数，指定受保护的地址空间。标志参数的值是 **CLK_LOCAL_MEM_FENCE** 和 **CLK_GLOBAL_MEM_FENCE** 的某种组合。**fence** 操作保证在其之后的数据加载和(或)存储操作开始之前，它前面的数据加载和(或)存储操作必须完成。单独的 **fence** 操作不会引发同步，**barrier** 操作不仅支持在一个或者两个地址空间的读(写)**fence** 操作，也保证在同一个 **work-group** 的所有 **work-item** 到达这个位置之前阻塞程序执行。

保证不同 **work-item** 间内存操作正确性的另一种方法是原子操作。这种方法特别有用，因为它可以在不影响其他 **work-item** 的前提下，保证数据读或者读写以及其他操作的正确性。然而，由于绝大多数硬件上浮点数原子操作的复杂性，OpenCL 只定义了整数类型的原子操作。原子操作可以是算数操作，例如 `int atomic_add( volatile __global int *p, int val)`，也可以只是数据操作，例如 `int atomic_xchg( volatile __global int *p, int val)`。在所有情况下，原子操作都将返回所操作位置的初始值。注意，如果返回值被忽略，编译器将自主选择没有返回值的原子操作，相对于普通原子操作，这在很多硬件架构上是非常高效的。

全局内存

全局内存，在 OpenCL C 代码中的定义为：一个带数据类型和地址限定符 `__global`(或者 `global`)的指针，或者是 `image2d_t` 或 `image3d_t` 的 `image` 类型。全局内存可以由设备上所有的计算单元(compute unit)使用。两种不同类型的对象在使用范围和使用方式上是不同的。

`__global` 地址空间限定符代表一个指向 `buffer` 对象的指针。如前所述, `buffer` 可以存放任意的标量数据类型、向量数据类型以及用户自定义的结构。无论哪一种 `buffer` 类型, 都可通过指针进行只读或者读/写访问。下面的代码就是一个有效利用 `buffer` 对象的例子。

```
typedef struct AStructure {
    float a;
    float b;
} AStructure;

__kernel void aFunction( __global AStructure *inputOutputBuffer ) {
    __global AStructure* inputLocation =
        inputOutputBuffer + get_global_id(0);
    __global AStructure* outputLocation =
        inputOutputBuffer + get_global_size(0) + get_global_id(0);

    outputLocation->a = inputLocation->a * -1;
    outputLocation->b = (*inputLocation).b + 3.f;
}
```

`image` 对象虽然从概念上看是位于 `__global` 内存空间, 但它的操作方式和 `buffer` 对象完全不同, 也不使用 `__global` 指针。`image` 对象是二维或者三维的, 通过 `image2d_t` 或者 `image3d_t` 类型限定符进行创建。与 `buffer` 对象不同, 在同一个 `kernel` 中, `image` 对象只能是只读或者只写, 而不能是读/写。这是 GPU 硬件设计中支持高性能缓存或者滤波的结果。在 `kernel` 代码中, 我们使用 `__read_only` 和 `__write_only` 访问限定符来指定 `image` 对象的访问方式。

`image` 是不可见的内存对象。尽管可以基于地址来读/写数据, 但我们无法确知 `image` 中两个不同值对应的内存地址之间的相对位置。因此, 为支持参数化读/写而不是通过指针读/写, 我们使用一组内建函数 `read_imagef`, `read_imagei`, `read_imageui`, `write_imagef` 来访问 `image` 对象。每一个 `image` 的读取函数都有三个参数:

```
float4 read_imagef(
    image2d_t image,
    sampler_t sampler,
    float2 coord)
```

最后的地址参数也可以是 `int2`(或者 `int4`, 如果 `image` 是 `image3d_t` 类型), 返回数据的确切含义取决于图像的格式。OpenCL 规范中完整列出了这些选项。

读取函数的第一个和第三个参数不言自明, 分别是 `image` 对象本身和读取坐标。第二个参数稍微复杂一些, 是一个 `sampler` 对象, 定义了硬件和运行时系统理解 `image` 的方式。`sampler` 对象定义的方式有两种: 一是直接在 OpenCL C 代码中声明一个 `sampler_t` 类型的常量数据; 二是在主机端代码中调用 `clCreateSampler` 函数创建一个 `sampler`, 然

后将其作为参数传递给 kernel 代码。下面的例子使用第一种方式创建 sampler。

```
__constant sampler_t sampler =
    CLK_NORMALIZED_COORDS_TRUE | CLK_FILTER_LINEAR;
__kernel void samplerUser(
    __read_only image2d_t sourceImage,
    __global float *outputBuffer) {

    float4 a = read_imagef(
        sourceImage,
        sampler,
        (float2)(
            (float)(get_global_id(0)),
            (float)(get_global_id(1))));

    outputBuffer[
        get_global_id(1) * get_global_size(0) +
        get_global_id(0)] = a.x + a.y + a.z + a.w;
}
```

float4 向量的返回值取决于创建 image 对象时所指定的 image 格式。例如，一个 CL_R 的 image 对象，其返回值只包含 x 通道的数据，w(alpha)通道的数据为 1.0。

除用写入数据代替 sampler 以外，写入函数采用的一组参数相似：

```
float4 write_imagef(image2d_t image, float2 coord, float4 color)
```

本地内存

OpenCL 支持的一部分硬件体系结构，包括许多 GPU 和 Cell，都实现了有别于 DRAM 和片上缓存体系结构的小块 scratchpad 内存——本地内存。这些体系结构中的本地内存和全局内存不相交，并通过独立的内存操作进行访问。因此，数据的拷入和拷出都需要编程来实现。根据硬件体系结构的不同，内存操作或者通过 DMA 传输(最有效的访存方式是使用 `async_work_group_copy` 函数)来实现，或者通过内存到内存的拷贝来实现。CPU 实现也支持本地内存，但是本地内存是标准的可缓存的内存，在这些类似情况下，因为鼓励 cache-aware 编程的缘故，使用本地内存依然是有好处的。

本地内存是非常有用的，因为它提供了同一个 work-group 内所有 work-item 之间进行通信的最有效方式。任何分配的本地内存都可以被整个 work-group 访问，因此，一个 work-item 对本地内存的写入，对 work-group 内的其他 work-item 都是可见的。记住，在 OpenCL 中，work-item 是一个逻辑概念，如没有特殊说明，是会独立执行的。

本地内存通过 `__local` 地址限定符定义。既可以在 kernel 内部定义，也可通过参数传递。下面的代码说明了这两种定义方式：

```

__kernel void localAccess(
    __global float* A,
    __global float* B,
    __local float* C )
{
    __local float aLocalArray[1];

    if( get_local_id(0) == 0 ) {
        aLocalArray[0] = A[0];
    }
    C[get_local_id(0)] = A[get_global_id(0)];
    barrier( CLK_LOCAL_MEM_FENCE );

    float neighborSum = C[get_local_id(0)] + aLocalArray[0];

    if( get_local_id(0) > 0 )
        neighborSum = neighborSum + C[get_local_id(0)-1];

    B[get_global_id(0)] = neighborSum;
}

```

图 5.11 是前面代码中数据流的图形化表示。注意，数据从全局内存读出并写入到两个本地数组 C 和 aLocalArray 的时间是不能预测的，因为 work-item 的执行是相互独立且无序的。然而在实际应用中，由于映射到硬件上的方式是可预见的，因此对于一个给定的设备是可以粗略预测的。

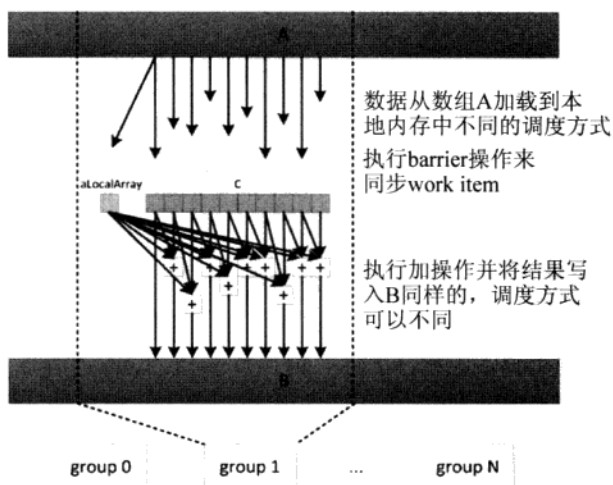


图 5.11

本地访存的数据流模式

例如，在 AMD GPU 中，本地内存的操作在一个宽 SIMD 向量上以时钟步的方式执行，这意味着同一个 work-item 的读和写操作顺序在整个向量上是有保证的。然而，这个特征并不是通用的。在一般情况下，我们必须插入 barrier 操作，因为只有在 barrier 操作处我们才能保证所有的本地内存写入以及相对应的全局内存读取在整个 work-group 中已经全部完成，数据对其内部的所有 work-item 都是可见的。barrier 操作执行完成后，数据就可以被整个 work-group 使用，如图 5.11 中的下半部分所示。

aLocalArray 的作用域是整个函数，而且对整个 work-group 都是可见的。也就是说，每一个 work-group 只有一个 32 位的本地内存变量，该 work-group 内所有的 work-item 都访问同一个 32 位的变量值。在这种情况下，在 barrier 执行完成后，work-item 0 已经完成了对 aLocalArray 的写入，因此，在这个 work-group 内部的所有 work-item 都可以访问它。

另一个创建本地内存数组的方法是通过 kernel 参数，如数组 C。这个版本是调用运行时 API 来创建的。为分配内存，就像给 kernel 传递全局内存数组一样，我们调用 clSetKernelArg 函数，不同的是需要将最后一个指针域设为 0。这样，我们就根据第三个参数的大小为每个 work-group 在本地内存上分配所需要的内存空间：

```
clErrNum = clSetKernelArg(  
    kernel object,  
    parameter index,  
    size in bytes,  
    0);
```

常量内存

常量地址空间，通过 __constant 限定符描述，其作用是将少量的常量数据同全局地址空间分离开来，以便运行时能够为其分配 cache 资源或者在可能的情况下有效利用常量内存的 bank。常量地址空间的数据分配通过调用 clSetKernelArg 函数来实现，而且能够被 kernel 内部的指针访问。对这部分数据的处理，不同硬件的处理方式是不同的。例如，AMD Radeon_ HD 6970 定义了以下三种常量数据的处理方式。

- 直接地址访问：数据地址是不变的而且被集成到指令中。因为这些数据位于硬件的常量缓存中，所以性能非常高，达到 16 bytes/cycle/core。
- 相同索引：整个 wavefront 访问同一个地址：性能达到 4 bytes/cycle/core。
- 不同索引：和全局内存的访问方式一样，但可能会被 L1 cache 缓存。

对每一个设备来说，OpenCL kernel 常量数据的参数个数以及缓存大小都是有限的，这两个值可以通过调用 clDeviceInfo 函数使用 CL_DEVICE_MAX_CONSTANT_ARGS 和 CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE 这两个参数进行查询。

创建 `__constant` 缓存时，内存分配函数必须使用 `CL_MEM_READ_ONLY` 标志。

私有内存

私有内存是指没有使用地址限定符进行声明的所有变量，非 `kernel` 函数内部的所有变量以及那些非指针的参数。理论上，私有数据都位于寄存器上，然而由于缺乏对溢出的管理以及硬件不能实现寄存器数组的动态索引，私有数据也可能放在全局内存上。私有内存分配的数量直接影响着 `kernel` 使用的寄存器的数量。

像本地内存一样，硬件架构的寄存器数量是有限的。使用大量寄存器对性能的影响因硬件架构不同而不同。

x86 架构 CPU 的寄存器数量相对较少。然而，由于 CPU 的 `cache` 较大，将寄存器数据加载到内存堆栈以及将其取回的操作开销非常小。寄存器变量可以高效地移进移出，以此来保证在寄存器中只保留最常用的数据。

通常情况下，GPU 并不会以这种奢侈的方式使用 `cache`。很多设备都没有读/写 `cache`，即使有，大小也是有限制的。因此，由大量线程导致的寄存器溢出很快也会导致 `cache` 溢出，从而导致数据重用时程序阻塞。如果数据溢出到 `DRAM` 上，会导致性能显著下降，因此，要尽量避免寄存器溢出。

当寄存器不溢出时，寄存器就会像 `LDS` 一样影响活动线程的数量。AMD Radeon HD 6970 每一个计算单元(CU)拥有 256 KB 的寄存器，对于一个 64 路的 `wavefront` 来说，每个线程可以使用 256 个 128 位(长度为 4 的向量)的寄存器。如果每个线程使用 100 个寄存器，那么每个计算单元只能同时调度 2 个 `wavefront`，这样除了指令延迟，什么也隐藏不了。如果每个线程使用 49 个寄存器，我们就可以同时部署 5 个 `wavefront`，以此来帮助隐藏延迟。

将数据放在寄存器上或许可以提高性能，但如果由此导致同时运行的 `wavefront` 数减少而不能充分隐藏延迟，就会有更多的程序挂起和更多的 GPU 时钟周期的浪费。

小结

在这一章中，我们讨论了 OpenCL 程序的内存一致性、并发性和同步性。为了有助于尽可能在多种硬件架构上高效实现，OpenCL 遵循宽松的执行模型和内存一致性模型。在稍后几章中，我们要讨论执行模型如何映射到特定硬件架构上，然后再通过例子来讨论如何优化 OpenCL 程序以更高效地使用硬件。

OpenCL 在 CPU/GPU 平台上的实现

引言

在第 3 章中，我们讨论了不同硬件架构所采用的权衡方案。在这些硬件架构中，许多都支持 OpenCL 程序的运行。OpenCL 的设计使得 OpenCL 能够被映射到不同的硬件架构上，并支持 kernel 代码的调优和加速。在本章中，我们将讨论 OpenCL 是如何映射到高端 AMD CPU 和 AMD Radeon HD6970 GPU 所组成的计算机系统的。尽管我们选用 AMD 系统来说明 OpenCL 的映射和实现，然而各相关厂商都已经实现了 OpenCL 到 NVIDIA GPU，Intel/ARM CPU 以及任何 OpenCL 兼容硬件架构的映射。

OpenCL 在 AMD PHENOM II X6 上的实现

AMD 的 OpenCL 实现，可以使 OpenCL 代码以统一的方式运行在 AMD GPU 和 AMD x86 CPU 上。所有主机端代码将像操作系统和通用应用程序代码一样运行在通用 x86 CPU 上。然而，通过 OpenCL 运行时提供的队列机制，OpenCL C 代码可以在 x86 设备上编译和运行。

OpenCL 能够运行在 AMD Phenom II X6 六个核中的任意一个核上。图 6.1 为 AMD Phenom II X6 的设计图示。

OpenCL 到这个硬件架构的映射如图 6.2 所示。整个芯片作为一个单独的设备供 OpenCL 使用，通过调用 `clGetDeviceIDs` 函数获得这个设备，并将其传递给 `clCreateContext` 和 `clBuildProgram` 函数。注意，设备为 CPU 时，需要将设备类型参数 `CL_DEVICE_TYPE` 传递给 `clGetDeviceIDs` 函数。

通过将整个 CPU 作为一个单独的设备，并行负载便可以通过一个单独的队列分配到 CPU 的所有核上，以有效利用系统的并行性。除此之外，我们还可以通过设备拆分技术将 CPU 拆分成多个设备，这将在下一章讨论。

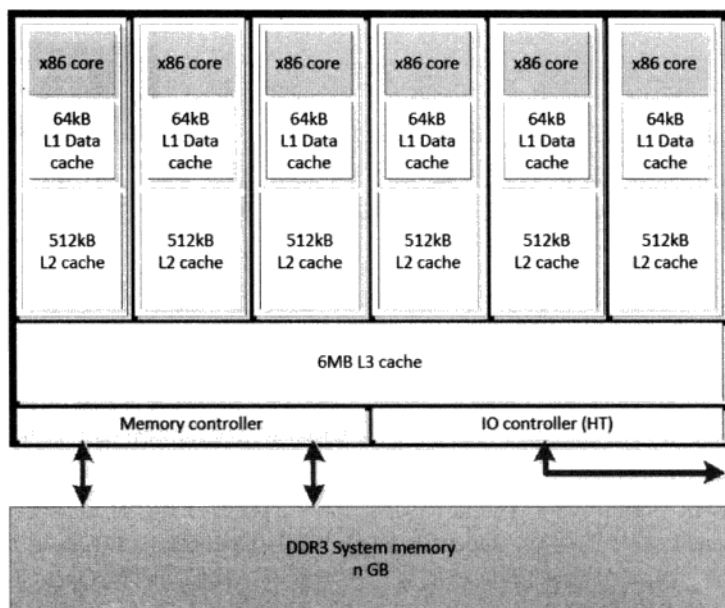


图 6.1

AMD Phenom II X6: 6 核和 3 级缓存

OpenCL CPU 运行时将创建一个运行在 CPU 每一个核上的线程，将它作为一个工作池来处理生成的 OpenCL kernel。任务队列的任务分配由核心管理者线程负责：从队列头部移出任务并将其分配给工作者线程。

任意给定的 OpenCL kernel 可能由数千个 work-group 组成。因此，必须为其准备相应的参数、分配内存并在必要时进行初始化和生成工作队列。

OpenCL 利用 barrier 和 fence 操作实现细粒度同步。在一个典型的基于 CPU 的计算机系统中，操作系统负责管理线程间的通信，然而，线程与操作系统交互所引起的开销阻碍了高效并行扩展的实现。除此之外，在多核上运行一个单独的 work-group 会导致 cache 共享问题。为缓解这些问题，OpenCL CPU 运行时在一个操作系统线程上运行 work-group。OpenCL 线程轮流运行同一个 work-group 内的每一个 work-item，当这个 work-group 内的所有 work-item 都运行完成后，再运行同一个工作队列中的下一个 work-group。因此，同一个 work-group 内的线程是没有并行性的，如果可能的话，多个操作系统线程将允许多个 work-group 并行执行。

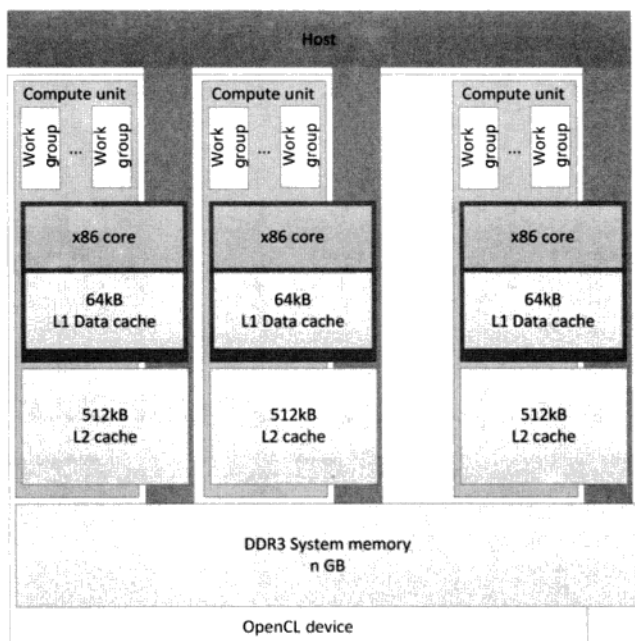


图 6.2

OpenCL 在 Phenom II X6 芯片上的映射。注意，主机代码将运行在作为 OpenCL 设备计算单元的核心上

在 OpenCL 的 barrier 同步操作中，同一个 work-group 的所有 work-item 是并发执行的。work-group 内的 work-item 必须完成 barrier 操作之前的所有代码，并等待其他 work-item 到达 barrier 操作处后，才能继续执行。在 barrier 操作中，一个 work-item 必须终止，而其他 work-item 将继续运行。出于性能因素的考虑，对 barrier 操作的支持通过操作系统用线程抢占方式(即，中断一个线程让其他线程运行)实现是不现实的，AMD 的 OpenCL CPU 运行时是通过 setjmp 函数和 longjmp 函数实现的。setjmp 存储系统状态，longjmp 恢复在 setjmp 被调用时的系统状态(Gummaraju et al., 2010)。在 AMD OpenCL 运行时中，提供了这两个函数的自定义版本，因为这两个函数需要与硬件分支预测器协同工作，并需要保持正确的程序栈对齐。

图 6.3 说明了一个使用 barrier 操作的 kernel:foo()。注意，尽管一个 CPU 线程会执行多个 work-group，但 CPU 线程只有在完成一个 work-group 的执行后，才执行下一个 work-group。

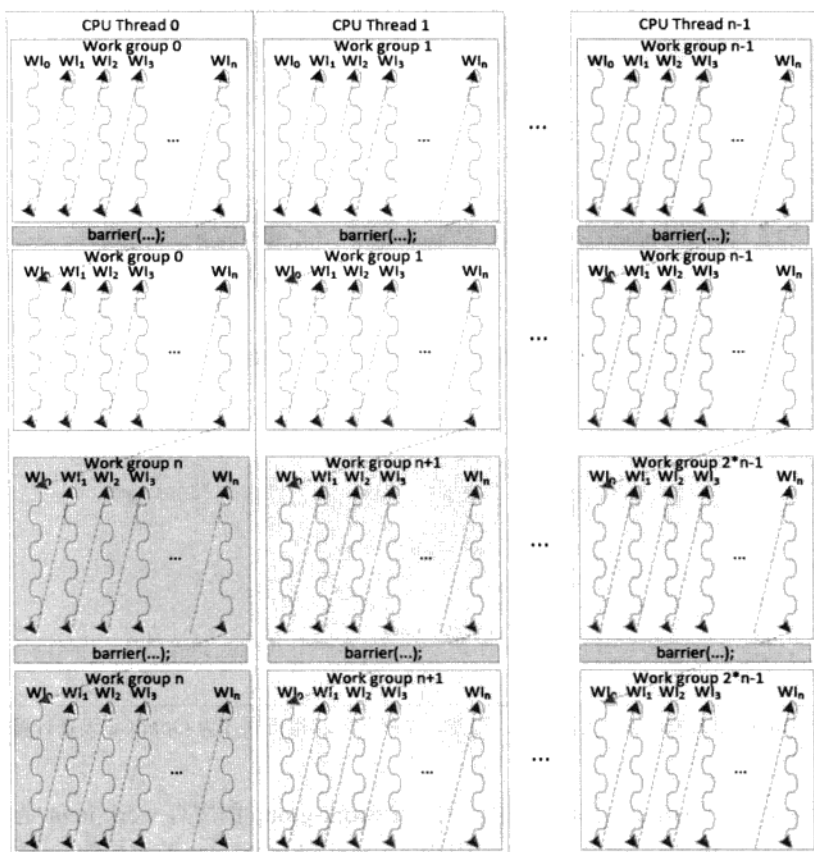


图 6.3

一个 OpenCL 工作者线程一次只能处理一个 work-group 内的一个 work-item。在 barrier 操作或 kernel 结束处，工作者线程才会处理下一个 work-item

当 kernel 中存在 barrier 操作时，CPU 线程首先将每一个 work-item 执行到 barrier 操作处，然后在 barrier 操作完成之后再继续执行每一个 work-item，以此来提供正确的 barrier 语义。

```
__kernel foo() {
    ...
    barrier(CLK_GLOBAL_MEM_FENCE);
    ...
}
```

AMD Phenom II 微架构包含 128 位的向量寄存器和 Streaming SIMD Extension(SSE)^① 各种版本的操作。同时, OpenCL C 定义了一组向量类型, 如 float2, float4, int4 等。这些向量类型的数学运算都进行了重载^②, 以支持如下操作:

```
float4 a = input_data[location];
float4 b = a + (float4){0.f, 1.f, 2.f, 3.f};
output_data[location] = b;
```

在 AMD Phenom 架构中, 这些向量类型存储在向量寄存器中, 而对它们的操作也会被编译成 SSE 指令。这便提供了一个非常重要的性能优化方法。就像我们在底层代码中讨论的那样, 向量加载和存储操作能够提高内存的操作效率。当前, 在一个 work-item 中对 SSE 向量的访问是显式的。我们将看到这个模型与下节将要讨论的 GPU 实现有何不同。

AMD Phenom II 设计中没有提供专门的 scratchpad(可擦除)cache。CPU 一般提供多级 cache 来隐藏访存延迟。由本地内存提供的数据本地化可以有效映射到 CPU cache 的层次架构上, 并允许 kernel 开发人员在没有真正的硬件 scratchpad cache 的情况下改进 cache 的性能。为改进 cache 本地化, 本地内存区域以每个 CPU 线程一个数组的方式进行分配, 并且可被这个线程执行的所有 work-group 重用。因为在 work-group 序列中禁止任何数据竞争和内存冲突, 所以没有必要为这种本地内存产生更进一步的 cache 未命中操作。同时, 重复调用内存分配函数(routine)是没有开销的。图 6.4 显示了如何将本地内存映射到 AMD Phenom II cache。

调用 setjmp 操作时, 存储在寄存器中的 work-item 的数据会被备份到位于主存的 work-item 栈中。数据的内存布局得以精心设置以减少 cache 竞争和 cache 冲突, 以提升对 cache 层次结构的利用。具体而言, work-item 的栈数据在内存中交错存储, 以减少内存冲突的几率。除此之外, 数据存储在较大的内存页上来保证向物理内存映射的连续性, 以减少 CPU 在 TLB(translation lookaside buffer)^③时的压力。

① SSE 是 SIMD 指令集在 x86 架构上的扩展, AMD 和 Intel 从 1999 年, 已经开始发布了多个版本的 SSE 指令集扩展。

② 重载是多态的一种形式, 支持使用不同参数的函数名重用。调用该函数时, 只需关心函数名称, 而不需要关心参数类型, 以此来简化代码。

③ TLB 是 CPU 上的从虚拟缓存到物理内存的地址转换硬件表。

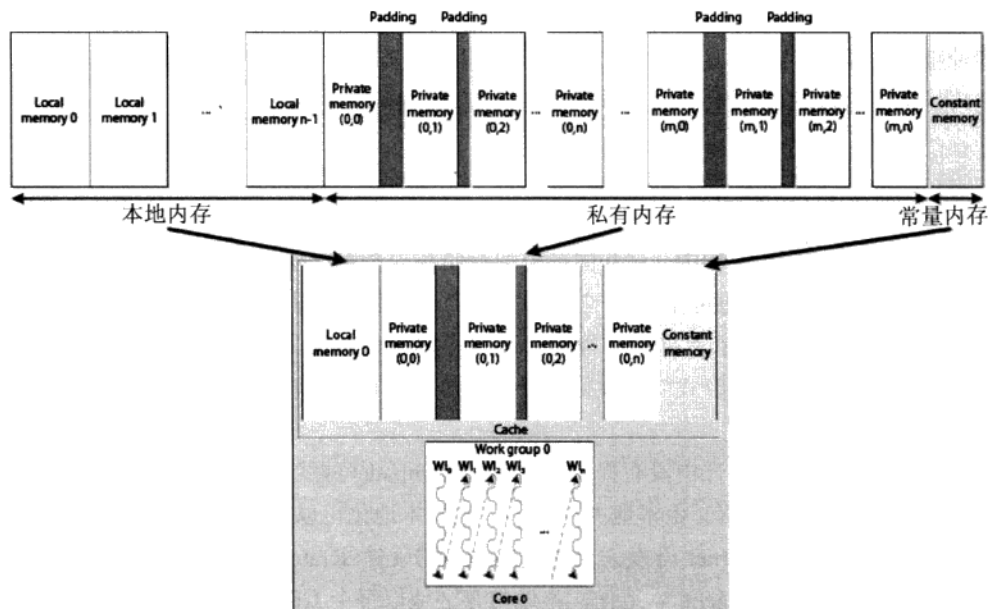


图 6.4

每一个 work-group 和 work-item 的本地、私有和常量内存都是连续存储的。这些数据作为连续的数据块加载到 cache 层次结构中，在指定 work-group 运行期间，尽可能保证 cache 本地化

OpenCL 在 AMD RADEON HD6970 GPU 上的实现

GPU 和 CPU 对 OpenCL 的实现有着明显的不同。读者必须记住，GPU 主要用来有效渲染三维图像，所以其底层架构和 CPU 明显不同。就像我们在第 3 章讨论的那样，这种差异可以总结为以下几点。

- 宽单指令多数据(SIMD)的执行：多个执行单元在不同的数据上执行相同的指令。
- 大量线程：支持大量的并发线程上下文。
- 硬件 scratchpad 内存：程序员可以完全控制的物理缓存。

下面的差异比较微妙但非常有趣，因为它们提供了改进 GPU 任务调度和通信延迟的方法。

- 硬件同步支持：支持并发硬件线程间的细粒度通信。

- 硬件管理任务及调度：硬件负责管理工作队列和负载平衡。

支持硬件同步减少了在 SIMD 核心上多个线程上下文间的同步开销，保证了线程间高效的细粒度通信。

由于深受三维图像处理的影响，GPU 提供了对任务调度的硬件支持。游戏任务负载管理涉及对复杂任务图的管理，任务图是在图像流水线上交替执行任务产生的。正如图 3.8 的 AMD Radeon HD6970 架构图所示，架构前端由一个命令处理器和组生成器构成，它们向两个硬件调度器传递结构化的组。这两个硬件调度器向设备上的 24 个 SIMD 核心安排计算任务。对于图像工作负载，AMD 还包含了一组位于命令处理器下面的硬件加速器。

- Tessellator：指运行时从一个大三角形中产生小三角形以适应不同模型复杂度的过程。
- Geometry assembler：打包几何信息供着色器进行处理。
- Rasterizer：将向量数据转化成光栅格式。
- Hierarchical Z processor：通过提供基于深度的提前拒绝像素的能力，维护场景深度的层次性，以减少负载。

总之，这些单元允许硬件像图 6.5 一样调度任务。为获得较高的 GPU 计算的加速性能，调度必须是非常有效的。

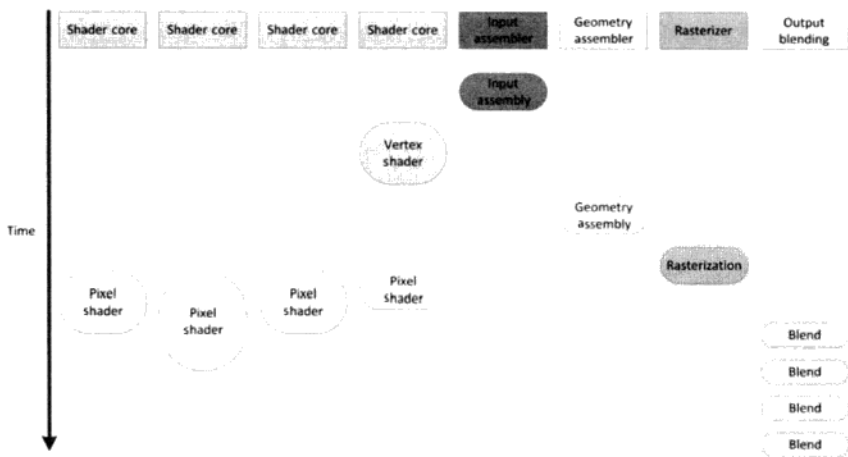


图 6.5

该示意图说明的是由硬件管理的、将一组任务分配到 GPU 上运行。有许多这样的任务一起调度时，硬件能够得以充分使用

线程调度开销需要维持在一个非常低的水平,因为 input assembly 所产生的工作块可能非常小——例如,一个由几个像素组成的三角形。单凭这点工作量是不能保持机器持续使用的,然而请记住,完整的图形流水线可以非常快速地同时组装、光栅化以及着色大量的三角形,如图 6.5 所示(注意:图中使用空白标记的未利用计算时间,当多个三角形被同时处理时,可以非常容易地填充)。这个例子充分说明了为什么 GPU 专为高吞吐量处理而设计,以及为什么工作负载需要恰当的映射到底层硬件以获取高性能。

在 OpenCL 中,任务调度是预定义的,并且只需要简单地将任务组装成 wavefront,因此,没有必要有很多这样的光栅和组装硬件。然而,要通过必要的优化工作使任务支持深度流水线命令处理器和任务产生器,在采用这种设计的 GPU 上取得高性能,我们需要做如下工作:

- 针对每一次 kernel 执行提供大量任务
- 批处理作业

通过为每一个 kernel 提供足够数量的任务,我们可确保组产生流水线被持续占用。这样,它总是有更多的任务提供给 wave 调度器,而 wave 调度器总是有更多的工作调度给 SIMD 单元。从本质上讲,我们通过创建大量的线程来占用机器:正如之前所讨论的那样,GPU 是一个面向吞吐量的设备。

第二点是 OpenCL 的队列机制。OpenCL 运行时从设备相关的任务队列中选择任务时,它会扫描队列中的任务,选择适当大小的任务块进行处理。在这一组任务中,它使用能够 GPU 流水线前端命令处理器所能理解的语言为 GPU 任务处理构建一个命令缓冲区。这个过程是需要时间的:1)构建队列;2)将队列存储在内存中的某个位置;3)告诉设备队列的位置;4)请求设备处理。特别是,这个过程运行在 kernel 空间并在驱动程序之后,因此需要大量的进出 kernel 空间的上下文切换来开启 GPU 的运行。至于 CPU,线程间的上下文切换会成为一个主要的开销,切换到线程模式和准备过于小的任务队列也是非常低效的。除此之外,还有一个进行任务队列调度时的相对恒定的消耗以及根据队列中任务多少的进一步的处理消耗。而这个恒定消耗必须通过为一个或者多个 kernel 提供足够多的任务来消除。

多线程和内存系统

图 6.6 展示了一个系统内存层次近似模型,它包含一个 AMD Phenom II CPU 和一个 AMD Radeon HD6970 GPU。CPU 的 cache 层次模型用来减少由单一内存访问而引起的访存延迟。

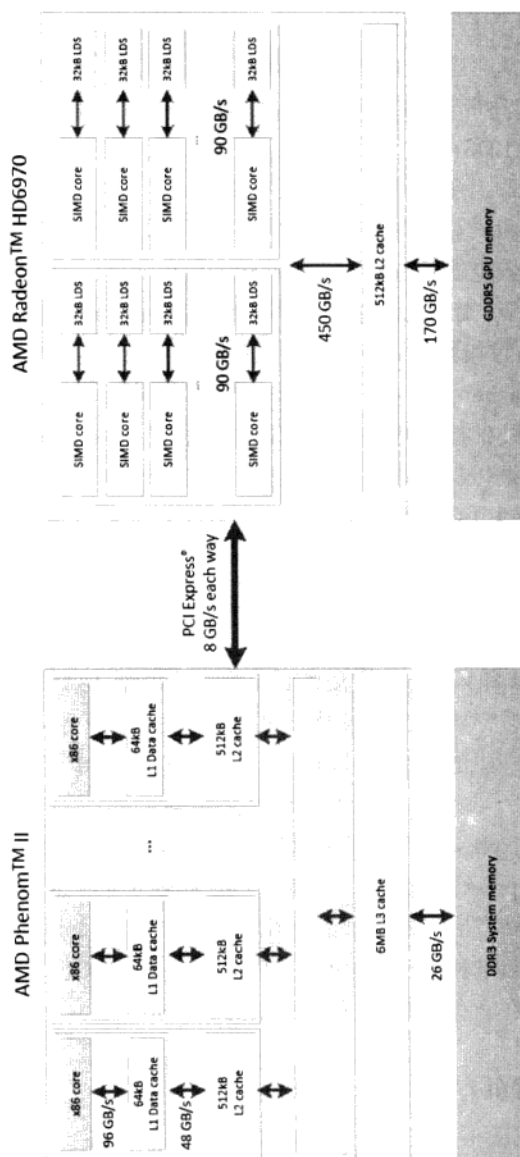


图 6.6

AMD Phenom II CPU 和 AMD Radeon HD6970 GPU 不同内存层次的大致带宽。特别要注意, 相对于其他层次, PCI-E 总线带宽很低, 尤其是相对于其接口两端的 cache

任何明显的延迟都会导致工作流挂起并降低执行效率。因为 GPU 的核使用多线程和宽 SIMD 设计来最大化吞吐量，这是以延迟作为代价的。内存系统也采用类似的设计来最大化带宽以满足吞吐量，当然也会有一些延迟的开销。

Radeon 设计上，和高带宽 GDDR5 相关联的有限缓存机制是通过下列方法实现的：

- 本地数据共享(LDS)
- 一个高层次的片上多线程
- 通过只读缓存和只写缓存来合并读写

LDS 具有高带宽、低延迟的特点，并允许程序员控制读/写访问。这种可编程的数据重用机制可减少资源浪费，还可以获得比硬件控制的高速缓存更有效的面积/功耗比。对垃圾数据的访问的减少(垃圾数据是指加载进 cache 但没有被用到的数据)意味着每单位面积上的 LDS 可以比 cache 小，并可以减少在更高级 cache 层次上的数据传输。

SIMD 核的多线程硬件控制允许硬件来隐藏访存延迟。执行访存操作时，运行在一个 SIMD 单元的线程将被挂起，并进入一个单独的内存控制器。这个线程在访存返回之前不会重新运行。为达到更高的性能和利用率，SIMD 核必须运行足够数量的线程。尽管 SIMD 核理论上可以同时部署 32 个 wavefront，8 个 OpenCL work-group，然而在很多情况下，每个 SIMD 核部署 4 个或 8 个 wavefront 就足够了。线程的部署必须要平衡寄存器和 LDS 的使用。

系统中的只读 cache 和只写 cache 提供了一种过滤机制，这种机制将向量访存操作中的复杂聚集写和离散读访存模式合并成最大的访存单元。在基于 DRAM 的系统中，大向量读取比从 x86 代码中产生的较小的按时间分布的读取高效得多，而且需要的临时高速缓存也少。

图 6.6 显示了 PCI Express 总线。在 CPU 主存和 GPU 之间的所有数据传输必须经过这条通道。因为 PCI Express 带宽远低于 DRAM 的访存带宽，甚至低于片上缓存带宽，因此，这会成为通信密集型应用的一个主要性能瓶颈。在 OpenCL 应用中，我们需要尽量减少内存拷贝操作的数量和大小。如果一个应用需要通过 PCI Express 总线反复来回拷贝数据，那么很难指望它能在 GPU 上获得高性能。

基于 clause 的 SIMD 执行

当前 AMD GPU 架构采用的是解耦执行方式，使整体设计在操作上是高度并行的(超标量)。在这样的解耦执行方式上编写的程序在 ISA 层次上形成了一个有趣的基于 clause 的格式。

在这里, **clause** 是一个由软件控制的机器调度单元, 它将任务按照已定义类型划分成块并在不同的超标量执行单元上执行。如同观察 x86 汇编代码中出现多少条 SSE 指令就可以指出性能问题的所在, 对 GPU 架构的汇编代码有一个清晰的认识对优化 OpenCL 代码是很有帮助的。下面是为 Radeon HD6970 架构编写的一个简单示例代码:

```
; ----- Disassembly -----
00 ALU_PUSH_BEFORE: ADDR(32) CNT(5) KCACHE0(CB0:0-15)
    0 x: LSHL      ___, R1.x, (0x00000007,9.809089250e-45f).x
    1 y: ADD_INT   R0.y, R0.x, PV0.x
    2 z: SETGT_UINT R0.z, KC0[0].y, PV1.y
    3 x: PREDNE_INT ___, R0.z, 0.0f UPDATE_EXEC_MASK UPDATE_PRED
01 JUMP POP_CNT(1) ADDR(15)
02 ALU: ADDR(37) CNT(1) KCACHE0(CB0:0-15)
    4 w: ADD_INT   R0.w, R0.y, KC0[0].x
03 ILX: ADDR(80) CNT(1)
    5 VFETCH R4.x___, R0.w, fc1
      FETCH_TYPE(NO_INDEX_OFFSET)
04 ALU_PUSH_BEFORE: ADDR(38) CNT(1)
    6 x: PREDGT    ___, R4.x, 0.0f UPDATE_EXEC_MASK UPDATE_PRED
05 JUMP POP_CNT(2) ADDR(15)
06 ALU: ADDR(39) CNT(2)
    7 z: LSHL      R0.z, R0.w, 1
      w: MOV       R4.w, 0.0f
07 ILX: ADDR(82) CNT(2)
    8 VFETCH R0.xy___, R0.z, fc0 FORMAT(32_32_FLOAT)
      FETCH_TYPE(NO_INDEX_OFFSET)
    9 VFETCH R1.x___, R0.w, fc2
      FETCH_TYPE(NO_INDEX_OFFSET)
08 ALU: ADDR(41) CNT(3)
    10 x: LSHL      R7.x, R0.y, (0x00000002,2.802596929e-45f).x
      w: LSHL      R0.w, R0.x, (0x00000002,2.802596929e-45f).x
09 ILX: ADDR(86) CNT(4)
    11 VFETCH R3.xyz_, R0.w, fc1b4 FORMAT(32_32_32_FLOAT)
      FETCH_TYPE(NO_INDEX_OFFSET)
    12 VFETCH R6.xyz_, R7.x, fc1b4 FORMAT(32_32_32_FLOAT)
      FETCH_TYPE(NO_INDEX_OFFSET)
    13 VFETCH R2.x___, R0.x, fc3
      FETCH_TYPE(NO_INDEX_OFFSET)
    14 VFETCH R0.x___, R0.y, fc3
      FETCH_TYPE(NO_INDEX_OFFSET)
10 ALU: ADDR(44) CNT(25) KCACHE0(CB0:0-15)
    15 x: ADD       R5.x, -R3.x, R6.x
      y: ADD       R0.y, -R3.y, R6.y
```

```

        z: ADD      R0.z, -R3.z,   R6.z
        w: MOV      R8.w, 0.0f
16   x: DOT4_e      _____, PV15.x, PV15.x
        y: DOT4_e      _____, PV15.y, PV15.y
        z: DOT4_e      _____, PV15.z, PV15.z
        w: DOT4_e      _____, (0x80000000, -0.0f).x, 0.0f
17   y: ADD      _____, R1.x,   PV16.x
        z: ADD      T0.z, R1.x,   -PV16.x
18   x: MOV      R1.x, R0.w
        y: MOV      R1.y, R0.w
        z: MOV      R1.z, R0.w
        w: MUL_e      _____, R4.x,   PV17.y
19   x: RCP_e      _____, PV18.w
        y: RCP_e      _____, PV18.w
        z: RCP_e      _____, PV18.w
20   z: MUL_e      _____, T0.z,   PV19.x
21   y: MUL_e      _____, KC0[0].z, PV20.z
22   x: MUL_e      _____, R2.x,   PV21.y
        w: MUL_e      R0.w, R0.x,   PV21.y   VLC_120
23   x: MULADD_e   R4.x, -R5.x,   PV22.x, R3.x
        y: MULADD_e   R4.y, -R0.y,   PV22.x, R3.y
        z: MULADD_e   R4.z, -R0.z,   PV22.x, R3.z
11 MLM_RAT_CACHLESS_STORE_DWORD__N1: RAT(0)[R1], R4, ARRAY_SIZE(4)
    MARK VPM
12 ALU: ADDR(69) CNT(6)
24   x: MULADD_e   R8.x, R5.x,   R0.w,   R6.x
        y: MULADD_e   R8.y, R0.y,   R0.w,   R6.y
        z: MULADD_e   R8.z, R0.z,   R0.w,   R6.z
25   x: MOV      R7.x, R7.x
        y: MOV      R7.y, R7.x
        z: MOV      R7.z, R7.x
13 MLM_RAT_CACHLESS_STORE_DWORD__N1: RAT(0)[R7], R8, ARRAY_SIZE(4)
    MARK VPM
14 POP(2) ADDR(15)
15 NOP NO_BARRIER
16 END
END_OF_PROGRAM

```

用 00, 01, 02 等标记并且缩进量最小的代码是控制程序。代码运行在 wave 调度器上, 与任意一个运行在该调度器上的 wavefront 交叉执行。这个程序和调度器通过使用 ALU 任务的返回值来处理所有分支以及进行标记管理。相对于执行两个不同 clause 的延迟大约是 40 或更多个时钟周期, 执行一个控制流指令的延迟相对较高。第一个 clause 返回调度器执行控制流指令时, 如果代码中包含足够的 ALU 操作, 并且有足够多的

wavefront 在同时运行, 这个延迟是很容易隐藏的。看一眼 ISA 代码很容易了解在有足够多 wavefront 运行的情况下如何进行有效调度。

第 02, 04 行等, 是控制程序内的 ALU 操作, 这些声明初始化 ALU clause。然后调度器将控制权交给其中一个 SIMD 单元, 这个 SIMD 单元将处理该 clause 内的 ALU 操作。一旦 ALU clause 在 SIMD 引擎上开始执行, 在它执行完之前不会有别的控制流在这个 SIMD 引擎上执行。注意, 这些 ALU clause 有两个非常重要的特征, 尽管只有第一个直接出现在代码中:

- 这是一个单路指令流, 即它们不是宽 SIMD 指令
- 这些 clause 含有 VLIW 指令, 形如 x, y, z, w

使用类多路模型对 SIMD 架构进行编程在前面已经进行了讨论。在这种情况下, 这里看到的 ISA 代码实际上就是由一个典型的 OpenCL kernel 实例产生的。SIMD 引擎的执行单位是被称为 wavefront 的 64 路逻辑 SIMD 单元。每一个 wavefront 使用一个指令解码器, 并且拥有自己的指令流, 因此可被视为一个单独的硬件线程上下文。wavefront 内的 64 个 work-item 在 16 路 SIMD 硬件单元上完成执行需要四个时钟周期。不同的 wavefront 在指令流中在不同的点执行。所有分支的执行粒度都是 wavefront, 任何 wavefront 内部的分支都会被该 wavefront 内的所有 work-item 执行, 因此, 在 wavefront 内的分支执行效率是非常低的。在面向竞争性架构(例如 NVIDIA 的 GTX580)的开发中以及在面向宽向量架构的软件开发中, 无论是手动向量化还是硬件向量化, 都存在着同样的问题。

执行这段代码的 SIMD 单元示意图如图 6.7 所示。每个 SIMD 单元包含一个 32 端口的 LDS, LDS 访存需要 4 个操作延迟并在其接口处有原子操作单元。这些原子单元意味着在 LDS 上执行的无返回原子操作可以和在 ALU 上执行的 ALU 操作同时进行, 从而提供了更进一步的并行性。如果 LDS 的读/写出现冲突, 就重复执行直到所有的操作都完成。对于读操作, 这会导致 ALU 挂起。

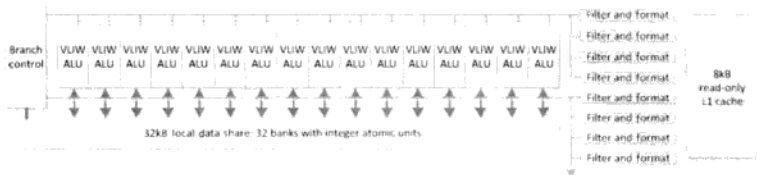


图 6.7

Radeon HD6970 架构上的 SIMD 核。这个 16 路 SIMD 单元用 8 个时钟周期同时交替完成 2 个 64 路的 wavefront。每个 SIMD 单元有 32 kB 的片上 LDS

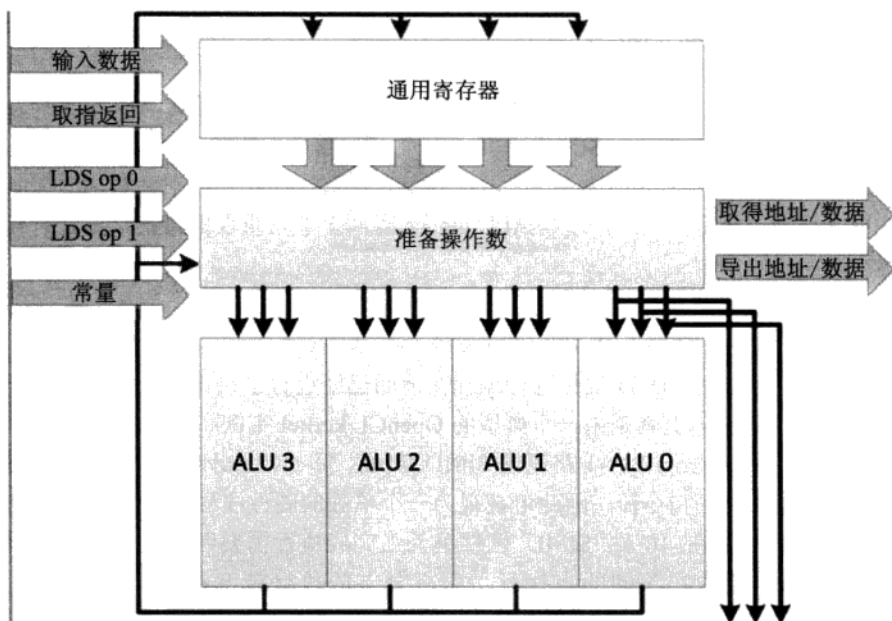


图 6.8

SIMD 单元中的一路，显示了执行硬件的 VLIW 格式，这个格式可以在指令流中的 ALU clause 内部看到。这 4 个 VLIW 单元可以通过各种联合方式执行基本的算术操作。例如，利用 4 个单元，可以执行 4 个 32 位(或者一个 64 位)的浮点数乘加运算或者 FMA(fused multiply-add)运算、两个 64 位的浮点数加法、一个 32 位的浮点数特殊运算(sin 和 log 等)

如图 6.8 所示，VLIW 包可以非常灵活地在 GPU 架构上执行，可以运行在 4 个结构近似的 ALU 上。VLIW 单元的每一路都可以执行 32 位的浮点数运算、24 位的整型数运算(使用浮点尾数)以及 32 位的整型数加法。这些操作不需要相同，就像代码中第 15 个代码包那样：包含 3 个浮点数加法和一個 move 操作。VLIW 可以使用各种方式联合资源来执行双精度和整型数运算。例如，每个时钟周期可执行一次双精度浮点数乘法，它比单精度浮点数乘法慢 4 倍。

在一个具有强大 ALU 计算能力和少量逻辑控制的架构上，VLIW 的执行方式具有非常高的性能。

然而，在绝大多数情况下，编译器提取 VLIW 包的必要性也导致效率的显著降低。

在前面的代码实例中，我们可以看到只有 4 个 ALU 包是完全满操作的。当 VLIW 包少于 4 路执行时，就可能造成执行资源的浪费。而当代码中不存在依赖性时，编译器可以非常好地从代码中抽取 VLIW 包，例如，循环展开或向量化编程都可以提高执行资源的利用率，因为它们为编译器提供了尽可能多的相互独立的指令。在这种情况下，不同于 AMD CPU 的 OpenCL 实现采用完全显式向量化，AMD Radeon HD6970 架构代码的向量化采用了显式和隐式两种方式。隐式向量化是指根据向量架构进行编码，并根据硬件的向量化特征将 ISA 代码表示的任务映射到硬件上。显式向量化是指将小部分代码向量化以协助填充 VLIW 包，从而最大化利用内存带宽和向量寄存器资源。

第 7 行代码初始化一个纹理 `clause`。在纹理 `clause` 中，读内存操作是在纹理读取硬件上执行的。这个操作与在 SIMD 单元上执行的 ALU 操作是相互独立的。因此，一个 `wavefront` 的内存操作可以与另一个 `wavefront` 的 ALU 操作同时进行。

资源分配

GPU SIMD 单元寄存器和 LDS 存储空间是有限的。每一个 SIMD 单元有 256 kB 的寄存器，对一个 `wavefront` 来说，每一个 `work-item` 可分配 256 个 128 位寄存器。这些寄存器将根据在 SIMD 单元上同时运行的 `wavefront` 的数目进行分配。每个 SIMD 单元有 32 kB 的 LDS，LDS 是一个具有 32 个 bank 的随机存取 SRAM，并为每个 `work-item` 提供了两个端口进行读/写。通过定义在 `kernel` 内部的本地内存分配请求以及 OpenCL 运行时参数传递机制可为运行在 SIMD 单元的 `work-group` 分配 LDS。

因为 OpenCL 程序的运行是标准映射，所以在 SIMD 单元上执行一个单独的 `kernel` 时，很有可能会遇到资源瓶颈。如图 6.9 所示，有两个 `work-group`，每个 `work-group` 包含两个 `wavefront`，每个 `work-item` 需要 42 个寄存器，每个 `work-group` 需要 12 kB 的 LDS。由于 LDS 的限制，每个 SIMD 上只能同时运行 4 个 `wavefront`，这个数也接近隐藏访存延迟所需要的 `wavefront` 的最小数。隐藏指令延迟和充分利用 SIMD 单元最少需要三个 `wavefront`：两个在 SIMD 单元的两个指令插槽中执行 ALU 操作，另一个执行访存操作。然而，在实际的情况中，由于控制流的原因，3 个 `wavefront` 是不够的。如果能够在 SIMD 上同时运行 6 个或者更多的 `wavefront`，我们就能更好地隐藏控制流延迟，特别是访存延迟。总之，同时运行的线程越多，越能更好地隐藏延迟。

有2个wavefront的workgroup，有128个workitem
 每个wavefront需要42个寄存器
 每个workgroup需要12kB of LDS

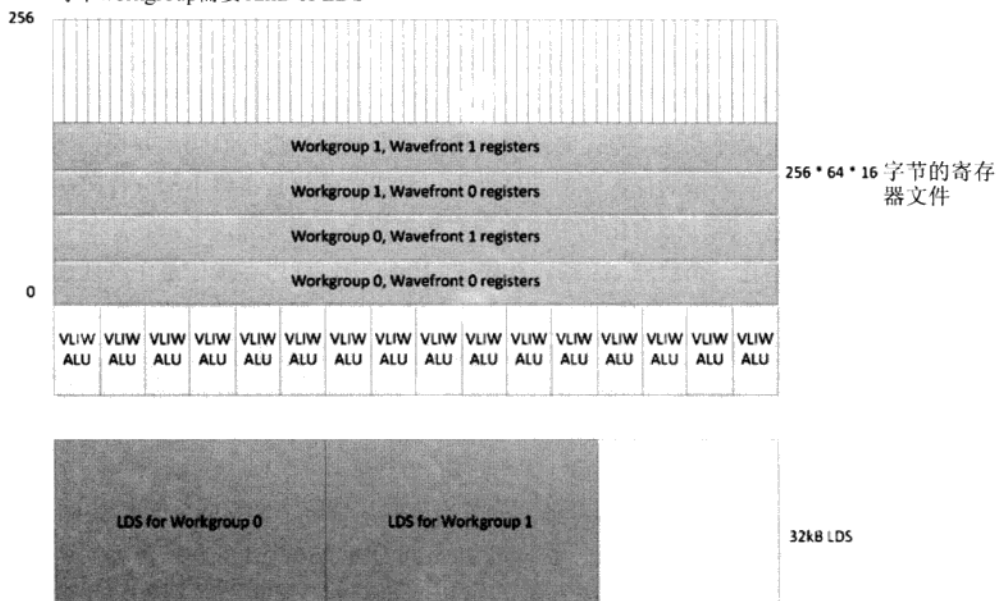


图 6.9

SIMD 资源分配。一个需要 12 kB LDS 的 work-group 包含 128 个 work-item，每个 work-item 需要 42 个寄存器。由于 LDS 限制，我们只能在每个 SIMD 单元上同时部署两个 work-group，即 4 个 wavefront

由于是 LDS 的限制，如果算法可行，最好首先将每个 work-group 的 wavefront 数目增加到三个。另外一种选择是减少每个 work-group 分配的 LDS，以使我们能够在每个 SIMD 上运行三个 work-group。这个方法对于一个 wavefront 正在等待同步或者访存而未在 SIMD 单元上运行的情况是非常有效的。

每个 work-group 只能运行在一个 SIMD 单元上。这样做的原因在我们看到每个 work-group 需要的状态存储量时就非常清楚了：每个 work-group 需要 12 kB 的 LDS 和 84 kB 的寄存器。如果将这些数据转移到内存上，再转移到另外一个 SIMD 单元上，将是一个非常大的数据传输量。另外，控制流代码在 wave 调度器上运行时，如果没有另外一个带有 ALU 操作的 wavefront 已经准备好在 SIMD 单元上调度运行，硬件会被闲置下来。

OpenCL 的内存性能

OpenCL 全局内存

在第 3 章中，我们就内存的空间和时间局部性进行了讨论。OpenCL 程序的性能与内存的有效使用有很大关系。不幸的是，高效内存访问高度依赖于运行 OpenCL 程序的设备。访存模式在 GPU 上是高效的，然而在 CPU 上可能是低效的。甚至我们将 OpenCL 程序在不同厂商的 GPU 平台上迁移时，我们也会看到访存效率的显著差异。然而，依然有许多通用的方法可以使 OpenCL 程序在多个硬件设备上高效运行。

无论什么情况，分析内存性能的一种有用方式都是判断 kernel 的实际吞吐量。一个简单方法是计算 kernel 的内存带宽：

$$EB = (B_r + B_w) / T$$

其中：

EB 是有效带宽

B_r 是从全局内存中读取的字节数

B_w 是写入到全局内存的字节数

T 是 kernel 的执行时间

T 可以通过使用程序剖析工具例如 AMD Stream Profiler(我们将在第 11 章中讨论)来获得。 B_r 和 B_w 可通过计算每个 work-item 读取或者写入的字节数乘以 work-item 总数来获得。当然，在很多情况下这个数是估算的，因为我们可能在有数据依赖的分支中读/写数据。

一旦测得 kernel 的实际使用带宽，我们就可以和设备的峰值带宽相比较，以此来确定我们与峰值性能的差距。越接近峰值性能，内存系统的利用率就越高。如果与峰值性能差距较大，我们就需要考虑调整访存模式来改进带宽利用率。

空间局部性是 OpenCL 访存中一个非常重要的方面。运行 OpenCL 程序的绝大多数体系架构在一定程度上都是基于向量的(不管是类 SSE 向量指令还是如 AMD IL 或者 NVIDIA PTX 的硬件向量化类通道编程)，它们的内存系统都受益于向量化访存。另外，本地化访存也提供了缓存优势。

大多数现代 CPU 都有一个向量指令集，比如各种版本的 SSE 和 AVX。为了高效访存，我们需要仔细设计代码满足对齐和向量化读取以使用这些指令集。对于一个给定的短向量，进行向量化读取的最有效方式就是通过使用向量数据类型(例如 float4)，传递给编译器尽可能多的信息。这种访存方式可以充分利用 cache line，使数据在 cache 和寄存器间的迁移尽可能高效。

然而，对于很多 CPU 来说，缓存可以减小在执行小规模、非对齐及随机读取时的性能损失。图 6.10 和图 6.11 说明了连续读取和随机读取的差异。随机读取不仅会命中多个 cache line(造成更多的 cache 未命中)，而且也会导致数据在内存系统中的传输低效。

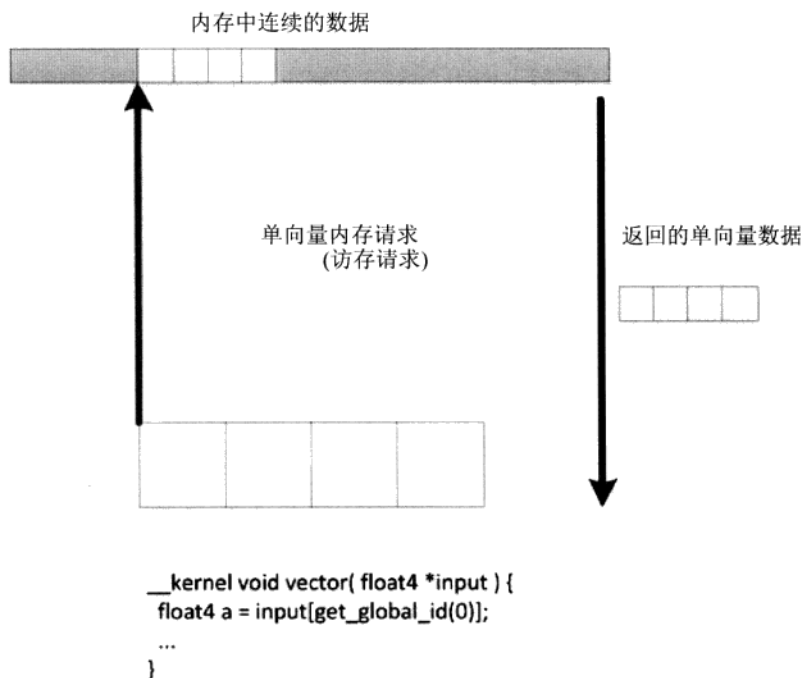


图 6.10

使用向量化读取，为通过内存系统高效返回数据提供更多的机会

如同在第 3 章中讨论的那样，GPU 和 CPU 的内存架构差异显著。GPU 使用多线程来掩盖访存延迟，更注重 ALU 能力而不是缓存和复杂的无序逻辑。为能有效利用传统 GPU 的大量计算资源，内存系统的高带宽变得越来越重要。许多现代 GPU 体系架构，特别是高性能的桌面版本，例如最新的 AMD Radeon 和 NVIDIA GeForce，都采用了宽 SIMD 架构。可以想象一下，图 6.11 的随机读取方式移植到 64 路硬件上可能的性能损失。

即使在这些架构中，高效访存模式也是不同的。对于带 SSE 指令集的 x86 CPU 来说，我们要使用 float4(128 位)的访存模式，而且要尽量让访存落在 cache line 内以减少 cache 未命中的数量。

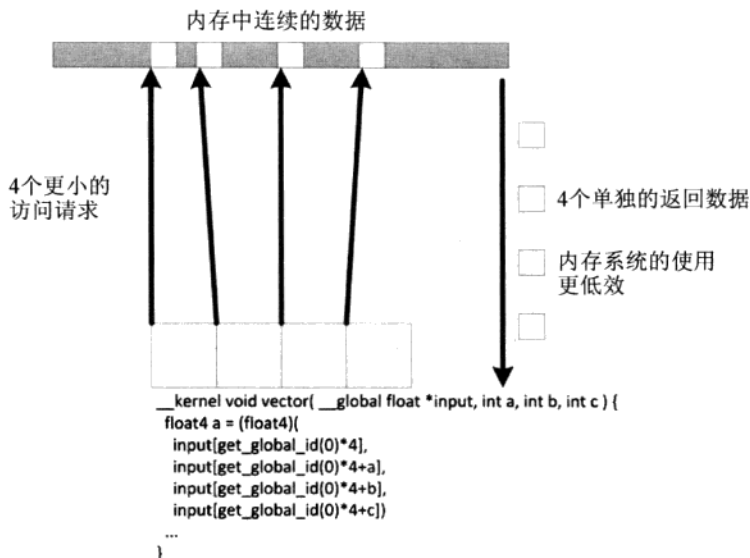


图 6.11

随机读取。如果我们采用更加随机的访存模式，也像图 6.10 那样传递 4 个 float 类型的数据，并返回更小的内存块。这样的内存系统利用是非常低效的。

对于 AMD Radeon HD6970 GPU 体系架构来说，16 个连续 work-item 的内存访问将合并成一次访存请求。如果不能有效响应这些访存请求，它们将在内存系统中延迟。最有效的访存方式是，每个 work-item 读取 128 位数据，16 个 work-item 的访存就可以形成一个连续且大小为 256 字节的内存区域。这样，内存系统将为这 16 个 work-item 的访存创建一个单独的访存请求。为实现在不同体系架构间的性能移植，一个有效的通用解决方案是尽量有效地压缩访存请求，以便宽向量机(AMD 和 NVIDIA GPU)和窄向量机(x86 CPU)都可以尽可能高效利用内存系统。为实现这一点，整个 work-group 访存请求的基址应该对齐到 $\text{workgroupSize} * \text{loadSize}$ ，loadSize 是每个 work-item 访问的字节数，其大小应该是合理的——对于 x86 CPU 以及 AMD GPU 架构来说是 128 位，如果是支持 AVX 的架构，其大小要扩展为 256 位。

处理不同内存系统的细节时，新的难题又产生了，如减少片外内存冲突。例如，让我们考虑 AMD Radeon HD6970 的地址分配方式。如图 6.12 所示，低 8 位用来选择位于同一个内存 bank 内的字节；给定 cache line 和子 cache line 的读取范围。

31:15 14:11 bank 10:8 channel 7:0 address

图 6.12

Radeon HD6970 架构全局内存的地址组织方式

如果我们试图从一个二维数组中读取一系列数据，就已经知道这是在低效使用片上总线。也意味着我们需要在设备上同时运行多个 **work-group** 以访问不同的内存 **channel** 和 **bank**。内存 **channel** 是与片外内存链接的片上内存控制器(如图 6.13)。我们希望访存能够尽量多使用内存系统中不同的 **bank** 和 **channel**，最大化数据访问的并发性。然而，如果一个 **wavefront** 的向量化访存占用了多个内存 **channel**(或者 **bank**)，会阻塞其他 **wavefront** 的内存访问，从整体上减少内存吞吐量。理想情况下，一个 **wavefront** 使用一个 **channel** 和 **bank**，以使多个 **wavefront** 可以并行访问多个 **channel**。这样一来，数据流就可以高效地进出内存。

为避免使用多个 **channel**，一个 **wavefront** 的访存地址范围应限制在大小为 64 个字的区域内，这可以通过 **wavefront** 内的所有 **work-item** 都是从连续的地址空间中读取 32 位的数据来实现。可能出现的最坏情况是多个 **wavefront** 的所有 **work-item** 的读取地址从第八位开始都相同：使用同一个 **channel** 和 **bank**，这时会串行化访存，实际利用带宽只能达到峰值带宽的很小一部分。AMD 架构在这方面的详细讨论可参考 AMD OpenCL 编程指南(Advanced Micro Devices, Incorporated, 2011)。其他架构的相关信息也可以从相关厂商处得到——例如“NVIDIA CUDA 编程指南”(NVIDIA Corporation, 2010)。

本地内存——软件管理的 cache

大多数支持 OpenCL 的设备都支持某种形式的缓存。由于面向图形设计，许多 GPU 都只为只读数据提供缓存功能，以提供一定程度的数据重用。

保证数据可广泛用于各种各样的设备，可用的最简单途径是使用 **image** 内存(参见第 5 章)。**image** 内存将数据集映射到纹理读取硬件上，并假定不需要复杂的过滤和二维访存模式，以改善 GPU 的内存访问效率。然而，相较于读取数据的活动线程数，GPU **cache** 是非常小的。在本地地址空间上使用程序员可控的 **scratchpad** 内存是缓存读取数据的有效方法。相较于硬件控制的 **cache**，空间浪费更少，能效比更高，在给定区域上性能更高，等等。同时也是同一个 **work-group** 内的 **work-item** 相互交换数据的一个更重要的，保证访存低延迟的有效方法。

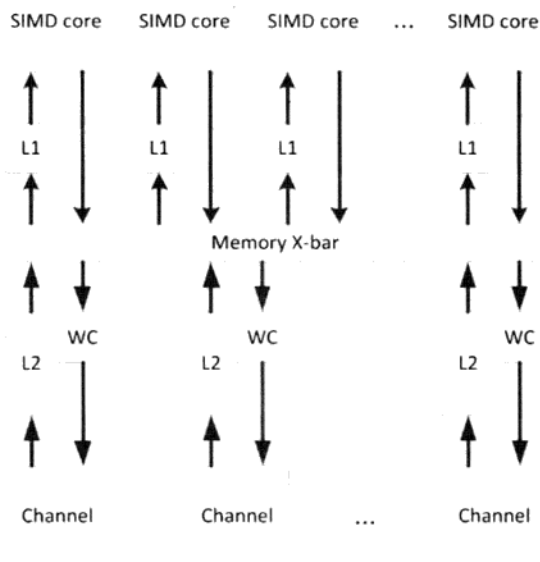


图 6.13

HD6970 的内存系统，说明多个 SIMD 核是如何通过交叉总线(crossbar)在同一个 channel 和 bank 发生冲突的

图 5.11 的代码就是这种方法的一个简单例子。这段代码将数据从 A 加载到 C，然后进行多次访问，避免了重复访问 DRAM。同时，这段代码从 `aLocalArray` 中加载一个值，虽然只加载一次却能被 work-group 内的所有 work-item 重用，大大减少了内存传输。在很多 CPU 的硬件实现中，很多这样的数据重用是在 cache 中实现的(特别是 x86 处理器)。只有了解底层的内存访问模式，我们才能够控制实际应用程序中数据的重用量。

当然，在考虑如何以最佳方式进行数据本地化优化时，也有折中情况。在很多情况下，将数据加载到本地内存，然后再传递给 ALU(可能通过寄存器)的额外复制指令是有开销的，可能比直接从 cache 重用数据低效。要对同一个位置进行大量读写；或者当写操作生命周期非常长，同时有大量读操作要使用其结果；或者通过手动 cache 分块能有效减少二维数据访问模式中经常出现的 cache 冲突时，将数据加载到本地内存是非常有用的。

在读/写操作中，特别是对于那些只有只读 cache 的硬件架构来说，本地内存的优势更加明显。例如，考虑下面较基本的前缀和算法：

```
void localPrefixSum(
    __local unsigned *prefixSums,
    unsigned numElements) {

    // Run through levels of tree halving sizes of the element set
    // performing reduction phase
    int offset = 1;
    for( int level = numElements/2; level > 0; level /= 2 ) {
        barrier(CLK_LOCAL_MLM_FENCE);

        for( int sumElement = get_local_id(0);
            sumElement < level;
            sumElement += get_local_size(0) ) {

            int ai = offset*(2*sumElement+1)-1;
            int bi = offset*(2*sumElement+2)-1;

            prefixSums[bi] = prefixSums[ai] + prefixSums[bi];
        }

        offset *= 2;
    }
    barrier(CLK_LOCAL_MLM_FENCE);

    // Need to clear the last element
    if( get_local_id(0) == 0 ) {
        prefixSums[ numElements-1 ] = 0;
    }

    // Push values back down the tree
    for( int level = 1; level < numElements; level *= 2 ) {
        offset /= 2;
        barrier(CLK_LOCAL_MLM_FENCE);

        for( int sumElement = get_local_id(0);
            sumElement < level;
            sumElement += get_local_size(0) ) {

            int ai = offset*(2*sumElement+1)-1;
            int bi = offset*(2*sumElement+2)-1;

            unsigned temporary = prefixSums[ai];
            prefixSums[ai] = prefixSums[bi];
            prefixSums[bi] = temporary + prefixSums[bi];
        }
    }
}
```

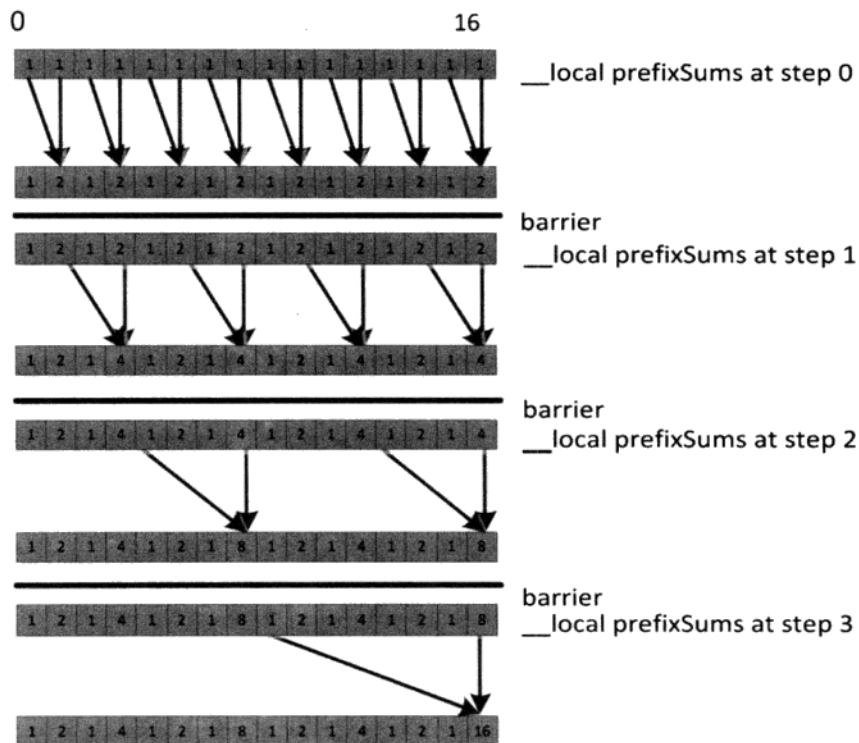


图 6.14

在本地内存中，使用 8 个 work-item 对 16 个元素求前缀和的算法步骤。每个累加阶段后都紧跟一个传播阶段，这个传播阶段将计算结果传递回计算树以保证每个计算位置结果的正确性

尽管前面的代码在很多架构上并不是最优的，但的确通过使用本地内存存在 work-item 间有效地共享数据。第一层循环的数据流如图 6.14 所示($\text{level} = \text{numElements} \gg 1$ to 0)。注意，每一次循环迭代都更新一系列值，在下一次迭代中可能要使用不同的 work-item。同时也要注意，在每次迭代中，work-item 的数量都会随着计算量的减小而减少。内层循环会关闭多余的 work-item，以避免分支执行。同时，我们插入 barrier 操作来保证 work-item 之间的同步，以确保数据已经为下次迭代执行做好了准备。

前面讨论的前缀和代码使用本地内存的方式对大多数宽 SIMD 架构来说是低效的，例如高性能 GPU。正如全局内存相关讨论所提到的那样，内存系统往往都通过 bank 化

来允许大量访存端口而不需要在每个内存位置都设定多个访存端口。`scratchpad` 内存硬件就采用了这种结构：每个 `bank` 都能进行多次读取或者并发读/写(或者其他多路配置)，其中多次读取将分散到多个 `bank` 上进行。当我们使用宽 SIMD 硬件进行访存时，这是一个重要的考虑因素。AMD GPU 的每个 16 路 SIMD 一个时钟周期可通过 VLIW 单元同时调度运行 4 条指令，包括两条加载指令。因此，当本地内存指令被包含进 VLIW 包时，SIMD 单元一个时钟周期内可进行 32 次 LDS 读/写。如果每个 `bank` 都支持一个访存端口，只有当所有的访存都访问不同的 `bank` 时才能实现这样的吞吐量，因为每个 `bank` 只能提供一个值。在这个架构中，因为每个 `work-item` 一个时钟周期能执行两次读/写操作，所以本地内存由 32 个 `bank` 组成。在其他竞争性架构上也有相似的情况，例如，NVIDIA Fermi 架构也有 32 个 `bank` 的本地内存。

本地内存的问题并不像全局内存那么严重。在全局内存中，分散的随机访存模式可能会导致 `cache line` 未命中从而引起访存延迟。在本地内存中，至少在实际的 `scratchpad` 架构中，程序员知道数据是何时加载进去的。唯一需要保证的是读/写指令的 16 次访存都访问的是不同的 `bank`。

图 6.15 说明了一个简单的比较：前缀和算法(图 6.14)的第二步访问本地内存的 8 个 `bank`，每个 `work-item` 一个时钟周期能执行一次本地内存操作。在这种情况下，本地内存一个时钟周期可返回 8 个值。当前缀和算法的第一步执行一组访存操作时，我们会得到什么结果呢？

注意，本地内存的 16 个元素被扩展为两行。每一列是一个 `bank`，每行是 `bank` 内的内存地址。假设 `bank` 的宽度为 32 位，内存地址的拆分如同图 6.15 顶部所示。两个连续的内存字位于不同的 `bank` 中。像全局内存那样，当一个合适长度的 SIMD 向量指令访问连续地址时，就可以有效访问本地内存 `bank` 而不产生冲突。然而，如图 6.15 所示，我们看到的是一个不同的行为所示。如下代码，第二次访问本地内存时，试图从 `prefixSums[bi]` 中位置为 3, 7, 11, 15 处读取数据：

```
prefixSums[bi] = prefixSums[ai] + prefixSums[bi];
```

如同图 6.15 所示，3 和 11 位于 `bank 3` 中，7 和 15 位于 `bank 7` 中。从一个 `bank` 中同时读取两行是不可能的，访存请求会被硬件串行化，从而导致读延迟。为了获取高性能，我们可能希望重构代码来避免冲突。一个有用的技术就是添加向内存地址填充，图 6.16 便是一个相关的简单例子。

内存地址:

n:5 row	4:2 bank	1:0 byte
---------	----------	----------

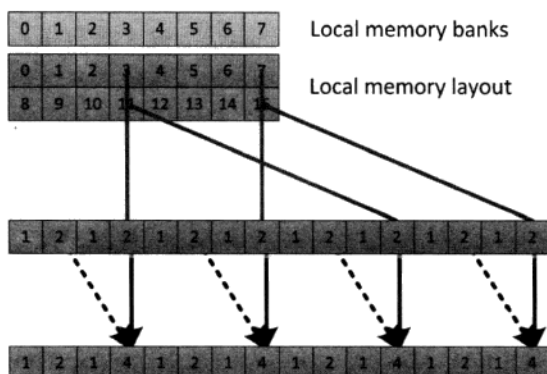


图 6.15

图 6.14 第一步在有 8 个 bank 的本地内存上的操作，一个 work-item 一个时钟周期进行一次访存

内存地址:

n:5 row	4:2 bank	1:0 byte
---------	----------	----------

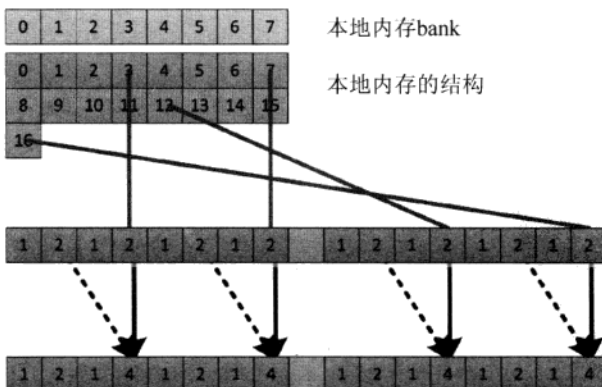


图 6.16

在图 6.15 的数据结构中增加 padding 以解决 bank 冲突

在第一次 bank 对齐后,通过地址转换,我们可以改变平均访存跨度来避免冲突。不幸的是,这样做增加了地址计算开销,可能比 bank 冲突的开销还要大。因此,这是一个与架构相关的调优折中的例子。

在进行本地内存分配时应该小心。任何使用非硬件管理 scratchpad 内存的设备,其本地内存都是有限的。比如 AMD Radeon HD6970 GPU,本地内存的大小是 32 kB,是 OpenCL 标准的最低要求。32 kB 的本地内存由同时运行在 SIMD 单元上的所有 work-group 共享,认识到这点非常重要。此外,由于 GPU 通过在每个 SIMD 单元上部署大量线程来隐藏延迟增大吞吐量,因此,部署的 work-group 越多,越能充分利用硬件。如果每个 work-group 需要使用 16 kB 的本地内存,那么每个 SIMD 单元同时只能部署两个 work-group。如果这些 work-group 只含有少量的 wavefront(两个),就不会有足够的硬件线程来隐藏访存延迟。因此,在一个多线程设备上,本地内存分配要在由于数据共享而得到的效率收益和由于硬件线程的减少而造成的效率损失间进行权衡。

OpenCL 定义了查询设备上可用本地内存大小的 API,而且这个值可以在程序编译和调度前通过参数传递给 kernel。如下代码中,第一次 API 调用查询本地内存类型以确定本地内存是专用的还是在全局内存中(可能缓存,也可能不缓存,这也是可以查询的),第二次调用将返回本地内存大小:

```
cl_int err;
cl_device_local_mem_type type;
err = clGetDeviceInfo(
    deviceId,
    CL_DEVICE_LOCAL_MEM_TYPE,
    sizeof(cl_device_local_mem_type),
    &type,
    0);
cl_ulong size;
err = clGetDeviceInfo(
    deviceId,
    CL_DEVICE_LOCAL_MEM_SIZE,
    sizeof(cl_ulong),
    &size,
    0);
```

小结

本章介绍了一个非常具体的 OpenCL 到硬件架构的实现,同时也介绍了 OpenCL 到 CPU 架构与到 GPU 架构映射的细微差异。

本章的核心原则适用于所有 CPU 和 GPU 架构，然而通过改变向量长度(在 NVIDIA GPU 为 32，AMD GPU 为 32/64，在 CPU 上要短得多)以及改变线程上下文管理和指令调度策略，很容易产生性能差异。显然，一本书不可能涵盖所有的硬件架构，但可以讨论其中的一个典型例子。同时也希望无论本节是否提及 OpenCL 设备，我们都能够通过深入研究各厂商提供的技术文档写出高效的代码。

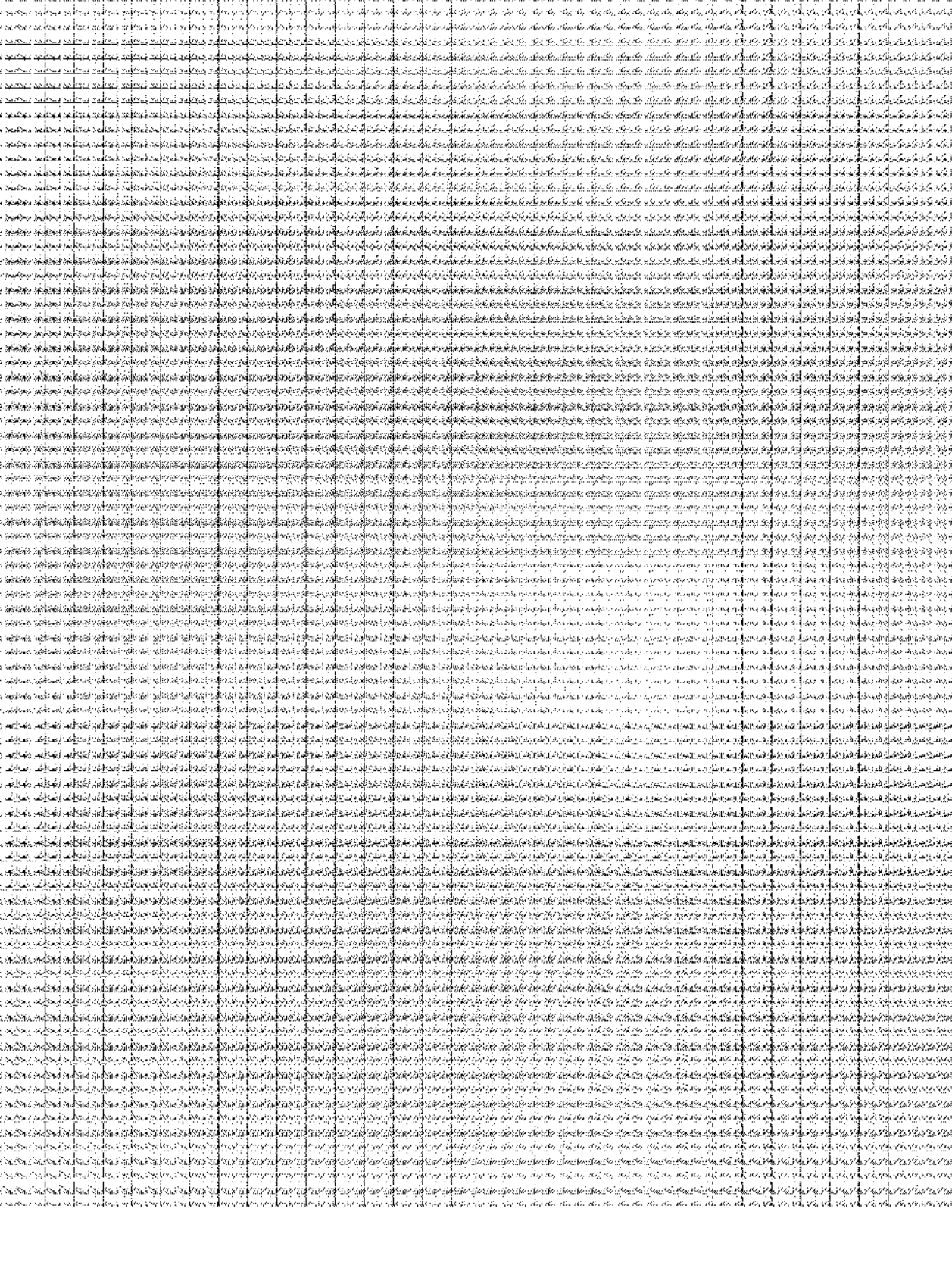
参考文献

Advanced Micro Devices, Incorporated. (2011). *The AMD Accelerated Parallel Processing—OpenCL Programming Guide*. Sunnyvale, CA: Advanced Micro Devices, Inc.

Association for Computing Machinery. (2010). *PACT 010: Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques: September 11–15, 2010, Vienna, Austria*. Association for Computing Machinery.

Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B. R., & Zheng, B. (2010). *Twin Peaks: A Software Platform for Heterogeneous Computing on General-Purpose and Graphics Processors*. New York: Association for Computing Machinery.

NVIDIA Corporation. (2010). *NVIDIA CUDA C Programming Guide*. Santa Clara, CA: NVIDIA Corporation.



OpenCL 案例学习 1：卷积

引言

在第 4 章中，我们介绍了使用 OpenCL image 实现卷积的简单实例。image 的优点在于自动处理越界访问(通过 clamping 或者 wrapping 访问)，简化了卷积滤波器访问图像越界时所需的编码处理，因此可减少控制流的开销。image 还支持数据缓存和数据访问转换，能够提高内存系统的性能。在 GPU 环境中 image 的自动缓存机制胜于不缓存的性能。然而在许多情况下，有效使用本地内存能得到比 image 更好的性能。本章以卷积滤波为例，提供一些关于如何有效使用本地内存的直观概念，我们鼓励读者比较这两种不同的实现方法，在具体的使用场合选择合适的优化技术。

计算卷积的 kernel

OpenCL 卷积 kernel 可自然分成 3 个步骤：(1) 将输入数据从全局内存缓存到本地内存；(2) 执行卷积；(3) 将输出数据写回全局内存。本章重点关注第一步，即为底层内存系统优化数据缓存，在执行卷积的上下文中还会讨论循环展开。写回阶段非常简单，将不详细分析。在讨论过程中，虽然我们使用一个 7×7 的滤波器来阐释优化细节，但本章研究的优化原则能推广到使用不同滤波参数的实现中。整章内容将展示 OpenCL kernel 优化技术及相关的主机端代码。代码清单中提供了完整的参考实现。

选择合适的 workgroup 大小

如前所述，执行卷积时每个 work-item 根据滤波窗口的尺寸访问周围像素。滤波半径指除当前像素以外每个方向访问的像素数。例如，7×7 的滤波器需要在每个方向上访

问 3 个额外的像素，所以其半径为 3。从图 4.5 中很容易看出，相邻的输出点具有数据访问的二维局部性。

每个工作区域根据输入滤波器尺寸可能需要填充像素。因此为了提高效率，我们应该使用二维区域访问，并且最大限度地减少边界填充元素相对输出数据大小的比例来提高输入输出效率。对于该实例，我们考虑的 `work-item` 映射是：每个 `work-item` 产生一个输出结果，在第 9 章中的直方图实例中将考虑每个 `work-item` 产生多个输出的优化策略。图 7.1 显示的是一个 OpenCL `workgroup` 中对给定的工作区域填充像素。

在 OpenCL 编程时，必须同时考虑 `work-item` 的创建和算法设计，尤其是在使用本地内存进行优化时。对卷积而言，`workgroup` 的大小与本地内存上的数据缓存算法密切相关。缓存数据有两种简单的方法。第一种方法是创建与缓存在本地内存中的数据元素数一样多的 `work-item`。换言之，创建的 `work-item` 数等于输出像素和填充像素的数之和。使用这种方法，每个元素只需简单地从全局内存复制一个像素到本地内存，表示边界像素的 `work-item` 在卷积计算中将处于闲置状态。这种方法的缺点在于对更大的滤波窗口，每个 `workgroup` 不能计算输出太多的像素。对 GPU 而言，`wavefront` 可能会被碎片化，导致 ALU 计算周期被浪费。第二种方法是创建与需要执行卷积的像素数相同的 `work-item`。在这种方法中，因为 `work-item` 的数小于本地内存中缓存的像素数目，所以一些 `work-item` 需要拷贝多个元素，但在卷积计算中不会出现闲置的 `work-item`。这种做法显然更适合于大尺寸滤波器，因为填充元素的数不会限制产生输出像素的 `work-item` 数。对于这个例子，我们使用第二种方法创建 `work-item`，因为它更适合于面向 GPU 的 OpenCL 环境。

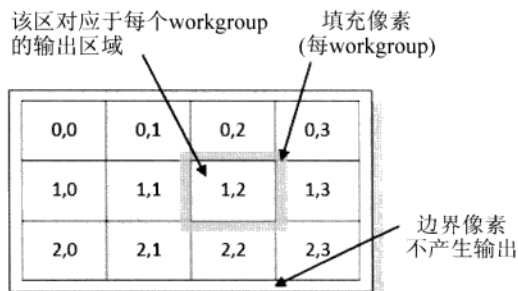


图 7.1

每个 `workgroup` 有自己的输出区域，其中存放该 `workgroup` 的计算结果。每个 `workgroup` 缓存和自己输出区域对应的源图像数据，并依据滤波半径填充像素

更进一步采取这种优化方法，可以创建比每个 workgroup 输出像素数更少的 work-item。读者很容易从之前讨论的算法出发，体会各种 work-item 数量情况下的权衡。寻找最优组合意味着需要探索很大的设计空间。

选择合适的 workgroup 大小使程序运行高效需要考虑到底层的内存架构。第 5 章讨论了在面向向量处理器进行优化时，特别是 AMD 6970 GPU，重要的是确保多个连续的 work-item 发出的访存请求能合并为单个内存访问，16 个连续的 work-item 均读取地址对齐的 128 位内存能最大利用内存带宽。而在 NVIDIA 平台上，当 32 个 work-item 以 128 字节对齐地址的方式读取 128 字节时，能合并为一次访问请求，从而最有效地利用内存带宽(NVIDIA, 2009)。这意味着从 128 字节对齐的地址开始，32 个 work-item 顺序访问 4 字节元素，这是最为理想的访问模式。此外还支持 64 和 32 字节的内存访问。因此，通过创建宽为 32 或 16 (work-item)的 workgroup 可以在不考虑具体平台的情况下实现高效的内存访问。workgroup 的 Y 维不影响内存访问的性能。AMD GPU 中 workgroup 中 work-item 的数目限制为 256，因此选择宽度为 32 时高度为 8，宽度为 16 时高度为 16。NVIDIA 可以支持更大的 workgroup 大小，但是“理想”的尺寸取决于硬件资源之间的相互作用关系。程序取得最佳性能的 workgroup 的大小，需要权衡内存访问效率和计算效率。本章的代码和分析中，我们使用 workgroup 大小为 16×16 的卷积算法。

执行从全局到本地内存的读取操作时，每个 workgroup 需要在每个维度上额外拷贝滤波半径两倍的数据。对于 7×7 的滤波器，意味着每个维度有六个额外的像素。计算 NDRange 大小时，图像周围边界像素的滤波半径(对 7×7 的滤波器值为 3)将不会计算输出值，因为它们会导致滤波器越界访问^①。对于尺寸为 imageWidth 和 imageHeight 的图像，每个维度上分别仅需要(imageWidth-2 *filterRadius)×(imageHeight-2* filterRadius)个 work-item。由于图像尺寸很可能不是 workgroup 大小的整数倍，必须在 X 和 Y 维度上创建额外的 workgroup(图 7.2)。每个维度上最后的 workgroup 可能没有被充分利用，在 OpenCL 的 kernel 中必须考虑这点。下面的函数将给定的某个数值(例如图像的宽度)四舍五入到另一个数(例如 workgroup 的宽度)的整数倍：

```
// This function takes a positive integer and rounds it up to
// the nearest multiple of another provided integer
unsigned int roundUp(unsigned int value, unsigned int multiple) {
```

① 该算法可通过检测越界访问并返回有效值来进行修改，使其能让边界像素产生输入输出值。

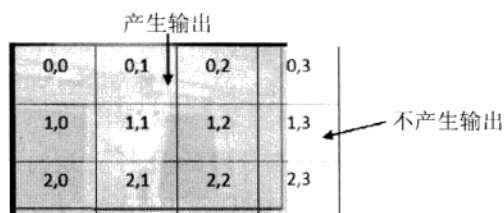


图 7.2

每个维度最后的 workgroup 可能包含越界像素，kernel 需要对越界像素对应的 work-item 进行处理

```
// Determine how far past the nearest multiple the value is
unsigned int remainder = value % multiple;

// Add the difference to make the value a multiple
if(remainder != 0) {
    value += (multiple - remainder);
}

return value;
}
```

计算图像尺寸为 `imageWidth * imageHeight` 的 NDRange 大小的代码如下：

```
// Selected workgroup size is 16x16
int wgWidth = 16;
int wgHeight = 16;

// When computing the total number of work-items, the
// padding work-items do not need to be considered
int totalWorkItemsX = roundUp(imageWidth - paddingPixels,
    wgWidth);
int totalWorkItemsY = roundUp(imageHeight - paddingPixels,
    wgHeight);

// Size of a workgroup
size_t localSize[2] = {wgWidth, wgHeight};
// Size of the NDRange
size_t globalSize[2] = {totalWorkItemsX, totalWorkItemsY};
```

将数据缓存到本地内存

使用本地内存缓存数据首先需要在本地内存中分配空间，可在 OpenCL kernel 代码中硬编码静态实现，或者通过指定大小将其作为参数传递给运行时动态分配。由于程序根据滤波尺寸缓存的数据量不同，下面代码动态分配本地内存空间，并将其作为第 7 个参数(参数索引为 6)传递给 OpenCL kernel：

```

int localWidth = localSize[0] + paddingPixels;
int localHeight = localSize[1] + paddingPixels;
size_t localMemSize = (localWidth * localHeight *
    sizeof(float));
...
// Dynamically allocate local memory (per workgroup)
clSetKernelArg(kernel, 6, localMemSize, NULL);

```

编写 kernel 时, 从全局内存到本地内存拷贝数据的过程最需要深思熟虑, 并且最容易出错。work-item 首先需要确定拷贝数据的全局内存地址, 然后确保不会访问到其工作区域或图像边界以外的地址。下面的代码显示了每个 work-item 要操作的内存区域(由全局内存和本地内存组成), 然后将数据从全局内存拷贝到本地内存中。图 7.3 描述了该实例执行数据拷贝使用的变量:

```

__kernel
void convolution(__global float* imageIn,
    __global float* imageOut,
    __constant float* filter,
    int rows,
    int cols,
    int filterWidth,
    __local float* localImage,
    int localHeight,
    int localWidth) {
    // Determine the amount of padding for this filter
    int filterRadius = (filterWidth/2);
    int padding = filterRadius * 2;

```

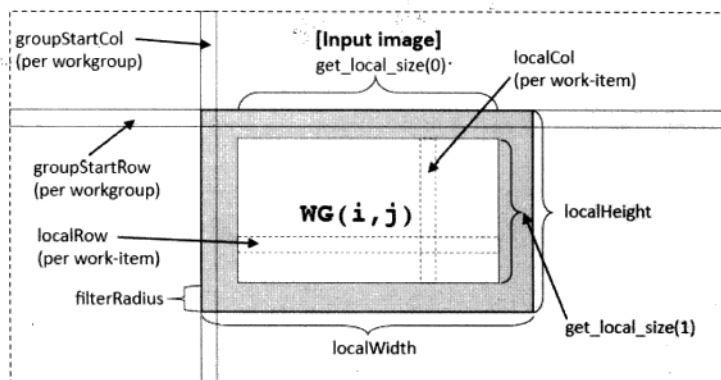


图 7.3

描述了 work-item 读取输入图像地址所需的变量


```

// Determine where each workgroup begins reading
int groupStartCol = get_group_id(0)*get_local_si.
int groupStartRow = get_group_id(1)*get_local_si.

// Determine the local ID of each work-item
int localCol = get_local_id(0);
int localRow = get_local_id(1);

// Determine the global ID of each work-item. work-
// representing the output region will have a unique
// global ID
int globalCol = groupStartCol + localCol;
int globalRow = groupStartRow + localRow;

// Cache the data to local memory

// Step down rows
for(int i = localRow; i < localHeight; i +=
    get_local_size(1)) {

    int curRow = groupStartRow+i;

    // Step across columns
    for(int j = localCol; j < localWidth; j +=
        get_local_size(0)) {

        int curCol = groupStartCol+j;

        // Perform the read if it is in bounds
        if(curRow < rows && curCol < cols) {
            localImage[i*localWidth+j] =
                imageIn[curRow*cols+curCol];
        }

    }

}

barrier(CLK_LOCAL_MLM_FENCE);

// Perform the convolution
...

```

数据拷贝之后需要 **barrier** 同步，因为 **work-item** 完成各自的数据拷贝需要的时间不一样，只有 **workgroup** 中所有 **work-item** 都完成数据传输后才能开始计算卷积。

内存访问对齐

在 NVIDIA 和 AMD 的 GPU 上对齐访问全局内存中数据都会提高程序性能。特别是对 NVIDIA，在对齐到 128 字节的地址上访问 128 字节数据能理想地映射到内存硬件。然而在该例中，宽度为 16 的 **workgroup** 只访问 64 字节数据段，所以数据应该对齐到 64 字节的地址。这意味着每个 **workgroup** 访问的首列地址应对齐到 64 字节。

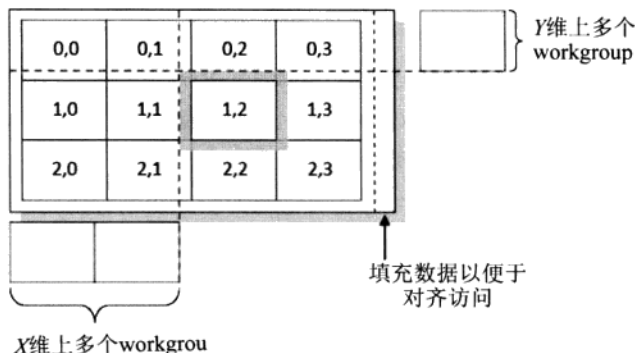


图 7.4

每个 workgroup 读取第一个填充像素的位置是每个维度中 workgroup 大小的倍数。确保对齐内存访问需要合理填充图像并适当调整 workgroup 的 X 维大小

在该例中，选择让边界像素对应的 work-item 不计算输出值，决定了所有 workgroup 的偏移量是 workgroup 尺寸的倍数(例如对 16x16 的 workgroup，workgroup <N, M>开始访问的数据列为 $N*16$)。图 7.4 的例子显示了每个 workgroup 的偏移值。为了确保每个 workgroup 正确的对齐，唯一的要求是给输入数据填充额外的列使其宽度变成 workgroup X 维的整数倍。

在主机端对数组数据进行手工填充复杂而耗时，有时候行不通。为了避免这种繁琐的手工修正，OpenCL 引入了 `clEnqueueWriteBufferRect()` 命令将主机端数组拷贝到一个较大的设备 buffer 中。当创建 buffer 时，用来确定大小的列数应是满足对齐后所需的元素数量。对于大小为 16x16 的 workgroup，列数应四舍五入到最近的 16 的倍数。调用 `clEnqueueWriteBufferRect()` 将数据从主机端拷贝到填充后的设备缓冲区的代码如下：

```
// Pad the number of columns (assuming 16x16 workgroup)
int deviceWidth = roundUp(imageWidth, 16);
// No padding needed for rows
int deviceHeight = imageHeight;

// Copy the input data on the host to the padded buffer
// on the device
clEnqueueWriteBufferRect(queue, d_inputImage, CL_1D,
    buffer_origin, host_origin, region,
    deviceWidth*sizeof(float), 0, imageWidth*sizeof(float),
    0, inputImage, 0, NULL, NULL);
```

在 Radeon 6970 GPU 上将数据对齐到 64 字节边界能够带来 10% 的性能提升。在 NVIDIA Fermi 架构上, 非对齐访问会增加读操作, 但是额外的数据被缓存到片上 L1 Cache 中, 能被下一次用到, 因此在 GTX480 上的性能提升微乎其微。GT200 系列架构不支持缓存, 因此内存对齐扮演着重要的角色。我们在 GTX 285 GPU 上使用 NVIDIA Visual Profiler 可以看到, 对齐到 64 字节地址边界时会减少内存访问事务, 相应每个事务读取更多数据, 内存性能相较于最初版本提高了 8%。

通过向量读取提高效率

AMD GPU 对每路 SIMD 优化了 128 位读操作, 因此执行矢量读取能够提升性能(例如读 float4 类型数据, 相较于读 float 类型数据取得的内存带宽更接近理论峰值)。尽管使用向量类型计算卷积并不直观, 如果从全局内存缓存到本地内存的数据宽度是 4 的倍数, 向量类型可用来完成算法中的数据传输部分。此外, 如果滤波器半径不大于 8(滤波器至多是 17×17), 大小为 16×16 的 workgroup 内每个 work-item 只需读取单个 float4 便能拷贝所有的填充像素和输出像素。

为支持 float 数据类型, 第一步是调整本地内存分配, 使其包含所需要的额外像素:

```
int localWidth = roundUp(localSize[0]+padding, 4);
int localHeight = localSize[1]+padding;
size_t localMemSize = (localWidth*localHeight*sizeof(float));
```

将数据从全局内存向量化传输到本地内存的代码如下所示。为了实现向量读取, 且仍然使用标量数据计算卷积, 输入图像类型为 float4, 而本地内存缓存的数据类型为 float。执行读操作时, 将一个临时 __local float4 类型的指针指向需要的本地内存地址, 数据读取使用该指针完成。由于输入图像的数据类型使用 float4, 每列存储四个值, 图像的宽度(列数)应除以 4。图像的行数不变。

```
__kernel
void convolution_read4(__global float4* imageIn,
                      __global float* imageOut,
                      __constant float* filter,
                      int rows,
                      int cols,
                      int filterWidth,
                      __local float* localImage,
                      int localHeight,
                      int localWidth) {
```

```

// Vector pointer that will be used to cache data
// scalar memory
__local float4* localImage4;

// Determine the amount of padding for this filter
int filterRadius = (filterWidth/2);
int padding = filterRadius * 2;

// Determine where each workgroup begins reading
int groupStartCol = get_group_id(0)*get_local_size(0)/4;
int groupStartRow = get_group_id(1)*get_local_size(1);

// Flatten the localIds 0-255
int localId = get_local_id(1)*get_local_size(0) +
    get_local_id(0);
// There will be localWidth/4 work-items reading per row
int localRow = (localId / (localWidth/4));
// Each work-item is reading 4 elements apart
int localCol = (localId % (localWidth/4));

// Determine the row and column offset in global memory
// assuming each element reads 4 floats
int globalRow = groupStartRow + localRow;
int globalCol = groupStartCol + localCol;

// Set the vector pointer to the correct scalar location
// in local memory
localImage4 = (__local float4*)
    &localImage[localRow*localWidth+localCol*4];
// Perform all of the reads with a single load
if(globalRow < rows && globalCol < cols/4 &&
    localRow < localHeight) {
    localImage4[0] = imageIn[globalRow*cols/4+globalCol];
}
barrier(CLK_LOCAL_MLM_FINCL);

// Perform the convolution
...

```

在 AMD Radeon 6970 上, 通过使用向量读取能够带来显著的性能提升。相对于算法的最初实现, 使用对齐的 float4 数据类型访存后内存性能提高了 42%。128 位读在 NVIDIA GPU 内存架构上优势不太明显, 额外寄存器压力会降低整体的性能, 尽管使用向量类型使内存访问事务的总数下降, 在这个例子中, 性能反而有轻微下降。

执行卷积

一旦数据缓存在本地内存中，就可以高效访问来执行卷积。下面代码给出了一个对应于第 4 章 C 版本的算法实现。每个 **work-item** 计算一个输出像素，通过对该像素所在的周围像素执行乘加操作来完成卷积。对于那些由于滤波器尺寸限制需要越界访问原始图像之外的像素，则不产生输出结果。对于 7×7 的滤波器，第一个有效的输出像素将位于索引(3, 3)，对填充了像素后发起的访问开始索引是(0, 0)。由于我们希望所有的 **work-item** 产生输出数据，全局索引为(0, 0)的 **work-item** 产生的输出值将在(3, 3)。换句话说，每个 **work-item** 输出值的坐标相对于其全局 ID 在 X 和 Y 方向上的位移等于滤波半径。执行卷积的代码如下所示：

```
// Only allow work-items mapping to valid locations in
// the output image to perform the convolution
if(globalRow < rows - padding && globalCol < cols - padding) {

    // Each work-item will filter around its start location
    // (starting from the filter radius left and up)
    float sum = 0.0f;
    int filterIdx = 0;

    // The convolution loop
    for(int i = localRow; i < localRow+filterWidth; i++) {
        int offset = i*localWidth;
        for(int j = localCol; j < localCol+filterWidth; j++) {
            sum += localImage[offset+j] * filter[filterIdx++];
        }
    }

    // Write the data out
    imageOut[(globalRow+filterRadius)*cols +
              (globalCol+filterRadius)] = sum;
}
```

通过循环展开提高性能

使用 AMD APP Profiler 可以看到，对于该 **kernel** 其 VLIW 计算功能部件的利用率较低(43%)。这意味着仅使用了不到一半的功能部件，即编译器无法找到足够的指令来充分利用 VLIW 功能部件。对该卷积 **kernel** 而言，过于紧凑的计算循环很可能是罪魁祸首。如果能静态确定滤波尺寸(即在编译时已确定)，我们可以通过对内层循环进行展开来增加 ALU 指令的相对比例。

```

// Only allow work-items mapping to valid locations in
// the output image to perform the convolution
if(globalRow < rows - padding && globalCol < cols - padding) {
    // Each work-item will filter around its start location
    //(starting from the filter radius left and up)
    float sum = 0.0f;
    int filterIdx = 0;

    // Inner loop unrolled
    for(int i = localRow; i < localRow+filterWidth; i++) {
        int offset = i*localWidth+localCol;
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
        sum += localImage[offset++] * filter[filterIdx++];
    }

    // Write the data out
    imageOut[(globalRow+filterRadius)*cols +
              (globalCol+filterRadius)] = sum;
}
}

```

在 AMD Radeon 6970 上使用 600×400 图像, 滤波尺寸为 7×7, 对最内层循环进行展开能得到 2.4 倍的加速比, 与此同时功能部件的利用率提高到了 79%。在 GTX 285 和 480 平台上结论类似, 即分别能得到 2.6 倍和 2.2 倍的性能加速。

对卷积中内层和外层循环完全展开会增加代码规模, 因此对大尺寸滤波 kernel 可能不可行。然而一般而言, 循环展开在 AMD 和 NVIDIA GPU 设备上都能带来可观的加速比。对于 7×7 的滤波器, 在 Radeon 6970 平台上相对于不使用循环展开能得到 6.3 倍的性能加速比, GTX 285 和 480 平台分别能得到 3.2 和 2.9 的性能加速比。

小结

本章讨论一个经典的计算 kernel 即卷积, 它在机器视觉、统计和信号处理等领域有广泛应用。我们展示了针对 AMD 或者 NVIDIA GPU 优化该 kernel 的方法, 探讨了不同的内存优化策略带来的好处, 并证明性能在很大程度上依赖于设备底层的内存架构。尽管如此, 将卷积中典型的双重循环进行适当的循环展开在所有设备上都获得了显著的性

能提升。一般而言，很多性能优化策略都依赖于底层设备硬件架构。为了获得最佳性能，程序员应具备这些知识。

代码清单

主机端代码

```
#define WGX 16
#define WGY 16

// Uncomment each of these to run with the corresponding
// optimization
#define NON_OPTIMIZED
//#define RLAD_ALIGN4
//#define RLAD4

// This function takes a positive integer and rounds it up to
// the nearest multiple of another provided integer
unsigned int roundUp(unsigned int value, unsigned int multiple)
{
    // Determine how far past the nearest multiple the value is
    unsigned int remainder = value % multiple;

    // Add the difference to make the value a multiple
    if(remainder != 0) {
        value += (multiple - remainder);
    }

    return value;
}

int main(int argc, char** argv) {
    // Set up the data on the host

    // Rows and columns in the input image
    int imageHeight;
    int imageWidth;

    // Homegrown function to read a BMP from file
    float* inputImage = readImage("input.bmp", &imageWidth,
        &imageHeight);

    // Size of the input and output images on the host
    int dataSize = imageHeight*imageWidth*sizeof(float);

    // Pad the number of columns
```

```

#ifdef NON_OPTIMIZED
    int deviceWidth = imageWidth;
#else // READ_ALIGNED || READ4
    int deviceWidth = roundUp(imageWidth, WGX);
#endif
int deviceHeight = imageHeight;
// Size of the input and output images on the device
int deviceDataSize = imageHeight*deviceWidth*sizeof(float);

// Output image on the host
float* outputImage = NULL;
outputImage = (float*)malloc(dataSize);
for(int i = 0; i < imageHeight; i++) {
    for(int j = 0; j < imageWidth; j++) {
        outputImage[i*imageWidth+j] = 0;
    }
}

// 45 degree motion blur
float filter[49] = {
    0,    0,    0,    0,    0, 0.0145,    0,
    0,    0,    0,    0, 0.03/6, 0.1283, 0.0145,
    0,    0,    0, 0.03/6, 0.1283, 0.03/6,    0,
    0,    0.03/6, 0.1283, 0.03/6,    0,    0,    0,
    0.0145, 0.1283, 0.03/6,    0,    0,    0,    0,
    0, 0.0145,    0,    0,    0,    0,    0,    0};
int filterWidth = 7;
int filterRadius = filterWidth/2;
int paddingPixels = (int)(filterWidth/2) * 2;

// Set up the OpenCL environment

// Discovery platform
cl_platform_id platform;
clGetPlatformIDs(1, &platform, NULL);

// Discover device
cl_device_id device;
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &device,
    NULL);

// Create context
cl_context_properties props[3] = {CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platform), 0};
cl_context context;
context = clCreateContext(props, 1, &device, NULL, NULL,
    NULL);

// Create command queue
cl_command_queue queue;

```



```

queue = clCreateCommandQueue(context, device, 0, NULL);

// Create memory buffers
cl_mem d_inputImage;
cl_mem d_outputImage;
cl_mem d_filter;
d_inputImage = clCreateBuffer(context, CL_MEM_READ_ONLY,
    deviceDataSize, NULL, NULL);
d_outputImage = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    deviceDataSize, NULL, NULL);
d_filter = clCreateBuffer(context, CL_MEM_READ_ONLY,
    49*sizeof(float), NULL, NULL);

// Write input data to the device
#ifdef NON_OPTIMIZED
    clEnqueueWriteBuffer(queue, d_inputImage, CL_TRUE, 0,
        deviceDataSize,
            inputImage, 0, NULL, NULL);
#else // READ_ALIGNED || READ4
    size_t buffer_origin[3] = {0,0,0};
    size_t host_origin[3] = {0,0,0};
    size_t region[3] = {deviceWidth*sizeof(float),
        imageHeight, 1};
    clEnqueueWriteBufferRect(queue, d_inputImage, CL_TRUE,
        buffer_origin, host_origin, region,
        deviceWidth*sizeof(float), 0, imageWidth*sizeof(float), 0,
        inputImage, 0, NULL, NULL);
#endif

// Write the filter to the device
clEnqueueWriteBuffer(queue, d_filter, CL_TRUE, 0,
    49*sizeof(float), filter, 0, NULL, NULL);

// Read in the program from file
char* source = readSource("convolution.cl");

// Create the program
cl_program program;

// Create and compile the program
program = clCreateProgramWithSource(context, 1,
    (const char**)&source, NULL, NULL);
cl_int build_status;
build_status = clBuildProgram(program, 1, &device, NULL, NULL,
    NULL);

// Create the kernel
cl_kernel kernel;
#ifdef NON_OPTIMIZED || defined READ_ALIGNED
    // Only the host-side code differs for the aligned reads
    kernel = clCreateKernel(program, "convolution", NULL);

```

```

#else // READ4
    kernel = clCreateKernel(program, "convolution_read4", NULL);
#endif

    // Selected workgroup size is 16x16
    int wgWidth = WGX;
    int wgHeight = WGY;

    // When computing the total number of work-items, the
    // padding work-items do not need to be considered
    int totalWorkItemsX = roundUp(imageWidth-paddingPixels,
        wgWidth);
    int totalWorkItemsY = roundUp(imageHeight-paddingPixels,
        wgHeight);

    // Size of a workgroup
    size_t localSize[2] = {wgWidth, wgHeight};
    // Size of the NDRange
    size_t globalSize[2] = {totalWorkItemsX, totalWorkItemsY};

    // The amount of local data that is cached is the size of the
    // workgroups plus the padding pixels
    #if defined NON_OPTIMIZED || defined READ_ALIGNED
        int localWidth = localSize[0] + paddingPixels;
    #else // READ4
        // Round the local width up to 4 for the read4 kernel
        int localWidth = roundUp(localSize[0]+paddingPixels, 4);
    #endif
    int localHeight = localSize[1] + paddingPixels;

    // Compute the size of local memory (needed for dynamic
    // allocation)
    size_t localMemSize = (localWidth * localHeight *
        sizeof(float));

    // Set the kernel arguments
    clSetKernelArg(kernel, 0, sizeof(cl_mem), &c_inputImage);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &c_outputImage);
    clSetKernelArg(kernel, 2, sizeof(cl_mem), &c_filter);
    clSetKernelArg(kernel, 3, sizeof(int), &deviceHeight);
    clSetKernelArg(kernel, 4, sizeof(int), &deviceWidth);
    clSetKernelArg(kernel, 5, sizeof(int), &filterWidth);
    clSetKernelArg(kernel, 6, localMemSize, NULL);
    clSetKernelArg(kernel, 7, sizeof(int), &localHeight);
    clSetKernelArg(kernel, 8, sizeof(int), &localWidth);

    // Execute the kernel
    clEnqueueNDRangeKernel(queue, kernel, 2, NULL, globalSize,
        localSize, 0, NULL, NULL);

    // Wait for kernel to complete

```

```

    clFinish(queue);

    // Read back the output image
#ifdef NON_OPTIMIZED
    clEnqueueReadBuffer(queue, d_outputImage, CL_FALSE, 0,
        deviceDataSize, outputImage, 0, NULL, NULL);
#else // READ_ALIGNED READ4
    // Begin reading output from (3,3) on the device
    // (for /x/ filter with radius 3)
    buffer_origin[0] = 3*sizeof(float);
    buffer_origin[1] = 3;
    buffer_origin[2] = 0;

    // Read data into (3,3) on the host
    host_origin[0] = 3*sizeof(float);
    host_origin[1] = 3;
    host_origin[2] = 0;

    // Region is image size minus padding pixels
    region[0] = (imageWidth - paddingPixels)*sizeof(float);
    region[1] = (imageHeight - paddingPixels);
    region[2] = 1;

    // Perform the read
    clEnqueueReadBufferRect(queue, d_outputImage, CL_FALSE,
        buffer_origin, host_origin, region,
        deviceWidth*sizeof(float), 0, imageWidth*sizeof(float), 0,
        outputImage, 0, NULL, NULL);
#endif

    // Homegrown function to write the image to file
    storeImage(outputImage, "output.bmp", imageHeight,
        imageWidth);

    // Free OpenCL objects
    clReleaseMemObject(d_inputImage);
    clReleaseMemObject(d_outputImage);
    clReleaseMemObject(d_filter);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    return 0;
}

```

kernel 代码

```

__kernel
void convolution(__global float* imageIn,
    __global float* imageOut,

```

```

    __constant float* filter,
        int rows,
        int cols,
        int filterWidth,
    __local float* localImage,
        int localHeight,
        int localWidth) {

    // Determine the amount of padding for this filter
    int filterRadius = (filterWidth/2);
    int padding = filterRadius * 2;

    // Determine the size of the workgroup output region
    int groupStartCol = get_group_id(0)*get_local_size(0);
    int groupStartRow = get_group_id(1)*get_local_size(1);

    // Determine the local ID of each work-item
    int localCol = get_local_id(0);
    int localRow = get_local_id(1);

    // Determine the global ID of each work-item. work-items
    // representing the output region will have a unique global
    // ID
    int globalCol = groupStartCol + localCol;
    int globalRow = groupStartRow + localRow;

    // Cache the data to local memory

    // Step down rows
    for(int i = localRow; i < localHeight; i +=
        get_local_size(1)) {

        int curRow = groupStartRow+i;

        // Step across columns
        for(int j = localCol; j < localWidth; j +=
            get_local_size(0)) {

            int curCol = groupStartCol+j;

            // Perform the read if it is in bounds
            if(curRow < rows && curCol < cols) {
                localImage[i*localWidth + j] =
                    imageIn[curRow*cols+curCol];
            }
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the convolution
    if(globalRow < rows-padding && globalCol < cols-padding) {
        // Each work-item will filter around its start location

```

```

        //(starting from the filter radius left and up)
        float sum = 0.0f;
        int filterIdx = 0;

        // Not unrolled
        for(int i = localRow; i < localRow+filterWidth; i++) {
            int offset = i*localWidth;
            for(int j = localCol; j < localCol+filterWidth; j++){
                sum += localImage[offset+j] *
                    filter[filterIdx++];
            }
        }

        /*
        // Inner loop unrolled
        for(int i = localRow; i < localRow+filterWidth; i++) {
            int offset = i*localWidth+localCol;
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
            sum += localImage[offset++] * filter[filterIdx++];
        }
        */

        // Write the data out
        imageOut[(globalRow+filterRadius)*cols +
            (globalCol+filterRadius)] = sum;
    }

    return;
}

__kernel
void convolution_read4(__global float4* imageIn,
    __global float* imageOut,
    __constant float* filter,
        int rows,
        int cols,
        int filterWidth,
    __local float* localImage,
        int localHeight,
        int localWidth) {

    // Vector pointer that will be used to cache data
    // scalar memory
    __local float4* localImage4;

    // Determine the amount of padding for this filter
    int filterRadius = (filterWidth/2);
    int padding = filterRadius * 2;

```

```

// Determine where each workgroup begins reading
int groupStartCol = get_group_id(0)*get_local_size(0)/4;
int groupStartRow = get_group_id(1)*get_local_size(1);

// Flatten the locals 0-255
int localId = get_local_id(1)*get_local_size(0) +
    get_local_id(0);
// There will be localWidth/4 work-items reading per row
int localRow = (localId / (localWidth/4));
// Each work-item is reading 4 elements apart
int localCol = (localId % (localWidth/4));

// Determine the row and column offset in global memory
// assuming each element reads 4 floats
int globalRow = groupStartRow + localRow;
int globalCol = groupStartCol + localCol;

// Set the vector pointer to the correct scalar location
// in local memory
localImage4 = (__local_float4*)
    &localImage[localRow*localWidth+localCol*4];

// Perform all of the reads with a single load
if(globalRow < rows && globalCol < cols/4 &&
    localRow < localHeight) {
    localImage4[0] = imageIn[globalRow*cols/4+globalCol];
}
barrier(CLK_LOCAL_MLM_FLNCL);

// Reassign local IDs based on each work-item processing
// one output element
localCol = get_local_id(0);
localRow = get_local_id(1);

// Reassign global IDs for unique output locations
globalCol = get_group_id(0)*get_local_size(0) + localCol;
globalRow = get_group_id(1)*get_local_size(1) + localRow;
// Perform the convolution
if(globalRow < rows-padding && globalCol < cols-padding) {
    // Each work-item will filter around its start location
    // (starting from half the filter size left and up)
    float sum = 0.0f;
    int filterIdx = 0;

    // Not unrolled
    for(int i = localRow; i < localRow+filterWidth; i++) {
        int offset = i*localWidth;
        for(int j = localCol; j < localCol+filterWidth; j++) {
            sum += localImage[offset+j] *
                filter[filterIdx++];
        }
    }
}
/*

```

```

// Inner loop unrolled
for(int i = localRow; i < localRow+filterWidth; i++) {
    int offset = i*localWidth+localCol;

    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];
    sum += localImage[offset++] * filter[filterIdx++];

}
*/

/*
// Completely unrolled
int offset = localRow*localWidth+localCol;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];
sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

offset += localWidth;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];
sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

offset += localWidth;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];
sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

offset += localWidth;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];

```

```

sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

offset += localWidth;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];
sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

offset += localWidth;

sum += localImage[offset+0] * filter[filterIdx++];
sum += localImage[offset+1] * filter[filterIdx++];
sum += localImage[offset+2] * filter[filterIdx++];
sum += localImage[offset+3] * filter[filterIdx++];
sum += localImage[offset+4] * filter[filterIdx++];
sum += localImage[offset+5] * filter[filterIdx++];
sum += localImage[offset+6] * filter[filterIdx++];

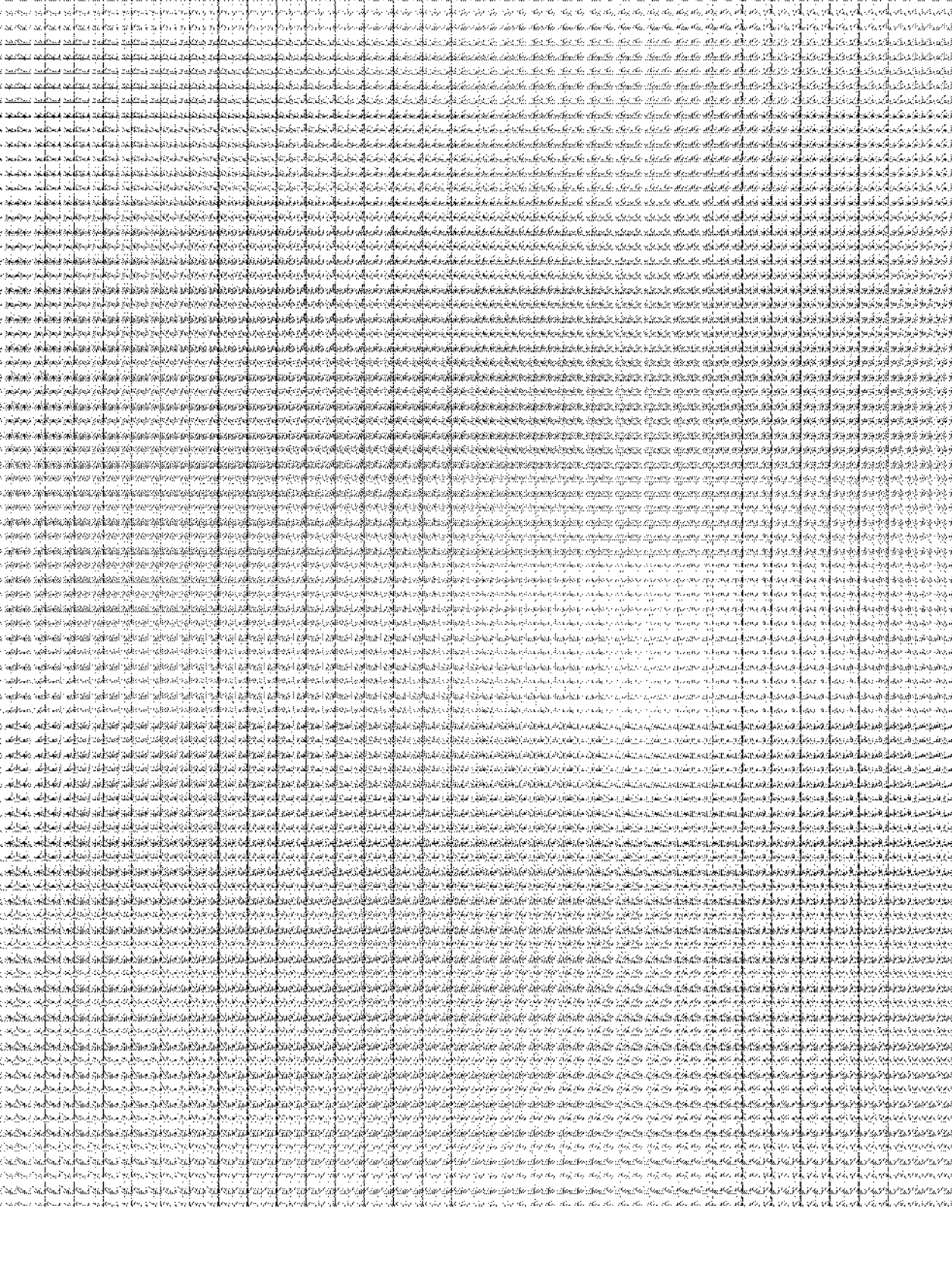
*/

// Write the data out
imageOut[(globalRow+filterRadius)*cols +
         (globalCol+filterRadius)] = sum;
}
return;
}

```

参考文献

NVIDIA Corporation. (2009). *NVIDIA OpenCL Programming Guide for the CUDA Architecture*. Santa Clara, CA: NVIDIA Corporation.



OpenCL 案例学习 2：视频处理

引言

本章，我们将讨论一个有趣的应用案例，以加深我们对 OpenCL 各方面的认识。这个应用是一个视频处理程序，输入是一段视频流，通过使用 OpenCL 对视频内容进行一些图像处理操作(后文也称 effect)，输出结果是另一段可以实时播放的视频流。该应用可以扩展成对一段视频流进行多种处理或对多个输入视频流进行融合输出单个视频流。

为了完成这个任务，我们需要考虑使用多种重要的技术，包括使用 OpenVideo 解码 API 进行专业硬件解码的方法、各种将已解码的视频帧传输到计算设备的方法、高级的利用事件列表计算依赖的方法以及各种提高性能的方法。同时在 OpenCL 与 OpenGL 之间用共享缓存来播放视频。这种 API 之间最终共享缓存的行为，我们称之为协同操作。

在考察各种不同的方式来完成手头任务的时候，性能是这个案例的核心目标。这个应用的性能调优颇有挑战性，因为有很多方面都可以加速。然而，视频流的输出速度是我们要测试的基准，目标是让它能再次实时播放。

获得视频帧

在我们讨论这个 OpenCL 实例之前，首先需要将视频输入到视频处理程序以满足我们这个视频处理程序的输入需求。在这里，输入的视频是以压缩的形式存储在计算机中的。从磁盘读取了输入数据之后，RGB 格式的视频帧会用于进行一些图像处理操作和制作输出结果。视频文件必须被解码，以从压缩的视频中获得我们需要的 RGB 格式。

CPU 上的解码

目前有很多利用 CPU 在各种平台上进行视频解码的程序库。假设我们使用 GPU 作为 OpenCL 的计算设备，在一个系统中通常是把视频显示作为最终目标。在异构计算中非常直接的方式是，我们在 CPU 上进行视频解码，然后在 GPU 上执行视频处理的过程。如图 8.1 所示，在异构计算中一般是根据任务的边界来进行工作的划分。采用这种方式的异构计算有一个内在局限，即：程序的总体性能将受限于任何一个任务的整体性能或者是将计算结果从一个设备传输到另一个设备所涉及的开销。

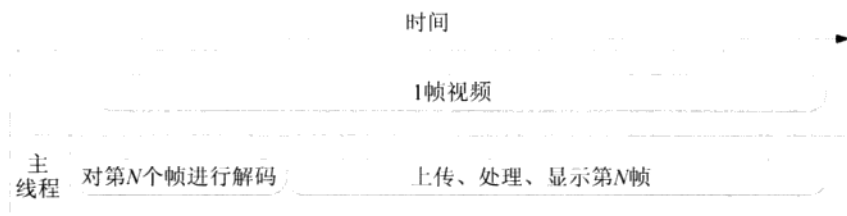


图 8.1

在 CPU 上解码然后在 GPU 上处理和显示视频，这就是处理视频的方式

在本例中，我们选择使用 VLC 项目中的 libvlc 库，libvlc 库不仅功能完善，使用广泛，性能很高，而且可以用在 Linux、Windows 和 OS X 上运行，不依赖于某一个特定的平台。使用 OpenCL 也有好处即 OpenCL 开放标准已经在 Linux、Windows 和 OS X 上实现。之后我们会看到这个例子中的 OpenCL 代码也同样可以运行在这三个平台上(显示代码中有少数例外)。这个例子中的最后一个主要部分就是将输出结果显示给用户，我们使用另一个开放标准(OpenGL)来完成这个过程。

为了解码一帧视频，我们必须首先初始化解码框架，让它打开待解码文件。然后解码框架将以指定的格式解码一帧视频。解码框架包含各种不同的方法来返回一帧，最基本的方式是简单阻塞调用，一旦帧已经准备好了就将控制权返回给调用者。另一种方式是解码框架可能依赖回调函数，回调函数在解码帧准备好之后被调用，从而支持异步解码请求。

相比阻塞调用，使用回调的方式有一些优点。然而，不管选择什么方式，最重要的是考虑性能和怎样解码对处理视频剩余时间的影响。如果程序员使用单线程带循环的方式来编写这个应用，那么回调方法可以允许一些视频解码和其他工作的重叠，仅当数据准备好时才唤醒程序。主循环将接着执行最少量的工作，最好是只拷贝已经解码的帧。

然而, 如果程序使用阻塞调用的方式, 主循环必须发起一个对解码库的调用, 阻塞直到下一帧被解码为止。这种阻塞调用减少了本能够并行执行的解码工作, 如果磁盘输入或者输出必须获得视频帧, 这将是一个非常重要的问题。

多线程解码

鉴于我们的目标是最大限度地提高性能, 即使在主循环中做最少的解码工作所涉及的开销远远高于什么都不做。为了将全部解码工作从主循环中移除, 我们创建工作者线程来完成这些工作。对每一个输入的待解码视频都创建一个工作者线程, 同时异步解码和其它工作的执行通过应用级别的线程而不是 API 回调来实现的。作为阻塞调用的替代, 程序专注于为主循环的高效执行做好准备。为了保持主循环一直运行不需要等待任何一个工作者线程, 示例代码使用简单的先进先出(FIFO)的方式为主循环提供已解码帧缓存。解码帮助者线程监控 FIFO 队列, 并尝试让它一直保持满队状态, 而 FIFO 的大小被设计成足够大以适应解码端的最大延迟(图 8.2)。

在 GPU 上解码视频

很多现代 GPU 拥有针对确定格式视频进行解码的特定功能部件。设计此功能旨在利用专业硬件解码相对于通用计算的高效性, 让用户在观看视频时通过减少 CPU(或 GPU)的功耗来延长电池使用时间。这样的硬件有很多名字: AMD 把这个特性称为 Unified Video Decode, NVIDIA 称为 PureVideo, 它同样也可以在 Intel 的图形设备上找到。微软的 Windows 的 DXVA API 通常用来在 Windows 上访问这些硬件。

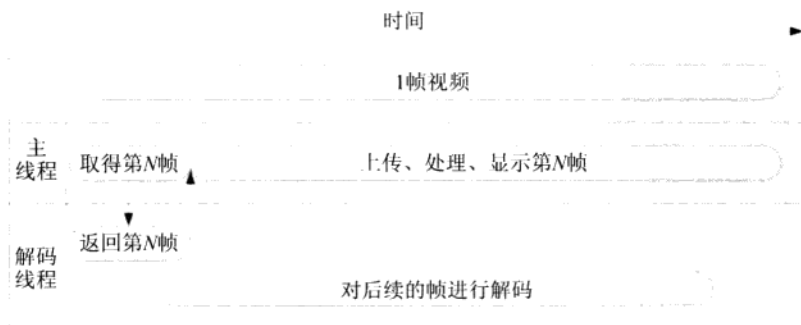


图 8.2

主应用线程添加一个帮助者线程来解码视频

使用视频解码硬件的方式是将压缩视频数据流传输到 GPU 设备上,在那里进行解码,然后将解压缩的视频帧写入 GPU 内存(如图 8.3)。为了在 OpenCL 中使用解压缩后的视频帧,我们需要将视频帧映射到 OpenCL 的内存空间。因为不同的 API 使用不同的方式来管理内存,所以即使视频帧已经在 GPU 内存里面了,也需要这么做。另一种方式是将解码后的帧从 GPU 内存拷贝回系统内存,然后再上传到 OpenCL 设备。后一种方法由于通过系统总线进行多次拷贝,所以通常会比第一种方式代价更高。更好的方式是让解码视频数据直接可以被 OpenCL 访问。

OpenVideo 解码

在本例中充分发挥特定功能解码硬件的优势,从而避免使用各种复杂的方式来访问解码帧内存,本例中使用 OpenVideo Decode API。这个 API 支持从 OpenCL 访问解码视频帧。API 是这样编写的:它操作一个类似于 OpenCL 的、从单独 API 域申请的、平台独立的上下文。

在使用 Open Video Decode API 之前,我们需要检查设备并检查视频的格式,确认它们是否互相兼容。在 OpenCL 中,这通过检查是否具有利用 `cl_amd_open_video` 与 OpenVideo 进行互操作的能力来实现。在 OpenVideo 中,我们可以通过查看 `OVDcodeGetDeviceInfo()` 和 `OVDcodeGetDeviceCap()` 这两个函数的结果来确定视频加速器是否可用以及它们能处理哪些内容。

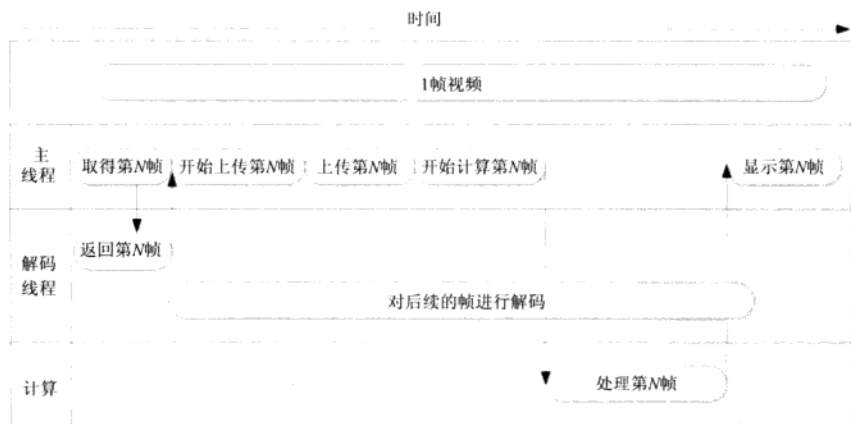


图 8.3

使用 GPU 并行处理进一步提高解码帧的效率

```

if ( !(ret = OVDecodeGetDeviceInfo(&numDevices, 0)) )
    //handle getting device info not succeeded
}

if(numDevices == 0){
    //handle no devices being available
}

ovdecode_device_info *deviceInfo =
    newovdecode_device_info[numDevices];
ret = OVDecodeGetDeviceInfo(&numDevices, deviceInfo);

for(unsignedint i = 0; i < numDevices; i++){
    ovdecode_cap *caps =
        newovdecode_cap[deviceInfo[i].decode_cap_size];
    ret = OVDecodeGetDeviceCap(
        deviceInfo[i].device_id,
        deviceInfo[i].decode_cap_size, caps);
}
...

```

下一步,我们先创建一个 OpenCL 上下文,然后在 Open Video 中将 OpenCL 上下文传递给 OVDecodeCreateSession()函数来创建一个解码会话(decode session)。

我们必须对每一个待解码视频流创建一个解码会话,此外,应用程序还必须协调好由 OpenCL 和 Open Video 共享使用的内存。这通过在 OpenCL 中创建内存对象,然后使用 clGetPlatformMem-Handle()将它们转换成平台独立的句柄来实现。此时,OpenCL 和 Open Video 都拥有了这块内存的句柄,访问时需要同步。

```

OPContextHandle ovdContext = clCreateContext(
    properties,
    1,
    &clDeviceID,
    0,
    0,
    &err);
ov_session session = OVDecodeCreateSession (
    ovdContext,
    ovDeviceID,
    profile,
    output_format,
    output_Width,
    output_Height);

```

随着基本设置的完成和一条在 OpenCL 和 Open Video API 之间直接链路的建立,我们接下来考虑在主解码循环中使用 Open Video 的解码功能。为了内存访问同步,应用程序首先使用 OVDecodeAcquireObject()函数从 Open Video Decode 获得内存对象,这样解码

视频可以写入到这块内存里面。为了进行解码，OVDecodePicture()函数被用来传递压缩视频流，该函数根据解码参数读取解析码流。创建解析的码流需要对压缩视频格式有详细的了解；因此，我们用一个第三方库来完成这个任务。解码完成之后，内存对象从 Open Video 释放，从而 OpenCL 运行时可以从它的域里面获取这个对象，并且能对已解码的帧进行访问。最终，当这些操作完成之后，内存对象将被释放。我们重复迭代这个循环(内存访问同步，解码，访问已经解码的视频)直到视频处理完成。

需要的命令序列如下：

```
OVresultOVAcquireObject(
    session,
    unsigned int num_handle,
    ov_handle *decode_handle,
    1,
    &pre_event,
    &acquire_event);

res = OVDecodePicture(
    session,
    &picture_parameter,
    &pic_parameter_2,
    pic_parameter_2_size,
    &bitstream_data,
    bitstream_data_read_size,
    slice_data_control,
    slice_data_control_size,
    output_surface,
    1,
    &acquire_event,
    &eventRunVideoProgram,
    0);

ret = OVReleaseObject(
    session,
    unsigned int num_handle,
    ov_handle *decode_handle,
    1 /* num_event_in_wait_list */,
    &decode_event,
    &release_event);

status = clEnqueueNDRangeKernel(...)
```

虽然内存对象的同步访问导致了较大的开销，但是使用特定功能硬件来进行视频解码解放了计算资源，减少了解压后视频来回的传输。由于只有压缩视频需要在 GPU 和系统之间的总线上进行传输，因此也减轻了总线负担。然而，由于特定硬件解码功能特性

的原因,这样做有一些限制。硬件设计上同时解码的视频流的数量通常是有限的。大部分高端系统可以同时解码两个流(包括 picture-in-picture 脚本)。由于这个限制,我们必须提供一个 CPU 版解码的备用路径。

在 OpenCL 中处理一个视频

假设我们可以访问输入数据,然后我们讨论在 OpenCL 中如何处理这些数据。本实例的视频处理过程的输入和输出分别使用单一的视频帧。它允许不同的参数设置控制产生不同的效果。视频处理参数通过对 kernel 设置不同的参数来实现。

OpenCL 的一些特点导致访问图像数据非常容易。首先,向量类型如<类型>4(即:char4, int4, float4)可以方便的映射到一个四种颜色(红, 绿, 蓝, alpha(RGBA))的像素上。甚至更方便的支持长度为 3 的向量映射成 RGB、YUV 等类型的像素数据。

OpenCL 访问图像数据的另一个有效方法是使用 image 对象。OpenCL 的 buffer 对象在内存中的存储方式类似于 C 语言中的数组,按连续方式存储,一般用于通用目的的存储。image 不仅更加容易访问二维和三维数据,也支持将数据存储成各种不同格式,同时具有对这些数据取样的能力。例如, image 访问函数可以非常高效的对数据按线性或者双立方(bi-cubic)进行插值。image 的格式可以设置和查询,一旦数据已经存在于 image 中, OpenCL kernel 就可以调用 read_image[u,f]()函数来将数据按各种格式输出。

视频处理例子使用 image 作为数据格式存储视频帧,我们使用一个简单的 effect 来详细的说明这些。选择的 effect 使用一个参数,以线性的方式使帧变暗或变亮。OpenCL kernel 有三个参数:输入视频帧、输出视频帧和一个代表亮度的浮点标量。当亮度值是 1.0 的时候,输出图像不变,亮度值大于 1.0 表示输出图像变得更亮,当亮度值小于 1.0 的时候,输出更暗。

```
__kernel void
brighten(
    read_only image2d_t ir,
    write_only image2d_t out,
    float brightness)
{
    float4 val;
    int2 coordinate = (int2)(get_global_id(0), get_global_id(1));
    const sampler_t sampler =
```



```

    CLK_NORMALIZED_COORDS_FALSE |
    CLK_ADDRESS_NONE |
    CLK_FILTER_NEAREST;

    val = read_imagef(in, sampler, coordinate);

    val *= (float4)(brightness, brightness, brightness, 1.0f);
    write_imagef(val, coordinate);
}

```

虽然视频帧可能是以每个通道 8 比特 RGBA 格式存在,但是亮度参数是一个浮点数,能很好进行调节。在本实例中,我们使用的是 OpenCL 的 `image` 对象,因此可以调用 `read_imagef()` 函数对每一个像素返回一个 `float4` 类型的数,然后在使用 `write_imagef()` 函数写输出像素之前,执行亮度修正操作。

在输入视频帧被上传到 `image` 之后,我们创建专门的 OpenCL effect kernel,然后将它应用于输入数据。为了应用专门的 effect,应用程序编译 kernel 然后正确设置 kernel 参数。对只应用一个过滤的基本情况,输出帧将被写入一个最便于显示的位置。

在多个视频上处理多个不同 effect

为了处理多个不同的 effect,除了将多个 effect 链接在一起以外,我们按照和处理单视频单 effect 同样的步骤。我们可以想像一幅中间图像,它由一个 effect 输出,作为输入传给下一个 effect。由于每一个 effect 必须等待前一个 effect 完成才可以开始执行,因此我们可以利用视频帧缓存直接代替中间缓存。如果算法中当前像素的值取决于其他几个像素(这被称为 `gather` 算法),或者当输出像素用于改变其他输出像素的时候(这被称为 `spreading` 算法),这个方法就会有一些问题。在这些情况下,中间缓存是必须的。然而,为了使工作更加有效,我们需要移除不必要的中间缓存,我们的目标是在任何情况下都不使用中间缓存。

事件链

OpenCL 使用命令队列模型来进行任务调度,使用事件来获取每一个入队任务的信息。这些事件使命令之间建立起一种顺序依赖关系。更进一步,这些事件能被应用程序用来查询队列的状态。每一个事件都通过调用 `clEnqueue{*}` 来创建,一个依赖事件列表可以被赋给 `clEnqueue{*}` 操作。运行时系统会检查每一个依赖,直到满足后才开始启动

任务(或提交任务)。然而, 由于 `clEnqueue {*}` 正在创建事件, 我们不能对将来调用的 `clEnqueue {*}` 提交依赖, 因为那个事件尚未创建。

根据单个视频的例子, 我们入队了多个 `effect`, 包括它们之间的依赖, 同样的方法可以用在多个视频流上。因为视频处理进程同一个 `effect` 可能需要两个或多个视频流作为输入, 所以在组合新 `effect` 之前必须保证这些视频流之前入队的 `effect` 都已经处理完成。OpenCL 运行时利用事件可以处理顺序依赖。

当使用多个工作流的时候, 还需要考虑另外一个问题, 即哪个已经处理的视频帧可以移开, 用来放中间缓存。与单一的视频包括多个 `effect` 一样, 我们把结果输出到目标流的视频帧缓存, 从而以此方式去掉中间缓存。

最终输出显示到屏幕

在处理的最后一个环节的目标是将处理好的视频通过屏幕显示给用户。就概念而言, 将一个处理好的视频帧显示只需要简单地将数据拷贝到与屏幕交互的缓存帧或者其他数据结构中即可。考虑到移植性, 我们的例子中使用 OpenGL 作为输入, OpenGL 的输出视频被移动到一个双缓存的纹理当中。纹理在一个四边形中进行渲染, 从输出窗口顶部开始输出, 大小和输出窗口大小一致。

在 OpenGL 画纹理四边形之前, 输出数据必须从 OpenCL 拷贝(至少在逻辑上)到 OpenGL 的纹理当中。

OpenCL/OpenGL 协同工作能力

OpenCL 提供一个在 OpenCL 和 OpenGL 之间共享内存对象的可扩展选项。OpenCL 扩展将在第 10 章详细叙述, 它允许增加一些超出核心规范的特性。由于这些特性是可选的, 在使用这些特性之前, 应用程序必须先测试这些特性是否可用。使用 OpenCL C++ 封装 API, 可以通过使用 `cl::Device` 类的 `getInfo()` 方法来查询某一个具体特性, 返回 `CL_DEVICE_EXTENSIONS` 参数信息。结果字符串通过与特定的字符串对比可以测试出是否支持 OpenGL 共享。

```
//check that the device for support OpenGL/CL interop
if(devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find(
    "cl_khr_gl_sharing") == std::string::npos) {
    //OpenGL/CL interop is not available
}
```

因为内存对象共享模型是在上下文中定义的，我们在创建上下文的时候用参数描述哪一个 OpenGL 上下文被用来支持共享。不同操作系统获得 OpenGL 上下文的方式是不同的，这会稍微损失一点前面提到的可移植性。这种不同是由于属性名称在不同环境下的差异而造成的。下面的代码在 Windows 环境下使用 OpenGL 共享创建一个 OpenCL 上下文：

```
cl_context_properties props[7] =
{
    CL_CONTEXT_PLATFORM,
    (cl_context_properties) amdplatform(),
    CL_GL_CONTEXT_KHR,
    (intptr_t) wglGetCurrentContext(),
    CL_WGL_HDC_KHR,
    (intptr_t) wglGetCurrentDC(), //HDC handle of the display
    0
};

context = cl::Context(devices, props, NULL, NULL, &err);
if (CL_SUCCESS != err) {
    std::cerr << "Creating context failed with value "
    << err << std::endl;
}
```

一旦使用 OpenGL 共享创建了 OpenCL 上下文，应用程序必须在 OpenGL 中创建一个 texture 或者 buffer 用于在 OpenCL 中创建一个 image 或者 buffer。因为 OpenGL 需要 OpenCL texture，所以下面的视频处理应用程序会使用 image。

```
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
glTexParameteri(
    GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER,
    GL_LINEAR);
glTexParameteri(
    GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER,
    GL_LINEAR);
glTexParameteri(
    GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_S,
    GL_CLAMP);
glTexParameteri(
    GL_TEXTURE_2D,
    GL_TEXTURE_WRAP_T,
    GL_CLAMP);
glTexEnvf(
```

```

GL_TEXTURE_LNV,
GL_TEXTURE_LNV_MODL,
GL_DLOCAL);
glEnable(GL_TEXTURE_2D);

```

使用 `cl::Image2DGL` 类, 我们在这个 `texture` 基础上创建 OpenCL 的 `Image2D` 内存对象。

```

glImageBuffer = cl::Image2DGL(
    ctx->getContext(),
    CL_MEM_READ_WRITE,
    GL_TEXTURE_2D,
    0,
    texture,
    &err);

```

除了需要同步来保证 OpenCL 和 OpenGL 不访问脏数据(与 OpenVideo 解码 API)以外, 结果图片可以像其他任何图片一样被使用。这个同步是调用 `enqueueAcquireGLObjets()` 和 `enqueueReleaseGLObjets` 函数来加在队列对象上。这些命令可以访问全部 OpenCL 事件架构, 因为他们和其它命令一样以同样的方式在排队。在调用这些同步之前, 应用程序必须保证对内存对象的写操作全部完成。这种 `barrier` 同步操作在 OpenGL 中使用 `glFinish()` 来实现, 而在 OpenCL 中通过等待写入内存对象必然产生的事件来实现。下面的代码为了简洁省略了检查返回值或使用事件(用假设一个排好序的队列来替代):

```

std::vector<cl::Memory> mem_objs;
mem_objs.push_back(glImageBuffer[buffer_idx]);
ret = cq.enqueueAcquireGLObjets(&mem_objs, NULL, NULL);
ret = cq.enqueueWriteImage(
    ImageBuffer[buffer_idx],
    CL_FALSE,
    origin,
    size,
    0,
    0,
    data,
    NULL,
    NULL);
ret = cq.enqueueReleaseGLObjets(&mem_objs, NULL, NULL);

```

小结

在这一章中,我们讨论了 Open Video 解码 API 的使用,和 OpenCL 与 OpenGL image 结构的交互。Open Video 解码 API 支持高效的视频解码,而且功耗不高。OpenGL 和 OpenCL 之间的交互使高效数据共享成为可能,当数据都在一个给定的设备内存中的时候避免了重复的数据结构。本章提供了再一次利用 OpenCL 框架强大功能的机会。

OpenCL 案例学习 3：直方图

引言

本章将以图像直方图算法为例讨论访存密集型 kernel 的优化方法。该算法源图像的每个像素大小为 8 位，目标数据为一个长度为 256 的数组，数组元素的大小为 32 位。

图像直方图算法的原理是对图像的每个元素做如下操作：

```
for( many input values ) {  
    histogram[ value ]++;  
}
```

该算法在一个小的直方图数据结构中进行许多离散的读—修改—写操作，有很高的数据重用率，因此该算法在 CPU 实现中会使用 cache。然而在 GPU 实现中，这些操作都访问全局内存，这会导致极低的性能。

在全局内存访问中，会出现严重的访存冲突。为解决这个问题，我们选择以 work-group 为单位，并行化直方图计算的算法：对源数据进行分割，每个 work-group 只计算其中一部分生成局部直方图。这样最终会产生许多存储在本地内存上的局部直方图，然后将这些局部直方图进行规约，生成一个存储最终结果的全局直方图。算法描述如图 9.1 所示。

选择适量的 work-group

当我们将一个 OpenCL kernel 映射到具体的硬件架构上时，我们会遇到 work-item 和 workgroup 的数量方面的限制，这些限制来自于算法设计、目标架构特点以及各种内存大小的限制，可能会影响同时运行在硬件架构上的 work-group 和 work-item 的数量。workgroup 内的所有 work-item 共享本地内存，并将其计算结果存储在本地内存上，因此这些 work-item 会访问同一块本地内存区域，相互协作以减少访存开销。直方图计算的最后一个步骤是对每个 work-group 存储在本地内存上的局部直方图进行全局规约，如果

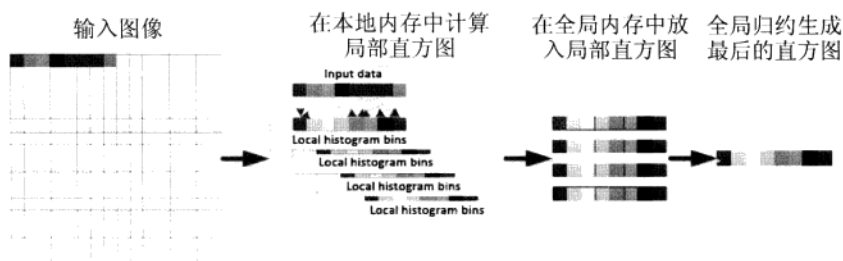


图 9.1

直方图计算：首先每个 work-group 计算生成局部直方图，然后将结果输出到全局内存，最后进行全局规约以产生最终直方图

创建过多的 work-group，不仅会增加本地内存到全局内存的数据传输和本地内存使用量，会增大规约操作的规模。

为能最有效地使用本地内存，减少到全局内存的数据传输开销，work-group 的数量应该尽可能接近设备上计算单元或者 SIMD 单元的数量。本地内存位于 AMD 或 NVIDIA GPU 的每个 SIMD 核上，在 OpenCL 标准中，其访存效率至少与全局内存一样。最大化利用这块专用内存区域的高访存带宽是非常重要的，并且我们会使用尽可能多的本地内存来并行化内存访问。就像刚才提到的，我们应该避免创建的 work-group 在数量上超出实际 SIMD 核过多，因为这样会浪费全局内存带宽。

我们可以通过 OpenCL API 在运行时查询设备上计算单元的数量：

```
clGetDeviceInfo(..., CL_DEVICE_MAX_COMPUTE_UNITS, ...);
```

在该算法中，一个 work-item 处理一个像素的方法是非常低效的。这主要是因为映射到每个局部直方图的像素数量越少，在规约阶段产生的开销就越高。同时，如果 work-item 数量过多，每个 work-item 和 work-group 的执行开销就会增大，同样也会降低算法性能。

选择最优的 work-group 大小

针对特定算法的 work-group 最优大小是非常难以确定的。其最小值应是硬件调度单元宽度的偶数倍。这是因为目标架构大多是基于向量的，如果仅仅使用一半单元来调度执行 work-item，将是非常低效的。在 AMD GPU 上，硬件调度单元是 wavefront——一个运行在硬件 SIMD 单元上，包含 32 或者 64 个 work-item 的逻辑向量。OpenCL

work-group 的执行过程以 wavefront 为单位。

每个计算单元只部署一个 wavefront 是不能充分利用硬件计算资源的。硬件每次交替调度两个 wavefront 给 SIMD 单元执行 ALU 运算, 同时调度其他 wavefront 执行访存操作。因此, 对于 AMD GPU, 每个计算单元至少要部署 3 个 wavefront, 才能充分利用其硬件资源。然而, 为能充分的隐藏访存密集型 kernel 的全局内存访存延迟, 每个 SIMD 最少需要部署 7 个 wavefront。

我们不可能超过设备所支持的 work-group 能够包含的 work-item 的最大值, 对于 AMD GPU 这个值为 256。最小情况下, 每个 work-group 包含 64 个 work-item, 每个计算单元部署三个 work-group。最多情况下, 每个 work-group 包含 256 个 work-item, 而每个计算单元部署的 work-group 在数量上受限于硬件资源。不过, 部署过多的 work-item 也是没有好处的。一旦部署的 work-item 在数量上足以充分利用计算资源和隐藏访存延迟, 再部署额外的 work-item 也不会提升性能, 反而可能产生额外的开销。根据这两个范围, 通过细致调优找到最佳性能点是非常有必要的。

全局内存访存优化

直方图计算是一个访存密集型操作: 输入图像的每个像素只进行少量的算术操作。因此, 我们不能依赖 OpenCL 设备强大的浮点数计算能力来获取高性能。相反, 我们必须尽量高效利用内存系统, 当数据从设备的一种内存迁移到另一种内存时, 要对访存进行优化。

对于直方图操作, 全局内存的数据读取顺序是相互独立的。这就意味着我们可以自由优化访存模式而不用考虑数据读取顺序的正确性。

AMD GPU 和 CPU 系统访存的最优方案是进行 128 位读操作, 正好和包含 4 个 32 位浮点数的像素大小相匹配。在 GPU 中, 单独的读操作是低效的, 向量化硬件可以容纳多达 64 个 work-item, 因此, 同时执行多次 128 位读操作可以充分利用 GPU 的高速内存数据通道。对于所有同时执行的 work-item 来说, 其理想读取模式是每个 work-item 读取连续的 128 位数据, 这样在一个计算单元上并行执行的 16 个 work-item 就会读取 256 个字节的数据, 直接映射到 GPU 内存的 channel(通道)上。

我们采用的访存模式如下: 每个 work-item 读取 4 个无符号整型数据, 组成长度为 4 的向量(unit4)。每个 work-item 以其全局索引值为相对起始读取地址, 下次读取的地址跨度等于所有 work-item 读取的字节数。

一个高效的数据读取方式如以下代码和图 9.2 所示:

```
uint gid = get_global_id(0);
uint Stride = get_global_size(0);

for( i=0, icx = gid; i < n4VectorsPerWorkItem; i++, icx += Stride ) {
    uint4 temp = Image[icx];
    ...
}
```

尽管代码相似, 图 9.3 是一个低效的 GPU 实现(每个 work-item 的读取操作都是串行的)。然而, 当系统中的 work-item 在数量上并没有明显超过 CPU 核时, 这个实现对 CPU 来说是完美的。

```
uint gid = get_global_id(0);

for( i=0, icx = gid * n4VectorsPerWorkItem;
      i < n4VectorsPerWorkItem;
      i++, icx++)

    uint4 temp = Image[icx];
    ...
}
```

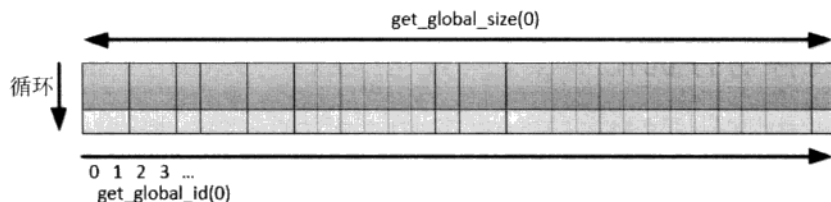


图 9.2

一个高效的 GPU 读取方式: work-item 读取连续的地址, 然后经过一个内存跨度循环读取下一个数据

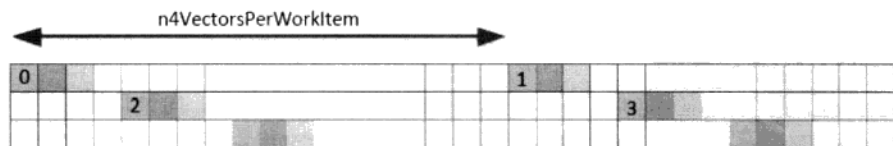


图 9.3

串行化读取: 每个 work-item 读取一段连续的内存。这种读取方式对 CPU 来说是高效的, 因为一个 work-item 的空间局部性比串行执行多个 work-item 的合并读取更高效

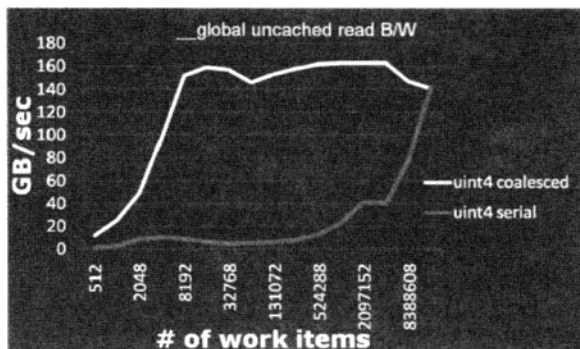


图 9.4

串行读取与合并读取比较: 在采用向量化串行、合并读取两种访存模式下, AMD Radeon HD 6970 GPU 内存系统吞吐量随 work-item 数量变化的示意图。注意, 在数据集大小固定的情况下, 当 work-item 数量非常多时, 两种读取模式的性能曲线趋于一点, 此时两者具有相同的性能。该例子的数据集大小为 256 MB

从图 9.4 中我们可以看到, 随 work-item 数量的变化, 两种访存模式在性能上的差异。采用合并读取模式, 开始我们可以看到一个明显的性能增长, 但随着 work-item 数量的增多, 性能开始降低, 因为此时系统中存在过多的活动 work-item。采用串行读取模式, 当每个 work-item 读取数据非常多时, 性能非常低, 但随着读取数据的减少, cache line 的数据重用与内存系统中的合并读取开始提升性能直到两种读取模式的性能曲线汇于一点, 此时一个 work-item 读取一个元素。

使用原子操作计算局部直方图

当这个应用在 CPU 上运行时, 每个 work-group 只有一个线程, 因此不存在读后写的危害。然而, 在 GPU 上运行时, 我们可以使用 SIMD 向量以及提供细粒度线程。这意味着可以并行执行多个 work-item(在高端 GPU 上, 为 16 个 work-item), 而且我们可以同时运行多个 wavefront, 以使指令交替进行。因此, 在局部直方图计算中, 我们不能保证读写依赖的顺序。

为了解决这个问题, 我们可以多次重新生成局部直方图, 然而这么做需要为 work-group 内的所有 work-item 对每个直方图单元进行一次拷贝操作(我们并不知道它们的执行顺序)。这么做的结果是内存压力相当大。

另一种解决方案是使用硬件原子操作。如同在第 6 章讨论的那样, AMD Radeon HD6970 架构的硬件原子操作和每条通道上的本地数据存储相关, 并且能显著改善性能。我们可以使用这些原子操作而不用顾虑算法的开销。

在本地内存中执行直方图操作就如同下面代码一样简单。值得注意的是, 由于我们同时读取了 16 个像素共 128 位的数据(每个像素为单通道, 大小为 8 位), 因此我们需要在主循环中重复调用这些代码, 并使用掩码和偏移操作来提取这些数据。在一个 uint4 向量上执行掩码和偏移操作可以有效映射到底层硬件架构上。

```
for( int i=0, idx=gid; i<n4VectorsPerWorkItem; i++, idx+= Stride ) {
    uint4 temp = Image[idx];
    uint4 temp2 = (temp & mask);

    (void) atomic_inc( subhists + temp2.x );
    (void) atomic_inc( subhists + temp2.y );
    (void) atomic_inc( subhists + temp2.z );
    (void) atomic_inc( subhists + temp2.w );

    temp = temp >> 8;
    temp2 = (temp & mask);
    ...
}
```

本地内存访存优化

直接在全局内存上计算局部直方图的算法是不可扩展的(当我们加入额外的硬件资源时, 性能得不到相应的提升)。因为在执行直方图单元操作时存在着一个严重的性能瓶颈: 不管使用原子操作还是每个 work-group 使用多个局部直方图, 都会由于内存传输量的显著增长而导致 DRAM 接口对性能的限制。

为避免这个问题, 我们选择为每个 work-group 在本地内存上创建独立的局部直方图。不管程序运行在 CPU 还是 GPU 上, 本地内存的带宽都非常高, 因此使用局部直方图不仅保证了 CPU 上 cache 的独立性, 而且保证了 GPU 上专用高带宽内存(LDS)的有效使用。因为我们的访存操作使用宽 SIMD 向量, 所以在访问本地内存时要非常小心。以 AMD Radeon 6970 GPU 为例, 其本地内存架构是一个具有 32 个 bank 的 SDRAM 数组, 这些 bank 由内存地址的 2~7 位标记。如果并行执行的 16 路 SIMD 中有多个线程试图同时访问同一个 LDS bank 时, 就会发生 bank 冲突。

相对输入数据, 我们使用了少量的 workgroup, 所以每个 work-item 会处理大量的输入像素。

这样可以减小最后规约算法的规模, 将工作集中在本地内存上, 使数据传输本地化以改进性能。这么做的另外一个好处是我们可以略微扩大每个 work-group 使用的本地内存大小, 如此一来每个 work-group 可以计算一组直方图, 而不只限于计算一个直方图, 最后再进行一次规约操作。每个 work-item 计算大量的像素, 所以可以忽略规约阶段的开销。

如果内存布局得当, 使用多个局部直方图所带来的性能提升是非常明显的。图 9.5 显示了我们如何通过多个子直方图来减少 bank 冲突。AMD 硬件的 LDS 有 32 个 bank, 直方图单元可能会映射到同一 bank 上, 因此一条 SIMD 指令内的 16 个 work-item 可能同时计算位于同一 bank 上的直方图单元而导致 bank 冲突, 这种冲突将导致应用挂起。

为避免应用挂起, 为每个 work-item 复制一份局部直方图, 并存储在 LDS 的同一个 bank 内, 以保证每个 work-item 无论计算哪个直方图单元, 都不会引起 bank 冲突。这样做的代价是增加了每个 work-group 本地内存的使用和一次本地规约操作。

图 9.6 说明了这种方法对性能影响。当访问一个随机数据集时, 由于冲突很少, 规约开销是非常明显的。当访问均匀数据集时, 如只使用一个局部直方图会产生大量的 bank 冲突, 所以增加局部直方图的数量之后, 性能提升是非常明显的。

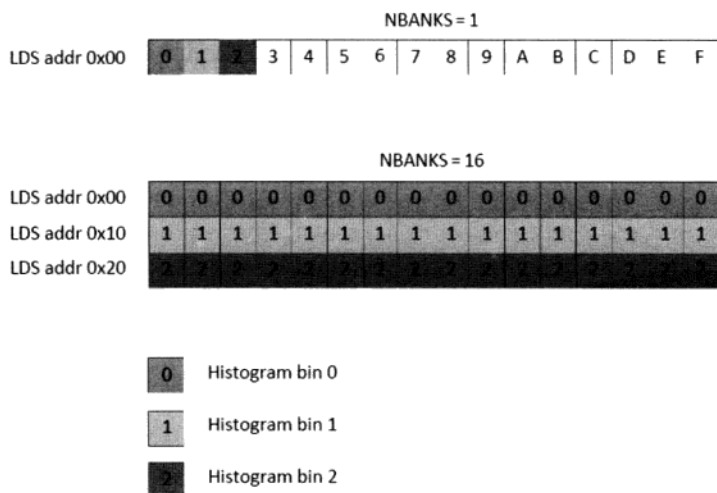


图 9.5

只有一个局部直方图和 16 个硬件 bank 时, 每隔 16 个单元就会映射到同一个硬件 bank。这样, 同一个 SIMD 向量的任意两个 work-item 访问直方图单元 a 和 b, 当 $a \% 16 = b \% 16$ 时, 就会发生冲突。通过使用多个局部直方图, 我们消除了这些冲突

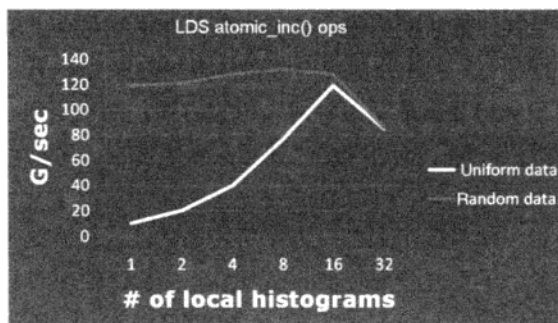


图 9.6

局部直方图的数量对性能的影响。随着一个 work-group 的局部直方图数量的增加，LDS 冲突的数量也随之减少。同时也付出了两个代价：增加了一次本地规约操作；由于资源限制，系统中并发执行的线程数减少

实际数据介于这两个值之间。即使我们使用的评估平台 Radeon HD6970 总共有 32 个 bank，性能峰值性能却出现在 bank 数为 16 时。这是因为一个 SIMD 向量只能同时部署 16 个 work-item，并且每条 SIMD 通道的原子操作被限制为 32 位，所以我们最多只能使用其中的 16 个 bank。

接下来，我们修改之前的代码，根据局部直方图的数量计算偏移量：

```
#define NBANKS 16

uint offset = (uint) lid % (uint) (NBANKS);

for( int i=0, idx=gic; i<n4VectorsPerWorkItem; i++, idx+= Stride ) {
    uint4 temp = Image[idx];
    uint4 temp2 = (temp & mask) * (uint4) NBANKS + offset;

    (void) atomic_inc( subhists + temp2.x );
    (void) atomic_inc( subhists + temp2.y );
    (void) atomic_inc( subhists + temp2.z );
    (void) atomic_inc( subhists + temp2.w );
    ...
}
```

局部直方图的规约

在总结局部直方图的讨论之前，我们必须对多个局部直方图进行规约，并将最终结果加载到全局内存上。我们既可以屏蔽掉多余的 work-item，也可以在局部直方图上进行

迭代操作(直方图单元在数量上超过 **work-item** 时)。然后, 将局部直方图加载到全局内存上:

```
barrier( CLK_LOCAL_MEM_FENCE );
for( int binIndex = lid;
    binIndex < NBINS;
    binIndex += get_local_size(0) ) {

    uint bin = 0;

    for( int i=0; i<NBANKS; i++ ) {
        bin += subhists[
            (lid * NBANKS) + ((i + lid) % NBANKS) ];
    }

    globalHistogram[
        (get_group_id(0) * NBINS) + binIndex ] = bin;
}
```

注意, 在前面的代码中, 我们修改代码以避免求和循环中的 **bank** 冲突, 否则会产生 **NBANKS** 路冲突。

全局规约

直方图算法的最后一步是对所有的局部直方图进行全局规约。由于我们精心限制了 **work-group** 的数量, 所以全局规约的工作量非常小。如此一来, 我们可避免使用复杂的基于树的规约算法和多通路算法, 而是采用与局部直方图规约相似的一个求和 **kernel**:

```
__kernel void reduceKernel(
    __global uint * globalHistogram,
    uint nSubHists ) {

    uint gid = get_global_id(0);
    uint bin = 0;

    for( int i=0; i < nSubHists; i++ )
        bin += globalHistogram [ (i * NBINS) + gid ];
    globalHistogram [ gid ] = bin;
}
```

完整的 kernel 代码

```
__kernel __attribute__((reqd_work_group_size(NBINS,1,1)))
void histogramKernel(__global uint4 *Image,
                    __global uint *globalHistogram,
                    uint n4VectorsPerWorkItem){
```

```

__local uint subhists[NBANKS * NBINS];

uint gid    = get_global_id(0);
uint lid    = get_local_id(0);
uint Stride = get_global_size(0);
uint4 temp, temp2;
const uint shift = (uint) NBINS;
const uint mask = (uint) (NBINS-1);
uint offset = (uint) lid % (uint) (NBANKS);

uint localItems = NBANKS * NBINS;
uint localItemsPerWorkItem;
uint localMaxWorkItems;

// parallel LDS clear

// first, calculate work-items per data item.
// at least 1:
localMaxWorkItems = min( 1, get_local_size(0) / localItems );
// but no more than we have items:
localMaxWorkItems = max( 1, localMaxWorkItems / localItems );
// calculate threads total:
localMaxWorkItems = localItems / localMaxWorkItems;
// but no more than LDS banks:
localMaxWorkItems = min( get_local_size(0), localMaxWorkItems );

localItemsPerWorkItem = localItems / localMaxWorkItems;

// now, clear LDS
__local uint *p = (__local uint *) subhists;

if( lid < localMaxWorkItems ) {
    for(i=0, idx=lid;
        i<localItemsPerWorkItem;
        i++, idx+=localMaxWorkItems)
    {
        p[idx] = 0;
    }
}

barrier( CLK_LOCAL_MLM_ILNCL );

// read & scatter phase
for( int i=0, idx=gid;
    i<n4VectorsPerWorkItem;
    i++, idx += Stride )

    temp = Image[idx];
    temp2 = (temp & mask) * (uint4) NBANKS + offset;

    (void) atomic_inc( subhists + temp2.x );
    (void) atomic_inc( subhists + temp2.y );
    (void) atomic_inc( subhists + temp2.z );

```

```

(void) atomic_inc( subhists + temp2.w );
temp = temp >> shift;
temp2 = (temp & mask) * (uint4) NBANKS + offset;

(void) atomic_inc( subhists + temp2.x );
(void) atomic_inc( subhists + temp2.y );
(void) atomic_inc( subhists + temp2.z );
(void) atomic_inc( subhists + temp2.w );

temp = temp >> shift;
temp2 = (temp & mask) * (uint4) NBANKS + offset;

(void) atomic_inc( subhists + temp2.x );
(void) atomic_inc( subhists + temp2.y );
(void) atomic_inc( subhists + temp2.z );
(void) atomic_inc( subhists + temp2.w );

temp = temp >> shift;
temp2 = (temp & mask) * (uint4) NBANKS + offset;

(void) atomic_inc( subhists + temp2.x );
(void) atomic_inc( subhists + temp2.y );
(void) atomic_inc( subhists + temp2.z );
(void) atomic_inc( subhists + temp2.w );
}

barrier( CLK_LOCAL_MEM_FENCE );

// reduce __local banks to single histogram per work group
for( int binIndex = lid;
      binIndex < NBINS;
      binIndex += get_local_size(0) ) {

    uint bin = 0;

    for( int i=0; i<NBANKS; i++ ) {
        bin += subhists[
            (lid * NBANKS) + ((i + lid) % NBANKS) ];
    }

    globalHistogram[
        (get_group_id(0) * NBINS) + binIndex ] = bin;
}
}

__kernel void reduceKernel(
    __global uint *globalHistogram,
    uint nSubHists ) {

    uint gid = get_global_id(0);
    uint bin = 0;

```

性能和小结

结合这些方法，我们可以高效地处理大量输入数据。在 AMD Radeon HD6970 平台上，使用 256 个直方图 bin，我们可以达到如下性能。

1. 158 GB/s 将输入数据从全局内存读取到寄存器中。
2. 140 GB/s 在 1 基础上，增加数据到局部直方图的加载。
3. 13x9 GB/s 在 1 和 2 的基础上增加本地规约操作。
4. 128 GB/s 在 1~3 的基础上，增加最后的全局规约，需要入队并执行第二个 kernel。

本章集中讨论了访存密集型 kernel 的优化方法和优化步骤。讨论了如何调整数据读取模式和使用本地内存来减少访存开销。通过细致而有效的优化，我们可以将其优化到与只读取数据的 kernel 差不多的性能。减少本地内存 bank 冲突和提高全局内存性能的技术在其他算法中也有广泛的应用。

OpenCL 案例学习 4：混合粒子模拟

引言

本案例使用 CPU 和 GPU 实现了混合尺度的粒子模拟。在不丧失一般性的情况下，本章仅限于考察二维问题以使 kernel 代码更容易理解。此外，本例还使用了 OpenCL 在 CPU 和 GPU 间可以共享数据的特性，这将更适合于 AMD APU 这类高度集成的设备。

在 GPU 这样支持数据并行的硬件上，实现单一大小的粒子的模拟是相对容易的。但不幸的是，如果这种假设不成立(比如粒子大小是变化的)，模拟的效率会降低。造成低效的原因是不均衡的计算粒度，特别是碰撞检测，它是整个粒子模拟中的主要计算部分。假设有一个大的粒子和许多小的粒子，那么在大粒子上检测到的碰撞数量要显著高于小粒子。大多数 GPU 采用宽向量的 SIMD 方式，如果一个 GPU SIMD 引擎在大多数的 SIMD 路上执行小粒子相关计算，而只有一条 SIMD 路计算大粒子，那么“大粒子”的路将比“小粒子”的路运行更长的时间。这意味着，计算小粒子碰撞的路必须等待着大粒子的路完成。随着向量宽度的增加，任务的不平衡性也越来越严重，设备利用率越来越低。CPU 没有处理这种计算粒度变化的问题。因此，我们使用了这种方法：使用 CPU 计算大粒子的碰撞，使用 GPU 计算统一尺度(大小)的小粒子。本章介绍如何使用 OpenCL 来实现 GPU 和 CPU 的协作方式，从而完成混合尺度的粒子模拟。

计算概览

图 10.1 是模拟的屏幕截图。可以看到渲染成蓝色的大量小粒子，和其他大小的绿色

和红色的粒子。其中，绿色和红色分别代表了动态和静态粒子。粒子间的碰撞分为 3 种类型：小-小，大-小和大-大。小-小碰撞的粒度是均匀的，所以这部分工作由 GPU 进行。如前所述，大-小和大-大的粒度是变化的，所以 CPU 更适合运行这部分计算。如果没有大-小的碰撞，那么剩下的两个是完全独立的，也就是说 GPU 模拟小粒子，CPU 模拟大粒子。混合粒子模拟可以看成这两种独立模拟的扩展。如何将存储在 GPU 显存的小粒子和 CPU 内存的大粒子实现高效的相互作用是一大挑战。如果不考虑分配内存的优化，那么在每次计算相互作用时，GPU 的数据必须要传输回主机端。但是，通过使用 OpenCL 的 CPU 和 GPU 可共享访问的 buffer，那么这个数据传输就不是必须的了。

算法概览如图 10.2 所示，其中还表示了主机 CPU 上的 GPU 管理线程和 CPU 工作线程间的交互。每次计算迭代分为 3 个主要步骤：(1)数据结构构造；(2)计算碰撞的作用力 (3)计算力的合成。

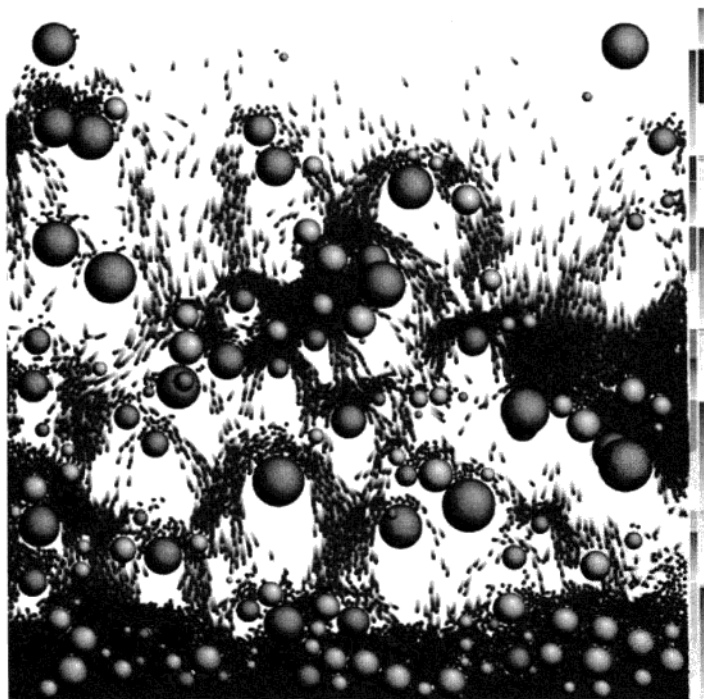


图 10.1

混合粒子模拟的屏幕截图(请参考本书封面)

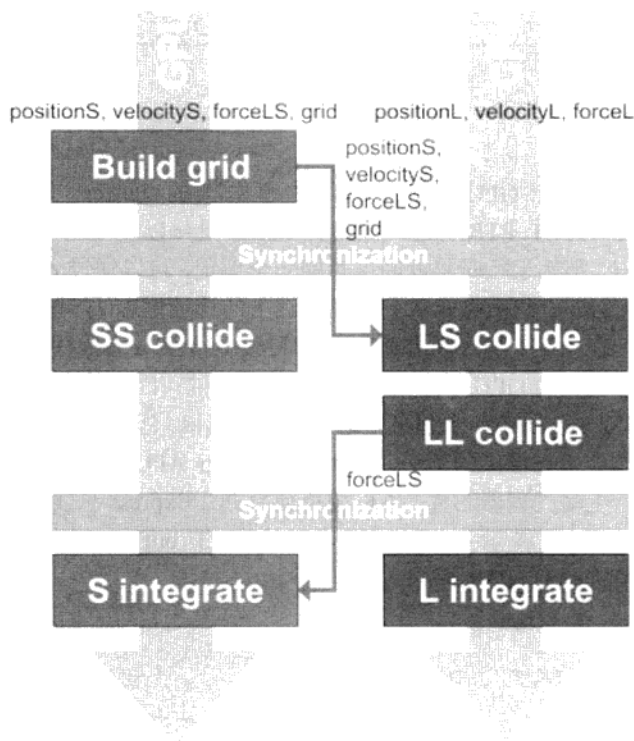


图 10.2

单次迭代的计算流程概览

首先，生成 GPU kernel 所用的数据结构。之后，GPU 和 CPU 进行同步，确保新数据结构上的计算完成，与 CPU 线程共享小粒子的更新数据和物理属性。在第一次同步之后，开始进行碰撞的计算。GPU 计算小-小的碰撞，CPU 计算大-小和大-大的碰撞。为了加速小粒子和大粒子间的碰撞检测，GPU 和 CPU 使用相同的数据结构。碰撞的结果是一组在粒子上的作用力。针对小粒子使用两个力的 buffer：一个用来保存小-小碰撞产生的力，另外一个用来与 CPU 共享，保存大-小碰撞的结果。在碰撞计算结束后，GPU 和 CPU 线程再次进行同步，保证数据全部写完，用于之后的合成阶段。之后，GPU 合成小粒子上的力，CPU 合成大粒子的力。请读者注意，CPU 和 GPU 对于共享的 buffer 没有写冲突。CPU 只是在大-小碰撞阶段写共享的 buffer，而 GPU 只在同步后的合成阶段访问共享 buffer。

GPU 实现

创建 buffer

首先，需要创建在 GPU 和 CPU 间共享数据的 buffer。可以在创建时设置 CL_MEM_ALLOC_HOST_PTR 标志，指明此 buffer 是可以被 CPU 访问的。如果 buffer 创建时设置此标志，那么运行时将返回主存的一个地址，可以通过 map 和 unmap 函数访问其中的数据。我们还有将数据永久驻留在主存上的选项标志(尽管这不是必须的)。^① 如果数据是永久驻留在内存，那么 buffer 的 map 和 unmap 函数的执行开销非常低，只需要同步开销。但对于独立的 GPU，这种好处也是有代价的。因为数据保存在 CPU 内存上，所以任何 GPU 访问 buffer 的请求都必须通过 PCI Express 总线。与访问 GPU 内存上的数据相比，这种操作会带来巨大的延迟。AMD 的 Fusion APU 架构和类似的 GPU 与 CPU 共享内存的异构片上多处理器芯片，可以避免使用 PCI Express 总线的开销，从而得到更好的性能。以下代码展示如何创建一个共享的 buffer 请注意传入参数中的 CL_MEM_ALLOC_HOST_PTR 标志：

```
buffer = clCreateBuffer(
    context,
    CL_MEM_READ_WRITE | CL_MEM_ALLOC_HOST_PTR,
    bufferSize,
    0,
    &status );
```

尽管此 buffer 创建为 CPU 和 GPU 共享，但是由于 OpenCL 内存空间的不统一，所以我们不能通过这个 buffer 的指针访问数据。为了在 CPU 上访问 buffer 的数据，我们必须对 buffer 进行 map 操作：

```
ptr = clEnqueueMapBuffer(
    commandQueue,
    buffer,
    false,
    CL_MAP_READ | CL_MAP_WRITE,
    0,
    bufferSize,
    0,
    0,
    0,
    &status );
```

^① 注意，CL_MEM_ALLOC_HOST_PTR 保证 buffer 调用 map 函数后，传入的指针会返回成为 CPU 可访问的指针。返回指针所指向的数据在调用 map 函数后和 unmap 函数前是有效的。

`clEnqueueMapBuffer` 函数返回指向 `buffer` 数据的指针。如果 `buffer` 没有使用那个标志创建, 那么 `map` 操作会将数据从设备拷贝回主机上, 时间开销非常高。但是, 如果使用那个标志, 在适当的运行时和硬件支持下, `map` 操作的开销就会很低, 之后 `buffer` 就会存在于主存上。

构造加速结构

在加速粒子碰撞检测中, 我们使用了每个网格单元功能一致的均匀网格(即网格内各单元的空间尺寸不变)。如果粒子模拟不需要扩展到无限空间, 那么均匀网格的快速构建和查询的特性非常适合粒子模拟中的碰撞检测。均匀网格有 2 个 `buffer`: 一个计数器 `buffer` 保存网格单元的计数, 一个索引 `buffer` 保存粒子的索引或者网格单元的数据。每个 `work-item` 对应一个粒子, 将粒子的空间位置信息从全局内存中读取到寄存器中, 之后转换成网格坐标。然后, `work-item` 从计数器 `buffer` 中读取在此网格单元内的粒子数量, 把这些粒子的索引添加到索引 `buffer` 中。在访问计数器 `buffer` 时必须小心谨慎, 因为一些粒子可能会落到同样的网格单元内, 所以这些 `work-item` 是试图读/写同一个计数器。为了保证正确的内存访问, 我们需要使用原子操作, 如下所示:

```
for( all particles ) {
    idx = convertToGridIdx( particle[i] );
    count = atomic_inc( grid_count[idx] );
    grid[ calcAddr( idx, count ) ] = 1;
}
```

在后面展示粒子模拟的 `kernel` 代码中, 还包括额外的逻辑, 用于处理边界情况和防止过多粒子都落在一个网格单元中。

计算碰撞

在此步中, 计算作用于每个粒子的来自碰撞粒子的力。均匀网格用于快速定位每个粒子所关联的碰撞粒子。粒子的空间位置已经转换为网格坐标, 可以计算出网格单元的内存地址。由于网格单元的大小固定为粒子的直径, 所以可能碰撞的粒子只存在周围的 3^2 个网格单元中(三维情况下是 3^3 个)。因为我们计算的粒子直径相同, 所以计算的粒度是统一的, 所有 `work-item` 都要检查 9 个单元。一个 `work-item` 需要依次访问 9 个单元, 读取每个单元的粒子数量和坐标, 计算到本单元的作用力。作用力是一个碰撞粒子的位置和速度的函数。在计算过粒子间的碰撞后, 每个 `work-item` 都会检查粒子碰撞的边界条件。

最后，将这些力累加并写到作用力的 **buffer** 中。

```
for( all particles ) {
    force = 0;

    // compute particle interaction
    for( neighborhood ) {
        for( particles in grid cell[n] ) {
            j = getStoredParticleIdx();
            force += calcForce( particle[i], particle[j] );
        }
    }
    // compute boundary interaction
    force += calcForce( particle[i], top );
    force += calcForce( particle[i], bottom );
    force += calcForce( particle[i], left );
    force += calcForce( particle[i], right );
    forceOut[i] = force;
}
```

合成

在通过 CPU 和 GPU 计算作用力之后，需要更新粒子的位置和速度，我们把这个过程称之为合成。将小-小粒子的相互作用力和大-小粒子的作用力进行求和，之后计算速度。粒子的位置用速度乘以时间步长 dt 进行更新。最终，将与 CPU 共享数据的 **buffer** 清零，为下一次迭代做准备。这个步骤的伪码非常简单，如下所示：

```
for( all particles ) {
    vel[i] += forceSmallSmall[i] + forceLargeSmall[i];
    pos[i] += vel[i]*dt;

    // zero forces for the CPU side
    forceBigParticles[i] = 0;
}
```

CPU 实现

CPU 实现与 GPU 版本差别不大，不同之处是相比 GPU 的向量，CPU 更偏好于标量的处理。因为粒子的碰撞都是串行处理的，所以我们不需要考虑粒度的问题。大粒子和小粒子的碰撞，也是使用 GPU 上构建的均匀网格，与 CPU 进行共享。首先，计算大粒子在网格空间的范围，查找相关的单元。图 10.3 展示了大粒子所覆盖的网格单元，如图所示，大粒子占据了很多单元，而小粒子相对较少。如果这部分代码在 GPU 上执行，会浪费许多 SIMD 路的执行周期。

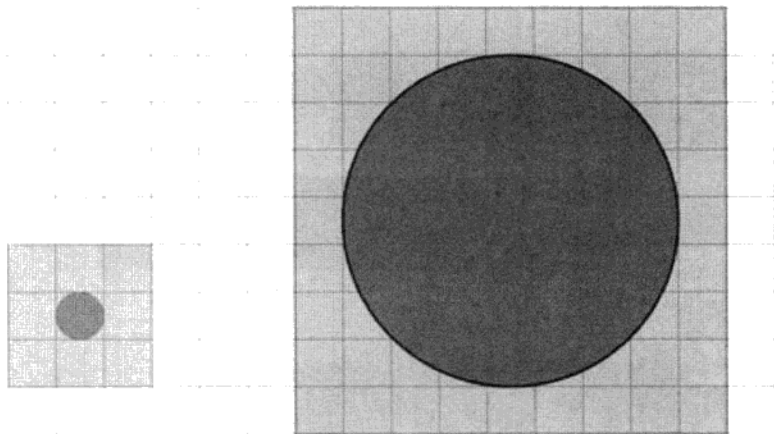


图 10.3

映射到均匀网格上的小粒子和大粒子

负载均衡

尽管在图 10.2 中, 我们展示了使用两个线程的模拟流程, 但是这种实现方式的计算负载很不均衡。这种不均衡造成 CPU 的利用率很低。当然, 不均衡的程度与小粒子和大粒子的数量有关系。请看 CPU 工作线程, 显然, 大-小和大-大碰撞比合成的计算量更大。因此, 在第 1 次和第 2 次同步间的时间最长。另一方面, 在 GPU 端, 计算量最大的是小-小碰撞, 之后是均匀网格的生成。尽管 GPU 的计算能力很强, 但是构建均匀网格的开销还是不可忽略的。如果模拟如图 10.2 的方式执行, 那么如同图 10.4A 所示, CPU 必须等着第一个同步, GPU 必须等待第二个同步。

可以调整计算顺序来改进负载的不均衡。图 10.2 只表示了一次迭代的流程, 如果我们假设粒子在上次迭代结束和本次迭代开始之前是不动的, 那么网格的生成可以不放在流程的第一步。这个假设在通常情况下是可满足的, 除非粒子可以直接移动, 比如通过用户交互的方式, 但这种情况是非常少见的。基于这个观察, 我们可以把网格的生成移到上一次迭代的末尾。通过这种改动, 使得 GPU 的后半部分有两个大块的计算, 而相应的 CPU 部分只包含粒子的合成(这个计算量小)。如果我们可以把 CPU 的一些计算从前半部分移到后半部分, 那么也可以缓解负载不均衡的情况。仔细观察流水线上的数据依赖

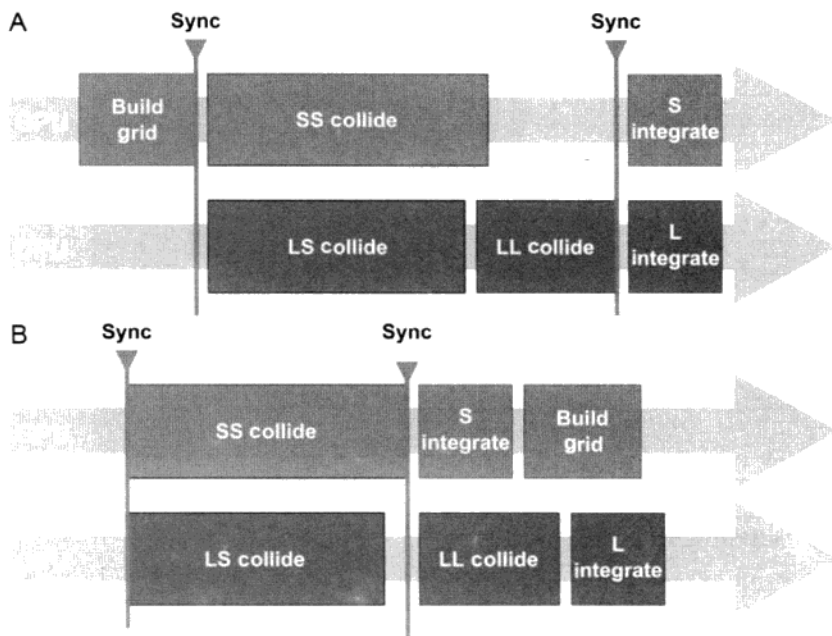


图 10.4

负载均衡

关系后，可以看到第二次同步只是针对包含小粒子作用力的 buffer 是必须的。所以，之后 GPU 的执行并不依赖于 CPU 上大-大粒子的碰撞。这意味着，我们可以把大-大粒子的碰撞移到后半部分进行计算(如图 10.4B 所示)。通过这种方法，我们改进了 CPU 上的负载均衡，最终减少了整个模拟时间。

性能和小结

图 10.1 是混合粒子模拟的屏幕截图，大约有 8200 个小粒子和 150 个大粒子，运行平台是 AMD A-series Fusion APU，其 CPU 和 GPU 集成在一块芯片上。图右侧的两条柱代表了两个线程的计算时间(每个线程对应一条)。整个柱的高度对应 1/60 秒，蓝色和绿色的块分别对应花费在 GPU 和 CPU 上的时间。由于每时间步有 4 次迭代，所以每次迭代近似耗时大约 4 ms。最开始的 4 个块对应第一次迭代，可以看到 GPU 计算后的同步操作。CPU 等待 GPU 的执行完成，产生了第一个绿色块和同步之间的缝隙。

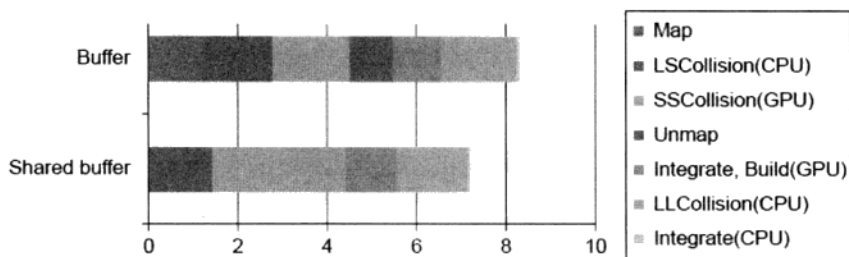


图 10.5

模拟时间的细分图

如果没有使用 `CL_MEM_ALLOC_HOST_PTR` 标志创建 `buffer`, 那么在 `buffer` 的 `map` 和 `unmap` 操作中须得拷贝数据。`map` 操作的影响可以通过比较使用普通 `buffer` 的模拟和使用共享 `buffer` 的模拟, 进行定量的分析, 图 10.5 是二者的时间细分图。尽管模拟本身是相同的, 但可以清楚看到使用和不使用数据共享的差别, 使用共享的 `buffer` 具有明显的优势。也需要注意到, 在共享数据后, GPU 部分执行小-小碰撞的时间变长, 因为通过共享的内存接口访问数据带来了额外的延迟。

在本章, 我们讨论了混合粒子模拟的实现, 利用了 CPU 和 GPU 两者的优势。此外, 在如 AMD Fusion APU 这种 CPU 和 GPU 紧密集成的设备上, 数据的共享可以消除多余的数据拷贝, 从而提高计算效率。尽管本例为了便于理解而有意简化, 但是同样的技术可以应用于很多复杂的应用。

生成均匀网格的 kernel 代码

```
typedef struct
{
    float4 m_max;
    float4 m_min;
    int4 m_nCells;
    float m_gridScale;
    u32 m_maxParticles;
}ConstBuffer;

__kernel
void GridConstructionKernel(
    __global float4* gPosIn,
```

```

    __global int* gridG,
    __global int* gridCounterG,
    ConstBuffer cb )
{
    int glidx = get_global_id(0);

    if( glidx >= cb.m_maxParticles ) {
        return;
    }

    float4 iPos = gPosIn[glidx];

    int4 gridCrd = ugConvertToGridCrd(
        iPos*cb.m_min, cb.m_gridScale );

    if( gridCrd.x < 0 || gridCrd.x >= cb.m_nCells.x
        || gridCrd.y < 0 || gridCrd.y >= cb.m_nCells.y ) {
        return;
    }

    int gridIdx = ugGridCrdToGridIdx(
        gridCrd,
        cb.m_nCells.x,
        cb.m_nCells.y,
        cb.m_nCells.z );

    int count = atom_add(&gridCounterG[gridIdx], 1);

    if( count < MAX_IDX_PLR_GRID ) {
        gridG[ gridIdx*MAX_IDX_PLR_GRID + count ] = glidx;
    }
}

```

粒子模拟的 kernel 代码

```

typedef struct
{
    float4 m_g;
    int m_numParticles;
    float m_dt;
    float m_scale;
    float m_e;

    int4 m_nCells;
    float4 m_spaceMin;
    float m_gridScale;
}ConstBuffer;

__kernel
void CollideGridKernel(
    __global float4* posIn,

```

```

__global float4* velln,
__global int* grid,
__global int* gridCounter,
__global float4* forceOut,
ConstBuffer cb)

{
    int gIdx = get_global_id(0);

    if(gIdx >= cb.m_numParticles) {
        return;
    }

    int4 nCells = cb.m_nCells;
    float4 spaceMin = cb.m_spaceMin;
    float gridScale = cb.m_gridScale;
    float dt = cb.m_dt;
    float e = cb.m_e;

    float4 f = make_float4(0,0,0,0);

    float4 x_i = posIn[ gIdx ];
    float4 v_i = velln[ gIdx ];

    float sCoeff, dCoeff;
    calcCoeffs(v_i.w, v_j.w, sCoeff, dCoeff);

    int4 iGridCrd = ugConvertToGridCrd( x_i-spaceMin, gridScale );

    //1. collide particles
    for(int i=-1;i<=1;i++) for(int j=-1;j<=1;j++) {
        int4 gridCrd = make_int4(
            iGridCrd.x+i,
            iGridCrd.y+j,
            iGridCrd.z+k,
            0);

        if( gridCrd.x < 0 || gridCrd.x >= nCells.x
            || gridCrd.y < 0 || gridCrd.y >= nCells.y ) {
            continue;
        }

        int gridIdx = ugGridCrdToGridIdx(
            gridCrd,
            nCells.x,
            nCells.y,
            nCells.z );

        int numElem = gridCounter[ gridIdx ];
        numElem = min(numElem, MAX_IDX_PER_GRID);

        for(int ie=0; ie<numElem; ie++) {
            int jIdx = grid[ MAX_IDX_PER_GRID*gridIdx + ie ];

```

```

        if( jIdx == gIdx ) {
            continue;
        }
        float4 x_j = posIn[jIdx];
        float4 v_j = velIn[jIdx];
        f += calcForce(
            x_i,
            x_j,
            v_i,
            v_j,
            x_i.w,
            x_j.w,
            v_i.w,
            v_j.w,
            dt,
            sCoeff,
            dCoeff );
    }
}

//2. collide with boundary
{
    float sCoeff, dCoeff;
    calcCoeffs(v_i.w, sCoeff, dCoeff);

    float4 planes[4];
    planes[0] = make_float4(0.1,0,cb.m_scale);
    planes[1] = make_float4(-1,0,0,cb.m_scale);
    planes[2] = make_float4(1,0,0,cb.m_scale);
    planes[3] = make_float4(0,-1,0,cb.m_scale);

    for(int j=0; j<4; j++) {
        float4 eqn = planes[j];
        float dist = dot3wl( x_i, eqn );
        f += calcForceB(
            x_i,
            v_i,
            x_i.w,
            dist,
            eqn,
            dt,
            sCoeff,
            dCoeff );
    }
}

forceOut[ gIdx ] = f;
}

```

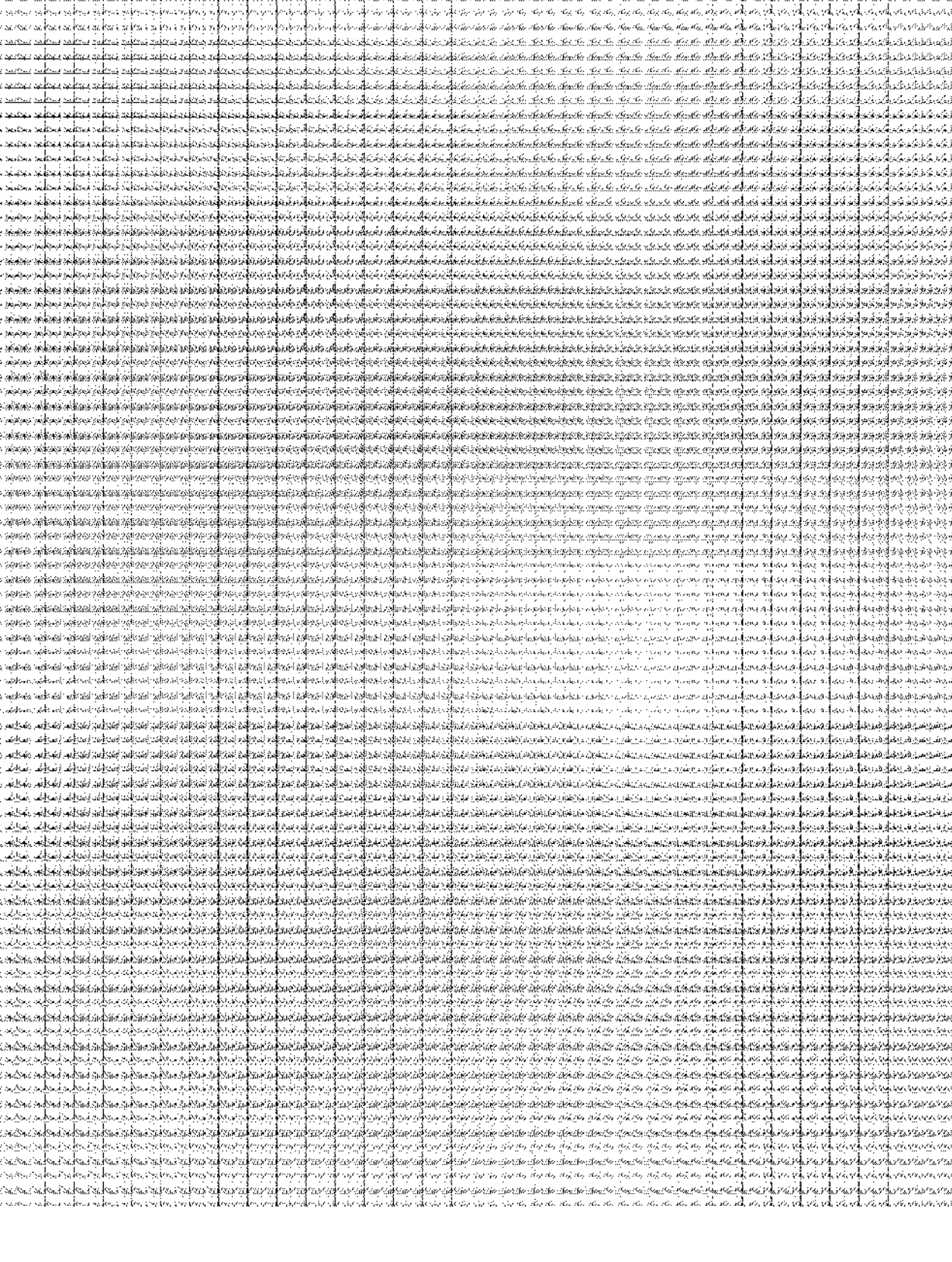
```
__kernel
void IntegrateKernel(
    __global float4* pos,
    __global float4* vel,
    __global float4* forceSS,
    __global float4* forceLS,
    ConstBuffer cb)
{
    int gIdx = get_global_id(0);
    if( gIdx >= cb.m_numParticles ) {
        return;
    }

    float4 x = pos[gIdx];
    float4 v = vel[gIdx];

    v += (forceSS[gIdx]+ forceLS[gIdx])*cb.m_dt/v.w+cb.m_g;
    x += v*cb.m_dt;

    pos[gIdx] = make_float4(x.x, x.y, x.z, pos[gIdx].w);
    vel[gIdx] = make_float4(v.x, v.y, v.z, vel[gIdx].w);

    forceLS[gIdx] = make_float4(0,0,0,0);
}
```



OpenCL 扩展

引言

和大多数编程语言、编程框架类似，OpenCL 规范也支持一些可选择的扩展。它们代表了一部分数量少但很重要的 OpenCL 扩展。在本章，我们将讨论一些这样的扩展，这些扩展能为程序员提供一些有助于实现 OpenCL 应用的工具。

扩展机制概览

OpenCL 定义了三类不同的扩展。

- **KHR 扩展：**KHR 扩展需要由 OpenCL 工作组正式批准，同时任何声称支持该扩展的应用必须通过一套一致性测试。所有 KHR 扩展都是 OpenCL 规范的一部分。一般来说，我们目标是保持 KHR 扩展集相对较小，从而避免各厂商实现之间的不同。遗憾的是，在一些情况下，由于某个特性不是所有平台都支持，所以必须使用扩展的方式。与 DirectX 的互操作就是一个很好的例子，因为它只和支持 Microsoft Windows 的实现有关。每一个 KHR 扩展有一个唯一的字符串，字符串的格式为 `cl_khr_<name>`，`name` 代表特定扩展的名字。
- **EXT 扩展：**EXT 扩展是由 OpenCL 工作组的一个或多个成员开发，但是没有经过 OpenCL 工作组的正式批准。EXT 扩展不需要通过一致性测试。EXT 扩展在早期通常用于访问一些只能在 OpenCL 工作组部分成员的设备上运行的特性，它们中的大部分在之后将成为 OpenCL 核心规范或 KHR 扩展的一部分。每一个 EXT 扩展具有一个唯一的字符串，字符串的格式为 `cl_ext_<name>`，`name` 是特定扩展的名字。
- **厂商扩展：**由某个厂商单独开发，主要用于支持一些只能在某个厂商的硬件设

备或者平台上使用的特性(因此不适合开发一般应用)。

如 AMD 的设备属性查询, 它允许开发者查询针对 AMD 设备的一些附加的信息。每一个厂商扩展具有一个唯一的字符串, 字符串的格式为 `cl_<vendor>_<name>`, `vendor` 是由厂商定义的特定字符串(如: `amd` 代表 AMD), `name` 是特定扩展的名字。

扩展可以和一个给定的 OpenCL 平台关联起来, 在这种情况下, 扩展一般是被启用的。例如: AMD 的事件回调扩展(`cl_amd_event_callback`)总是被启用的。其他扩展一般仅限于某一些特定的设备, 如双精度支持(`cl_khr_fp64`)。

一般来说, 应用程序需要通过查询来了解平台或特定设备支持哪些扩展。下面的代码描述了如何使用 C++ 封装 API 查询平台和设备扩展的信息。

```
std::vector<cl::Platform> platforms;

cl::Platform::get(&platforms);
std::string platformLxts = platforms[0].getInfo<CL_PLATFORM_LXILNSIONS>();

cl_context_properties cprops[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platforms[1](), 0};

cl::Context context(devtype, cprops);
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DLVICLS>();

// Check that double is supported
std::string deviceLxts = devices[0].getInfo<CL_DLVICL_LXILNSIONS>();
```

当然, 因为我们现在已经有了关于该平台和设备的 `std::string` 类型的扩展信息, 我们可以简单利用 `find()` 方法来查找其是否支持某一个扩展。例如, 要验证一个设备是否支持 64 位的原子操作(由 `cl_khr_int64_base_atomics` 定义), 可以使用如下代码:

```
bool has64Atomics = deviceLxts.find("cl_khr_int64");
```

由于扩展是可选的, 因此并没有要求 OpenCL 的实现必须提供可以由主机端应用程序调用的外部接口。相应的, 一个主机端程序不能被静态链接到任何一个扩展 API 上。但我们可以用如下 API 来查询特定函数的地址:

```
void * clGetExtensionFunctionAddress(const char *funcname);
```

该函数返回 `funcname` 对应的扩展函数地址, 返回的指针可以转换成所期望类型的函数指针。typedef 每一个为所有扩展 typedef 每一个声明指向 API 入口的指针扩展声明的函数指针。例如, EXT 扩展原子计数(Atomic Counters), 包含在 OpenCL 扩展头文件

cl_ext.h 里面, 提供了低延迟 append/consume 计数器的原子操作, 它的定义如下:

```
typedef CL_API_NAME cl_counter_ext (CL_API_CALL * clCreateCounter-
LXI_fn)(
    cl_context          /* context */,
    cl_counter_flags_amd /* flags */,
    cl_uint             /* value */,
    cl_int *            /* error_ret */) CL_API_SUFFIX__VERSION_1_0;
```

然后, 假设某一个特定的 OpenCL 设备支持 cl_ext_atomic_counters 扩展, 下面的调用将把 clCreateCounterEXT_pfn 设置成一个非 NULL 值:

```
clCreateCounterEXT_fn clCreateCounterEXT_pfn =
    clGetExtensionFunctionAddress("clCreateCounterEXT");
```

在实践中, 通常可以使用下面的宏来快速分配这些函数:

```
#define __INIT_CL_EXT_FCN_PTR(name) \
    if(!##name_fn) { \
        ##name_pfn = (##name_fn) \
            clGetExtensionFunctionAddress(#name); \
        if(!##name_pfn) { \
            } \
    }
```

如下代码可以获得与前面定义函数指针代码同样的效果:

```
__INIT_CL_LXI_FCN_PTR(clCreateContextLXI)
```

本章后面的例子将进一步使用 OpenCL 的 C++ 封装 API, 如果一个相应的扩展可用, 直接提供显式的类接口并自动分配和查询函数指针。

只要应用程序已经确定设备支持某个扩展, 并且已经分配了所需要的 API 函数地址, 就可以使用相应的 OpenCL C 源码扩展。这些扩展也必须在本次使用的 OpenCL C 源码中被开启。这些特性由一些编译制导指令来控制, 形式如下:

```
#pragma OPENCL_EXTENSION extension_name : behavior |
```

extension_name 指需要开启或关闭的扩展的名字, 如 cl_khr_in64。behavior 可以是 enable 或 disable。开启或关闭一个扩展, 从当前行开始生效, 直到遇到下一个对应的指令或文件结束。

例如，在 OpenCL C 源码程序中，可以使用下面的代码来开启 64 位原子操作扩展：

```
#pragma OPENCL EXTENSION cl_khr_int64 : enable
```

`image` 是一个例外，它与前面所描述的规则不同。要想检查设备是否支持该特性，可以使用一个单独的设备查询语句：

```
bool hasImages = devices[0].getInfo<CL_DEVICE_IMAGE_SUPPORT>;
```

如果该函数调用返回的结果为 `true`，表明设备支持 `image`，而且不需要在 OpenCL C 源码中开启该特性。

现在有很多 KHR、EXT 和厂商扩展，我们不能在本章详细进行介绍。本章余下部分将详细讨论目前被广泛使用的两个扩展。

- 设备拆分：将一些设备拆分成较小的子设备的能力。
- 双精度支持：设备是否像支持浮点数据类型一样，也支持双精度数据类型。

设备拆分

EXT 扩展设备拆分(Bellows et al., 2010)提供了一个用于将 OpenCL 设备拆分成多个子设备的接口。例如，一个 AMD 的 6 核 Istanbulx86 CPU(如图 11.1A 所示)。默认情况下，我们把所有 6 个核视为一个 OpenCL 设备，如图 11.1B 所示。使用设备拆分，这 6 个核可以拆分成 6 个 OpenCL 设备，如图 11.1C 所示。每一个都可以支持一个或多个命令队列。因为这些命令队列是异步的并且各自在自己的线程中运行，因此可以用设备拆分来创建一个简单而强大的基于任务并行的多线程应用程序。到目前为止，设备拆分只支持类 CPU 设备，但这个功能将扩展到各个平台的所有设备，包括 GPU。下一步，我们使用一个例子来说明设备拆分的基本用法。

使用设备拆分的一个有趣的应用是用它来创建和分析并发运行时，类似于微软的并发运行时(Microsoft 2010)和英特尔的 TBB(Threading Building Blocks, Intel, 2011)。下面我们来分析一个例子，创建一个简单的 C++类，将任务分配到一组由设备拆分而得到的子设备上，从而支持并行 `for` 结构。虽然这个例子与 `ConcRT`、`TBB` 等在运行时生成函数有一些相似，但它充分表明一点：不需要费什么工夫，我们就能用 OpenCL 创建优于基本数据并行的、可移植、可扩展的并行应用程序。这个例子的目的是实现一个简单的 `Parallel` 类，用于提供 `parallelFor` 方法来将任务分配到 x86 的各个核上，每个任务对应一

个子设备。**Parallel** 类拥有如下公共接口：

```
class Parallel
{
public:
    Parallel();
    static unsigned int atomicAdd(
        unsigned int inc,
        volatile unsigned int *dest);
    bool parallelFor(int range, std::function<void (int i)> f);
};
```

我们以一个计算随机生成的数组中素数个数的应用为例来说明这个类的用途。我们使用 **OpenCL C++** 封装 API 来实现该应用，为了开启设备拆分扩展，需要定义 **USE_CL_DEVICE_FISSION** 宏并包含 **cl.hpp** 头文件(Gaster 2010)^①。

```
#define USL_CL_DLVIOL_FISSION 1
#include <CL/cl.hpp>
```

如前所述，我们在 **Parallel** 类中封装了一些简单的运行时的并行特性。程序中所有 **OpenCL** 状态都将在类定义的私有部分处理。默认的构造器将初始化 **OpenCL** 并将 CPU 设备拆分成单核的子设备。和标准的 **OpenCL** 初始化一样，我们首先需要查询平台，创建上下文(在本例中，使用 **CL_DEVICE_TYPE_CPU**)，最后检查设备列表(如图 11.1B 所示，这通常作为一个单独的步骤)：

```
std::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);

cl_context_properties properties[] =
{
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)(platforms[1]),
    0
};

context_ = cl::Context(CL_DEVICE_TYPE_CPU, properties);

std::vector<cl::Device> devices =
    context_.getInfo<CL_DEVICE_TYPES>();
```

在我们使用设备拆分之前，必须先确认设备是否支持该功能。

① 如程序清单 11.1 所示，我们也在完整的源代码文件中为 **OpenCL** 启用了 C++ 异常。

```

if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find(
    "cl_ext_device_fission") == std::string::npos) {

    std::cout << "Required that device support "
        << "cl_ext_device_extension"
        << std::endl;
    exit(-1);
}

```

指定一个OpenCL `cl::Device`，在本例中是`devices[0]`，用`createSubDevices`方法创建子设备：

```

cl_int createSubDevices(
    const cl_device_partition_property_ext * properties,
    VEC10R_CLASS<Device>* devices)

```

给定一组拆分属性(见表 11.1 的定义)，将创建一组子设备，子设备通过 `devices` 参数返回。

下面的代码将使用 `CL_DEVICE_PARTITION_EQUALLY_EXT` 宏来创建一组相同的子设备，每个子设备和核之间一一对应。

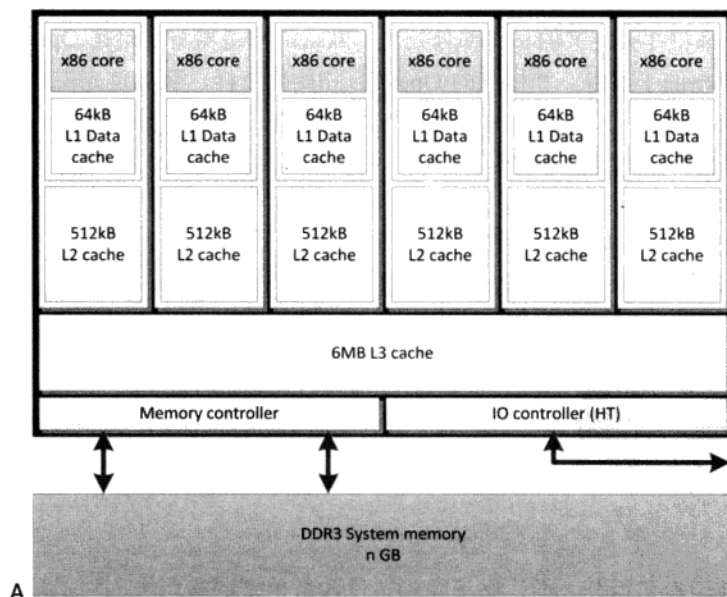


图 11.1

从 OpenCL 的角度来看进行了设备拆分和没有进行设备拆分的 CPU。(A)6 核 Phenom II；(B)6 核 Phenom II 的全部核作为单个 OpenCL 设备；(C)6 核 Phenom II 每个核做为一个 OpenCL 设备

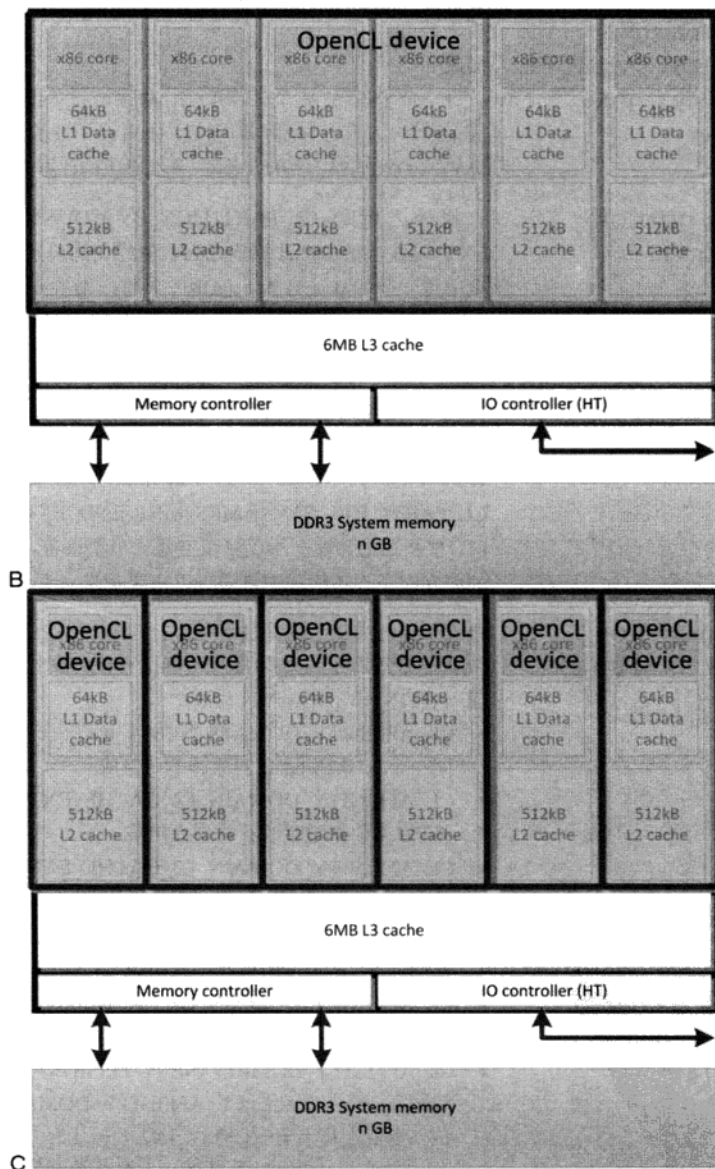


图 11.1 (续)

从 OpenCL 的角度来看进行了设备拆分和没有进行设备拆分的 CPU。(A)6 核 Phenom II；(B)6 核 Phenom II 的全部核作为单个 OpenCL 设备；(C)6 核 Phenom II 每个核作为一个 OpenCL 设备

表 11.1 子设备拆分属性

cl_device_partition_property_ext	描述
CL_DEVICE_PARTITION_EQUALLY_EXT	按每一个子设备 N 个计算单元, 将聚合设备尽可能多的拆分成子设备。 N 和该属性同时传递。如果 N 不能被 CL_DEVICE_MAX_COMPUTE_UNITS 整除, 那么余下的计算单元将不会使用
CL_DEVICE_PARTITION_BY_COUNTS_EXT	该属性之后是 CL_PARTITION_BY_COUNTS_LIST_END_EXT 属性, 指出在计算单元列表中终止的位置, 如果要为列表中每一个非 0 计数 M , 创建一个拥有 M 个计算单元的子设备, 那么 CL_PARTITION_BY_COUNTS_LIST_END_EXT 应定义为 0
CL_DEVICE_PARTITION_BY_NAMES_EXT	该属性之后是一个计算单元名字列表, 每一个列表以一个 CL_PARTITION_BY_NAMES_LIST_END_EXT 属性为开始, 计算单元名字是从 0 到计算单元数量减 1 的整数。CL_PARTITION_BY_NAMES_LIST_END_EXT 被定义成 -1 时, 使用该选择器一次仅可以创建一个子设备。每个计算单元的名字在子设备描述中最多可以出现一次
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN_EXT	将设备拆分成较小的聚合设备, 这些聚合设备包含一个或多个共享一部分某级 cache 的计算单元。伴随该属性的值来自如下 CL_AFFINITY_DOMAIN 列表: • CL_AFFINITY_DOMAIN_NUMA_EXT——子设备由共享同一个 NUMA band 的计算单元组成 • CL_AFFINITY_DOMAIN_L4_CACHE_EXT——子设备由共享 L4 数据缓存的计算单元组成 • CL_AFFINITY_DOMAIN_L3_CACHE_EXT——子设备由共享 L3 数据缓存的计算单元组成 • CL_AFFINITY_DOMAIN_L2_CACHE_EXT——子设备由共享 L2 数据缓存的计算单元组成 • CL_AFFINITY_DOMAIN_L1_CACHE_EXT——子设备由共享 L1 数据缓存的计算单元组成 • CL_AFFINITY_DOMAIN_NEXT_FISSIONABLE_EXT——根据下一个可分裂的 CL_AFFINITY_DOMAIN 来拆分设备。实现上将沿着 NUMA, L4, L3, L2, L1, 的层级顺序, 选择第一层级对可以进一步拆分的设备或子设备按照共享该层缓存的计算单元进行拆分。用户可以通过调用 clGetDeviceInfo (CL_DEVICE_PARTITION_STYLE_EXT) 来判断子设备上发生了什么

```

cl_device_partition_property_ext subDeviceProperties[] =
{
    CL_DEVICE_PARTITION_EQUALLY_EXT,
    1,
    CL_PROPERTIES_LIST_END_EXT,
    0
};

devices[0].createSubDevices(subDeviceProperties, &subDevices_);
if (subDevices_.size() <= 0) {
    std::cout << "Failed to allocate sub-devices" << std::endl;
    exit(-1);
}

```

下面的代码放在默认的构造函数中，通过遍历设备列表，为每一个设备创建一个命令队列：

```

for (auto i = subDevices_.begin(); i != subDevices_.end(); i++) {
    queues_.push_back(cl::CommandQueue(context_, *i));
}

```

图 11.2 展示了 CPU 每一个核匹配一个关联的命令队列。特别注意不同的命令队列提交命令是以并发的方式分别提交的。

实现 `atomicAdd` 非常简单，后面我们会提供实现方式。这里，我们的焦点主要集中于定义 `parallelFor` 上，它带有两个参数。第一个参数用来表示迭代空间的边界，第二个参数是一个将被应用于迭代空间每一个元素的函数对象。第二个参数是一个原始的 C++ 函数对象^①，但是 OpenCL 的 `clEnqueueNDRangeKernel` 操作仅对 OpenCL C kernel 有效，因此我们需要一个另外的方法来支持原始的 C++ 函数。非常幸运，OpenCL 恰好有一个这样的函数，`clEnqueueNativeKernel`，它可以入队一个 C 函数。通过精心处理，它也可以用于 C++ 函数。原始的 kernel 已经在第 5 章描述过了。

使用 `clEnqueueNativeKernel`，定义 `parallelFor` 非常简单^②：

```

bool parallelFor(int range, std::function<void(int i)> f)
{
    std::vector<cl::Event> events;

    size_t args[2];

    args[0] = reinterpret_cast<size_t>(&f);
}

```

① “原始的”(native)代表这个函数是在主机端编译，而不是使用 OpenCL 的 C 编译器。

② 注意，如前面所述，我们使用的是 OpenCL 的 C++ 封装 API(而不是 C API 中的 `clEnqueueNativeKernel`)。

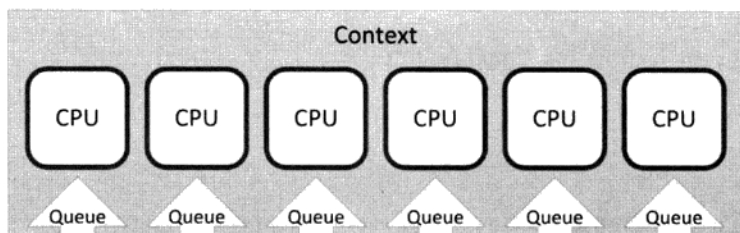


图 11.2

OpenCL 子设备和关联的命令队列

```

int index = 0;
for (int x = 0; x < range; x++) {
    int numQueues =
        range - x > queues_.size() ? queues_.size() : range - x;

    cl::Event event;
    while(numQueues > 0) {
        args[1] = static_cast<size_t>(index++);
        queues_[numQueues-1].enqueueNativeKernel(
            funcWrapper,
            std::make_pair(
                static_cast<void*>(args),
                sizeof(size_t)*2),
            NULL,
            NULL,
            NULL,
            &event);

        events.push_back(event);

        numQueues--;
        x++;
    }
    cl::Event::waitforEvents(events);
}
return true;
}

```

函数的第一部分是设置参数，在这里参数是我们实际要运行的函数。之所以需要这样，是因为 OpenCL 需要一个 C 函数，但这里却是一个 C++ 函数对象(通常称为一个 functor)。因此，我们提供一个叫 funcWrapper 的封装函数并把一个指向 functor 的函数指针作为参数传递给它。当执行的时候，封装函数再解封装 functor，再调用相应的函数对象。函数的主体是在 0 到 range 的范围内循环执行 functor，在内层循环则是把 0 到 range 的一些子集映射到实际的子设备上。需要说明的是这些提交都是异步的，每一个调用都会在提交结束后返回一个事件，我们需要在每组提交的最后进行等待。一个更优的版本应该是我们在大量的提交之后才进行等待，控制好使用 queue.flush() 的粒度，等等。我们把这个作为一个练习留给读者。

最后，我们把这些都放在代码清单 11.1 里面，包括 main 函数在内，它完整展示了我们的素数检查实例的实现过程。一个重要且有趣的地方是这个例子没有使用 OpenCL C 设备端代码，它只使用了 OpenCL 运行时作为一个轻量的线程库。

代码清单 11.1

```
// Lnable OpenCL C++ exceptions
#define __CL_LNABLL_LXCLPTIONS

// Lnable Device Lission
#define USL_CL_DLVLICL_LISSION 1

#include <CL/cl.hpp>

#include <cstring>
#include <cstdlib>
#include <cassert>
#include <cmath>
#include <iostream>
#include <fstream>
#include <vector>

#include <functional>
#include <memory>

class Parallel
{
private:
    cl::Context context_;
    std::vector<cl::Device> subDevices_;
    cl::CommandQueue queue_;
    std::vector<cl::CommandQueue> queues_;

    static void CL_CALLBACK funcWrapper(void * a)
    {
        size_t * args = static_cast<size_t*>(a);

        std::function<void (int i)> * f =
            reinterpret_cast<std::function<void (int i)>*>(args[0]);
```

```

        (*f)(static_cast<int>(args[1]));
    }
public:
    Parallel()
    {
        std::vector<cl::Platform> platforms;
        cl::Platform::get(&platforms);

        cl_context_properties properties[] =
        {
            CL_CONTEXT_PLATFORM,
            (cl_context_properties)(platforms[1])(),
            0
        };

        context_ = cl::Context(CL_DEVICE_TYPE_CPU, properties);

        std::vector<cl::Device> devices =
            context_.getInfo<CL_DEVICE_TYPES>();

        // Check that device fission is supported
        if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find(
            "cl_ext_device_fission") == std::string::npos) {
            std::cout << "Required that device support "
                << "cl_ext_device_fission"
                << std::endl;
            exit(-1);
        }

        cl_device_partition_property_ext subDeviceProperties[] =
        {
            CL_DEVICE_PARTITION_LOCALLY_L1,
            1,
            CL_DEVICE_PARTITION_LIST_END,
            0
        };

        devices[0].createSubDevices(subDeviceProperties, &subDevices_);
        if (subDevices_.size() <= 0) {
            std::cout << "Failed to allocate sub-devices" << std::endl;
            exit(-1);
        }

        for (auto i = subDevices_.begin(); i != subDevices_.end(); i++) {
            queues_.push_back(cl::CommandQueue(context_, *i));
        }

        std::cout << "Number of sub-devices "
            << subDevices_.size()
            << std::endl;

        static unsigned int atomicAdd(
            unsigned int inc,
            volatile unsigned int *dest)
    {

```

```

#if defined(_MSC_VLR)
    return (unsigned int)_InterlockedLxchangeAdd(
        (volatile long*)dest,
        (long)inc);
#else
    return __sync_fetch_and_add(dest, inc);
#endif
}

bool parallelFor(int range, std::function<void (int i)> f)
{
    std::vector<cl::Lvent> events;

    size_t args[2];

    args[0] = reinterpret_cast<size_t>(&f);

    int index = 0;
    for (int x = 0; x < range; x++) {

        int numQueues =
            range - x > queues_.size() ? queues_.size() : range - x;

        cl::Lvent event;
        while(numQueues > 0) {

            args[1] = static_cast<size_t>(index++);

            queues_[numQueues-1].enqueueNativeKernel(
                funcWrapper,
                std::make_pair(
                    static_cast<void*>(args),
                    sizeof(size_t)*2),
                NULL,
                NULL,
                NULL,
                &event);

            events.push_back(event);

            numQueues--;
            x++;
        }

        cl::Lvent::waitForLvents(events);
    }

    return true;
}

};

const unsigned int numNumbers = 1024;

```

```

int
main(int argc, char** argv)
{
    volatile unsigned int numPrimes = 0;
    int * numbers = new int[numNumbers];
    ::srand(2009);

    // Random initialize
    for (size_t i = 0; i < numNumbers; ++i) {
        numbers[i] = ::rand();
    }

    try {
        Parallel parallel;

        parallel.parallelFor(numNumbers, [numbers, &numPrimes]
            (int x) {

                auto isPrime = [] (unsigned int n) -> bool {
                    if (n == 1 || n == 2) {
                        return true;
                    }
                    if (n % 2 == 0) {
                        return false;
                    }

                    for (unsigned int odd = 3;
                        odd <= static_cast<unsigned int>(
                            sqrt(static_cast<float>(n)));
                        odd += 2) {

                        if (n % odd == 0) {
                            return false;
                        }
                    }

                    return true;
                };

                if (isPrime(numbers[x])) {
                    Parallel::atomicAdd(1, &numPrimes);
                }
            });

        std::cout << "Number of primes found = " << numPrimes <<
            std::endl;
    }

    catch (cl::Error err) {
        std::cerr
            << "ERROR: "
            << err.what()
            << "("
            << err.err()
            << ")"
    }
}

```

```
        << std::endl;
    return LX11_FAILURL;
}
delete[] numbers;
return LX11_SUCCCLSS;
}
```

双精度

浮点格式允许程序员使用非常大或者是非常小的非整型数值。对于很多应用来说，单精度浮点数不能提供足够的表示范围或者是精度。很多应用(尤其是在科学和工程方面)需要部分或全部使用双精度来进行计算。OpenCL 不需要特别的计算设备来支持双精度。对一些需要双精度的关键应用，OpenCL 提供了一个 `cl_khr_fp64` 扩展选项。这个可以在任何需要使用双精度的 OpenCL C 程序中用下面的制导语句来开启：

```
#pragma cl_khr_fp64 : enable
```

一旦开启，将支持如下双精度数据类型的访问：

类 型	说 明
<code>double</code>	双精度浮点数
<code>double1</code>	长度 2 的 <code>double</code> 向量
<code>double2</code>	长度 3 的 <code>double</code> 向量
<code>double4</code>	长度 4 的 <code>double</code> 向量
<code>double8</code>	长度 8 的 <code>double</code> 向量
<code>double16</code>	长度 16 的 <code>double</code> 向量

双精度数据的存储格式遵循 IEEE 754 标准。

这些数据类型和单精度浮点数据类型是一一对应的。在 AMD CPU 上，当 OpenCL 设备是 `CL_DEVICE_TYPE_CPU` 时，向量类型将直接映射到 SSE 和 AVX 打包寄存器 (packed register)。如果一个向量类型长度超出了基本硬件向量长度，那么将使用多个硬件向量来存储该向量。所有在 OpenCL 规范中定义的与 `float` 类型有关的数据转换规则都适用于 `double` 类型。与 `float` 类型相关的内建数学函数也可相应地用于 `double`，`double2`，

double3, double4, double8 和 double16 等数据类型。

作为一个简单的例子, 代码清单 11.2 展示了 OpenCL C 代码实现的一个双精度的矩阵乘法操作。注意, OpenCL 源文件的第一行开启了相应的扩展。

代码清单11.2

```
#pragma OPENCL_EXTENSION cl_khr_fp64 : enable
#include "matrixmul.h"

#define AS(i, j) As[(j) * BLOCK_SIZEL + (i)]
#define BS(i, j) Bs[(j) * BLOCK_SIZEL + (i)]

__kernel void
matrixMul(
    __global double* C,
    __global double* A,
    __global double* B,
    int wA,
    int wB,
    __local double* As,
    __local double* Bs)
{
    // Block index
    int bx = get_group_id(0);
    int by = get_group_id(1);

    int tx = get_local_id(0);
    int ty = get_local_id(1);

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZEL * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZEL;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZEL * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZEL * wB;

    // Csub is used to store the element of the block sub-matrix
    // that is computed by the thread
    double Csub = 0;

    // Loop over all the sub-matrices of A and B
    // required to compute the block sub-matrix
    for (int a = aBegin; b = bBegin;
        a <= aEnd;
```

```

        a += aStep, b += bStep) {

    // Load the matrices from device memory
    // to local memory; each work-item loads
    // one element of each matrix
    AS(ty, tx) = A[a + wA * ty + tx];
    BS(ty, tx) = B[b + wB * ty + tx];

    // Synchronize to make sure the matrices are loaded
    barrier(CLK_LOCAL_MEM_FENCE);

    // Multiply the two matrices together;
    // each work-item computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += AS(ty, k) * BS(k, tx);

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    barrier(CLK_LOCAL_MEM_FENCE);
}
// Write the block sub-matrix to device memory;
// each work-item writes one element
int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
C[c + wB * ty + tx] = Csub;
}

```

主机端的 OpenCL 程序，使用的是 OpenCL C++ 封装 API，如代码清单 11.3 所示，共享头文件 `matrixmul.h` 如代码清单 11.4 所示。这个例子非常简单，但有两点是值得在以后进行详细讨论。首先，为了避免不可预料的运行时错误，程序必须检查设备是否支持 `cl_khr_fp64` 扩展。通过如下代码来实现：

```

if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find("cl_khr_fp64")
==
    std::string::npos) {

    std::cout << "Required that device support cl_khr_fp64" << std::
endl;
    exit(-1);
}

```

在这种情况下，如果设备不支持 `cl_khr_fp64` 扩展，程序将直接退出。更健壮的解决方案是返回到主机端，让主机端来计算出相应的结果。其次，这个例子在命令队列中使用程序剖析 API 来执行获得矩阵相乘的时间信息。

代码清单11.3

```

#include <CL/cl.hpp> // C++ Wrapper API (no need to include cl.h)

#include <cstring>
#include <cstdlib>
#include <cassert>
#include <cmath>
#include <iostream>
#include <fstream>
#include <vector>
#include <matrixmul.h>

template <size_t X, size_t Y>
class Matrix
{
private:
    cl_double data_[X * Y];

public:
    template <typename Functor>
    Matrix(Functor init) { init(data_, X, Y); }
    Matrix() { ::memset(data_, '\0', X * Y * sizeof(cl_double)); }

    Matrix<X,Y> operator += (const Matrix<X,Y>& rhs)
    {
        for (size_t i = 0; i < (X * Y); ++i) {
            data_[i] += rhs.data_[i];
        }
        return *this;
    }

    Matrix<X,Y> operator - (const Matrix<X,Y>& rhs)
    {
        Matrix<X,Y> result = *this;
        return result -= rhs;
    }

    bool operator == (const Matrix<X,Y>& rhs)
    {
        for (size_t i = 0; i < (X * Y); ++i) {
            if (data_[i] != rhs.data_[i]) {
                return false;
            }
        }
        return true;
    }

    bool operator != (const Matrix<X,Y>& rhs)
    {
        return !(*this == rhs);
    }
};

```

```

bool compareL2fe(const Matrix<X,Y>& reference, cl_double
epsilon)
{
    cl_double error = 0.0f;
    cl_double ref = 0.0f;

    for (size_t i = 0; i < (X * Y); ++i) {
        cl_double diff = reference.data_[i] - data_[i];
        error += diff * diff;
        ref += reference.data_[i] * reference.data_[i];
    }

    cl_double normRef = ::sqrt((double) ref);
    if (::fabs((double) ref) < 1e-7f) {
        return false;
    }
    cl_double normError = ::sqrtf((double) error);
    error = normError / normRef;

    return error < epsilon;
}

void printOn(std::ostream& out)
{
    for (size_t y = 0; y < Y; ++y) {
        for (size_t x = 0; x < X; ++x) {
            out << data_[y * X + x] << " ";
        }
        out << "\n";
    }
}

cl_double* data() { return data_; }
size_t size() const { return X * Y * sizeof(cl_double); }
};

static void
randomInit(cl_double* data, size_t width, size_t height)
{
    for (size_t i = 0; i < width * height; ++i) {
        data[i] = ::rand() / (double) RAND_MAX;
    }
}

static void
rop(cl_double*, size_t, size_t)
{
}

static void
computeGolg(
    cl_double* C, const cl_double* A, const cl_double* B,
    cl_uint hA, cl_uint wA, cl_uint wB)
{
    for (cl_uint i = 0; i < hA; ++i) {

```

```

        for (cl_uint j = 0; j < wB; ++j) {
            cl_double sum = 0;
            for (cl_uint k = 0; k < wA; ++k) {
                cl_double a = A[i * wA + k];
                cl_double b = B[k * wB + j];
                sum += a * b;
            }
            C[i * wB + j] = (cl_double)sum;
        }
    }

int
main(int argc, char** argv)
{
    cl_device_type devType = CL_DEVICE_TYPE_CPU;
    cl_int err, err1, err2, err3;
    ::srand(2009);

    Matrix<WA, HA> A(&randomInit);
    Matrix<WB, HB> B(&randomInit);

    // Initialize the OpenCL runtime.
    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    cl_context_properties cprops[] = {
        CL_CONTEXT_PLATFORM, (cl_context_properties)platforms[1]
        (), 0 };

    cl::Context context(devType, cprops);
    std::vector<cl::Device> devices = context.getInfo<CL_
    CONTEXT_DEVICES>();
    if (devices.size() == 0) {
        std::cerr << "Device not available\n";
        return LXII_FAILURL;
    }

    // Check that double is supported
    if (devices[0].getInfo<CL_DEVICE_EXTENSIONS>().find("cl_khr_
    fp64") ==
        std::string::npos) {
        std::cout << "Required that device support cl_khr_fp64"
        << std::endl;
        exit(-1);
    }

    std::ifstream file("matrixmul_kernels.cl");
    if (!file.is_open()) {
        std::cerr << "We couldn't load CL source code\n";
        return LXII_FAILURL;
    }

    std::string prog(

```

```

        std::istreambuf_iterator<char>(file),
        (std::istreambuf_iterator<char>()));

cl::Program::Sources source{
    1,
    std::make_pair(prog.c_str(), prog.length()+1));

cl::Program program(context, source);

err = program.build(devices, "-l.");
if (err != CL_SUCCESS) {
    std::string str;
    str = program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]);
    std::cout << "Program Info: " << str;
    exit(1);
}

cl::Kernel matrixMul(program, "matrixMul", &err);
if (err != CL_SUCCESS) {
    std::cerr << "Could not create kernel \"matrixMul\"\n";
    return EXIT_FAILURE;
}

cl::Buffer in0(context, CL_MLM_USL_HOST_PIR, A.size(), A.data(), &err1);
cl::Buffer in1(context, CL_MLM_USL_HOST_PIR, B.size(), B.data(), &err2);
cl::Buffer out(context, CL_MLM_ALLOC_HOST_PIR,
    WC * HC * sizeof(cl_double), NULL, &err3);
if (err1 != CL_SUCCESS || err2 != CL_SUCCESS || err3 != CL_SUCCESS) {
    std::cerr << "Could not create memory objects\n";
    return EXIT_FAILURE;
}

err = matrixMul.setArg(0, out);
err |= matrixMul.setArg(1, in0);
err |= matrixMul.setArg(2, in1);
err |= matrixMul.setArg(3, WA);
err |= matrixMul.setArg(4, WB);
const size_t localSize = sizeof(double[BLOCK_SIZE][BLOCK_SIZE]);
err |= matrixMul.setArg(5, cl::__local(localSize));
err |= matrixMul.setArg(6, cl::__local(localSize));
if (err != CL_SUCCESS) {
    std::cerr << "Could not set matrixMul's args\n";
    return EXIT_FAILURE;
}

cl_command_queue_properties properties = CL_QUEUE_PROFILING_ENABLE;
cl::CommandQueue queue(context, devices[0], properties, &err);
if (err != CL_SUCCESS) {
    std::cerr << "Could not create the command queue\n";
    return EXIT_FAILURE;
}

```

```

    }

    cl::Event event;
    err1 = queue.enqueueNDRangeKernel(
        matrixMul, cl::NullRange,
        cl::NDRange(WC, HC),
        cl::NDRange(BLOCK_SIZE, BLOCK_SIZE),
        NULL,
        &event);

    Matrix<WC, HC> C(&nop);
    err2 = queue.enqueueReadBuffer(out, CL_TRUE, 0, C.size(), C.data());
    if (err1 != CL_SUCCESS || err2 != CL_SUCCESS) {
        std::cerr << "matrixMul execution failed\n";
        return EXIT_FAILURE;
    }

    Matrix<WC, HC> reference(&nop);
    computeGold(reference.data(), A.data(), B.data(), HA, WA, WB);

    if (!C.compareL2fe(reference, 1e-6f)) {
        Matrix<WC, HC> difference = reference - C;
        difference.printOn(std::cout);
        std::cout << "FAILED\n";
        return EXIT_FAILURE;
    }

    if ((properties & CL_QUEUE_PROFILING_ENABLE) != 0) {
        cl_ulong start = event.getProfilingInfo<CL_PROFILING_COMMAND_START>();
        cl_ulong end = event.getProfilingInfo<CL_PROFILING_COMMAND_END>();
        std::cout << "Elapsed time: "
                    << (double)(end - start) / 1e6
                    << "ms\n";
    }

    std::cout << "PASS!\n";
    return EXIT_SUCCESS;
}

```

代码清单11.4

```

// Thread block size
#define BLOCK_SIZE 16

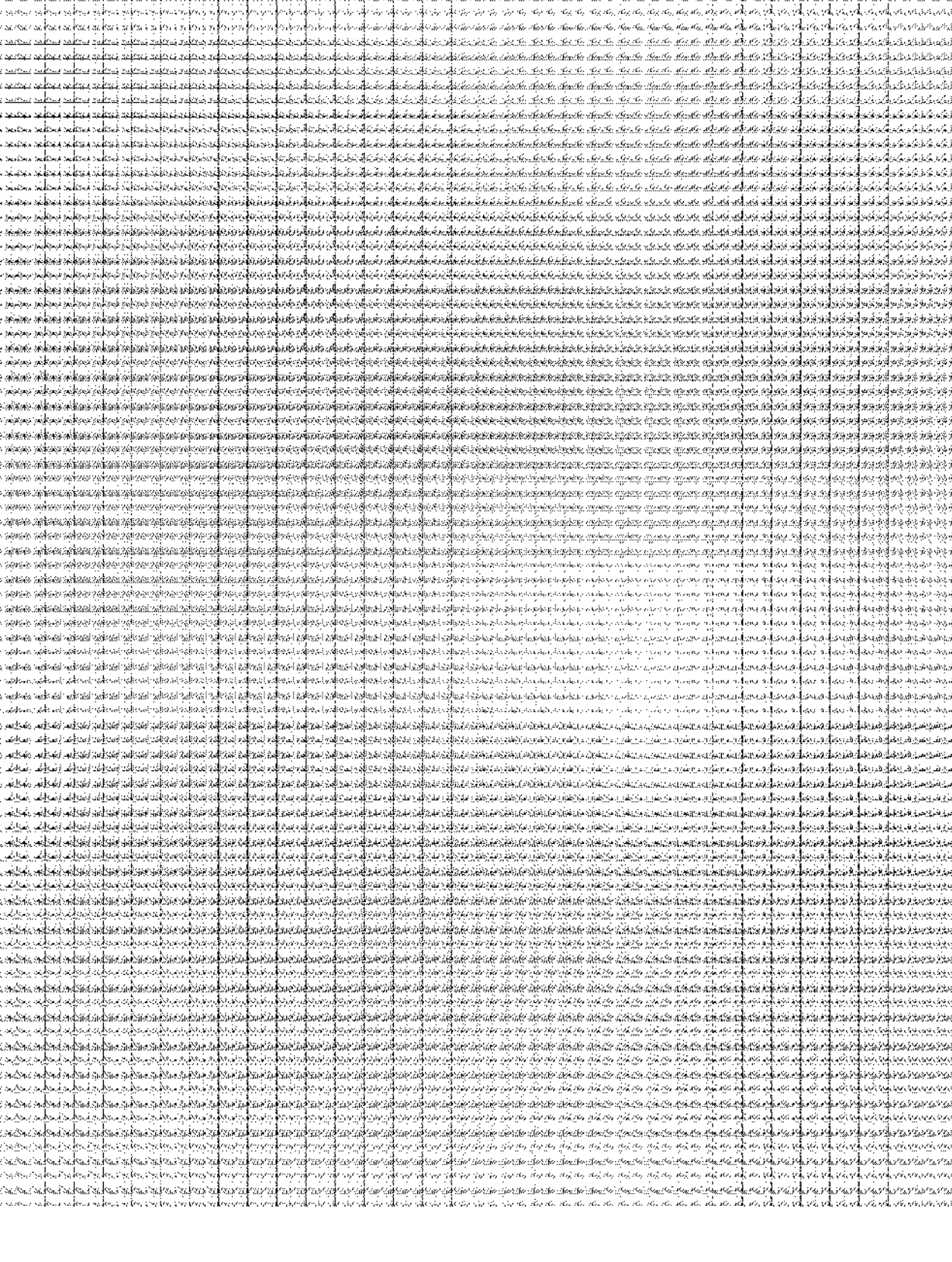
// Matrix dimensions
// (chosen as multiples of the thread block size for simplicity)
#define WA (3 * BLOCK_SIZE) // Matrix A width
#define HA (5 * BLOCK_SIZE) // Matrix A height
#define WB (8 * BLOCK_SIZE) // Matrix B width

```

```
#define HB WA // Matrix B height  
#define WC WB // Matrix C width  
#define HC HA // Matrix C height
```

参考文献

- Bellows, G., Gaster, B. R., Munshi, A., Ollmann, I., Rosenberg, O., & Watt, B. (2010). *Device Fission*. Khronos OpenCL Working Group.
- Gaster, B. R. (2010). *The OpenCL C++ Wrapper API, Version 1.1*. Khronos OpenCL Working Group.
- Intel. (2011). *Threading Building Blocks 3.0 for Open Source*. <http://threadingbuildingblocks.org>.
- Microsoft. (2010). *Concurrency Runtime: Visual Studio 2010*. <http://msdn.microsoft.com/enus/library/dd504870.aspx>.



OpenCL 的性能剖析和调试

引言

我们使用 OpenCL 编程的目标不只是编写孤立的高性能的核心模块，而是加速并行应用的运行。前面几章讨论了如何运用架构的特性来优化 OpenCL kernel 的性能。在本章中，我们将讨论设备上的计算 kernel 和主机如何进行交互。我们需要从整体上测量和研究应用程序，以此来理解性能瓶颈。

OpenCL 应用包含一些 kernel 和大量主机与设备间的输入/输出(IO)操作。剖析这样的应用程序可以帮助我们回答与应用相关的以下问题，从而提升应用的性能。

- 应用中存在多个 kernel 时，应该优化哪个 kernel？
- kernel 在命令队列中的等待时间和真正执行的时间分别是多少？
- 应用中，执行时间与 OpenCL 环境初始化和编译 kernel 时间的比例是多少？
- 应用中，主机和设备间 IO 的时间与计算时间的比例是多少？

本章前两小节介绍 OpenCL API 提供的剖析应用性能的基本特性，如何使用操作系统的 API 对代码段进行计时。在之后的两小节中，将介绍 AMD 提供的帮助分析应用程序的两个工具。

- AMD Accelerated Parallel Processing (APP) Profiler，是一个性能分析工具，在 OpenCL 程序执行期间收集 OpenCL 运行时和 AMD Radeon GPU 的数据
- AMD Accelerated Parallel Processing (APP) KernelAnalyzer，是一个静态分析工具，可以为 AMD Radeon GPU 编译、分析和反汇编 OpenCL kernel 代码。

通过这 4 种不同的剖析方法，可以帮助开发人员迅速判断一个程序的执行是否符合预期，在本章最后所述的调试特性配合下，可以极大改进开发流程。

在本章结尾，我们将讨论 OpenCL 代码的调试。调试并行程序比传统的串行代码要难得多，原因是存在一些难于检测和重现的 bug，比如竞态条件。我们将简要介绍一个 OpenCL 的调试和分析工具，gDEBugger。之后，简要讲解 AMD 的 printf 扩展，它可用于查看 kernel 数据。

基于事件的剖析

OpenCL 对提交到命令队列的命令使用 64 位计时，比如，clEnqueueNDRangeKernel() 等 clEnqueueXX() 命令。通常来说，命令进入队列是异步过程。如前面几章节所述，开发人员利用事件来跟踪命令的状态和确保依赖关系。事件提供了记录命令历史的途径：对应的命令什么时候进入队列？什么时候提交到设备？什么时候开始执行和结束的？通过调用 clGetEventProfilingInfo API 可以获得事件的信息，查询计时数据：

开启剖析

基于事件的 OpenCL 程序剖析，需要显式针对每个命令队列开启，即在命令队列创建的时候，设置 CL_QUEUE_PROFILING_ENABLE 标志。

一旦命令队列已经创建完毕，便无法再开启或关闭事件剖析功能。

```
cl_int clGetEventProfilingInfo (
    cl_event event,
    cl_profiling_info param_name,
    size_t param_value_size,
    void *param_value,
    size_t *param_value_size_ret)
```

第 1 个参数 event 代表待查询的事件，第 2 个参数是描述查询的枚举值。正确的枚举值如下表所示。

CL_PROFILING	返回类型	返回的 param_value 的含义
CL_PROFILING_COMMAND_QUEUED	cl_ulong	64 位, 当前设备的时间计数器值 (纳秒), 表示 event 代表的命令在主机上进入命令队列的时刻
CL_PROFILING_COMMAND_SUBMIT	cl_ulong	64 位, 当前设备的时间计数器值 (纳秒), 表示 event 代表的命令从主机提交到设备的时刻
CL_PROFILING_COMMAND_START	cl_ulong	64 位, 当前设备的时间计数器值 (纳秒), 表示 event 代表的命令在设备上开始执行的时刻
CL_PROFILING_COMMAND_END	cl_ulong	64 位, 当前设备的时间计数器值 (纳秒), 表示 event 代表的命令在设备上结束执行的时刻

如前所述, OpenCL 的命令队列是异步执行的, 即命令进入队列后, 函数会立刻返回。为此, 调用获取 OpenCL 事件时间戳的 `clGetEventProfilingInfo` 之前, 需要调用 `clFinish` 函数或者其他的事件同步函数, 以确保事件相关的任务已经执行完毕。

事件剖析信息默认不是打开的, 需要为每个命令队列进行开启。为了开启一个命令队列的事件剖析功能, 必须在创建时传入如下参数:

```
CL_QUEUE_PROFILING_ENABLE
```

一旦命令队列已经创建完毕, 将无法再开启或关闭事件剖析功能。

下面是一个通过事件剖析 kernel 执行的简单例子:

```
// Sample Code that can be used for timing kernel execution duration
// Using different parameters for cl_profiling_info allows us to
// measure the wait time
cl_event timing_event;
cl_int err_code;
```

```

    //! We are timing the clEnqueueNDRangeKernel call and timing information will be
    //! stored in timing_event
    err_code = clEnqueueNDRangeKernel (
        command_queue, kernel,
        work_dim,
        global_work_offset,
        global_work_size,
        local_work_size,
        0,
        NULL,
        &timing_event);
    clFinish(command_queue);

    cl_ulong starttime;
    cl_ulong endtime;
    err_code = clGetEventProfilingInfo(
        timing_event,
        CL_PROFILING_COMMAND_START,
        sizeof(cl_ulong),
        &starttime, NULL);

    kerneltimer = clGetEventProfilingInfo(
        timing_event,
        CL_PROFILING_COMMAND_END,
        sizeof(cl_ulong),
        &endtime,
        NULL);

    unsigned long elapsed = (unsigned long)(endtime - starttime);

    printf("Kernel Lxecution\t%d ns\n", elapsed);

```

AMD APP Profiler

AMD APP(Accelerated Parallel Processing)Profiler 是 AMD 公司提供的—个性能分析工具，在 OpenCL 应用的执行期间，收集 OpenCL 运行时和 AMD Radeon GPU 的数据。利用这些信息，我们可以发现应用中的性能瓶颈，从而找到在 AMD 平台上优化程序性能的方法和路径。后面，我们把 AMD APP Profiler 简称为 Profiler。

Profiler 已经包含在 AMD APP SDK 的安装包中。也可以使用独立的 Profiler 安装包，从 AMD 开发者网站下载：<http://developer.amd.com>。

在本小节中，我们将介绍 Profiler 2.2 版本的主要功能，此版本已经包含在 AMD APP SDK 2.4 中。因为 Profiler 还处在快速发展中，所以请查阅 Profiler 的文档以了解最新特性。

Profiler 支持两种使用方式。

1. Microsoft Visual Studio 2008 或者 2010 的插件方式。
2. 命令行方式，支持 Windows 平台和 Linux 平台。

我们推荐使用 Visual Studio 插件的形式，这样可以以多种方式可视化和分析结果。在 Visual Studio 中使用 Profiler 插件非常简单，加载一个解决方案，选择 C/C++ 项目作为启动(Start-up)项目。之后，点击 APP Profiler Session Explorer 面板的 Collect Application Trace 或者 Collect GPU Performance Counters 按钮。默认情况下，APP Profiler Session Explorer 面板和 Visual Studio Solution Explorer 面板位于同一个窗口面板上。开发人员不需要修改代码和项目配置，就可以剖析和分析应用。Profiler 会查询 Visual Studio 中运行应用所必要的项目设置信息。应用运行结束后，Profiler 会生成并显示相关的分析信息。

命令行方式适合用来收集没有源代码应用的数据。Profiler 的输出文本可以直接进行分析，也可以载入到 Visual Studio 的插件中进行可视化分析。

Profiler 支持两种工作方式：收集 OpenCL 程序的轨迹信息，或者收集 OpenCL kernel 的 GPU 性能计数器信息。

收集 OpenCL 程序轨迹

OpenCL 程序的轨迹信息(也即跟踪信息)会列出程序中所有调用的 OpenCL API 函数。对每一个 API 调用，Profiler 会记录输入的参数和输出结果。此外，Profiler 也会记录主机端代码的 CPU 时间戳和取自 OpenCL 运行时的设备的时间戳。输出数据会保存成基于文本的 AMD 自定义文件格式，称之为 Application Trace Profile 文件，请浏览文档以获取详细说明。这种方式特别适合帮助开发人员理解一个复杂应用程序的高层结构。

根据 OpenCL 应用程序的轨迹数据，我们可以执行以下任务。

- 在 Timeline 视图中查看程序的高层结构。我们可以了解程序中 OpenCL 上下文和创建的命令队列的数量，以及它们的相互关系。应用程序代码，Kernel 执行和数据传输操作都以时间线的方式进行展示。
- 判断程序是受限于 kernel 执行，还是数据传输。通过 Summary Pages 视图，查询应用最耗时的前 10 个 kernel 和数据传输，也可以查询 API 的热点(调用频率最高或者最耗时)。
- 通过 API Trace 视图，查看和调试所有 API 调用的输入参数和输出结果。

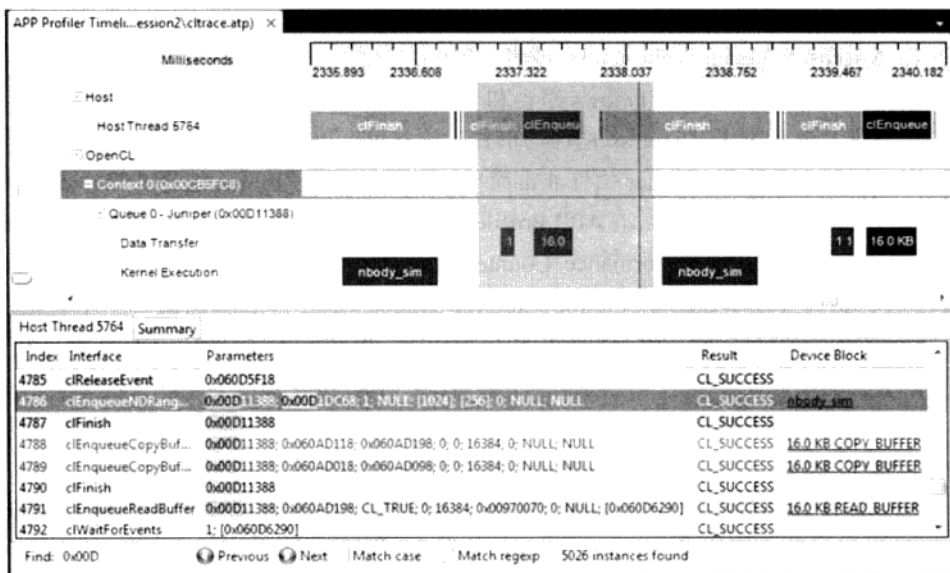


图 12.1

Microsoft Visual Studio 2010 中 AMD APP Profiler 的 Timeline 和 API Trace 视图

Timeline 视图(图 12.1)可以可视化方式展示了应用程序的执行。窗口的上方是时间轴(毫秒),表示应用消耗的全部时间,从第一个 OpenCL 调用开始时进行计时,在最后一个 OpenCL 调用时结束。在时间轴的下方是每个调用 OpenCL 函数的主机(OS)线程。对于每个主机线程,都沿着时间轴,画出所调用的每个 OpenCL API 的开始时间和结束时间。在主机线程下方是 OpenCL 树,表示此应用中创建的所有上下文和队列。我们可以在 Timeline 视图中进行多种操作(比如:缩放,拖动,收起或展开)和选择一段感兴趣的区域等。从 Timeline 视图中,我们可以跳转到 API Trace 视图对对应的 API 调用,也可以反过来,从 API Trace 视图对应到 Timeline 视图。

Timeline 视图对于调试 OpenCL 程序非常有用,体现在以下三个方面。

- 可以容易的判断程序的高层结构是否正确。通过查看时间线,可以验证创建的队列和上下文的数量是否符合预期。
- 可以检查程序中的同步是否执行正确。例如,如果 kernel A 依赖于一个 buffer 操作和 kernel B 的输出,那么时间轴上 kernel A 应该出现在 buffer 操作和 kernel B 完成的后面。这种同步错误是很难被传统的调试技术发现的。

- 最后，可以查看程序对硬件的使用是否高效。比如，在时间线上，独立的 kernel 执行和数据传输操作应该并行运行。

Summary 视图

Summary 视图展示了 OpenCL 应用程序的多种统计结果，可以大致的确定应用的性能瓶颈。此外，也提供了许多有用的信息，比如每个上下文创建的 buffer 和 image 数量，最耗时的 kernel 调用，等等。

Summary 视图提供以下独立的页面。

- API Summary 页：此页显示程序中所有 OpenCL API 调用的统计信息，可以用来定位 API 的热点。
- Context Summary 页：此页显示了每个上下文的所有 kernel 分发和数据传输操作的统计信息，在每个上下文中创建的 buffer 和 image 数量，如图 12.2 所示。
- Top 10 Data Transfer Summary 页：此页显示了前 10 个最耗时的单个数据传输操作。
- Top 10 Kernel Summary 页：此页显示了前 10 个最耗时的单个 kernel 执行操作。

通过这些汇总页面，可以让我们判断一个应用是受限于 kernel 执行还是数据传输 (Context Summary 页)。如果程序受限于 kernel 执行，那么我们可以找出哪个设备是性能的瓶颈。在 Kernel Summary 页，我们可以找出总运行时间最长的 kernel。或者，通过 Top 10 Kernel Summary 页，找到最长运行时间的单个 kernel 实例。如果这个性能瓶颈是运行在 GPU 设备上的 kernel，那么可以使用 GPU 的性能计数器以查找 kernel 内部的瓶颈。我们将在本章的后面介绍 GPU 性能计数器视图。

Host Thread 0x24 | Host Thread 0x26 | Host Thread 0x28 | Summary

Context Summary

Context ID	# of Buffers	# of Images	# of Kernel Dispatches (CPU Device)	Total Kernel Dispatches (CPU Device)	# of Kernel Dispatches (GPU Device)	Total Kernel Dispatches (GPU Device)	Total Kernel Time (ms)	# of Memory Transfers	Total Memory Time (ms)	# of Read	Total Read Time (ms)	# of Write	Total Write Time (ms)	# of Copy	Total Copy Time (ms)	# of Copy	Total Copy Time (ms)
0	2	0	1	67.43130	1	15.86324	2	1.21196	0	0	0	2	1.21196	542.00	0	0	0
1	2	0	1	16.13000	NA	NA	1	0.13787	0	0	0	1	0.13787	256.00	0	0	0
2	2	0	NA	NA	1	1.38867	1	1.71825	0	0	0	1	1.71825	236.00	0	0	0
3	2	0	1	16.12796	NA	NA	1	0.13444	0	0	0	1	0.13444	256.00	0	0	0
4	2	0	NA	NA	1	1.38915	1	1.66952	0	0	0	1	1.66952	256.00	0	0	0
Total	10	0	3	115.86428	1	18.64731	6	4.87205	0	0	0	6	4.87205	1.50	0	0	0

图 12.2

Microsoft Visual Studio 2010 中 AMD APP Profiler 的 Context Summary 页视图

如果应用是受限于数据传输,那么可以通过 Context Summary 页来确定最耗时的数据传输类型(读、写、复制或者映射)。我们可以探索是否有必要修改算法,以最小化此类型的数据传输。借助于 Timeline 视图,我们可以查看数据传输是否是最高效的方式,即和 kernel 执行同时运行。

API Trace 视图

API Trace 视图(图 12.1)列出了应用程序中所有的 OpenCL API 调用。

每个执行过至少一次 OpenCL 调用的主机线程,都会有一个单独的标签页。这个标签页包含此线程所有 API 调用的列表。此列表显示每个调用的索引号(代表了执行顺序)、API 函数的名称,以分号分隔的传递参数队列以及函数的返回值。在显示参数时,Profiler 会尽量多给出函数的传入或返回数据的信息,如,试图取回指针所指的值,解码枚举变量的值,等等。双击 API Trace 视图中的一项,会转到 Timeline 视图中此 API 调用所在的主机线程行。

通过这个视图中,我们可以针对每一个 API 调用分析和调试输入参数和返回结果。例如,我们可以轻松检查是否所有的 API 调用都返回 CL_SUCCESS,或所有的 buffer 都是使用正确的标志创建。我们也可以通过此视图找出冗余的 API 调用。

收集 OpenCL GPU Kernel 性能计数器

GPU kernel 性能计数器可以用来寻找 kernel 执行中可能的性能瓶颈。在工具的文档中,已经列出了 AMD Radeon GPU 支持的所有性能计数器执行以下任务。

一旦通过轨迹数据发现了哪个 kernel 最需要进行优化,我们就可以收集 GPU 性能计数器以深入分析 kernel 在 GPU 设备上的执行。通过性能计数器,我们可以做到以下几点。

- 查询为此 kernel 分配的资源数量(通用寄存器,本地内存和 Flow Control Stack 大小)。这些资源影响在 GPU 上可同时运行的 wavefront 的数量。如果 wavefront 的数量越多,就可以更好地隐藏数据延迟。
- 确定 GPU 执行的 ALU、全局内存指令和本地内存指令的数量。
- 确定读取和写入全局内存的字节数。

Method	Execution	GlobalWorkSize	GroupWorkSize	Time	LocalMemSiz	DataTransfer	GPRs
WriteBuffer	1			2.58932		16	
WriteBuffer	2			0.08525		16	
tbody_sim_k1_juniper1	3	{ 1024 1 1 }	{ 256 1 1 }	0.65119	4096		12
CopyBuffer	4			0.18184		16	
CopyBuffer	5			0.10060		16	
ReadBuffer	6			0.13456		16	

图 12.3

Microsoft Visual Studio 2010 中 AMD APP Profiler 的 Session 视图

- 确定系统中 SIMD 引擎和内存部件的利用率。
- 查看 Shader 编译器的效率，是否有效地把 ALU 指令打包成 AMD GPU 使用的 VLIW 指令。
- 查看所有的本地内存(LDS)bank 冲突，即 SIMD 单元中的多路试图读或者写同一个 LDS bank，从而必须串行执行(导致访问延迟)。

Session 视图(图 12.3)展示了一个剖析会话中的性能计数器。输出数据可以保存成 CSV 格式的文件。也可以点击 Method 列中的 kernel 名称，显示这个 kernel 的 OpenCL kernel 源代码，AMD 中间语言，GPU ISA 或者 CPU 汇编代码。

AMD APP KernelAnalyzer

AMD APP(Accelerated Parallel Processing) KernelAnalyzer 是一款静态分析工具，可以为 AMD Radeon GPU 编译、分析和反汇编 OpenCL kernel。可以用图形化界面来交互式地调优 OpenCL kernel，也可以用命令行方式来生成详细的报告。后面，我们把此工具简称为 KernelAnalyzer。

KernelAnalyzer 已经包含在 AMD APP SDK 中，也可以用独立安装包进行安装。可从 AMD 开发者网站 <http://developer.amd.com> 下载 kernelAnalyzer 的安装包。

运行 KernelAnalyzer 要求系统已安装了 AMD OpenCL 运行时。但是，不需要系统上安装 GPU。

在 KernelAnalyzer 中编译一个 OpenCL kernel，只需要简单地把包含 OpenCL kernel 代码的源文件拖拽到 KernelAnalyzer 的主窗口中(图 12.4)。我们不需要整个 OpenCL 应用，就可以编译或者分析 OpenCL kernel。

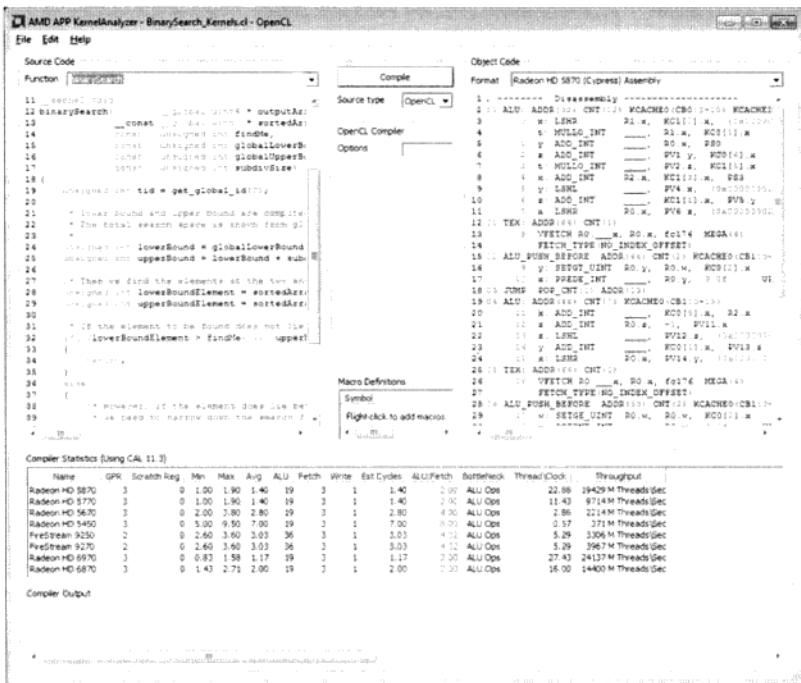


图 12.4

AMD APP KernelAnalyzer

通过 KernelAnalyzer，我们可以执行以下任务。

- 针对多种 Catalyst 驱动版本和 GPU 目标设备，编译、分析和反汇编 OpenCL kernel。
- 查看 OpenCL 运行时生成的 kernel 编译错误和警告信息。
- 查看 OpenCL 运行时生成的 AMD 中间语言代码。
- 查看 AMD Shader 编译器生成的 ISA 代码。特别的，通过分析 ISA 代码进行设备相关的 kernel 优化。
- 查看通过分析 ISA 代码生成的多种统计数据。
- 查看 kernel 分配的通用寄存器和溢出的寄存器数量^①。

因为 KernelAnalyzer 可以快速为多种 GPU 目标设备编译 kernel，所以它特别适合 OpenCLkernel 的快速原型开发。

① 译者注：即 kernel 中使用的寄存器数量超出了 GPU 通用寄存器的数量。

演示 AMD APP Profiler

在本小节中，我们将通过剖析 AMD APP SDK 2.4 版本中自带的矩阵乘法示例，演示 AMD APP Profiler 的用法。

我们将演示具体的步骤流程，包括运行 AMD APP Profiler，使用生成的数据发现应用的瓶颈，改进应用的性能。

启动 AMD APP Profiler

1. 在 Microsoft Visual Studio 中加载 AMD APP SDK 示例解决方案文件 (OpenCLSamplesVS10.sln)。
2. 设置 MatrixMultiplication 项目为启动(start-up)项目。
3. 确认此项目可以正常的编译和运行。
4. 确认已正确安装 APP Profiler
 - a. 在 Visual Studio 主菜单中，选择 Help -> About Microsoft Visual Studio 菜单项。在 Installed products 下，找到 AMD APP Profiler 项。如果没有，请重新安装 Profiler。
 - b. 检查 APP Profiler Session Explorer 已经和 Visual Studio Solution Explorer 停靠于同一个窗口面板上。如果没有看到 APP Profiler Session Explorer 面板，可选择 Visual Studio 主菜单的 View -> Other Windows -> APP Profiler Session Explorer 打开它。

使用应用程序的轨迹数据以发现性能瓶颈

1. 在 APP Profiler Session Explorer 上，点击 Collect Application Trace 按钮，开始收集应用的轨迹数据。
2. 在轨迹收集完成后，Visual Studio 编辑窗口的面板上会显示 APP Profiler Timeline (图 12.5)。
 - a. 通过图 12.5 的 API Trace 视图，我们可以验证所有 API 调用的返回值是否都是 CL_SUCCESS，以此来确认应用是否成功运行，OpenCL 运行时是否产生错误。在此视图中，我们也可以查看每个 API 调用的输入参数和输出结果。比如，最开始的两个 buffer 使用 CL_MEM_READ_ONLY 标志创建，第三个使用 CL_MEM_WRITE_ONLY 标志创建。这几个 buffer 都是 16 MB 大小。
 - b. 在图 12.5 的 Timeline 视图中，我们可以观察到应用有以下特点。
 - 包含一个 OpenCL 上下文和一个命令队列。
 - 命令队列是创建在名为 Juniper 的 GPU 设备上，这是 AMD Radeon HD5770 GPU 的内部名称。

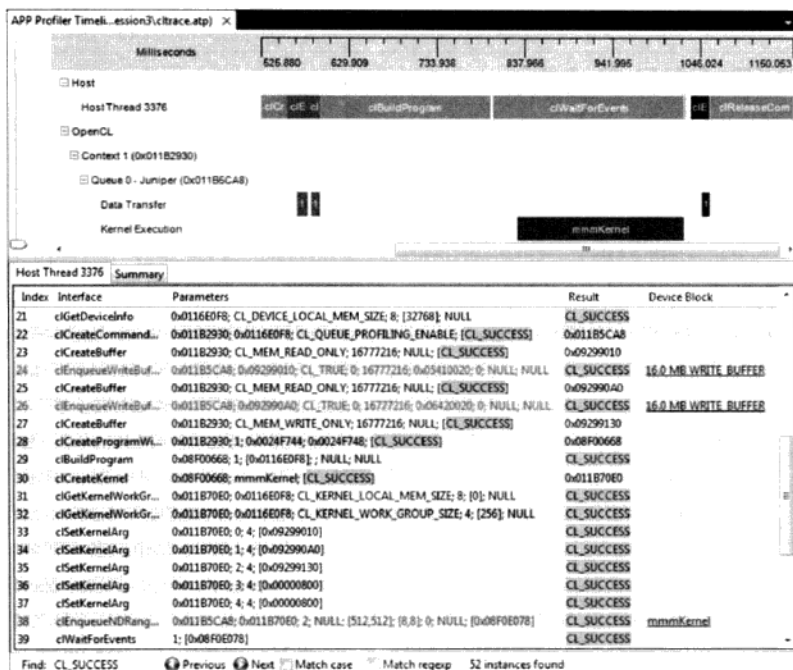


图 12.5

MatrixMultiplication 应用的 APP Profiler Timeline 和 API Trace 视图

- 应用包含三个数据传输操作：传输二个输入 buffer 到设备上，读取设备上一个输出 buffer。
- 只有一个 kernel 在 GPU 设备上执行。
- 此应用似乎有正确的数据传输和 kernel 执行的依赖关系。在 kernel 开始运行之前，两个输入 buffer 已经传输到设备上，在 kernel 运行结束后，读取设备上的输出 buffer。在缺少压力测试或者模型检测的情况下，这为正确的实现提供了良好的支持。

3. 点击Summary标签(图12.5)浏览Summary Pages视图。

- 在Summary Pages视图选择下拉组合框中的Context Summary转至Context Summary页(图12.6；在点击Summary标签后的默认视图)。
 - 在此页中，可以发现 GPU 设备的 kernel 执行是性能瓶颈。GPU 设备的 kernel 执行共计耗时 195 ms，相对的数据传输共计耗时 32 ms。

- b. 选择下拉菜单的 Top10 Kernel Summary 转至 Top10 Kernel Summary 页(图 12.7)
- 通过本页,我们可以发现 kernel 实例(mmmKernel)最需要进行优化。在本例中,只有一个 kernel 实例。

使用 GPU 性能计数器发现 kernel 的性能瓶颈

一旦发现最需要进行优化的 kernel 实例,那么我们就可以收集 GPU 性能计数器(点击 APP Profiler Session Explorer 面板上的 Collect GPU Performance Counters 按钮),从而引导我们发现 kernel 可能的性能瓶颈。

图 12.8 展示了采用 3 种不同方法进行两个 2048×2048 矩阵相乘的 GPU 性能计数器结果。

Host Thread 3376 Summary

Previous Next Context Summary

Context ID	# of Buffers	# of Images	# of Kernel Dispatches	Total Kernel Time (ms)	# of Memory Transfer	Total Memory Time (ms)	# of Read	Total Read Time (ms)	Size of Read	# of Write	Total Write Time (ms)	Size of Write	# of Map	Total Map Time (ms)	Size of Map	# of Copy	Total Copy Time (ms)	Size of Copy
1	3	0	1	195.90927	3	32.11332	1	9.13647	16.00 MB	2	22.97685	32.00 MB	0	0	0	0	0	0
Total	3	0	1	195.90927	3	32.11332	1	9.13647	16.00 MB	2	22.97685	32.00 MB	0	0	0	0	0	0

图 12.6
MatrixMultiplication 应用的 Context Summary 页面

Host Thread 3376 Summary

Previous Next Top10 Kernel Summary

Kernel Name	Context ID	Command Queue ID	Device Name	Duration (ms)	Global Work Size	Group Work Size	Thread ID	Call Index
mmmKernel	1	0	Juniper	195.90927	[512,512]	[8,8]	3376	38

图 12.7
MatrixMultiplication 应用的 Top10 Kernel Summary 页面

Method	Time	GPUs	Wavefronts	ALUBits	FetchInsts	WriteInsts	LDFetchInsts	LDWriteInsts	ALUBusy	ALUParking	Cachemiss	FetchInstsBusy	FetchInstsStall	LDInstsConflict
mmmKernel	195.90927	17	4096	11383	4096	4	0	0	13.32	77.33	0	99.10	42.29	0
mmmKernelLocal	134.06657	20	4096	20392	7394	4	4096	932	39.87	61.45	0	97.67	63.89	30.74
mmmKernelImage	23.40209	31	2048	23080	6144	8	0	0	83.76	84.87	69.71	98.06	0.09	0

图 12.8
三种不同方式的矩阵乘的 GPU 性能计算器结果, 矩阵大小为 2048×2048 。第一行只使用全局内存, 第二行使用了本地内存。第三行使用了 OpenCL image 对象

1. 第一行表示 `MatrixMultiplication` 示例的结果：此方法使用全局内存存储矩阵。
 - a. 根据结果所示，kernel 使用了 17 个通用寄存器(GPR)。这个值是不错的，同时运行的 wavefront 数量不大可能受到 GPR 使用的限制。总共生成了 4096 个 wavefront。当前设备最多有 24 个 SIMD 核，所以每个核至少有 170 个 wavefront——远远大于设备资源可同时运行的数量，足以轻易使设备保持忙碌状态。
 - b. 根据较低的 `ALUBusy` 和较高的 `FetchUnitBusy` 与 `FetchUnitStalled` 计数器值，我们可以推断此 kernel 的性能瓶颈是从全局内存读取数据。
2. 第二行表示 `MatrixMultiplication` 示例采用 `mmmKernel_local` kernel 的结果：此方法使用全局内存保存矩阵，但是也用本地内存缓存结果。
 - a. 根据结果所示，通过利用本地内存减少了从全局内存读取数据的压力，kernel 的执行时间从 195 ms 缩短到了 114 ms(改进 42%)。shader 编译器在打包 VLIW 指令上也稍有提高(`ALUPacking` 计数器)。
3. 第 3 行表示 `MatrixMullImage` 示例的结果：此方法使用 OpenCL image 对象保存矩阵。
 - a. 根据结果所示，kernel 执行时间显著缩短(减少到 21ms，相比第 1 种使用全局内存的方法改进了 89%)。从性能计数器中，我们可以发现 `FetchUnitStalled` 计数器的值几乎为 0%。`FetchUnitBusy` 的数值依旧很高；这表明这个 kernel 的瓶颈依旧是从全局内存中获取数据。可是，使用 image 对象起到了帮助作用，因为 image 对象有数据缓存(根据 `CacheHit` 计数器，缓存命中率为 69%)。`ALUBusy` 计数器的值现在非常高，这表明 SIMD 现在都是充分利用的。

从这三种方法，我们可以看出 `MatrixMultiplication` kernel 的性能瓶颈是从全局内存获取数据。第二和第三种方法分别通过利用本地内存和 image 对象缓存全局内存来尝试解决这个问题。

调试 OpenCL 应用程序

在前面的小节中，我们已经讲述了如何优化我们 OpenCL 代码的性能。但是，最高的要求是正确性。在本节中，我们将讨论异构环境下的调试问题以及概览所提供的工具。

传统上，调试并行程序比调试通常的串行代码更加复杂，因为一些不易察觉的 bug，比如竞态条件，是难以检测和重现的。调试运行在异构设备上的并行程序更加困难，因为加速并行平台的复杂性和“黑盒”的特质。

开发人员是在 API 的基础上开发，隐藏了并行平台的实现。调试器和剖析工具将开发人员的视角转换到了“白盒”模型，使开发人员可以仔细考察 OpenCL 中单独的命令是如何影响并行计算系统。如此一来，开发人员可以发现由于不正确的 OpenCL 使用造成的 bug，优化他们的应用程序。在本章的剩余部分，我们将简要介绍 gDEBugger，一种高级的 OpenCL 调试和剖析工具。同时，我们也将简要讲解 AMD 提供的 printf 扩展，一种更简单但有时比较有效的调试方法。

gDEBugger 概览

gDEBugger 是一个 OpenCL 和 OpenGL 的调试、剖析和内存分析工具。它可以帮助开发人员发现程序错误，优化 OpenCL 性能和内存消耗。

gDEBugger 由下列部分组成。

- gDEBugger Visual Studio 插件——在 Microsoft Visual Studio 中增加高级的 OpenCL 和 OpenGL 调试功能。
- gDEBugger(单机)——提供高级 OpenCL 和 OpenGL 调试与剖析功能，可运行在多种平台(Windows, Linux 和 Mac)。

图 12.9 是简化的 gDEBugger 与 OpenCL 设备交互的高层概览，展示了一些重要的模块和组件。使用 gDEBugger 时，拦截应用程序和 OpenCL Installable Client Driver 间的 API 调用。这使 gDEBugger 可以记录所有的 API 调用，识别所有使用的 OpenCL 对象和收集这些对象的数据。此外，gDEBugger 可以主动地修改一些 API 调用，增加调用以查询额外的信息，也可以消除一些调用来帮助开发人员分析性能瓶颈。

在后面的小节中，我们将简要描述 gDEBugger 的调试能力以显示在开发环境中的用处。

使用 gDEBugger 调试并行 OpenCL 应用程序

如前所述，在异构的应用程序中存在两个不同的代码区域。

1. API 级别的代码(clEnqueue 调用, clCreateBuffer 等)，在主机端运行。
2. 计算 kernel 代码，在设备端运行。

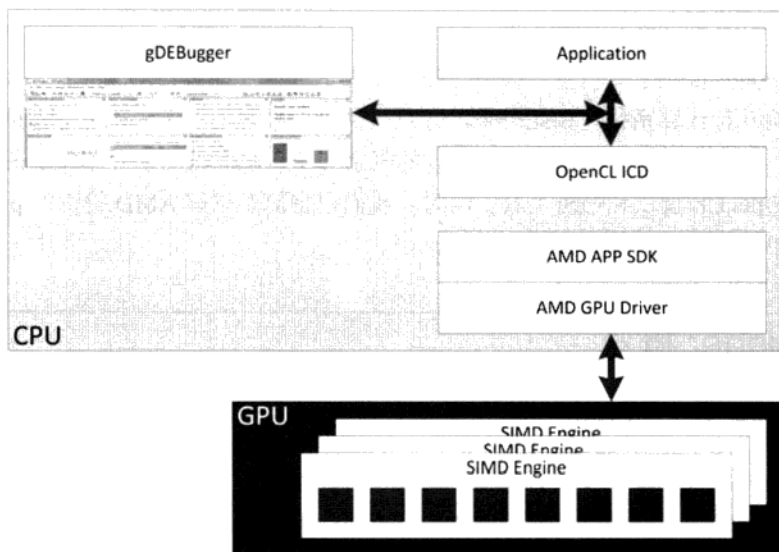


图 12.9

gDEBugger 连接到应用程序和 OpenCL 的高层概览

我们简要讲解 gDEBugger 调试 API 级别代码和计算 kernel 的细节。

API 级别调试

gDEBugger 提供的 API 级别调试可以查看调用运行时函数的参数。API 级别调试具有以下功能。

- **API 函数断点：**gDEBugger 可以在函数的运行前中断当前应用的执行。可以浏览本次函数调用的调用堆栈，也可以查看函数的参数。
- **记录 OpenCL API 调用历史：**所调试的进程挂起时，gDEBugger 会显示在当前选择的上下文中的最近 OpenCL 函数调用(和它的参数)。图 12.10 展示了 gDEBugger 如何回溯追踪程序中调用的 OpenCL 命令。
- **program 和 kernel 信息：**OpenCL 上下文含有多个 program 和 kernel 对象。gDEBugger 可以让我们验证每个上下文关联的 program。如果 program 采用 `clCreateProgramWithSource` 创建，那么我们也可以浏览传递给这个函数的源代码。
- **image 和 buffer 数据：**OpenCL 上下文中含有 buffer 和 image。gDEBugger 可以让我们浏览这些对象中的数据。对于 image 类型，gDEBugger 可以把 image 数据通过 Image View 可视化。

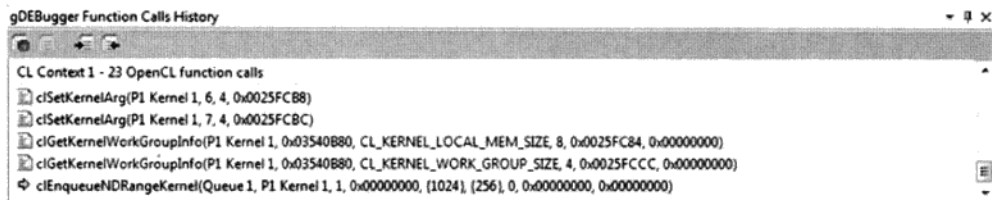


图 12.10

gDEDebugger 的函数调用历史

- 内存检查：gDEDebugger 可以浏览特定上下文 buffer 的内存消耗。gDEDebugger 中的内存检查功能可以用来追踪内存泄露和那些不需要创建或者没有及时释放的对象，这些对象会消耗设备内存而且使调试更加复杂。
- API 使用统计：gDEDebugger 提供当前选择上下文的 API 使用的统计信息。通过浏览上下文中 OpenCL API 调用的细分类别图，我们可以看到每个函数的调用次数。这可以使我们理解不同运行时函数在不同实现和设备上的性能差异。

调试 kernel

gDEDebugger 也可以调试运行在设备上的计算 kernel。有两种使用 gDEDebugger 调试 OpenCL kernel 的方法。

1. 在 kernel 代码中设置一个断点。
2. 从对应的 clEnqueueNDRangeKernel 调用开始单步执行 kernel。

普遍关注的问题是调试 OpenCL 应用程序时，在出现的大量 work-item 中如何保持追踪的状态。GPU 设备上的 kernel 通常会发起数以千计的 work-item。gDEDebugger 帮助开发人员关注于特定的活跃 work-item，显示 work-item 的变量值和执行可能的断点。图 12.11 是 gDEDebugger 调试 OpenCL kernel 代码时的截图。

AMD printf 扩展

AMD printf 是另一种有用的调试工具。在 kernel 代码中增加一行 `#pragma OPENCL EXTENSION cl_amd_printf: enable` 来打开 printf 扩展。OpenCL 扩展的细节已在第 11 章中描述。此 pragma 告诉 OpenCL 编译器在 kernel 内可能使用 printf 函数。

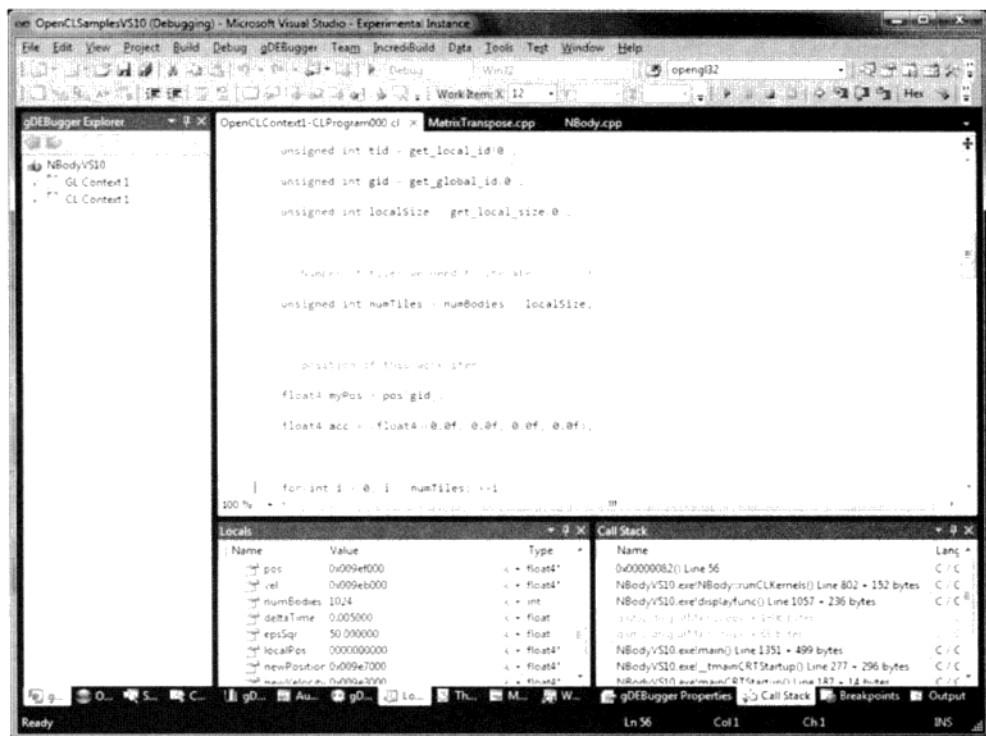


图 12.11

使用 gDBugger 调试应用程序 kernel 的 kernel 代码视图

如下向量加法代码所示, OpenCL kernel 中的 printf 用法与 C/C++ 编程中的用法类似。注意, 在 NDRange 中的每个 work-item 执行到 printf 时, 都会输出格式化字符串结果。

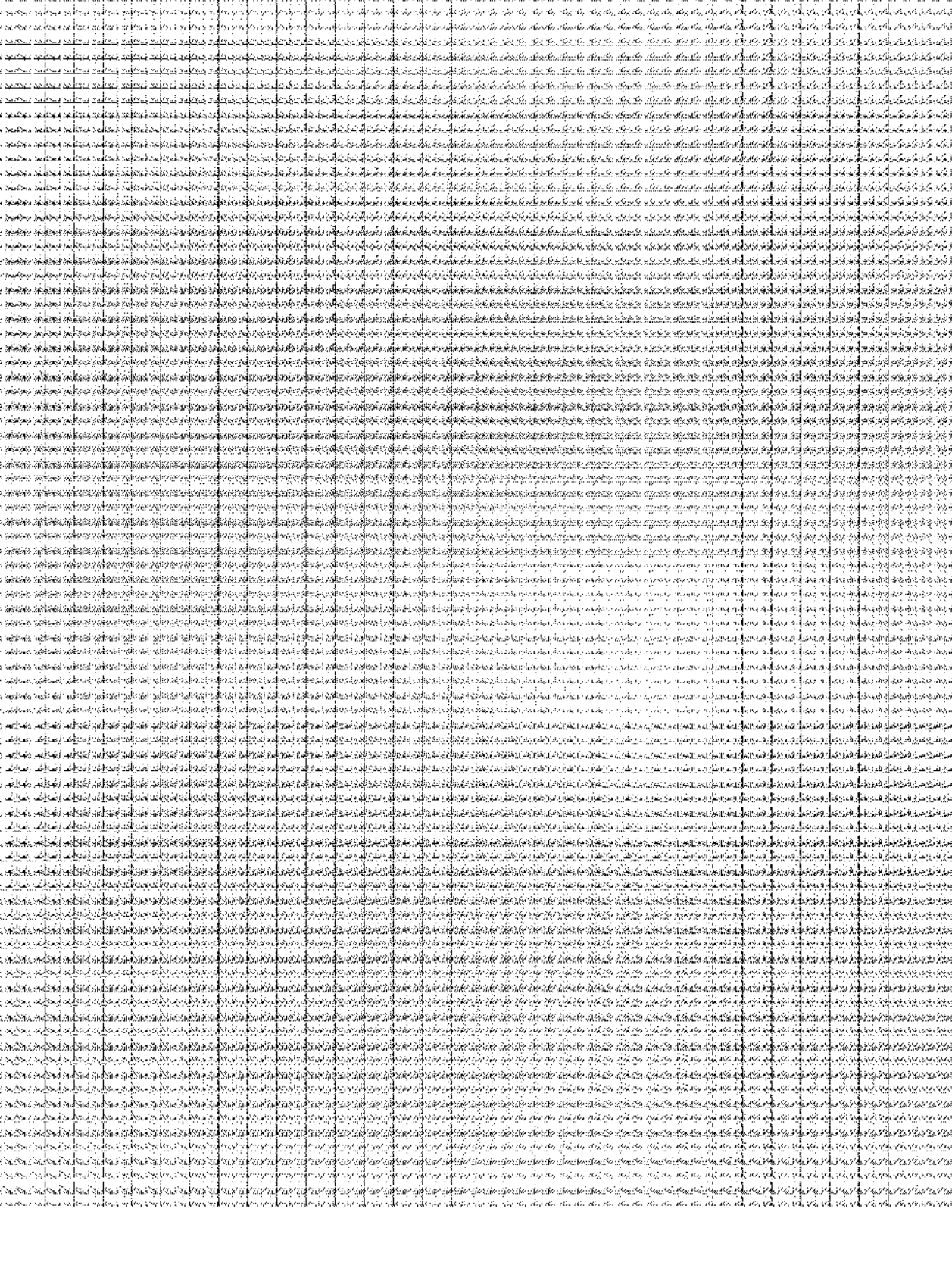
```
#pragma OPENCL EXTENSION cl_amd_printf : enable

//! Simple example showing usage of printf in a vector add kernel.
__kernel void vec_add(__global float * c_ip1, __global float * d_ip2,
__global float c_op, int N)
{
    int tid = get_global_id(0);
    if (tid < N)
```

```
float value_0 = d_ip0[tid];  
float value_1 = d_ip0[tid];  
//This line will print out the format string below 'N' times.  
printf("Values read in %f\t%f\n", value_0, value_1);  
d_op[tid] = value_0 + value_1;  
}
```

小结

在本章中，我们考察了不同的 OpenCL 剖析和调试工具。APP Profiler 和 KernelAnalyzer 等工具可以帮助我们理解代码中什么地方存在性能瓶颈。gDEBugger 可以帮助我们调试程序。



引言

相比于传统软件平台，万维网的优势如下：Web 上的应用程序或服务，可以在任何时间，从任何设备上访问，不需要安装，而且始终是最新的；便于灵活共享客户端和服务端之间的数据和计算。因此，未来将有越来越多的应用针对 Web 进行开发以及迁移到 Web 上。

同时，Web 应用程序也变得更加复杂、更具交互性以及更直观。为浏览器环境下高性能计算创造了需求。在过去几年里，浏览器厂商已经能够使 JavaScript 的运行性能得到数量级的提升。然而，JavaScript 引擎仍然无法比拟本机代码的性能，由于语言内在特性，将继续带来性能的滞后。特别是，JavaScript 不能利用现代 GPU，CPU 和数字信号处理器的多线程数据并行计算能力。

相反，OpenCL 明确地为这一目的而设计的。WebCL 是一组针对 OpenCL 的 JavaScript 绑定，旨在为 JavaScript 应用程序发挥这些设备的能力，允许 Web 开发人员，用来处理图像处理和物理现象模拟等计算密集的任务。仅使用 JavaScript 进行开发，这些类型的应用运行极端缓慢。从某种程度上说，WebCL (Khronos 组织) 已经满足了这些众多需求，但这只是部分而已：WebGL 着色器相较于 OpenCL kernel 严重受限，大量偏重于三维图形，而且只能在 GPU 上执行。

本章讨论在客户端上 Web 应用程序里使用 OpenCL 的前期工作。这里所描述的 WebCL 的实现版本已作为 Firefox 4.0 的一个扩展，模仿 WebGL 的 API 设计的。讨论的 WebCL 版本在 <http://webcl.nokiaresearch.com> 上可以获取。该系统可以工作在多个平

台上, 执行如图像处理的计算密集型任务, 相较于 JavaScript, 可以提供数量级以上的加速比。然而, 在安全性、健壮性、可移植性、与 WebGL 的互操作性、与 JavaScript 引擎的融合以及底层平台可用的 OpenCL 支持这些方面, 依然存在重大的挑战。

框架设计

WebCL 的最终目标是有效地利用所有客户端 Web 应用程序的计算资源。因此, 为保留和初始化资源以及执行控制, 我们还需要一个通用的框架、描述和管理 Web 处理的语言和库。

该方法应能够具有支持广泛设备的能力, 例如移动和嵌入式设备, 通常包括很大程度的异构性, 强大的执行环境只得到编程框架有限的支持。因此, 描述计算使用的编程语言, 应尽可能被众多公用平台广泛支持, 从开发人员的角度来看, 尽可能简单、通用。

C 语言是众所周知得到广泛应用和支持的。然而, C 本身并不具备支持并行编程的任何特点, 需要增强这种支持。当考虑建立一个并行处理的管理库时, 发现该设计类似于 OpenCL 框架。要开发一个 WebCL 框架, 与 OpenCL 相似的特点促使我们利用 WebGL 中的既定方法: 一层位于本地驱动程序上的 native API, 由于受限于 JavaScript 属性, 应尽可能类似于原有的 API(表 13.1), 该方法的好处如下: (1)开发的适配层尽可能“轻量级”, 这样便于维护;(2)移植 OpenCL 代码到 WebCL 只需付出最小的努力;(3)可能限制使用本地端上额外的抽象, 然而抽象并且用户友好的 JavaScript 工具库在该层实现之上。

为了可移植, 使用 web 的方法需要借助通用的语义, 编程人员应该不需要针对特定目标平台。该框架不应该承担二进制兼容性, 也不依赖于一套固定的系统库。因此, 该框架必须排除 OpenCL 的 native kernel 和二进制 kernel。此外, 不支持主机端指针, 因为在 JavaScript 没有直观的机制来实现他们, 并且主机端指针可能被用于暴露潜在的安全漏洞。

在 OpenCL 里, 检查返回的代码来进行错误管理, 代码分为返回值或者输出参数两种形式。每个操作的状态必须单独检查。JavaScript 使用异常进行错误处理, 从而使开发人员将异常包含在构造的 try - catch 代码块中。

表 13.1 与 OpenCL 相对应的 WebCL 抽象

OpenCL 抽象数据类型	Webcl 类
cl_command_queue	WebCLCommandQueue
cl_context	WebCLContext
—	WebCLDataObject
cl_device_id	WebCLDevice
cl_event	WebCLEvent
cl_kernel	WebCLKernel
cl_mem	WebCLMemoryObject
cl_platform_id	WebCLPlatform
cl_program	WebCLProgram
cl_sampler	WebCLSampler
—	WebCLUtils

WebCL 实验性实现

WebCL API 作为 Mozilla Firefox 浏览器的一个扩展，该框架先前已讨论，并以此作为实现的起点。WebCL 旨在吸收其它最先进的已发布或正在开发的 Web 技术，譬如 WebGL。作为 Firefox 的扩展，为我们提供了一个良好的实验起点。

Firefox 扩展

评估实现 WebCL 可选的方法，其中包括直接修改来源于 Mozilla 浏览器的扩展 NPAPI 插件，此方法将 WebCL 视为 Firefox 的扩展，作为 native XPCOM 代码(跨平台组件对象模型)组件来实现，使其能够脱离 Mozilla 代码库进行开发。这种实现方法无需用户安装和运行定制的浏览器。尽管 NPAPI 提供了具有吸引力的途径实现 WebCL(面向广泛的现有浏览器)，但是它不适合提供 WebCL API，因此在 JavaScript 中不能使用。

Mozilla Firefox 建立于 Mozilla 应用程序框架的上层。应用程序的用户界面使用 XUL 编写，XUL 是 Mozilla 中 XML 用户界面标记语言，内部主要使用 XPCOM。XPCOM 组件能够独立使用，作为功能单位与其他组件提供服务。XPCOM 提供一套核心的功能、组件和类，如组件管理。文件抽象，对象的消息传递，内存管理，组件可以使用各种语言来编写。为了方便访问 native OpenCL 代码库，我们选择 C++(尽管事实上 JavaScript 更普遍，实际上已被鼓励作为 Mozilla 的扩展)。

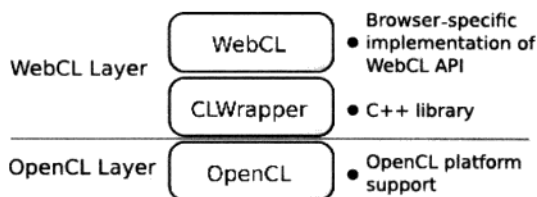


图 13.1

WebCL 实现的基本框架图

如图 13.1 所示，WebCL 通过两层来实现：WebCL 和 CLWrapper。为 WebCL 提供 API 的层次叫 simply WebCL，它包含特定浏览器的所有功能、XPCOM 组件和 JavaScript 互操作。WebCL 的构建基于 CLWrapper 之上，实质上是一个纯粹的 OpenCL C API 的 C++封装，只需执行一个新的 WebCL 层，使 WebCL 能移植到使用 C++语言编写的其他浏览器中。此外，CLWrapper 提供了对象生命周期管理以及基于类型机制的参数检索和 kernel 参数设置。

连接 JavaScript 和 OpenCL

C / C++与 JavaScript 环境之间最显著的差异之一是 JavaScript 支持资源管理。在 C 和 C++里，当动态分配的内存块不再需要时，程序员必须显式的释放。在 JavaScript 中，一旦系统检测到变量使用的内存不再使用，被称为垃圾回收器(garbage collection)的自动内存管理系统会释放它。但是，垃圾回收器的开销比较高，通常不会被立即激活，而是依赖于运行时系统决定何时释放内存。这可能会导致意想不到的不良行为，而且从应用程序的角度来看，因为性能会严重影响垃圾回收器。此外，如果垃圾回收器扩充到一般的资源任务管理，任何有限的资源可能都不会被及时释放。

一些 OpenCL 资源是非常有限的，如 OpenCL 中同时存在的上下文数。OpenCL 的内存缓冲区可以分配一块大的设备内存，再为同时分配的对象数目设置上限。在使用垃圾回收器的环境下，它可能不会确保这些资源准确释放，导致应用程序错误难以重现和校正。

由于 WebCL API 试图按照标准 JavaScript API 的特点，实现在很大程度上依赖于垃圾回收器。因此，应用程序应避免重新创建多个 WebCL 对象，尽可能重用它们。尽管 WebCL

API 变得稍微复杂一些但编程人员遇到的问题将减少, 因此, 它可能作为一个选项在未来增加的 OpenCL 相关资源中显式发布,

在许多情况下, OpenCL API 为任何数据提供一个 void 指针, 数据类型依据于使用的背景, 这在有限的 C 语言类型系统中是不幸的。例如, `clGetDeviceInfo` 需要一个称作 `param_name` 的枚举参数(代表使用设备的信息)和一个称为 `param_value` 的 void 指针参数(代表结果写入的内存缓冲区)。枚举值代表指针写入数据的上下文和含义。

由于 JavaScript 没有指针, 所以类型系统是动态而不是静态的, 类似的做法显然是无法实现的。因此, 调用时需要添加类型信息, 例如 `clGetDeviceInfo` 或 `clSetKernelArg`。有时可以省略类型, 因为当该值是一个特定类型的对象时, 类型可以推断出来。这种情况是可能的, 例如在使用内存对象的情况下。

许多 WebCL 用例(例如, 图像处理和信号处理)需要一个 JavaScript 上下文和在 OpenCL buffer 之间传输大量数据。总体而言, 数据的类型和特性只有在 OpenCL kernel 代码中才已知, 不能由 WebCL API 确定。鉴于 OpenCL 的 C API 使用的 void 指针类型以及往设备进行数据拷贝需要库来进行处理, 以这种方法使用 JavaScript 会不起作用, 因为 JavaScript 没有 void 指针。JavaScript 数组是一个不二之选, 但由于拷贝的单位变为数据项, 本质上是低效的。或许更重要的是数据的数组表示对数据施加了一定的结构性解释, 可能无法在所有情况下都有效, 相对于使用 void 指针引用二进制 blob 数据更受限制。

理想的解决办法是使用一个对象, 它能够保持一个任意大小的数据缓冲区, 同时为数据的含义提供多个不同的视角或解释。这样的对象以 `TypedArray` 和 `ArrayBuffer` 数据类型形式存在, 已经在 WebGL 的 API 中得到支持。虽然在即将发布的 Mozilla Firefox 中实现, 然而也只是得到部分的支持。例如, 缺乏 XPCOM 接口的定义, 这使得利用浏览器扩展 `TypedArray` 变得复杂。

虽然 WebCL 将来会直接使用 `TypedArray`, 但在实现中, 我们额外引入了称为 `WebCLDataObject` 的一层抽象来让 API 和其实现保持整洁, 并允许用户创建的浏览器不支持 `TypedArray` 的实现。这将成为在 WebCL API 使用的数据对象, 而 OpenCL 可使用 void 指针关联内存对象。此外, 数据对象提供了一定的实用工具, 以支持 canvas 元素处理图像数据, 这对以下小节中所讨论的实例很重要。

WebCL 动手练习

WebCL 的用法类似于编写 OpenCL 代码。与本地编码的区别是，在 Web 应用程序中，主机端代码现在变成了 JavaScript 代码。不同的环境下，从应用的角度来看，对于 C 和 JavaScript 自然会带来一些差异，但这些差异不会在本章中详细讨论。此外，我们选择忽略检验(check)，检验通常在开发高质量的应用程序中是必不可少的，这些实例重点突出了 OpenCL 和 WebCL 之间的差异。

Kernel 源代码:

```
__kernel void desaturate(
    __global uchar4* src,
    __global uchar4* dst,
    uint width,
    uint height) {

    int x = get_global_id(0);
    int y = get_global_id(1);
    int i = y * width + x;

    uchar4 color = src[i];
    uchar lum =
        (uchar)(0.30f * color.x + 0.59f * color.y + 0.11f * color.z);
    dst[i] = (uchar4)(lum, lum, lum, 255);
}
```

以下内容显示了 OpenCL 和 WebCL 主机端源代码的实现。为了便于读者理解，我们采用合乎逻辑的步骤，并行地比较了两种实现。首先，必须建立 OpenCL 上下文。为简单起见，选取第一个可用的(即默认的)平台作为设备。

以下代码描述了如何在 WebCL 中简化使用 OpenCL API 来获取信息。在 native OpenCL 中，首先需要获取平台数目，然后分配必要的内存缓冲区，最后再次获取平台信息并存储在分配的缓冲区中，而 Web 通过简单的返回值来实现。

普通的 JavaScript 数组、常量以及一些高层次的功能(如检索平台和创建上下文)可以传递给 property array 用于 WebCL 全局对象访问。与 OpenCL 显著的区别是遗漏错误报告/处理代码，WebCL 使用 JavaScript 的异常机制实现错误处理(回调可能会在未来实现)。

OpenCL 代码:

```

cl_uint numPlatforms;
cl_int status = clGetPlatformIDs(0, NULL, &numPlatforms);
cl_platform_id* platforms =
    (cl_platform_id*)malloc(sizeof(cl_platform_id)*numPlatforms);
status = clGetPlatformIDs(numPlatforms, platforms, NULL);
cl_context_properties clProperties[3] = {
    CL_CONTEXT_PLATFORM,
    (cl_context_properties)platforms[0],
    0 };
cl_context ctx = clCreateContextFromType(
    clProperties,
    CL_DEVICE_TYPE_DEFAULT,
    NULL,
    NULL,
    &status);

```

WebCL 代码:

```

var platforms = WebCL.getPlatformIDs();
var ctx = WebCL.createContextFromType(
    [WebCL.CL_CONTEXT_PLATFORM, platforms[0]],
    WebCL.CL_DEVICE_TYPE_DEFAULT);

```

下一步是构建 **buffer**，输入和输出 **buffer** 的大小 **bufSize** 等同于以字节为单位的图像像素数据的大小(即每个像素所占的字节数乘以图像的像素数)。由于 WebCL 采用了面向对象的编程范式，**createBuffer** 作为 **ctx** 对象的成员函数，而不是通过参数 **ctx** 指定。此外，WebCL 没有使用主机端指针。

OpenCL 代码:

```

cl_mem bufIn = clCreateBuffer(
    ctx,
    CL_MEM_READ_ONLY,
    bufSize,
    NULL,
    &status);
cl_mem bufOut = clCreateBuffer(
    ctx,
    CL_MEM_WRITE_ONLY,
    bufSize,
    NULL,
    &status);

```

WebCL 代码:

```
var bufIn = ctx.createBuffer(WebCL.CL_MLM_READ_ONLY, bufSize);
var bufOut = ctx.createBuffer(WebCL.CL_MLM_WRITE_ONLY, bufSize);
```

接下来, 需要根据 kernel 源代码来创建 OpenCL 程序, 然后编译。native OpenCL 的源代码为一个 C 字符串指针数组, 该数组可分成多个子字符串。在 WebCL 中, 不能分割源字符串, 因此子字符串数这一个参数也不需要。

OpenCL 代码:

```
cl_program program = clCreateProgramWithSource(
    ctx,
    1,
    (const char **)&kernelSrc,
    &kernelLength,
    &status);
size_t devicesLen;
status = clGetContextInfo(
    ctx,
    CL_CONTEXT_DEVICES,
    0,
    NULL,
    &devicesLen);
cl_device_id* devices = (cl_device_id *)malloc(devicesLen);
status = clGetContextInfo(
    ctx,
    CL_CONTEXT_DEVICES,
    devicesLen,
    devices,
    NULL);
status = clBuildProgram(program, 0, NULL, NULL, NULL);
```

WebCL 代码:

```
var program = ctx.createProgramWithSource(kernelSrc);
var devices = ctx.getContextInfo(WebCL.CL_CONTEXT_DEVICES);
try {
    program.buildProgram(devices, "");
} catch(e) { ... }
```

我们在下面几行源代码中, 创建 kernel 并设置其参数。对于正确的 WebCL 类型转换, kernel 参数的实际类型必须是已知的, JavaScript 与 C 显著不同的是变量类型是动态的。由于 kernel 代码需要数据的确切大小和类型, JavaScript 中值必须正确地转换。WebCL 对象, 如实例中内存对象 bufIn 和 bufOut, 并不需要具体的类型, 是因为在这种情况下没有解释的余地, 因此, 在该实例中使用的 bufOut 没有显式指定数据类型。

OpenCL 代码:

```

cl_kernel kernel = clCreateKernel(program, "desaturate",
    &status);
status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
    (void*)&bufIn);
status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
    (void*)&bufOut);
status = clSetKernelArg(kernel, 2, sizeof(cl_uint),
    (void*)&width);
status = clSetKernelArg(kernel, 3, sizeof(cl_uint),
    (void*)&height);

```

WebCL 代码:

```

var kernel = program.createKernel("desaturate");
kernel.setKernelArg(0, bufIn, WebCL.types.MEMORY_OBJECT);
kernel.setKernelArg(1, bufOut);
kernel.setKernelArg(2, width, WebCL.types.UINT);
kernel.setKernelArg(3, height, WebCL.types.UINT);

```

最后，我们需要一个命令队列，将拷贝源数据的写命令、kernel 执行与拷贝结果数据的读命令加入该队列。如前所述，与 OpenCL 内存对象相关的数据指针被 WebCL 中 WebCLDataObject 取代，它封装了二进制数据，无需使用 JavaScript 数组。

OpenCL 代码:

```

cl_command_queue cmdQueue = clCreateCommandQueue(
    ctx,
    devices[0],
    0,
    &status);

status = clEnqueueWriteBuffer(
    cmdQueue,
    bufIn,
    CL_FALSE,
    0,
    bufSize,
    pixels,
    0,
    NULL,
    NULL);

status = clEnqueueNDRangeKernel(
    cmdQueue,
    kernel,
    2,
    NULL,
    globalWS,

```

```

        localWS,
        0,
        NULL,
        NULL);
status = clEnqueueReadBuffer(
    cmdQueue,
    bufOut,
    CL_FALSE,
    0,
    bufSize,
    pixels,
    0,
    NULL,
    NULL);
status = clFinish(cmdQueue);

```

WebCL 代码:

```

var cmdQueue = ctx.createCommandQueue (devices[0], 0);

cmdQueue.enqueueWriteBuffer (bufIn, false, 0, bufSize,
dataObject, []);
cmdQueue.enqueueNDRangeKernel(kernel, globalWS.length, [],
globalWS, localWS, []);
cmdQueue.enqueueReadBuffer (bufOut, false, 0, bufSize, dataObject,
[]);
cmdQueue.finish();

```

在 WebCL 中，所产生的数据现在可被数据对象使用，并可以进一步被应用程序所利用。

Web 照片编辑器

为了基准测试 WebCL 在实际中应用的情况，我们提供了一个基于网络的照片编辑器原型，如图 13.2 所示。用于测试的该编辑器在 <http://webcl.nokiaresearch.com>。

Web 照片编辑器实例以触摸屏为特色，使用按钮和滑块以及一组图像处理操作优化用户界面。操作包括基本的调整，如剪切，缩放，旋转，消除红眼，模糊，平滑，边缘增强，失真等其他特效。应用程序完全运行在浏览器中，不需要任何服务器端处理。

借助 WebGL 和 WebCL，在常规的笔记本电脑上，大部分图像处理操作每秒处理的百万像素数能够达到几十甚至上百。不借助于现代 GPU，多核以及向量加速处理器的并



图 13.2

基于 WebCL, WebGL 和 HTML5 的照片编辑器原型

行处理能力，流畅而互动地编辑体验仍将是一个遥远的梦想。

编辑体验的核心在于绘画模式。该模式借助于画笔，对选定的图像片段交互地应用图像处理操作(图 13.3)。为了找出一张照片中的主体，可以先使用一个全局性的模糊滤镜，然后从原始图像“涂鸦”得出所需要的部分。

用户选择编辑的图像，需要分配四个 WebGL 纹理：(1)用于存储原始图像来源的纹理；(2)一个用于存储中间结果的临时纹理；(3)一个字母掩码纹理用于交互的涂鸦模式；(4)储存加工和绘制过的最终图像所需的目标纹理。对于待编辑的源和目标纹理的角色是否发生逆转的“应用”，取决于用户，所有的纹理与原始图像的大小相同。

照片编辑器的处理时间大多花费在 WebGL 着色器、WebCL kernel 和数据的来回传输上。当照片每次缩放、平移、绘画、调整窗口的大小或其他操作时，WebGL 的着色器需要调用画布重绘。因为所有的着色器和 kernel 都作用于全尺寸图像和计算量以及 CPU 和 GPU 之间来回数据传输操作，所以开销非常大。数据通信来源于目前 WebGL 和 WebCL 有限的纹理共享，实际上大部分的通信是多余的，

Threshold

Brush Size

Zoom & Pan

☐ Undo

☐ Cancel

☐ Apply



图 13.3

在照片编辑器的交互式绘画中，用户可以应用图像处理操作，如在这个实例中的 Sobel 边缘检测，在选取的图像上画一个圆形笔刷。点击左边的 Threshold 之后，会出现一个滑块，用于调整边缘检测器的灵敏度

讨论

WebCL 实现采用了一些仍处于开发阶段的新技术，这使得安装在目前商业化应用中会带来相对不稳定性。然而，通过这次 WebCL 性能评测的机会，表 13.2 为我们提供了

表 13.2 初步实验的测量数据，相较于 JavaScript 的加速比

过滤器	JavaScript	WebCL@CPU	WebCL@GPU
Grayscaleing	1.0	1.5	2.5
Bilateral filtering	1.0	20	200

早期的性能数据。在本试验中，一幅 2.5 MPix RGB 图像使用先前所描述的 grayscaleing 和双边过滤器进行过滤。双边过滤器是根据直双边定义的。表中的数字描述了相较于 JavaScript 实现取得的性能改善。实验结果在很大程度上取决于执行环境，因此这些结果对性能而言是具有指导性。

编写 WebCL 应用时，或者利用 OpenCL 将现有的代码移植到 WebCL，必须认识到目标环境的具体特点，它可以在结果的质量和有效性上带来重大影响。因为 WebCL 构建于 OpenCL API 之上，会有不可避免的开销(特别是当数据移动于 JavaScript 上下文和本地代码之间时)。出于这个原因，最好避免频繁的数据传输。一旦数据被传输到本机端，最好保持它，直到 JavaScript 应用程序需要用到它。

在应用程序开发过程中，对潜在的运行平台应该作出假设。现代 Web 应用程序应预计应用于多种设备，范围从台式机、移动设备到游戏机。当在本地执行时，更容易使应用程序利用特定类型平台所具有的优势，如桌面工作站或 AMD GPU。Web 应用程序在平台特化之前，应关注于收集足够多的执行环境相关信息。例如，在移动设备上运行时，kernel 编译时间可以高于预期。此外，与本地的 OpenCL 应用一样，应设置 kernel 执行参数，参数信息确定以后，可以获取参数数量的最大值。

如前所述，WebCL 的内存管理是基于 JavaScript 运行时内置的垃圾回收器。此外，管理有限的 OpenCL 硬件和运行时资源的机制是通过 WebCL API 实现的，过度使用可能会出现问題。因此，必须认真考虑整个应用程序中资源的利用。消耗大量资源的对象，如上下文或内存对象，应创建一次，然后尽可能重用。

在 kernel 中算法的设计应崇尚快速和高效。然而，在 Web 环境下，评估是基于容易占主导地位的 JavaScript 速度和垃圾回收器的开销，任何性能优化都是有意义的。此外，网络有利于跨平台的解决方案，因此，算法设计时应该考虑这点。

在未来，我们希望在 HTML 和 JavaScript 中看到对 TypedArray 数据类型广泛的支持。例如，canvas 元素将受益于 TypedArray 类型，具有读写完整图像像素数据的能力，这有别于目前支持的 pixel-by-pixel 方法。此外，WebCL 的输入和输出最好局限于 WebGL 纹理和 buffer，不需要任何中间 buffer。不幸的是，因为目前 WebCL 原型是一个 Firefox 插

件(而不是浏览器的组成部分), WebCL 扩展目前没有任何方法来访问 WebGL 的资源。这可以通过将 WebCL 整合到浏览器引擎中或说服浏览器厂商提供必要的 C/C++ API 扩展来弥补。

在 Web 环境下,任何浏览器向外界公开 API,无一例外地都将受到恶意软件的攻击。因此,WebCL 必须假设每一段主机段代码或 kernel 代码是潜在恶意的。如果底层的 OpenCL 驱动程序或平台不具备足够的安全和可靠,那么 WebCL 实现必须把驱动器/平台组合放入黑名单上,或通过进程隔离、内存保护、kernel 代码分析或其他方式提供更多的保障。然而不幸的是,目前 OpenCL 硬件不支持上下文切换,因此,对不良行为的代码进行驱动控制往往是苛刻并且繁琐的。

编写性能可移植性的 OpenCL C 程序非常具有挑战性。此外,除了智能手机或其他便携设备之外,大多数笔记本和台式电脑,不附带预安装的 OpenCL 驱动程序。要 WebCL 应用普遍,有必要使浏览器附带 OpenCL 的 CPU 实现。

小结

在本章中,我们描述了一个 Firefox 扩展,它提供了一套对应于 OpenCL 的 JavaScript API。它基于 native OpenCL 驱动程序之上,并将 OpenCL C kernel 的描述传递给该 OpenCL 驱动程序。

WebCL 的这种发展模式是具有前景的,尽管仍然存在着开放性和最明显的安全问题。然而 WebCL 使 Web 应用程序达到了一个新的层次,带来了显著的性能提升。虽然可以通过 WebGL 使用 GPU,但 WebCL 将进一步基于 Web 使用 GPU。在一个现代笔记本电脑上,对于计算 kernel,相较于纯 JavaScript, CPU 和 GPU 的性能分别达到了 20 倍和 200 倍的加速比。这些结果表明,该方法为实例提供了良好的性能,并且完全超出了 JavaScript 的范围。

参考文献

Khronos Group. (n.d.). *WebGL—OpenGL ES 2.0 for the Web*. <https://www.khronos.org/webgl>. Retrieved March 2011.

扩展阅读和相关网站

Khronos Group. (n.d.). *OpenCL Overview: OpenCL—The Open Standard for Parallel Programming of Heterogeneous Systems*. <https://www.khronos.org/opencvl>. Retrieved March 2011.

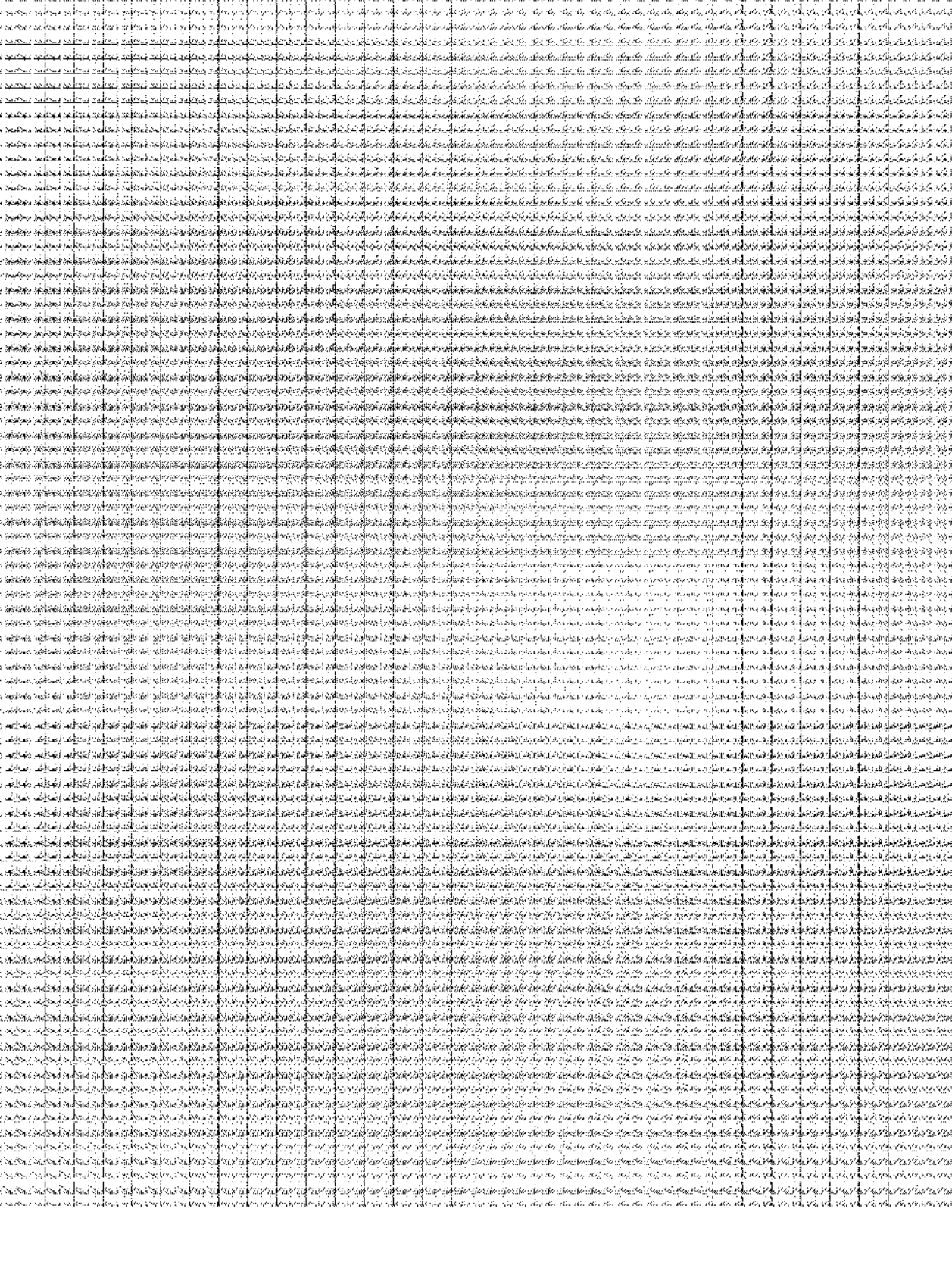
Khronos Group. (n.d.). *Khronos Typed Array API Registry*. <http://www.khronos.org/registry/typedarray>.

Khronos OpenCL Working Group. (2010.). *The OpenCL Specification*. (Version 1.1, Document Revision 36). Beaverton, OR: The Khronos Group.

Mozilla. (n.d.). *Firefox*. <http://www.mozilla.com/en-US/firefox>.

Mozilla. (n.d.). *Firefox Nightlies*. <http://www.mozilla.org/projects/minefield>.

Mozilla Developer Network. (n.d.). *Plugins*. <https://developer.mozilla.org/en/Plugins>.



索引

注意：本书参照原文版式，读者可根据下面的词条找到中译本中的相应描述。页码后有“f”字样表明是图，“t”字样表明是表。

A

- Accelerated processing unit (APU)
 - architecture, 205
 - GPU and CPU cores, 63–64
 - integration, SoCs, 53–54
 - Intel's Sandy Bridge design, 63, 65f
- AMD APP Profiler
 - application trace, 238
 - GPU kernel performance counters, 247
 - performance analysis, 238
- AMD APP KernelAnalyzer, 243–244
- AMD “Bobcat” CPU core *See* Central processing units
- AMD “Bulldozer” *See* Central processing units
- AMD E350 “Zacate” *See* Accelerated processing unit
- AMD E350 “Zacate”, 63, 64f
- AMD “Istanbul” *See* Central processing units
- AMD Phenom II *See* Central processing units
- AMD Radeon HD6970 *See* Graphics processing units
- Applications programming interface (API)
 - defined, 15
 - description, 16
 - Win32 and POSIX, 17–18
- APU. *See* Accelerated processing unit

B

- Buffers
 - array access syntax, 110
 - creation flags, 111
 - error codes, 111–112
 - reading and writing, 113

C

- Caching data
 - global memory to local memory copying, 155–156
- Cell processor, 55 *See also* Hardware trade-offs
- Central processing units (CPUs). *See also* CPU/GPU
 - OpenCL implementation
 - Intel Itanium, 2, 59
 - low-power
 - AMD “Bobcat” CPU core, 56
 - ARM ISA, 56
 - Intel Atom, 56–58
 - power/performance, 58

- mainstream desktop
 - AMD “Bulldozer”, 58–59
 - AMD “Istanbul” CPU, 214, 217f
 - AMD Phenom II, 58
 - Intel Sandy Bridge, 58
 - multi-core, 42
- Command queues
 - communication, 22–23
 - flush and finish commands, 26
 - host-device interaction, 23
 - multiple-queue scenarios, 99
- Concurrent runtime (ConcRT), 214–215
 - Microsoft concurrency runtime (ConcRT), 214
 - Threading building blocks (TBB), 214–215
- Constant memory (*See* Memory model)
- Contexts, 22
- Convolution (*See* Example applications)
- CPU/GPU OpenCL implementation
 - AMD Phenom II X6
 - architecture, mapping, 123, 125f
 - CPU core, 123–124
 - design, 123, 124f
 - memory mapping, 127, 128f
 - worker thread processes, 124, 125–127, 126f
 - 128-bit vector, 127
 - AMD Radeon HD6970
 - architecture, core, 135–136, 135f
 - clause-based SIMD execution
 - ALU operations, 135
 - ISA code and wavefronts, 135
 - software control, 132–135
 - texture clause, 137
 - VLIW packets, 136, 136f
 - hardware
 - synchronization support, 128–129
 - workload schedule, 129–130, 129f
 - performance, 130
 - queuing mechanism, 130
 - resource allocation
 - registers and LDS storage space, 137
 - SIMD core, OpenCL workloads, 137–138, 138f
 - threading and memory system, 132
 - high-bandwidth GDDR5, 132
 - memory hierarchy, 130–132, 131f
 - PCI express bus, 132

D

- Data sharing and synchronization
 - barriers/locks, primitives, 11
 - concurrent program, 11
- Debugging
 - AMD printf extension, 252–253
 - developer's view, 249
 - gDEBgger
 - API-level debugging, 250–251
 - kernel debugging, 251–252
 - parallel programs, 249

E

- Event chaining
 - OpenCL
 - command queue model, 180–181
 - runtime, 181
 - video frame buffer, 181
- Events
 - code, 96–99
 - command sequence, 96–99, 97f
 - defined, 23
 - devices, command queue, 99
 - error conditions, 103
 - event information, 103
 - execution models, 100
 - multiple command queues, 99–100, 99f
 - out-of-order queue, 96
 - parallel multidevice, 100, 101f, 102–103
 - read buffer operation, 103–104
 - task graph creation, 96–99
 - user, 103–104
 - wait list orders execution, 100–102
- Example applications
 - Convolution (buffers)
 - code listings
 - host code, 162–166
 - kernel code, 166–171
 - performance
 - code size, 161
 - loop unrolling, 160–161
 - memory optimization, 161
 - workgroup size selection, 151–154
 - Convolution (images)
 - compilation and execution, 82
 - description
 - blurring and vertical edge-detecting filters, 77–78, 79f
 - blurring kernel, 77
 - kernel, 83–84
 - sampler creation, 82
 - serial code, 78

Histogram

- description, 185
 - global memory data access patterns, 187–189
 - global reduction operation, 186f
 - kernel code, 193–196
 - local memory access
 - binning operations, 190
 - hardware banks, 191, 191f
 - reduction operation, 192f
 - SIMD vectors, 190
 - reduction
 - bank conflicts, 192–193
 - bins, 192–193
 - global, 193
 - using atomics
 - AMD Radeon HD6970 architecture, 190
 - memory pressure, 189
 - SIMD vector, 189
 - workgroups
 - global memory transactions, 186
 - local memory, 185–186
 - pixels mapping, 186
 - size, optimal, 186–187
- Image rotation**
- description, 73
 - buffer declaration and data movement, 76
- C++ bindings, 75, 76**
- coordinates and equations, 74
 - input decomposition, 74–75
 - program execution, 77
 - runtime kernel compilation, 76–77
- Matrix multiplication**
- buffer declaration and data movement, 70–71
 - implementation steps, 69, 70f
 - loops, 67–68
 - program execution, 72
 - runtime kernel compilation, 72
- Mixed particle simulation**
- computation
 - collisions, particles, 197–198
 - data and physical properties, 199
 - GPU vs. CPU, 198
 - screen shot, 197–198
 - small–small collisions, 199
 - CPU implementation, 202–203
 - data-parallel devices, 197
 - description, 197
 - GPU implementation
 - acceleration structure, 201
 - buffer creation, 200–201
 - computing collisions, 201–202
 - integration, 202

- kernels, 205–209
 - load balancing
 - CPU's thread workload, 203–204
 - pool imbalance, 203
 - structure, 204f
 - performance, 204–205
 - uniform grid creation kernel, 205–206
 - Vector addition, 32–38
 - Web photo editor
 - editing image, 265
 - painting mode, 265, 266f
 - prototype, 264, 265f
 - time spent, WebGL shaders and kernels, 265
 - touch screen optimization, 264
 - WebGL and WebCL, image processing operators, 264–265
 - Execution model
 - devices, 88
 - kernels, 17, 87
 - queuing and synchronization
 - callbacks, event, 104–108
 - command barriers and markers, 108–109
 - events, 96–104
 - finish operation, 95–96
 - memory consistency, 96
 - thread-safe command queues, 94
 - synchronization, 90–95, 91f
 - work-items, 17, 87
 - workgroups, 87, 88
 - wavefront and warp, 88
 - Extensions *See also* Open Computing Language types, 211
 - EXT extension, 211
- F**
- Firefox extension
 - layers implementation, 258, 258f
 - sources modification and NPAPI, 257
 - XUL and XML user interface, 257–258
- G**
- Global memory *See also* Memory model
 - access alignment
 - 64 and 128-byte segments, 156
 - data array padding, 157–158
 - GT200 series architecture, 158
 - global data access
 - AMD GPUs and CPUs, 187
 - arithmetic operations, 187
 - efficient access pattern, reading, 188–189, 188f
 - nonparallel work items, 188f
 - read converge, 189
 - serial coalesced trade-off, 189f
 - global performance
 - access memory, 139
 - analyzing performance, 139
 - bandwidth measurement, 139
 - 128-bit float4 accesses, 140–141
 - bits, 141–142, 142f
 - efficiency loss, 140–141, 141f
 - modern CPUs, vector instruction, 139–140
 - multiple wavefronts, 142
 - gDEBuzzer
 - API-level debugging and function, 250–251
 - components, 249
 - interaction, 249–250, 250f
 - kernel debugging, 251–252
 - OpenCL performance and memory consumption, 249
 - Graphic processing units (GPUs) *See also* CPU/GPU
 - OpenCL implementation
 - described, 60–61
 - handheld, 61
 - high-end desktop
 - AMD Radeon HD6970
 - cores, 62
 - lanewise SIMD model, 61–62
 - NVIDIA GTX580, 61, 62f
 - scratchpad memory buffer, 62–63
 - SIMD arrays and threads, 61
 - vs. CPU designs, 63
 - integration, 202
 - kernel performance counters, 242–243
- H**
- Hardware trade-offs
 - cache hierarchies and memory systems
 - access patterns, 54
 - GPUs and cell processor, 55
 - latency, 54–55
 - cores, 42
 - graphics APIs and pixel shaders, 41
 - heterogeneity, 42
 - multi-core architectures
 - AMD Istanbul CPU, 214, 217f
 - AMD Phenom II, 58
 - AMD Radeon HD6970 GPU architecture, 52, 53f
 - AMD's "Bulldozer" and "Bobcat" designs, 51–52, 52f
 - cloning, single core, 51
 - Intel Sandy Bridge, 58
 - Intel Itanium, 2, 59

- multi-core CPUs, 42
- multithreading, 48–51
- performance enhancement
 - clock frequency, 44
 - CMOS dynamic power consumption, 43
 - parallel coding, 43
 - voltage and frequency, 43
- SIMD and vector processing, 47–48
- SoC and APU, 53–54
- superscalar execution, 44, 45f
- VLIW, 44–47

Histogram *See* Example applications

I

Images

- channel order and type, 114
- data, 113
- multidimensional data structure, 114
- objects, 109–110
- runtime system and hardware, 114
- scalar and vector reads, 114–115
- transformations, 114
- vs. buffers, 113
- Z-order/Morton order memory
 - layouts, 114, 115f

Image rotation *See* Example applications

Intel Itanium, 2, 59 *See* Central processing units

Intel Sandy Bridge, 58 *See* Central processing units

Interoperability with OpenGL *See* Video processing

J

JavaScript and OpenCL

- characteristics, 258–259
- CL program, 262
- garbage collection, 258
- kernel arguments, 262
- memory objects, 262, 263
- resources and contexts, 258
- void pointer, 259
- WebCLDataObject use, 263–264

K

Kernels

- arguments, 107
- debugging, 251–252
- enqueue function, 28
- extraction, 27
- local memory allocations, 31
- mapping, 31
- performance counters, 242–243
- simulation, 206–209

- textual scope, 31
- uniform grid creation, 205–206

KHR extension, 211

L

Local data shares (LDS) *See also* Local memory

- allocation, 137–138
- availability, 138f
- memory latency, 137–138
- read/write access, 132
- SIMD, 135–136

Local memory *See also* Memory model

- data access
 - binning operations, 190
 - hardware banks, 191, 191f
 - reduction operation, 192f
 - SIMD vectors, 190
- performance
 - access phase, 145, 145f
 - balance efficiency, 148
 - behavior, 146f, 147
 - code loads, 143
 - data caches, 142
 - data structures, 147–148, 147f
 - HD6970 memory system and SIMD cores, 143f
 - images map data, 142–143
 - prefix sum, 146f, 147
 - read/write operations, 144–145
 - trade-offs, 143
 - VLIW packet, 145–146

M

Matrix multiplication *See* Example applications

Memory model

- device-side memory model
 - constant memory, 121–122
 - global memory, 117–119
 - local memory, 119–121
 - private memory, 122
 - relaxed consistency, 116–117
- host-side memory model
 - buffers, 110–113
 - images, 113–115

Memory objects

- buffers, 24–25
- definition, 23–24
- images, 25–26

Message-passing communication

- MPI, 10
- physical/arbitrary device, 9–10

Message passing interface (MPI), 10

Mixed particle simulation *See* Example applications

MPI *See* Message passing interface

Multithreading

- Cray/Tera MTA and XMT designs, 51
- extraction, instruction parallelism, 48
- SMT, 49, 49f
- time-sliced version, 49, 50f
- types, 48–49

N

NVIDIA GTX580 *See* Graphics processing units

O

OpenCL *See* Open computing language

Open computing language (OpenCL)

compilation

- AMD's implementation, 84
- dynamic libraries, 85
- Linux, 85

device architectures

- block-based parallelism model, 41
- design space
 - APU and APU-like designs, 63–64
 - CPU designs, 56–60
 - CPUs vs. GPUs, 55
 - GPU, 60–63
 - state storage and ALUs, 55–56, 57f

hardware trade-offs

- cache hierarchies and memory systems, 54–55
- cores, 42
- graphics APIs and pixel shaders, 41
- heterogeneity, 42
- multi-core architectures, 51–52
- multi-core CPUs, 42
- multithreading, 48–51
- performance increase, frequency, 43–44
- SIMD and vector processing, 47–48
- superscalar execution, 44

VLIW, 44–47

device fission, extensions

- class *Parallel*, 214–217
- command queues, 218, 220f
- creation, subdevices, 218
- exported extension, 217
- funcWrapper*, 221
- implementation, 221–225
- subdevice partition properties, 218, 219f

double precision, extensions

- C++ Wrapper API, 227–228
- data types, 225–226
- floating point formats, 225
- matrix multiplication implementation, 226–227

execution environment

- command queues, 22–23
- contexts, 22
- events, 23
- flush and finish command, 26
- memory objects, 23–26

execution model

- CPU concurrency models, 16–17
- data-parallel execution, 17–18
- hierarchical concurrency model, 18
- memory structures, 16
- NDRange, 18, 19f

framework, heterogeneous programming, 1–2

interoperability

- image2D memory object, 183
- memory objects, 182
- queue object, 183–184
- synchronization, 183–184
- texture, 182–183

kernels

- compiling kernels, 33, 36
- enqueue function, 28
- extraction, 27
- local memory allocations, 31
- mapping, 31
- textual scope, 31

querying, platform and device, 212

memory model *See* Memory model

parallel computing, 2

platform independence, 41

platform vendors, 2

profiling API, 228–233

program object creation *See* Programs

scope and applicability, 2

specification, models

- defined, 15–16
- execution, 15
- memory, 16
- parallel execution, 16
- platform, 16
- programming, 16

standard, 15

writing kernels

P

Parallel programming

- array multiplication, 5f
- computing, definition, 4
- and concurrency
 - assignments, 7–8
 - data sharing and synchronization, 11
 - message-passing communication, 9–10

- parallelism grains, 10–11
- receiving and processing input, 7
- subsets, program, 8, 8f
- threads and shared memory, 9
- control and data intensive, 1
- data and task-level parallelism, 5–6
- divide-and-conquer methods, 2–3
- elements multiplication, 4
- goals, 2
- GPUs, 3–4
- heterogeneity, 1
- image filtration, FFT, 5f
- multiple processors, 3
- OpenCL, 1–2
- parallelism and concurrency, classes, 4
- reduction concept, 6
- simple sorting and vector-scalar multiply, 3f
- structure, 11–12
- Parallelism grains
 - chunk size selection, 10–11
 - coarse-grained, 10
 - computation ratio, 10–11
 - fine-grained, 10
- Pragma directive, 213–214
- Private memory *See* Memory model
- Profiling events
 - command's queues and status, 236
 - enabling, 236
 - information, 237
 - kernel execution, 237–238
 - valid values enumeration, 236–237
- Programs
 - binary representation, 27
 - build process features, 27
 - dynamic library interface, 26–27
 - kernels, 26
 - runtime compilation, 26–27

Q

- Queuing and global synchronization
 - callbacks, event, 104–107
 - command barriers and markers
 - enqueueMarker, 109
 - event list, 108
 - finish operation, 95–96
 - synchronization, 109
 - task graphs, 109
 - events *See* Events
 - memory consistency
 - commands, 96
 - data, 96

- runtime, 96
- primary points, 94–95
- thread-safe command queues, 94

R

- RGBA format, 180 *See* OpenCL Images

S

- SIMD *See* Single instruction multiple data
- Simultaneous multithreading (SMT), 49, 49f
- Single instruction multiple data (SIMD)
 - ALU operations, 135
 - architecture, 135–136, 135f
 - cores, 52, 53f
 - ISA code and wavefronts, 135
 - OpenCL workloads, 137–138, 138f
 - software control, 132–135
 - texture clause, 137
 - threads, 61
 - vector processing
 - CPUs and GPUs, 48
 - execution advantage, 48
 - parallelism, 47
 - VLIW packets, 136, 136f
- SMT *See* Simultaneous multithreading
- SoC *See* Systems-on-chip
- STI Cell Processor, 54
- Sun Niagara design, 59–60 *See* Central processing units
- Superscalar execution, 44
- Systems-on-chip (SoC)
 - APU
 - benefits, 53–54
 - cell broadband engine processor, 54
 - multi-core design, 53
 - fused processors, 63

T

- Threading building blocks (TBB), 214–215
- Threads and shared memory
 - consistency model, defined, 9
 - definition, 9
 - global view, 9
 - OpenCL, 9
 - shared bus, 9
- Throughput computing, 50–51, 51f
 - See* Multithreading

V

- Vector addition *See* Example applications
- Vendor extensions, 211

Very long instruction word (VLIW)

- ALU packing, 160–161
- architectures, 42
- description, 44–45
- designs, 47
- double precision and integer operations, 136
- DSP chips, 47
- efficiency losses, 46
- four-way, 20
- low-level shader compiler, 61–62
- method, 44–45
- out-of-order execution, 45–46, 45f, 46f
- packet stream, 45–46
- SIMD lane, 53f

Video processing

- CPU decoding, frame
 - callback methodology, 174–175
 - framework and request, 174
 - heterogeneous computing, 173–174
 - multithreading, 175
 - VLC project, 174
 - display to screen
 - double-buffered texture, 181
 - OpenCL/OpenGL interoperability, 181–183
 - GPU decoding, frames
 - OpenVideo, 176–179
 - power consumption, 175–176
 - processing, 176, 176f
 - multiple videos with multiple special effects
 - chain effects, 180
 - event chaining, 180–181
 - OpenCL
 - features, 179
 - kernel, parameters, 179–180
 - RGBA format, 180
- VLIW *See* Very long instruction word

W

Web applications

- client-side, 255–256
- JavaScript code, 260
- visual and interaction, 255

WebCL

- advantages, World Wide Web, 255

- browser environment, 255
 - client-side web applications, 255–256
 - framework designing
 - development, 256
 - device capabilities, 256
 - error management, 256, 257t
 - goal, 256
 - requirement, Web usage, 256
 - garbage collector, 267
 - HTML and JavaScript, 267–268
 - OpenCL
 - color image, gray scale conversion, 260
 - ctx object function, 261–262
 - error reporting/handling, 260, 261
 - input and the output buffers, 261
 - JavaScript vs. C, 260
 - kernel source, 260
 - property array, 260
 - syntax, 260–264
 - performance, 266–267, 267t
 - photo editor, 264–265
 - pilot implementation
 - Firefox, 257–258
 - JavaScript and OpenCL, 258–259
 - portable devices, 268
 - web environment, 268
- Web photo editor *See* Example applications
- ### Workgroups
- definition, 88
 - histogram
 - global memory transactions, 186
 - local memory, 185–186
 - pixels mapping, 186
 - size, optimal, 186–187
 - size selection
 - aligning data, 158
 - caching data to local memory, 152, 154–159
 - efficiency, vector reading, 158–159
 - filter size, 151–152
 - input-output efficiency, 151–152
 - memory access aligning, 156–158
 - optimization approach, 153
 - out-of-bounds access, 153–154, 154f
- Work-items 17, 87

[General Information]

书名=OPENCL异构计算

作者=(美)贾斯特著

页数=277

出版社=北京市:清华大学出版社

出版日期=2012.06

SS号=13012464

DX号=000008288818

URL=<http://book.szdnnet.org.cn/bookDetail.jsp?dxNumber=000008288818&d=7E810E0C6074D238F6B8072E09F64BC4>