MORGAN & CLAYPOOL PUBLISHERS

通用图形处理器架构

托尔·M·阿莫德 威尔逊·维·伦·冯 蒂莫西·G·罗杰斯

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, *Series Editor*

通用图形处理器架构

计算机架构的合成讲座

编辑器
玛格丽特·马尔托诺西, ***Princeton University***

创始编辑荣誉称号马克·D·希尔,

University of Wisconsin, Madison

Synthesis Lectures on Computer Architecture 发布关于设计、分析、选择和互连硬件组件以创建满足功能、性能和成本目标的计算机的科学和艺术的 50 到 100 页的出版物。范围将主要遵循顶级计算机体系结构会议的范围，如 ISCA、HPCA、MICRO 和 ASPLOS。

通用图形处理器架构 Tor M. Aamodt，Wilson Wai Lun Fung 和 Timothy G. Rogers 2018

编译异构系统的算法 史蒂芬·贝尔，蒲京，詹姆斯·赫加提，马克·霍罗维茨 2018

虚拟内存的架构和操作系统支持 Abhishek Bhattacharjee 和 Daniel Lustig 2017

深度学习与计算机架构 Brandon Reagen, Robert Adolf, Paul Whatmough, Gu-Yeon Wei, 和 David Brooks 2017

片上网络（第二版）Natalie Enright Jerger, Tushar Krishna, 和 Li-Shiuan Peh 2017

时空计算与时序神经网络 詹姆斯·E·史密斯 2017

硬件和软件支持虚拟化 埃杜阿尔·布尼翁，杰森·尼赫，丹·萨弗里尔 2017

数据中心设计与管理：计算机架构师的视角 本杰明·C·李 2016

内存层次结构中的压缩入门 Somayeh Sardashti, Angelos Arelakis, Per Stenström, 和 David A. Wood 2015

研究基础设施用于硬件加速器 余坤 索非亚·邵和 大卫·布鲁克斯 2015

分析分析 Rajesh Bordawekar、Bob Blainey 和 Ruchir Puri 2015

可定制计算 陈宇婷，孔杰森，迈克尔·吉尔，格伦·瑞曼，和肖秉君 2015

堆叠架构 袁晔 和 赵继深 2015

单指令多数据执行 克里斯托弗·J·休斯 2015

节能计算机体系结构： Recent Advances Magnus Själander, Margaret Martonosi, 和 Stefanos Kaxiras 2014

FPGA加速计算机系统的仿真 Hari Angepat, Derek Chiou, Eric S. Chung, 和 James C. Hoe 2014

硬件预取入门 巴巴克·法尔萨菲和 托马斯·F·温尼奇 2014

片上光互连：计算机架构师的视角 克里斯托弗·J·尼塔，马修·K·法伦斯，文卡特什·阿凯拉 2013

计算机架构中的优化与数学建模 托尼·诺瓦茨基，迈克尔·费里斯，卡尔西凯扬·桑卡拉林甘，克里斯蒂安·埃斯坦，尼拉伊·瓦伊什和大卫·伍德 2013

计算机架构师的安全基础 Ruby B. Le
e 2013

数据中心作为计算机：仓库规模机器设计的入门，第二版 Luiz André Barroso, Jimmy Clidaras 和 Urs Hölzle 2013

共享内存同步 迈克尔·L·斯科特
2013

电压变化的弹性架构设计 Vijay Janapa Reddi 和
Meeta Sharma Gupta 2013

多线程架构 马里奥·涅米罗夫斯基
和 迪恩·M·图尔森 2013

通用图形处理的性能分析与调优 单位
(GPGPU) 2012年，Hyesoon Kim, Richard Vuduc, Sara Baghsorkhi, Je
e Choi 和 Wen-mei Hwu

自动并行化：基本编译器技术概述 塞缪尔·P·米德基夫 2012

相变存储器：从设备到系统 Moinuddin K. Qureshi, Sudhan
va Gurumurthi 和 Bipin Rajendran 2011

多核缓存层次结构 Rajeev Balasubramonian、Norman P. Jouppi
和 Naveen Muralimanohar 2011

内存一致性和缓存一致性入门 丹尼尔·J·索林、马克·D·希尔、戴维·A·伍德 2011

动态二进制修改：工具、技术与应用 Kim Hazelwood 2011

量子计算机架构师, 第二版 Tzvetan S. Metodi, Arvin I. Faruque, 和 Frederic T. Chong 2011

高性能数据中心网络：架构、算法和机遇 丹尼斯·阿布茨和约翰·金 2011

处理器微架构：一种实现视角 Antonio González, Fernando Latorre 和 Grigorios Magklis 2010

事务性内存，第二版 Tim Harris, James Larus 和 Ravi Rajwar 2010

计算机架构性能评估方法 Lieven Eeckhout 2010

可重构超级计算导论 马尔 к о·兰扎戈塔、斯蒂芬·比克和罗伯特·罗森伯格 2009

片上网络 Natalie Enright Jerger 和 Li-Shiuan Peh 2009

内存系统：你无法避免它，你无法忽视它，你无法伪装它 布鲁斯·雅各布 2009

故障容错计算机架构 丹尼尔·J·索林 2009

数据中心作为计算机：仓库规模机器设计简介

路易斯·安德烈·巴罗索和乌尔斯·
赫尔茨尔 2009

计算机架构的节能技术 斯特法诺斯·卡基拉斯 和 玛
格丽特·马尔托诺西 2008

芯片多处理器架构：提高吞吐量和延迟的技术 Kunle Olukotun, Lance Hammon
d 和 James Laudon 2007

事务性内存 詹姆斯·R·拉鲁
斯 和 拉维·拉杰瓦尔 2006

量子计算与计算机架构师 Tzvetan S. Metodi
和 Frederic T. Chong 2006

通用图形处理器架构 托尔·M·阿莫特，威尔逊·卫伦·冯，和蒂莫西·G·罗杰斯

www.morganclaypool.com

ISBN: 9781627059237 平装 ISBN: 97
81627056182 电子书 ISBN: 97816817
33586 精装

DOI 10.2200/S00848ED1V01Y201804CAC044

在摩根与克莱普尔出版社系列中的一篇出版物
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

讲座 #44
系列编辑：Margaret Martonosi，*Princeton University* 创始编辑名誉：M
ark D. Hill，*University of Wisconsin, Madison* 系列 ISSN 印刷版 1935-323
5 电子版 1935-3243

通用图形处理器架构

托尔·M·阿莫德特
University of British Columbia
威尔逊·韦伦·冯
Samsung Electronics
蒂莫西·G·罗杰斯
Purdue University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #44

MORGAN & CLAYPOOL PUBLISHERS

摘要

最初为了支持视频游戏而开发的图形处理器单元（GPU）现在越来越多地用于从机器学习到加密货币挖矿等通用（非图形）应用。与中央处理单元（CPU）相比，GPU通过将更大比例的硬件资源专用于计算，可以实现更高的性能和效率。此外，它们的通用可编程性使得现代GPU在与特定领域加速器的比较中对软件开发者更具吸引力。本书为有意研究支持通用计算的GPU架构的人提供了入门介绍。它收集了当前仅在各种离散来源中找到的信息。作者主导了GPGPU-Sim模拟器的开发，该模拟器在关于GPU架构的学术研究中被广泛使用。

本书的第一章描述了GPU的基本硬件结构，并简要概述了它们的历史。第二章总结了与本书其余部分相关的GPU编程模型。第三章探讨了GPU计算核心的架构。第四章探讨了GPU内存系统的架构。在描述现有系统的架构后，第三章和第四章提供了相关研究的概述。第五章总结了影响计算核心和内存系统的交叉研究。

本书应为希望了解用于加速通用应用程序的图形处理单元（GPU）架构的人士提供宝贵的资源，并为希望获得关于如何改善这些GPU架构的快速增长的研究领域入门的人士提供帮助。

关键词

GPGPU，计算机架构

内容

Preface

本书旨在帮助希望理解图形处理单元（GPU）架构的人士，并对不断增长的研究领域提供入门介绍，该领域探讨如何改进GPU设计。假定读者对计算机架构概念如流水线和缓存有一定了解，并对进行与GPU架构相关的研究和/或开发感兴趣。这类工作往往关注不同设计之间的权衡，因此本书旨在提供对这些权衡的洞察，以便读者能够避免通过反复试验来学习经验丰富的设计师已经知道的内容。

为了实现这一目标，本书将当前在专利、产品文档和研究论文等各种不同来源中找到的许多相关信息汇集到一个资源中。我们希望这将有助于减少刚开始进行自我研究的学生或从业者的生产时间。

虽然这本书涵盖了当前GPU设计的各个方面，但它也试图“综合”已发布的研究。这在一定程度上是出于必要，因为供应商对特定GPU产品的微架构几乎没有发言。在描述“基线”GPGPU架构时，本书既依赖于已发布的产品描述（期刊论文、白皮书、手册），在某些情况下还依赖于专利中的描述。在专利中找到的细节可能与实际产品的微架构有很大不同。在某些情况下，微基准研究为研究人员澄清了一些细节，但在其他情况下，我们的基线代表了我们基于公开可用信息的“最佳猜测”。尽管如此，我们相信这会有所帮助，因为我们的重点是理解已经研究过或者未来研究中可能有趣的架构权衡。

本书的几个部分集中于总结近年来关于改进GPU架构的众多研究论文。随着这一主题在近几年显著增长的受欢迎程度，书中要覆盖的内容实在太多。因此，我们不得不对涵盖的内容和遗漏的内容做出困难的选择。

托尔·M·阿莫德特，威尔逊· Wai Lun Fung 和蒂莫西·G·罗杰斯
2018年4月

致谢

我们要感谢我们的家人在写这本书期间的支持。此外，我们感谢我们的出版商迈克尔·摩根和编辑玛格丽特·马尔托诺西，在这本书的写作过程中所展现出的极大耐心。我们还要感谢Carole-Jean Wu、Andreas Moshovos、Yash Ukidave、Aamir Raihan和Amruth Sandhupatla对本书早期草稿提供的详细反馈。最后，我们要感谢马克·希尔分享他对撰写《综合讲座》的策略和对本书的具体建议的想法。

托尔·M阿莫德特，威尔逊· Wai Lun Fung 和蒂莫西·G·罗杰斯
2018年4月

CHAPTER 1

介绍

本书探讨了图形处理单元（GPU）的硬件设计。GPU最初是为了实现实时渲染而引入的，重点关注视频游戏。如今，GPU无处不在，从智能手机、笔记本电脑、数据中心一直到超级计算机。事实上，对Apple A8应用处理器的分析显示，它在集成GPU上投入的芯片面积比中央处理单元（CPU）核心更多 [A8H]。对越来越逼真的图形渲染的需求是GPU创新的初始驱动力 [Montrym and Moreton, 2005]。虽然图形加速仍然是它们的主要目的，但GPU越来越多地支持非图形计算。一个引起关注的突出例子是GPU在机器学习系统开发和部署中的日益增长的使用 [NVIDIA Corp., 2017]。因此，本书的重点是与提高非图形应用性能和能效相关的特性。

本介绍性章节简要概述了GPU。我们在第1.1节开始时考虑计算加速器的更广泛类别的动机，以了解GPU与其他选项的比较。然后，在第1.2节中，我们快速概述了当代GPU硬件。最后，第1.4节提供了本书其余部分的路线图。

1.1 计算加速器的前景

几十年来，后续一代计算系统的每美元性能呈指数增长。其根本原因是晶体管尺寸减小、硬件架构改进、编译器技术进步和算法改进的结合。据一些估算，这些性能提升中有一半是由于晶体管尺寸的减少，导致设备运行速度更快 [Hennessy 和 Patterson, 2011]。然而，自大约2005年以来，晶体管的缩放未能遵循现在被称为德纳德缩放的经典规则 [Dennard et al., 1974]。一个重要的结果是，随着设备变小，时钟频率的提高变得更加缓慢。要提高性能，需要寻找更高效的硬件架构。

通过利用硬件专业化，能将能效提高多达 500× [Hameed et al., 2010]。正如 Hameed 等人所示，实现这种效率提升有几个关键方面。转向向量硬件，例如 GPU 中的硬件，可以通过消除指令处理的开销，实现大约 10× 的效率提升。硬件专业化所带来的剩余大部分增益是通过最小化数据移动实现的。

可以通过引入复杂的操作来实现，这些操作执行多个算术运算，同时避免访问大型内存数组，如寄存器文件。

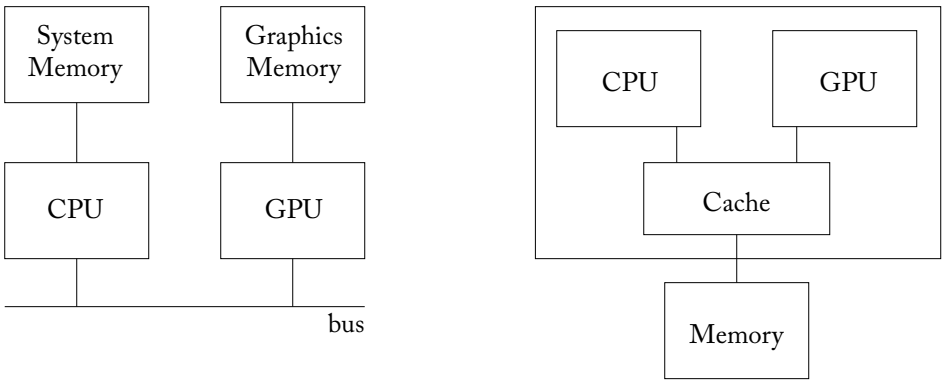
当今计算机架构师面临的一项关键挑战是找到更好的方法来平衡通过使用专用硬件获得的效率提升与支持广泛程序所需的灵活性之间的需求。在缺乏架构的情况下，只有能够为大量应用高效运行的算法才会被使用。一个新兴的例子是专门支持深度神经网络的硬件，例如谷歌的张量处理单元 [Jouppi et al., 2017]。虽然机器学习似乎很可能占用计算硬件资源的很大一部分，并且这些资源可能会迁移到专用硬件上，但我们认为仍然需要有效支持以传统编程语言编写的软件表达的计算。

一个对GPU计算强烈兴趣的原因是现代GPU支持图灵完备的编程模型。我们所说的图灵完备是指只要有足够的时间和内存，任何计算都可以被运行。相对于专用加速器，现代GPU是灵活的。对于能够充分利用GPU硬件的软件，GPU的效率可以比CPU高一个数量级 [Lee et al., 2010]。这种灵活性和效率的组合是非常理想的。因此，现在许多顶级超级计算机在峰值性能和能源效率方面都采用了GPU [top]。随着产品代的不断发展，GPU制造商不断完善GPU架构和编程模型，以提高灵活性，同时改善能源效率。

1.2 GPU硬件基础

通常第一次接触GPU的人会问它们是否最终会完全取代CPU。这似乎不太可能。在现有系统中，GPU不是独立的计算设备。相反，它们与CPU结合在一起，或者在单个芯片上，或者通过将仅包含GPU的插卡插入包含CPU的系统中。CPU负责启动GPU上的计算并传输数据到GPU和从GPU。CPU和GPU之间这种劳动分工的一个原因是，计算的开始和结束通常需要访问输入/输出(I/O)设备。虽然目前有一些正在进行的努力，以开发提供直接在GPU上提供I/O服务的应用程序编程接口（APIs），但到现在为止，这些都假定附近有一个CPU [Kim et al., 2014, Silberstein et al., 2013]。这些APIs通过提供便利的接口来隐藏管理CPU和GPU之间通信的复杂性，而不是完全消除对CPU的需求。为什么不消除CPU呢？用于访问I/O设备和提供操作系统服务的软件似乎缺乏某些功能，例如大规模并行性，这使它们不适合在GPU上运行。因此，我们首先考虑CPU和GPU之间的互动。

图 1.1 显示了一个典型系统的抽象图，该系统包含 CPU 和 GPU。左侧是一个典型的离散 GPU 设置，包括连接 CPU 和 GPU 的总线（例如 PCIe），适用于 NVIDIA 的 Volta GPU 等架构；右侧是典型集成 CPU 和 GPU 的逻辑图，例如 AMD 的 Bristol Ridge APU 或移动 GPU。注意，包括离散 GPU 的系统有针对 CPU（通常称为系统内存）和 GPU（通常称为设备内存）的独立 DRAM 内存空间。用于这些内存的 DRAM 技术通常是不同的（CPU 使用 DDR，GPU 使用 GDDR）。CPU DRAM 通常针对低延迟访问进行优化，而 GPU DRAM 则针对高带宽进行优化。相比之下，集成 GPU 的系统只有一个 DRAM 内存空间，因此必然使用相同的内存技术。由于集成 CPU 和 GPU 通常出现在低功耗移动设备上，因此共享的 DRAM 内存通常针对低功耗进行了优化（例如，LPDDR）。



(a) System with discrete GPU

(b) Integrated CPU and GPU

图 1.1：GPU 计算系统包括 CPU。

一个GPU计算应用在CPU上开始运行。通常，应用的CPU部分将分配和初始化一些数据结构。在较旧的NVIDIA和AMD离散GPU上，GPU计算应用的CPU部分通常会在CPU和GPU内存中为数据结构分配空间。对于这些GPU，应用的CPU部分必须协调将数据从CPU内存移动到GPU内存。更新的离散GPU（例如，NVIDIA的Pascal架构）具有软件和硬件支持，可以自动将数据从CPU内存传输到GPU内存。这可以通过利用虚拟内存支持来实现[Gelado等，2010]，无论是在CPU还是GPU上。NVIDIA称之为“统一内存”。在CPU和GPU集成到同一芯片并共享相同内存的系统中，不需要程序员控制从CPU内存到GPU内存的复制。然而，由于CPU和GPU使用缓存和

某些缓存可能是私有的，可能会出现缓存一致性问题，这需要硬件开发人员来解决 [Power et al., 2013b]。

在某些时候，CPU 必须在 GPU 上启动计算。在当前系统中，这通过在 CPU 上运行的驱动程序来实现。在启动 GPU 上的计算之前，GPU 计算应用程序指定应在 GPU 上运行的代码。该代码通常被称为内核（详细信息见第 2 章）。同时，GPU 计算应用程序的 CPU 部分还指定应运行多少线程，以及这些线程应在何处查找输入数据。要运行的内核、线程数量和数据位置通过在 CPU 上运行的驱动程序传达给 GPU 硬件。驱动程序将信息翻译并将其放置在 GPU 可访问的内存中，位置是 GPU 被配置为查找的地方。然后，驱动程序通知 GPU 它有新计算需要运行。

现代GPU由多个核心组成，如图1.2所示。NVIDIA称这些核心为*streaming multiprocessors*，而AMD称之为*compute units*。每个GPU核心执行与已启动在GPU上运行的内核相对应的单指令多线程（SIMT）程序。每个GPU上的核心通常可以运行大约千个线程。运行在单个核心上的线程可以通过临时存储器进行通信，并使用快速的屏障操作进行同步。每个核心通常还包含一级指令和数据缓存。这些缓存充当带宽滤波器，以减少发送到内存系统下层的流量。运行在核心上的大量线程用于隐藏访问内存的延迟，尤其在数据未找到于一级缓存时。

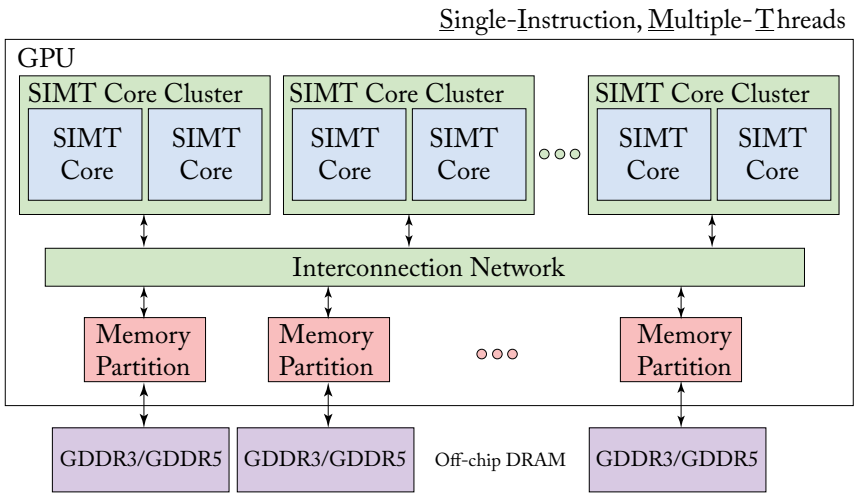


图 1.2：一种通用的现代 GPU 架构。

为了维持高计算吞吐量，有必要平衡高计算吞吐量与高内存带宽。这反过来又需要内存系统中的并行性。

在GPU中，这种并行性通过包含多个内存通道来实现。通常，每个内存通道都有一个与之相关的内存分区中的最后一级缓存部分。GPU核心和内存分区通过片上互连网络连接，例如交叉开关。也可以有其他组织方式。例如，直接与GPU在超级计算市场竞争的英特尔Xeon Phi，将最后一级缓存分配给核心。

GPU 在高并行工作负载下通过将更大比例的晶片面积分配给算术逻辑单元，并相应地减少控制逻辑的面积，可以在单位面积内获得比超标量乱序 CPU 更好的性能。为了深入理解 CPU 和 GPU 架构之间的权衡，Guz 等人 [2009] 开发了一个深入的分析模型，展示了性能如何随着线程数量的变化而变化。为了保持模型简单，他们假设一个简单的缓存模型，其中线程不共享数据，并且具有无限的外部内存带宽。图 1.3 重现了他们论文中的一个图，说明了他们模型中发现的一个有趣权衡。当一个大型缓存在少量线程之间共享时（正如多核 CPU 的情况），性能随着线程数量的增加而增加。然而，如果线程数量增加到缓存无法容纳整个工作集的程度，性能会下降。随着线程数量进一步增加，性能随着多线程隐藏长时间外部延迟的能力而增加。GPU 架构在该图的右侧表示。GPU 设计用来通过采用多线程来容忍频繁的缓存未命中。

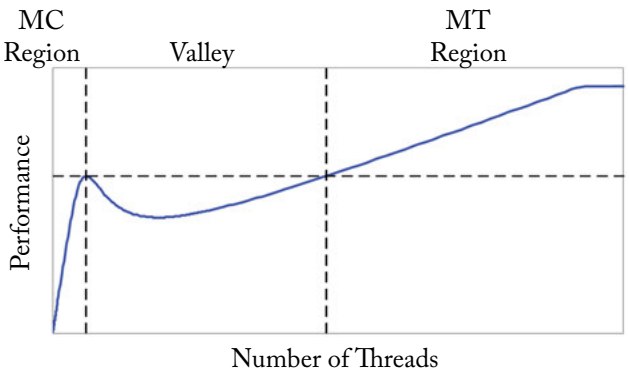


图 1.3：基于分析模型的多核 (MC) CPU 架构和多线程 (MT) 架构（如 GPU）之间性能权衡的分析表明，当线程数量不足以覆盖芯片外内存访问延迟时，可能会出现“ 性能谷 ”（基于 Guz 等人 [2009] 的图 1）。

随着德纳德缩放的结束 [Horowitz et al., 2005]，提高能量效率已成为计算机架构研究中的主要创新驱动因素。一个关键观察是，访问大型内存结构可能消耗与计算一样多或更多的能量。

例如，表1.1提供了在45纳米工艺技术中各种操作的能量数据[Han et al., 2016]。在提出新型GPU架构设计时，考虑能耗是很重要的。为此，最近的GPGPU架构模拟器如GPGPU-Sim [Bakhoda et al., 2009] 纳入了能量模型[Leng et al., 2013]。

表1.1：45纳米工艺技术各种操作的能耗（基于Han等人[2016]的表1）

Operation	Energy [pJ]	Relative Cost
32 bit int ADD	0.1	1
32 bit float ADD	0.9	9
32 bit int MULT	3.1	31
32 bit float MULT	3.7	37
32 bit 32KB SRAM	5	50
32 bit DRAM	640	6400

1.3 GPU的简史

这一部分简要描述了图形处理单元的发展历史。计算机图形学在1960年代开始出现，代表性项目包括伊凡·苏瑟兰的Sketchpad [Sutherland, 1963]。从最初的阶段开始，计算机图形学就已经成为动画离线渲染的重要组成部分，同时也促进了实时渲染的发展，以用于视频游戏。早期的视频卡由1981年的IBM单色显示适配器（MDA）开始，它仅支持文本。随后，视频卡引入了2D加速，接着是3D加速。除了视频游戏，3D加速器还针对计算机辅助设计。早期的3D图形处理器，如NVIDIA GeForce 256，相对固定功能。NVIDIA在2001年推出的GeForce 3中引入了GPU的可编程性，包括顶点着色器[Lindholm et al., 2001]和像素着色器。研究人员迅速学习如何使用这些早期的GPU实现线性代数，将矩阵数据映射到纹理中并应用着色器[Krüger and Westermann, 2003]。同时，关于将通用计算映射到GPU的学术工作也紧随其后，这样程序员便不需要了解图形知识[Buck et al., 2004]。这些努力激励了GPU制造商在图形之外直接支持通用计算。第一个商业化的产品是NVIDIA GeForce 8系列。GeForce 8系列引入了几个创新，包括支持从着色器写入任意内存地址和用于限制离芯带宽的临时存储器，这在早期的GPU中是缺乏的。下一个创新是通过NVIDIA的Fermi架构启用读写数据的缓存。后续的改进包括AMD的Fusion架构，它将CPU和GPU集成在同一芯片上，以及动态并行性，使得

从 GPU 本身启动线程。最近，NVIDIA 的 Volta 引入了专门针对机器学习加速的特性，如张量核心。

1.4 书籍大纲

本书的其余部分组织如下。

在设计硬件时，考虑其所支持的软件是很重要的。因此，在第二章中，我们提供了编程模型、代码开发过程和编译流程的简要总结。

在第三章中，我们探讨支持千线程执行的单个GPU核心的架构。我们逐步建立对支持高吞吐量和灵活编程模型所涉及权衡的越来越详细的理解。本章最后总结了与GPU核心架构相关的最新研究，以帮助快速让新进入该领域的人了解情况。

在第4章中，我们探讨了内存系统，包括GPU核心内的一级缓存和内存分区的内部组织。理解GPU的内存系统非常重要，因为在GPU上运行的计算通常受限于外部内存带宽。本章最后总结了与GPU内存系统架构相关的最新研究。

最后，第五章概述了其他关于GPU计算架构的研究，这些研究并不完全符合第三章或第四章的内容。

CHAPTER 2

编程模型

本章的目标是提供足够的上下文，以便那些没有GPU先前经验的人能够跟上后续章节的讨论，了解GPU如何用于非图形计算。我们在这里关注基本材料，将更深入的内容留给其他参考资料（例如，[Kirk 和 Wen-Mei, 2016]）。存在许多可以用于架构研究的GPU计算基准套件。学习如何编程GPU与对GPU计算感兴趣的计算机架构师相关，以便更好地理解硬件/软件接口，但如果您想在研究中探索更改硬件/软件接口，这就变得至关重要。在后者的情况下，现有基准可能不存在，因此可能需要创建，可能通过修改现有GPU计算应用程序的源代码。例如，探索在GPU上引入事务性内存（TM）的研究需要这样做，因为当前的GPU不支持TM（参见第5.3节）。

现代 GPU 采用宽 SIMD 硬件以利用 GPU 应用中的数据级并行。GPU 计算 API，如 CUDA 和 OpenCL，并没有直接将这些 SIMD 硬件暴露给程序员，而是提供了一种类似 MIMD 的编程模型，允许程序员在 GPU 上启动大量标量线程。这些标量线程可以遵循其独特的执行路径，并可以访问任意内存位置。在运行时，GPU 硬件以锁步方式在 SIMD 硬件上执行标量线程组，在 AMD 术语中称为 *warps*（或 *wavefronts*，以利用它们的规律性和空间局部性。这种执行模型称为单指令、多线程 (SIMT) [Lindholm et al., 2008a, Nickolls and Reusch, 1993]。

本章的其余部分将进一步讨论，并按如下方式组织。 在第2.1节中，我们探讨了最近GPU编程模型所使用的概念执行模型，并对过去十年发布的典型GPU的执行模型进行了简要总结。 在第2.2节中，我们探讨了GPU计算应用的编译过程，并简要回顾了GPU指令集架构。

2.1 执行模型

一个 GPU 计算应用在 CPU 上开始执行。对于离散 GPU，应用程序的 CPU 部分通常会分配内存以用于 GPU 上的计算，然后将输入数据传输到 GPU 内存中，最后在 GPU 上启动计算内核。对于集成 GPU，仅需要最后一步。计算内核是

由（通常）数千个线程组成。每个线程执行相同的程序，但可能会根据计算结果遵循该程序的不同控制流。下面我们将通过一个用CUDA编写的特定代码示例详细考虑这个流。在接下来的章节中，我们将从汇编层面看执行模型。我们的讨论不会侧重于GPU编程模型的性能方面。然而，Seo等人[2011]在OpenCL（一个类似于CUDA的可以编译到许多架构的编程模型）的背景下提出了一个有趣的观察：针对一个架构（例如GPU）进行精心优化的代码可能在另一个架构（例如CPU）上表现不佳。

图 2.1 提供了用于 CPU 实现著名操作 *single-precision scalar value A times vector value X plus vector value Y* 的 C 代码，这被称为 SAXPY。SAXPY 是著名的基础线性代数软件（BLAS）库的一部分 [Lawson et al., 1979]，对于实现更高层次的矩阵操作如高斯消去 [McCool et al., 2012] 非常有用。由于其简单性和实用性，它常常被用作教授计算机体系结构的例子 [Hennessy and Patterson, 2011]。图 2.2 提供了相应的 SAXPY 的 CUDA 版本，执行在 CPU 和 GPU 之间分配。

图2.2中的示例演示了CUDA和相关编程模型（例如，OpenCL [Kaeli et al., 2015]）提供的抽象。代码从函数`main()`开始执行。为了使示例专注于与GPU计算相关的细节，我们省略了分配和初始化数组`x`和`y`的细节。接下来，调用函数`saxpy_serial`。该函数的输入参数包括向量`x`和`y`中的元素数量（在参数`n`中），参数`a`中的标量值，以及用于表示向量`x`和`y`的数组指针。该函数对数组`x`和`y`的每个元素进行迭代。在每次迭代中，代码在第4行使用循环变量`i`读取值`x[i]`和`y[i]`，将`x[i]`乘以`a`然后加上`y[i]`，接着用结果更新`x[i]`。为简单起见，我们省略了CPU如何使用函数调用结果的细节。

接下来，我们考虑SAXPY的CUDA版本。与传统的C或C++程序类似，图2.2中的代码通过在CPU上运行函数`main()`开始执行。我们将首先突出GPU执行特有的方面，而不是逐行分析这段代码。

在GPU上执行的线程是由一个函数指定的计算 *kernel* 的一部分。在图2.2中展示的SAXPY的CUDA版本中，第1行的CUDA关键字 `__global__` 表示内核函数 `saxpy` 将在GPU上运行。在图2.2的示例中，我们对图2.1中的"`for`"循环进行了并行化。具体来说，图2.1中原始的仅CPU C代码第4行的"`for`"循环的每次迭代都被翻译成在图2.2第3-5行上运行的单个线程。

一个计算内核通常由数千个线程组成，每个线程都开始于运行相同的函数。在我们的例子中，CPU 在第 17 行使用 CUDA 的内核配置语法在 GPU 上开始计算。内核配置语法看起来很像 C 语言中的函数调用，只是多了一些额外的信息来指定包含的线程数。

```
void saxpy_serial(int n, float a, float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}
main() {
  float *x, *y;
  int n;
  // omitted: allocate CPU memory for x and y and initialize contents
  saxpy_serial(n, 2.0, x, y); // Invoke serial SAXPY kernel
  // omitted: use y on CPU, free memory pointed to by x and y
}
```

图 2.1：传统 CPU 代码（基于 Harris [201 2]).

```
__global__ void saxpy(int n, float a, float *x, float *y)
{
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if(i<n)
     y[i] = a*x[i] + y[i];
}
int main() {
  float *h_x, *h_y;
  int n;
  // omitted: allocate CPU memory for h_x and h_y and initialize contents
  float *d_x, *d_y;
  int nblocks = (n + 255) / 256;
  cudaMalloc( &d_x, n * sizeof(float) );
  cudaMalloc( &d_y, n * sizeof(float) );
  cudaMemcpy( d_x, h_x, n * sizeof(float), cudaMemcpyHostToDevice );
  cudaMemcpy( d_y, h_y, n * sizeof(float), cudaMemcpyHostToDevice );
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy( h_x, d_x, n * sizeof(float), cudaMemcpyDeviceToHost );
  // omitted: use h_y on CPU, free memory pointed to by h_x, h_y, d_x, and d_y
}
```

图 2.2：CUDA 代码（基于哈里斯 [2012]）。

在三重尖括号之间 (<<<>>>)。构成计算内核的线程被组织成由 *grid* 组成的层次结构，该层次结构由 *thread blocks* 和 *warps* 构成。在 CUDA 编程模型中，单个线程执行其操作数为标量值的指令（例如，32 位浮点数）。为了提高效率，典型的 GPU 硬件将线程组一起以锁步方式执行。这些组由 NVIDIA 称为 *warps*，由 AMD 称为 *wavefronts*。NVIDIA 的 warp 由 32 个线程组成，而 AMD 的 wavefront 则由 64 个线程组成。Warp 被分组到一个称为协作线程数组 (CTA) 或线程块的更大单位中，由 NVIDIA 定义。第 17 行指出计算内核应启动一个由 `nblocks` 个线程块组成的单一网格，其中每个线程块包含 256 个线程。CPU 代码传递给内核配置语句的参数被分配给 GPU 上每个运行线程的实例。

许多今天的移动设备系统芯片将CPU和GPU集成到一个芯片中，这与今天笔记本和台式计算机中的处理器类似。然而，传统上，GPU拥有自己的DRAM内存，这种情况在用于机器学习的数据中心中的GPU中依然存在。我们注意到英伟达推出了统一内存（Unified Memory），它可以透明地从CPU内存更新GPU内存，从GPU内存更新CPU内存。在启用统一内存的系统中，运行时和硬件负责代表程序员执行复制操作。鉴于机器学习的兴趣日益增长，并且本书的目标是理解硬件，在我们的示例中，我们考虑由程序员管理的独立GPU和CPU内存的一般情况。

基于许多 NVIDIA CUDA 示例中使用的风格，我们为在 CPU 内存中分配的内存命名指针变量时使用前缀 `h_`，为在 GPU 内存中分配的内存指针使用前缀 `d_`。在第 13 行，CPU 调用 CUDA 库函数 `cudaMalloc`。该函数调用 GPU 驱动程序并请求在 GPU 上分配内存供程序使用。对 `cudaMalloc` 的调用将 `d_x` 设置为指向一个 GPU 内存区域，该区域包含足够的空间来存放 `n` 个 32 位浮点值。在第 15 行，CPU 调用 CUDA 库函数 `cudaMemcpy`。该函数调用 GPU 驱动程序并请求将指向的 CPU 内存中数组的内容（由 `h_x` 指向）复制到 GPU 内存中数组（由 `d_x` 指向）。

让我们最后关注于在GPU上执行线程的过程。在并行编程中，一种常用策略是将每个线程分配到一部分数据。为了方便实施这一策略，GPU上的每个线程可以在线程块的网格中查找自己的身份。CUDA中实现这一机制使用了网格、块和线程标识符。在CUDA中，网格和线程块具有x、y和z维度。在执行过程中，每个线程在网格和线程块中都有一个固定的、唯一的非负整数的组合坐标 x、y 和 z。每个线程块在网格内有 x、y 和 z 坐标。类似地，每个线程在线程块中有 x、y 和 z 坐标。这些坐标的范围由内核配置语法（第17行）设置。在我们的示例中，y 和 z 维度没有指定，因此所有线程的 y 和 z 线程块和线程坐标均为零。在第3行，`threadIdx.x` 的值标识线程在其线程块内的 x 坐标以及 `blockIdx.x`。

指示线程块在其网格内的 x 坐标。值 `blockDim.x` 表示 x 维度中线程的最大数量。在我们的例子中，`blockDim.x` 的值为 256，因为这是在第 17 行指定的值。表达式 `blockIdx.x*blockDim.x + threadIdx.x` 用于计算用于访问数组 `x` 和 `y` 的偏移量 `i`。正如我们所看到的，使用索引 `i`，我们已经为每个线程分配了 `x` 和 `y` 的唯一元素。

在很大程度上，编译器和硬件的结合使得程序员无需关注线程在一个波束中的逐步执行特性。编译器和硬件使得每个线程在一个波束内独立执行的表现成为可能。在图 2.2 的第 4 行，我们比较了索引 `i` 的值与 `n`，数组 `x` 和 `y` 的大小。对于 `i` 小于 `n` 的线程，执行第 5 行。图 2.2 的第 5 行执行图 2.1 中原始循环的一次迭代。在网格中的所有线程完成后，计算内核在第 17 行将控制权返回给 CPU。在第 18 行，CPU 调用 GPU 驱动程序将由 `d_y` 指向的数组从 GPU 内存复制回 CPU 内存。

CUDA编程模型的一些额外细节在SAXPY示例中没有说明，但我们稍后会讨论，具体如下。

在CTA内部，各线程可以通过每个计算核心的临时存储器高效地相互通信。NVIDIA称这种临时存储器为*shared memory*。每个流式多处理器（SM）包含一个共享内存。共享内存中的空间在该SM上运行的所有CTA之间划分。AMD的图形核心下一代（GCN）架构[AMD, 2012]包含类似的临时存储器，AMD称其为*local data store* (LDS)。这些临时存储器较小，每个SM的大小在16到64 KB之间，并向程序员暴露为不同的内存空间。程序员通过在源代码中使用特殊关键字（例如，CUDA中的“`__shared__`”）来分配临时存储器。临时存储器充当软件控制的缓存。虽然GPU还包含硬件管理的缓存，但通过这样的缓存访问数据可能会导致频繁的缓存未命中。当程序员能够识别出频繁且可预测地重用的数据时，应用程序将受益于使用临时存储器。与NVIDIA的GPU不同，AMD的GCN GPU还包括一个所有GPU核心共享的*global data store* (GDS)临时存储器。临时存储器在图形应用中用于在不同的图形着色器之间传递结果。例如，LDS用于在GCN中的顶点着色器和像素着色器之间传递参数值[AMD, 2012]。

在一个CTA内的线程可以使用硬件支持的屏障指令高效同步。不同CTA中的线程可以进行通信，但需要通过一个所有线程都可以访问的全局地址空间来实现。访问这个全局地址空间在时间和能量上通常比访问共享内存更昂贵。

NVIDIA在Kepler代GPU中引入了CUDA动态并行性（CDP）[NVIDIA公司，a]。CDP的动机是观察到数据密集型不规则应用程序可能导致运行在GPU上的线程之间负载不平衡，从而导致

GPU硬件被低效利用。在许多方面，动机与动态扭曲形成（DWF）[Fung et al., 2007]及第3.4节中讨论的相关方法类似。

2.2 GPU 指令集架构

在本节中，我们简要讨论将计算内核从高层语言（如CUDA和OpenCL）翻译到由GPU硬件执行的汇编级别以及当前GPU指令集的形式。GPU架构与CPU架构有些不同的一个有趣方面是，GPU生态系统是如何演变以支持指令集的演变。例如，x86微处理器向后兼容于1976年发布的Intel 8086。向后兼容意味着为先前一代架构编译的程序将在下一代架构上运行而无需任何更改。因此，理论上40年前为Intel 8086编译的软件可以在今天的任何x86处理器上运行。

2.2.1 NVIDIA GPU 指令集架构

考虑到有时大量的供应商提供 GPU 硬件（每个供应商都有自己独特的硬件设计），通过 OpenGL 着色语言（OGSL）和微软的高级着色语言（HLSL）实现一层指令集虚拟化变得十分普遍，因为早期的 GPU 开始可编程。当 NVIDIA 在 2007 年初引入 CUDA 时，他们决定遵循类似的路径，并引入了自己的高层虚拟指令集架构用于 GPU 计算，称为并行线程执行 ISA，或 PTX [NVI, 2017]。NVIDIA 在每次发布 CUDA 时都对这一虚拟指令集架构进行了全面的文档记录，以至于本书的作者能够轻松开发 GPGPU-Sim 模拟器以支持 PTX [Bakhoda et al., 2009]。PTX 在许多方面类似于标准的精简指令集计算机（RISC）指令集架构，如 ARM、MIPS、SPARC 或 ALPHA。它还与优化编译器中使用的中间表示有相似之处。一个这样的例子是使用无限的虚拟寄存器集。图 2.3 展示了来自图 2.2 的 SAXPY 程序的 PTX 版本。

在 GPU 上运行 PTX 代码之前，有必要将 PTX 编译为硬件支持的实际指令集架构。NVIDIA 将这一层称为 SASS，短语为“ Streaming ASSembler ”[Cabral, 2016]。将 PTX 转换为 SASS 的过程可以通过 GPU 驱动程序或 NVIDIA 的 CUDA Toolkit 提供的独立程序 `ptxas` 来完成。NVIDIA 对 SASS 的文档并不完全。这使得学术研究人员难以开发能够捕捉所有编译器优化效果的架构模拟器，但让 NVIDIA 免受客户要求在硬件级别提供向后兼容性的压力，从而使得从一代到下一代的指令集架构的完全重新设计成为可能。不可避免地，希望在低级别理解性能的开发者开始创建他们自己的工具来反汇编 SASS。由 Wladimir Jasper van der Laan 发起的第一项此类努力被称为“ decuda ”[van der Lann]，于 2007 年底面世，适用于 NVIDIA 的 GeForce 8 系列（G80），大约是在首批支持 CUDA 的硬件发布后一年内。

```
.visible .entry _Z5saxpyifPfS_(
.param .u32 _Z5saxpyifPfS__param_0,
.param .f32 _Z5saxpyifPfS__param_1,
.param .u64 _Z5saxpyifPfS__param_2,
.param .u64 _Z5saxpyifPfS__param_3
)
{
.reg .pred %p<2>;
.reg .f32 %f<5>;
.reg .b32 %r<6>;
.reg .b64 %rd<8>;


ld.param.u32 %r2, [_Z5saxpyifPfS__param_0];
ld.param.f32 %f1, [_Z5saxpyifPfS__param_1];
ld.param.u64 %rd1, [_Z5saxpyifPfS__param_2];
ld.param.u64 %rd2, [_Z5saxpyifPfS__param_3];
mov.u32 %r3, %ctaid.x;
mov.u32 %r4, %ntid.x;
mov.u32 %r5, %tid.x;
mad.lo.s32 %r1, %r4, %r3, %r5;
setp.ge.s32 %p1, %r1, %r2;
@%p1 bra BB0_2;

cvta.to.global.u64 %rd3, %rd2;
cvta.to.global.u64 %rd4, %rd1;
mul.wide.s32 %rd5, %r1, 4;
add.s64 %rd6, %rd4, %rd5;
ld.global.f32 %f2, [%rd6];
add.s64 %rd7, %rd3, %rd5;
ld.global.f32 %f3, [%rd7];
fma.rn.f32 %f4, %f2, %f1, %f3;
st.global.f32 [%rd7], %f4;

BB0_2:
ret;
}
```

图 2.3：与图 2.2 中的计算内核对应的 PTX 代码（使用 CUDA 8.0 编译）。

decuda 项目对 SASS 指令集进行了充分详细的理解，以至于可以开发出汇编器。这有助于在 GPGPU-Sim 3.2.2 中为 SASS 提供对 NVIDIA GT200 架构的支持 [Tor M. Aamodt 等]。NVIDIA 最终推出了一种名为 `cuobjdump` 的工具，并开始部分记录 SASS。NVIDIA 的 SASS 文档 [NVIDIA Corporation, c] 当前（2018 年 4 月）仅提供了汇编操作码名称的列表，但没有关于操作数格式或 SASS 指令语义的详细信息。最近，随着在机器学习中使用 GPU 的爆炸性增长和对性能优化代码的需求，其他开发者为后续架构（如 NVIDIA 的 Fermi [Yunqing] 和 NVIDIA 的 Maxwell 架构 [Gray]）开发了类似于 decuda 的工具。

图2.4展示了为NVIDIA的Fermi架构编译的SAXPY内核的SASS代码[NVI, 2009]，并通过NVIDIA的`cuobjdump` (CUDA工具包)提取。图2.4的第一列是指令的地址。第二列是汇编，第三列是编码指令。如上所述，NVIDIA只部分文档化了它们的硬件汇编。比较图2.3和图2.4，可以注意到虚拟ISA和硬件ISA层之间的相似性和差异。在高层次上，有一些重要的相似性，例如都为RISC（都使用加载和存储来访问内存），且都使用了条件执行[Allen et al., 1983]。更微妙的差异包括：（1）PTX版本有一个实质上无限的寄存器集合可用，因此每个定义通常使用一个新的寄存器，类似于静态单赋值[Cytron et al., 1991]，而SASS使用有限的寄存器集合；（2）内核参数通过银行化的常量内存传递，该内存可被SASS中的非加载/存储指令访问，而在PTX中，参数被分配到它们自己独立的“ 参数 ”地址空间中。

图2.5展示了由同一版本的CUDA为NVIDIA的Pascal架构生成的SAXPY的SASS代码，并通过NVIDIA的`cuobjdump`提取。比较图2.5与图2.4，很明显NVIDIA的ISA发生了显著变化，包括指令编码方面的变化。图2.5包含一些没有反汇编指令的行（例如，第3行地址为0x0000）。这些是NVIDIA Kepler架构中引入的特殊“ 控制指令 ”，旨在消除使用看板进行显式依赖检查的需要[NVIDIA Corporation，b]。Lai和Seznec [2013]探讨了Kepler架构中控制指令的编码。正如Lai和Seznec [2013]所指出的，这些控制指令似乎类似于Tera计算机系统上的显式依赖前瞻[Alverson等，1990]。Gray描述了他们能够推断出的NVIDIA Maxwell架构的控制指令编码的详细信息。根据Gray，在Maxwell中每三个常规指令就有一个控制指令。这在图2.5中也适用于NVIDIA的Pascal架构。根据Gray，Maxwell上的64位控制指令包含三个组的21位编码，分别针对以下三个指令提供以下信息：暂停计数；yield提示标志；以及写、读和等待依赖屏障。Gray还描述了常规指令上寄存器重用标志的使用，这在图2.5中也可以看到（例如，用于第一个源的`R0.reuse`）。

```
Address     Dissassembly                                    Encoded Instruction
========    ============================================    ======================
/*0000*/         MOV R1, c[0x1][0x100];                     /* 0x2800440400005de4 */
/*0008*/         S2R R0, SR_CTAID.X;                        /* 0x2c00000094001c04 */
/*0010*/         S2R R2, SR_TID.X;                          /* 0x2c00000084009c04 */
/*0018*/         IMAD R0, R0, c[0x0][0x8], R2;              /* 0x2004400020001ca3 */
/*0020*/         ISETP.GE.AND P0, PT, R0, c[0x0][0x20], PT; /* 0x1b0e40008001dc23 */
/*0028*/     @P0 BRA.U 0x78;                                /* 0x40000001200081e7 */
/*0030*/    @!P0 MOV32I R5, 0x4;                            /* 0x18000000100161e2 */
/*0038*/    @!P0 IMAD R2.CC, R0, R5, c[0x0][0x28];          /* 0x200b8000a000a0a3 */
/*0040*/    @!P0 IMAD.HI.X R3, R0, R5, c[0x0][0x2c];        /* 0x208a8000b000e0e3 */
/*0048*/    @!P0 IMAD R4.CC, R0, R5, c[0x0][0x30];          /* 0x200b8000c00120a3 */
/*0050*/    @!P0 LD.E R2, [R2];                             /* 0x840000000020a085 */
/*0058*/    @!P0 IMAD.HI.X R5, R0, R5, c[0x0][0x34];        /* 0x208a8000d00160e3 */
/*0060*/    @!P0 LD.E R0, [R4];                             /* 0x8400000000402085 */
/*0068*/    @!P0 FFMA R0, R2, c[0x0][0x24], R0;             /* 0x3000400090202000 */
/*0070*/    @!P0 ST.E [R4], R0;                             /* 0x9400000000402085 */
/*0078*/         EXIT;                                      /* 0x8000000000001de7 */
```

图 2.4：与图 2.2 中的计算内核对应的低级 SASS 代码（使用 CUDA 8.0 为 NVIDIA Fermi 架构 sm_20 编译）。

整数短乘加指令中的操作数XMAD（见第7行）。这似乎表明，从Maxwell开始，NVIDIA GPU中增加了“ 操作数重用缓存 ”（参见第3.6.1节相关研究）。这个操作数重用缓存似乎使得寄存器值可以在每次主寄存器文件访问中被多次读取，从而降低能耗和/或提高性能。

2.2.2 AMD 图形核心下一代指令集架构

与NVIDIA不同，当AMD推出他们的Southern Islands架构时，他们发布了完整的硬件级ISA规范 [AMD, 2012]。Southern Islands是AMD第一代图形核心下一代（GCN）架构。AMD硬件ISA文档的可用性帮助了学术研究人员开发在较低层次工作的模拟器 [Ubal et al., 2012]。AMD的编译流程还包括一个称为HSAIL的虚拟指令集架构，作为异构系统架构（HSA）的一部分。

AMD 的 GCN 架构与 NVIDIA GPU（包括 NVIDIA 最近的 Volta 架构 [NVIDIA Corp., 2017]）之间的一个关键区别是分开的标量和向量指令。图 2.6 和 2.7 重现了来自 AMD [2012] 的高层 OpenCL（类似于 CUDA）代码及其相应的机器指令示例。

```
Address     Dissassembly                                        Encoded Instruction
========    ================================================    ======================
                                                                /* 0x001c7c00e22007f6 */
/*0008*/       MOV R1, c[0x0][0x20];                            /* 0x4c98078000870001 */
/*0010*/       S2R R0, SR_CTAID.X;                              /* 0xf0c8000002570000 */
/*0018*/       S2R R2, SR_TID.X;                                /* 0xf0c8000002170002 */
                                                                /* 0x001fd840fec20ff1 */
/*0028*/       XMAD.MRG R3, R0.reuse, c[0x0] [0x8].H1, RZ;      /* 0x4f107f8000270003 */
/*0030*/       XMAD R2, R0.reuse, c[0x0] [0x8], R2;             /* 0x4e00010000270002 */
/*0038*/       XMAD.PSL.CBCC R0, R0.H1, R3.H1, R2;              /* 0x5b30011800370000 */
                                                                /* 0x081fc400ffa007ed */
/*0048*/       ISETP.GE.AND P0, PT, R0, c[0x0][0x140], PT;      /* 0x4b6d038005070007 */
/*0050*/    @P0 EXIT;                                           /* 0xe30000000000000f */
/*0058*/       SHL R2, R0.reuse, 0x2;                           /* 0x3848000000270002 */
                                                                /* 0x081fc440fec007f5 */
/*0068*/       SHR R0, R0, 0x1e;                                /* 0x3829000001e70000 */
/*0070*/       IADD R4.CC, R2.reuse, c[0x0][0x148];             /* 0x4c10800005270204 */
/*0078*/       IADD.X R5, R0.reuse, c[0x0][0x14c];              /* 0x4c10080005370005 */
                                                                /* 0x0001c800fe0007f6 */
/*0088*/       IADD R2.CC, R2, c[0x0][0x150];                   /* 0x4c10800005470202 */
/*0090*/       IADD.X R3, R0, c[0x0][0x154];                    /* 0x4c10080005570003 */
/*0098*/       LDG.E R0, [R4];        }                         /* 0xeed4200000070400 */
                                                                /* 0x0007c408fc400172 */
/*00a8*/       LDG.E R6, [R2];                                  /* 0xeed4200000070206 */
/*00b0*/       FFMA R0, R0, c[0x0][0x144], R6;                  /* 0x4980030005170000 */
/*00b8*/       STG.E [R2], R0;                                  /* 0xeedc200000070200 */
                                                                /* 0x001f8000ffe007ff */
/*00c8*/       EXIT;                                            /* 0xe30000000007000f */
/*00d0*/       BRA 0xd0;                                        /* 0xe2400fffff87000f */
/*00d8*/       NOP;                                             /* 0x50b0000000070f00 */
                                                                /* 0x001f8000fc0007e0 */
/*00e8*/       NOP;                                             /* 0x50b0000000070f00 */
/*00f0*/       NOP;                                             /* 0x50b0000000070f00 */
/*00f8*/       NOP;                                             /* 0x50b0000000070f00 */
```

图 2.5：与图 2.2 中的计算内核对应的低级 SASS 代码（使用 CUDA 8.0 为 NVIDIA Pascal 架构编译，sm_60）。

恩岛架构。在图2.7中，标量指令以s_开头，而向量指令以v_开头。在AMD GCN架构中，每个计算单元（例如，SIMT核心）包含一个标量单元和四个向量单元。向量指令在向量单元上执行，并为波前中的每个线程计算不同的32位值。相反，标量指令在标量单元上执行，为波前中的所有线程计算一个共享的32位值。图2.7中所示的例子中，标量指令与控制流处理有关。特别是，exec是一个特殊寄存器，用于对SIMT执行中的个别向量通道的执行进行预判。在第3.1.1节中对GPU上控制流处理的掩蔽使用进行了更详细的描述。在GCN架构中，标量单元的另一个潜在好处是，SIMT程序中的计算某些部分通常会独立于线程ID计算相同的结果（见第3.5节）。

```
float fn0(float a,float b)
{
    if(a>b)
        return(a * a - b);
    else
        return(b * b - a);
}
```

图 2.6：OpenCL 代码（基于 AMD [2012] 中的图 2.2）。

```
// Registers r0 contains "a", r1 contains "b"
// Value is returned in r2
    v_cmp_gt_f32 r0, r1 // a>b
    s_mov_b64 s0, exec  // Save current exec mask
    s_and_b64 exec, vcc, exec // Do "if"
    s_cbranch_vccz label0 // Branch if all lanes fail
    v_mul_f32 r2, r0, r0 // result = a * a
    v_sub_f32 r2, r2, r1 // result = result - b
label0:
    s_not_b64 exec, exec // Do "else"
    s_and_b64 exec, s0, exec // Do "else"
    s_cbranch_execz label1 // Branch if all lanes fail
    v_mul_f32 r2, r1, r1 // result = b * b
    v_sub_f32 r2, r2, r0 // result = result - a
label1:
    s_mov_b64 exec, s0   // Restore exec mask
```

图 2.7：南方岛屿（图形核心下一步）微代码（基于 AMD [2012] 中的图 2.2）。

AMD 的 GCN 硬件指令集手册 [AMD, 2012] 提供了许多关于 AMD GPU 硬件的有趣见解。例如，为了启用长延迟操作的数据依赖解析，AMD 的 GCN 架构包括 `S_WAITCNT` 指令。对于每个波前，有三个计数器：向量内存计数、本地/全局数据存储计数和寄存器导出计数。每个计数器表示给定类型的未完成操作数量。编译器或程序员插入 `S_WAITCNT` 指令，以使波前等待，直到未完成操作的数量降到指定阈值以下。

CHAPTER 3

SIMT核心：指令和寄存器数据流

在本章及下一章中，我们将考察现代GPU的架构和微架构。我们将GPU架构的讨论分为两个部分：(1) 本章考察实现计算的SIMT核心，(2) 下一章查看内存系统。

在它们传统的图形渲染角色中，GPU 访问的数据集，如详细的纹理图，往往大得无法完全缓存于芯片上。为了实现高性能的可编程性，这在图形中是非常重要的，因为这不仅能降低随着图形模式增加而带来的验证成本，还能使游戏开发者更轻松地区分他们的产品 [Lindholm et al., 2001]，因此，有必要采用能够支持大规模离芯带宽的架构。因此，今天的 GPU 同时执行数以万计的线程。尽管每个线程的片上存储器存储量很小，但缓存仍然可以有效减少大量的离芯内存访问。例如，在图形工作负载中，相邻像素操作之间存在显著的空间局部性，这可以通过片上缓存捕获。

图 3.1 说明了本章节讨论的 GPU 流水线的微架构。该图展示了图 1.2 中单个 SIMT 核心的内部组织。流水线可以分为 SIMT 前端和 SIMD 后端。流水线由三个调度“循环”组成，这些循环在一个单一的流水线中协同工作：指令获取循环、指令发出循环和寄存器访问调度循环。指令获取循环包括标记为 Fetch、I-Cache、Decode 和 I-Buffer 的块。指令发出循环包括标记为 I-Buffer、Scoreboard、Issue 和 SIMT Stack 的块。寄存器访问调度循环包括标记为 Operand Collector、ALU 和 Memory 的块。在本章的其余部分，我们将通过考虑与这些循环各自相关的架构关键方面，帮助您全面理解图中每个单独块的功能。

由于全面理解该组织涉及许多细节，我们将讨论分为几个部分。我们按顺序安排这些部分，目的是逐步形成对核心微架构越来越详细的视图。我们首先从整体 GPU 流水线的高层次视角开始，然后逐步填充细节。我们将这些越来越准确的描述称为“近似”，以承认即使在我们最详细的描述中也省略了一些细节。由于当今 GPU 的中央组织原则是多线程，我们对此类“近似”进行整理。

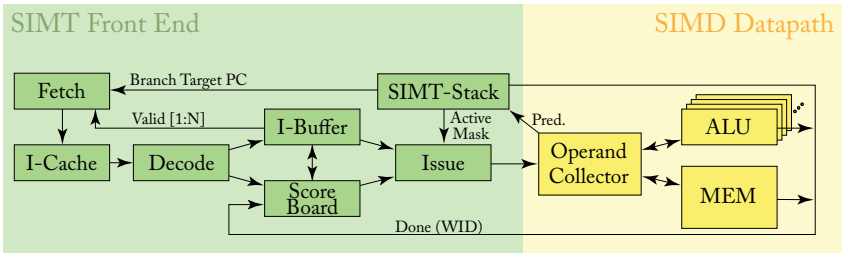


图 3.1：通用 GPGPU 核心的微架构。

围绕上述三个调度循环，我们发现将本章组织为考虑三个逐渐精确的“近似循环”是方便的，这些循环逐步考虑这些调度循环的细节。

3.1 单圈近似

我们首先考虑一个具有单一调度器的GPU。这种对硬件的简化视角与如果有人仅阅读CUDA编程手册中找到的硬件描述时对硬件的期望并无太大不同。

为了提高效率，线程被NVIDIA组织成称为“warp”的组，AMD则称之为“wavefront”。因此，调度的单位是一个warp。在每个周期，硬件选择一个warp进行调度。在单循环近似中，warp的程序计数器用于访问指令存储器，以查找要为warp执行的下一条指令。在获取指令后，该指令被解码，并从寄存器文件中获取源操作数寄存器。与从寄存器文件中获取源操作数并行，确定SIMT执行掩码值。以下小节描述了如何确定SIMT执行掩码值，并将其与现代GPU中使用的条件执行进行对比。

在执行掩码和源寄存器可用后，执行以单指令多数据的方式进行。每个线程在与通道关联的功能单元上执行，前提是设置了SIMT执行掩码。与现代CPU设计一样，功能单元通常是异构的，意味着给定的功能单元仅支持一部分指令。例如，NVIDIA GPU包含一个*special function unit*(SFU)、*load/store unit*、*floating-point*、*function unit*、*integer function unit*，以及截至Volta的一个*Tensor Core*。

所有功能单元名义上包含与一个warp中线程数量相同的多个执行通道。然而，一些GPU采用了不同的实现，其中一个warp或波前在多个时钟周期内执行。这是通过以更高的频率对功能单元进行时钟脉冲来实现的，这可以在增加能量消耗的代价下实现更高的单位面积性能。实现功能单元更高时钟频率的一种方法是对其执行进行流水线处理或增加其流水线深度。

3.1.1 SIMT 执行掩码

现代 GPU 的一个关键特性是 SIMT 执行模型，从功能的角度来看（尽管性能并非如此），它向程序员提供了每个线程完全独立执行的抽象。这个编程模型可能仅通过谓词化就能实现。然而，在当前的 GPU 中，实际上是通过传统谓词结合一系列谓词掩码的组合来实现的，我们将其称为 *SIMT stack*。

SIMT堆栈有效地处理了在所有线程可以独立执行时发生的两个关键问题。第一个是嵌套控制流。在嵌套控制流中，一个分支依赖于另一个分支的控制。第二个问题是当一个波束中的所有线程都避免控制流路径时，完全跳过计算。对于复杂的控制流，这可以带来显著的节省。传统上，支持预测的CPU通过使用多个谓词寄存器来处理嵌套控制流，并且文献中提议支持跨通道谓词测试。

GPU使用的SIMT栈可以处理嵌套控制流和跳过计算。专利和指令集手册中描述了几种实现。在这些描述中，SIMT栈至少部分是通过专门用于此目的的特殊指令进行管理的。相反，我们将描述一个在学术作品中介绍的稍微简化的版本，该版本假设硬件负责管理SIMT栈。

为了描述SIMT栈，我们使用一个示例。图3.2展示了包含两个嵌套在do-while循环中的分支的CUDA C代码，图3.3展示了相应的PTX汇编。图3.4 reproduces了Fung等人[Fung et al., 2007]的图5，展示了这段代码与SIMT栈的交互，假设GPU每个warp有四个线程。

图 3.4a 显示了与图 3.2 和图 3.3 中的代码相对应的控制流图 (CFG)。如 CFG 顶部节点内的标签 “ `A/1111` ” 所示，最初，warp 中的所有四个线程都在执行基本块 A 中的代码，这对应于图 3.2 中第 2 到第 6 行和图 3.3 中第 1 到第 6 行的代码。这四个线程在执行图 3.3 中第 6 行的分支后，按照不同的（不同的）控制流进行。该分支对应于图 3.2 中第 6 行的 “ if ” 语句。具体而言，如图 3.4a 中标签 “ `B/1110` ” 所示，前三个线程进入基本块 B。这三个线程分支到图 3.3 中的第 7 行（图 3.2 中的第 7 行）。如图 3.4a 中标签 “ `F/0001` ” 所示，执行分支后，第四个线程跳转到基本块 F，这对应于图 3.3 中的第 14 行（图 3.2 中的第 14 行）。

类似地，当在图 3.3 中的基本块 B 中执行的三个线程到达第 9 行的分支时，第一个线程分支到基本块 C，而第二个和第三个线程则分支到基本块 D。然后，所有三个线程到达基本块 E，并如图 3.4a 中标签 “ `E/1110` ” 所示共同执行。在基本块 G，所有四个线程一起执行。

GPU硬件如何使得一个warp中的线程能够在代码中沿着不同的路径执行，同时又采用了只允许每个周期执行一条指令的SIMD数据通路？

```
do {
  t1 = tid*N;       // A
  t2 = t1 + i;
  t3 = data1[t2];
  t4 = 0;
  if( t3 != t4 ) {
    t5 = data2[t2]; // B
    if( t5 != t4 ) {
      x += 1;       // C
    } else {
      y += 2;       // D
    }
  } else {
    z += 3;         // F
  }
  i++;              // G
} while( i < N );
```

图 3.2：示例 CUDA C 源代码，用于说明 SIMT 堆栈操作。

```
A:    mul.lo.u32      t1, tid, N;
      add.u32         t2, t1, i;
      ld.global.u32   t3, [t2];
      mov.u32         t4, 0;
      setp.eq.u32     p1, t3, t4;
@p1   bra             F;
B:    ld.global.u32   t5, [t2];
      setp.eq.u32     p2, t5, t4;
@p2   bra             D;
C:    add.u32         x, x, 1;
      bra             E;
D:    add.u32         y, y, 2;
E:    bra             G;
F:    add.u32         z, z, 3;
G:    add.u32         i, i, 1;
      setp.le.u32     p3, i, N;
@p3   bra             A;
```

图 3.3：示例 PTX 汇编代码，用于说明 SIMT 堆栈操作。

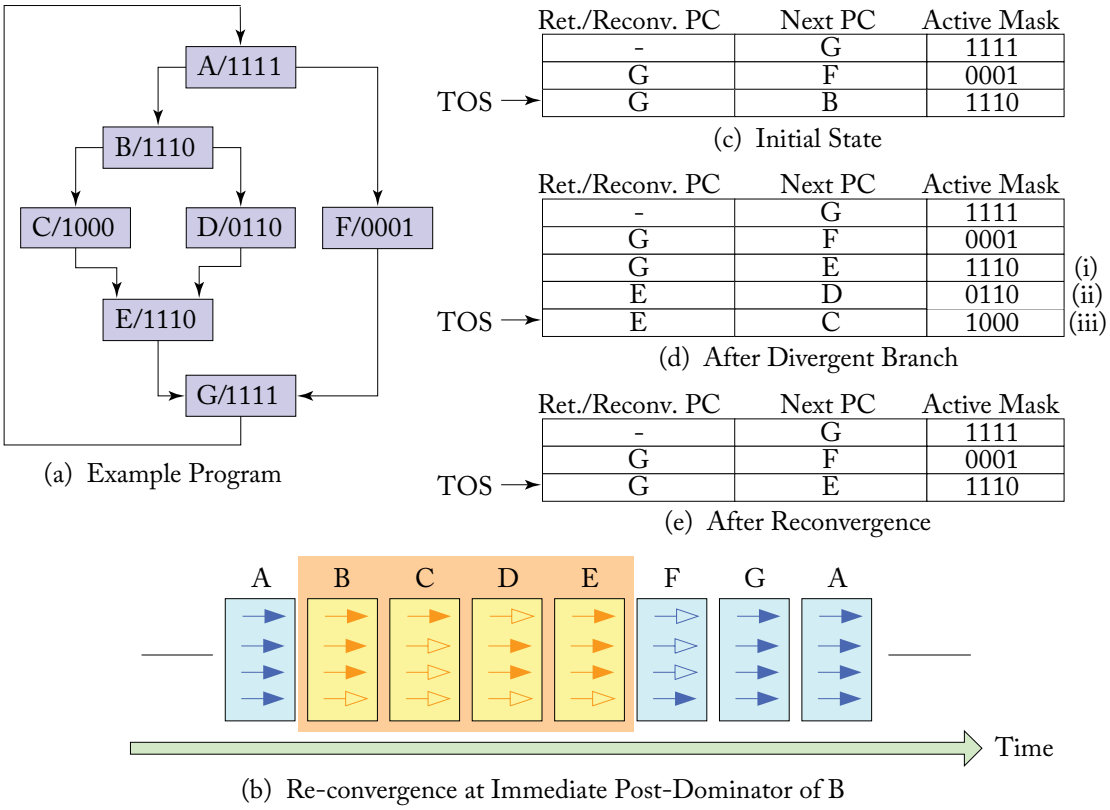


图3.4：SIMT堆栈操作示例（基于Fung等人[2007]的图5）。

当前 GPU 使用的方法是对在给定 warp 中沿不同路径执行的线程进行序列化。这在图 3.4b 中得到了说明，其中箭头表示线程。填充的箭头表示线程正在执行对应基础块中的代码（由每个矩形顶部的字母指示）。带空心箭头的箭头表示线程被屏蔽。图中的时间向右推进，如底部的箭头所示。最初，每个线程在基础块 B 中执行。然后，在分支之后，前面三个线程执行基础块 B 中的代码。请注意，此时线程四被屏蔽。为了维持 SIMD 执行，第四个线程在不同的时间（在这个例子中是几个周期后）通过基础块 F 执行替代代码路径。

为了实现发散代码路径的序列化，一种方法是使用如图3.4c – e所示的堆栈。该堆栈中的每个条目包含三个条目：一个重归并程序计数器（RPC），下一个要执行的指令的地址（Next PC），以及一个活动掩码。

图 3.4c 说明了在图 3.3 中第 6 行的分支执行后，堆栈的状态。由于有三个线程分支到基本块 B，和一个线程分支到基本块 F，因此在堆栈顶部 (TOS) 添加了两个新条目。瓦片执行的下一条指令是使用堆栈顶部 (TOS) 条目的下一个 PC 值来确定的。在图 3.4c 中，这个下一个 PC 值是 B，表示基本块 B 中第一条指令的地址。相应的活动掩码条目“ 1110 ”表示只有瓦片中的前三个线程应该执行这条指令。

在图3.3的第9行的分支处，第一个线程继续从基本块B执行指令。执行完这个分支后，它们如前所述分叉。这种分支分叉导致堆栈发生三个变化。首先，在执行分支之前，TOS条目的下一条PC条目，标记为图3.4d中的(i)，被修改为分支的*reconvergence point*，即基本块E中第一条指令的地址。然后，增加了两个条目，标记为图3.4d中的(ii)和(iii)，每个条目对应于分支后线程在warp中所遵循的路径。

重合点是在程序中线程分歧后可以被强制顺序执行的位置。一般来说，优先选择最近的重合点。在给定程序执行的最早时刻，可以在编译时保证分歧线程可以再次顺序执行的是导致分支分歧的分支的直接后支配者。在运行时，有时可以在程序的更早处重新汇聚[Coon 和 Lindholm，2008，Diamos 等，2011，Fung 和 Aamodt，2011]。

一个有趣的问题是“ 在跟随一个发散分支时应该使用什么顺序将条目添加到栈中？ ”为了将重聚栈的最大深度减少到与一个波在程序中线程数量的对数成正比，最好首先将具有最多活动线程的条目放入栈中，然后再放入具有较少活动线程的条目[AMD, 2012]。在图3.4的(d)部分我们遵循了这个顺序，而在(c)部分我们使用了相反的顺序。

3.1.2 SIMT 死锁和无栈 SIMT 架构

最近，NVIDIA披露了他们即将推出的Volta GPU架构的细节[NVIDIA Corp., 2017]。他们强调的一个变化是遮罩在发散下的行为，以及这如何与同步相互作用。基于栈的SIMT实现可能导致一种叫做“ ElTantawy和Aamodt [2016] ”的死锁状态，称为“ SIMT死锁 ”。学术研究描述了一种替代的SIMT执行硬件[ElTantaway等，2014]，这种硬件经过一些小的改动[ElTantawy和Aamodt，2016]，可以避免SIMT死锁。NVIDIA称他们的新线程发散管理方法为独立线程调度。独立线程调度的描述表明，他们实现了类似上述学术提案所获得的行为。下面，我们首先描述SIMT死锁问题，然后描述一种避免SIMT死锁的机制，该机制是一致的。

与NVIDIA关于独立线程调度的描述相符，且在最近的一项NVIDIA专利申请中披露 [Diamos et al., 2015]。

图3.5的左侧部分给出了一个CUDA示例，说明了SIMT死锁问题，中间部分显示了相应的控制流图。A行将共享变量mutex初始化为零，以指示锁是空闲的。在B行，每个warp中的线程执行`atomicCAS`操作，该操作对包含mutex的内存位置执行比较并交换操作。`atomicCAS`操作是一个编译器内置函数，会被转化为`atom.global.cas` PTX指令。从逻辑上讲，比较并交换首先读取mutex的内容，然后将其与第二个输入0进行比较。如果mutex的当前值为0，则比较并交换操作将mutex的值更新为第三个输入1。`atomicCAS`返回的值是mutex的原始值。重要的是，比较和交换对每个线程执行上述一系列逻辑操作是原子的。因此，由不同线程在同一warp内对任何单个位置的多次访问被串行化。由于图3.5中的所有线程都访问相同的内存位置，仅有一个线程会看到mutex的值为0，其余线程将看到值为1。接下来，在考虑SIMT堆栈的情况下，想一想在`atomicCAS`返回后，B行的`while`循环会发生什么。不同的线程会看到不同的循环条件。具体来说，一个线程希望退出循环，而其余线程希望留在循环中。退出循环的线程将到达重合点，因此在SIMT堆栈上不再活跃，从而无法执行C行的`atomicExch`操作来释放锁。留在循环中的线程将在SIMT堆栈顶部处保持活跃并将无限期地自旋。线程之间的循环依赖引入了一种新形式的死锁，ElTantawy和Aamodt [2016]称之为SIMT死锁，如果线程在MIMD架构上执行，这种死锁将不会存在。

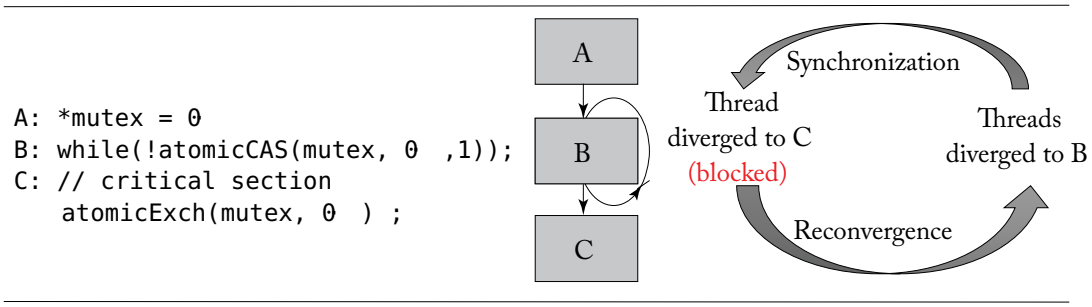


图 3.5：SIMT 死锁示例（基于 ElTantawy 和 Aamodt [2016] 的图 1）。

接下来，我们总结了一种无栈分支重合机制，与最近NVIDIA的一项美国专利申请[Diamos等，2015]中的机制类似。该机制与NVIDIA迄今为止对Volta重合处理机制的描述一致[Nvidia，2017]。关键思想是用每个warp的重合屏障替代栈。图3.6展示了NVIDIA专利申请中描述的每个warp维护的各种字段，图3.8

提供了一个相应的例子来说明收敛障碍的操作。实际上，提案提供了对多路径IPDOM [ElTantaway et al., 2014] 的替代实现，这将在第3.4.2节中与早期的学术作品一起描述。收敛障碍机制与Fung和Aamodt [2011]中描述的*warp barrier*概念有一些相似之处。为了帮助解释下面的收敛障碍机制，我们考虑在图3.8中的代码上执行单个warp，图3.8显示了由类似于图3.7中所示的CUDA代码产生的控制流图。

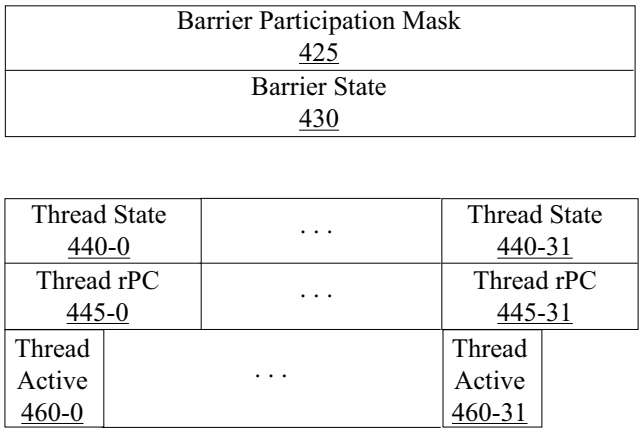


图 3.6：NVIDIA 最近描述的无堆栈备用收敛障碍基于分支分歧处理机制（基于 Diamos 等人 [2015] 的图 4B）。

```
1  // id = warp ID
2  // BB_A Basic Block "A"
3  if(id%2==0){
4      // BB_B
5  }else{
6      // BB_C
7      if(id==1){
8          // BB_D
9      }else{
10         // BB_E
11     }
12     // BB_F
13 }
14 // BB_G
```

图 3.7: 嵌套控制流示例（基于 ElTantaway 等人 [2014] 的图 6(a)）。

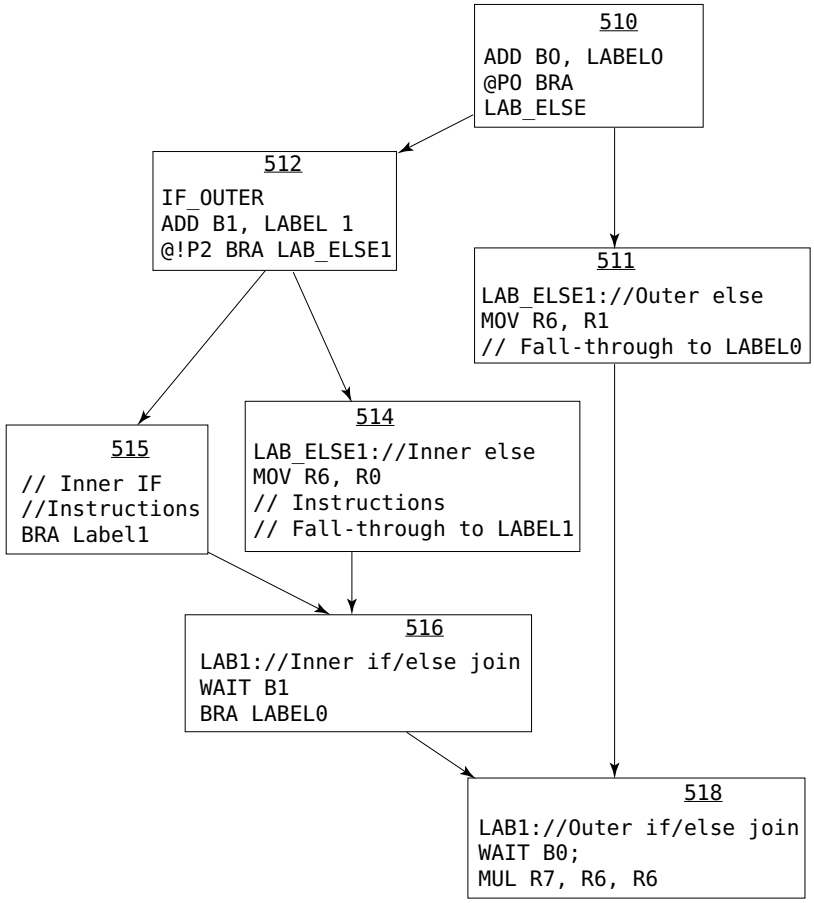


图 3.8：NVIDIA 最近描述的收敛障碍分支分歧处理机制的代码示例（基于 Diamos 等人 [2015] 的图 5B）。

接下来，我们描述图3.6中的字段。这些字段存储在寄存器中，并由硬件warp调度器使用。每个*Barrier Participation Mask*用于追踪给定warp中的哪些线程参与了给定的收敛障碍。对于给定的warp，可能会有多个障碍参与掩码。在常见情况下，通过给定障碍参与掩码追踪的线程将等待彼此到达程序中的某个公共点，该点位于一个分支后的收敛点，从而重新收敛。为了支持这一点，*Barrier State*字段用于追踪哪些线程已经到达给定的收敛障碍。*Thread State*追踪warp中每个线程是否准备好执行、在收敛障碍处被阻塞（如果是，则是哪个），或者已经让出。似乎让出状态可能被用来使warp中的其他线程能够在一种情况下越过收敛障碍并继续前进。

否则会导致SIMT死锁。线程rPC字段跟踪每个不活动线程的下一条待执行指令的地址。线程活动字段是一个指示波浪中相应线程是否处于活动状态的位。

假设一个 warp 包含 32 个线程，屏障参与掩码的宽度为 32 位。如果一个位被设置，则意味着对应的线程在这个汇聚屏障中参与。当线程执行分支指令时，例如图 3.8 中基本块 510 和 512 末尾的指令，线程会发生分歧。这些分支对应于图 3.7 中的两个“ if ”语句。屏障参与掩码被 warp 调度器用来在特定的汇聚屏障位置停止线程，该位置可以是分支的立即后支配者或其他位置。在任何给定时刻，每个 warp 可能需要多个屏障参与掩码以支持嵌套的控制流结构，例如图 3.7 中的嵌套 if 语句。图 3.6 中的寄存器可能使用通用寄存器、专用寄存器或两者的某种组合来实现（专利申请并未说明）。考虑到屏障参与掩码仅为 32 位宽，如果每个线程都拥有屏障参与掩码的副本（例如，如果天真地使用通用寄存器文件来存储它），将是冗余的。然而，由于控制流可以嵌套到任意深度，给定的 warp 可能需要任意数量的屏障参与掩码，这使得对掩码的软件管理变得可取。

为了初始化收敛屏障参与掩码，将使用一种特殊的“ ADD ”指令。当warp执行此ADD指令时，所有处于活动状态的线程在ADD指令指示的收敛屏障中其位被设定。在执行分支后，某些线程可能会发生分歧，这意味着接下来要执行的指令的地址（即PC）将会不同。当这种情况发生时，调度器将选择一组具有相同PC的线程并更新线程活动字段，以便为这些warp中的线程启用执行。学术提案将这样一组线程称为“ warp分裂 ”[ElTantaway et al., 2014, ElTantawy and Aamodt, 2016, Meng et al., 2010]。与基于栈的SIMT实现相比，使用收敛屏障实现的调度器可以在分歧线程组之间自由切换。这使得在某些线程已经获得锁而其他线程尚未获得锁时，warp内的线程能够继续向前推进。

“ WAIT ”指令用于在达到收敛屏障时停止一个warp分裂。如NVIDIA的专利申请中所述，WAIT指令包括一个操作数，用于指明收敛屏障的身份。WAIT指令的作用是将warp分裂中的线程添加到该屏障的Barrier State寄存器中，并将线程的状态更改为阻塞。一旦收敛屏障参与掩码中的所有线程执行了相应的WAIT指令，线程调度器便可以将原始warp分裂中的所有线程切换为活动状态，从而保持SIMD效率。图3.8中的例子有两个收敛屏障，B1和B2，分别在基本块516和518中有WAIT指令。为了实现warp分裂之间的切换，NVIDIA描述了使用YIELD指令以及其他细节，比如对间接分支的支持，我们在本讨论中省略了这些细节[Diamos et al., 2015]。

图3.9展示了基于堆栈的重汇聚时序示例，图3.10则通过独立线程调度说明了潜在时序，正如NVIDIA的Volta白皮书中所描述的。在图3.10中，我们可以看到语句A和B与语句X和Y交错出现，这与图3.9中的行为形成对比。这种行为与上述描述的汇聚障碍机制一致（以及多路径IPDOM [ElTantaway等, 2014]）。最后，图3.11展示了无堆栈架构如何执行图3.5中的自旋查找代码，从而避免SIMT死锁。

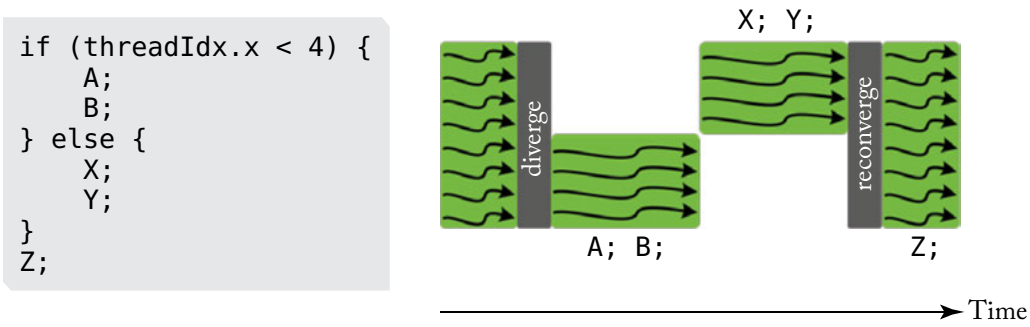


图 3.9：示例展示基于堆栈的重聚行为（基于 Nvidia [2017] 的图 20）。

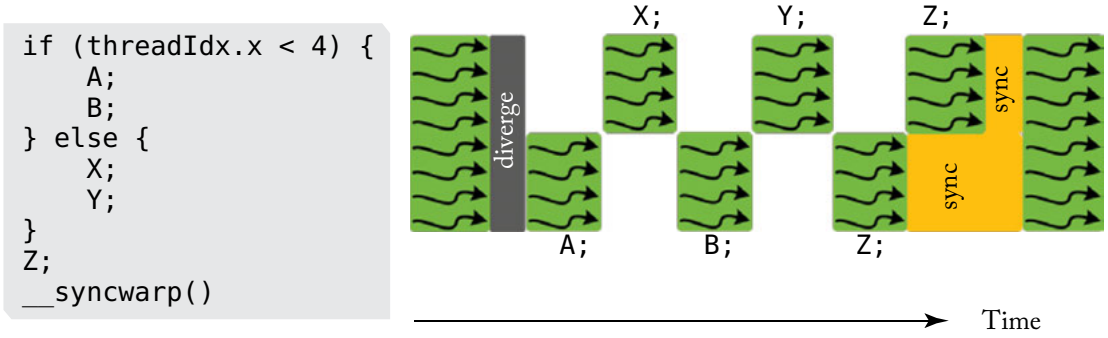


图 3.10：示例显示了 Volta 重新汇聚的行为（基于 Nvidia [2017] 的图 23）。

3.1.3 WARP 调度

每个 GPU 核心包含许多 warp。一个非常有趣的问题是这些 warp 应该以什么顺序调度。为了简化讨论，我们假设每个 warp 在调度时仅发出一条指令，并且 warp 不具备资格

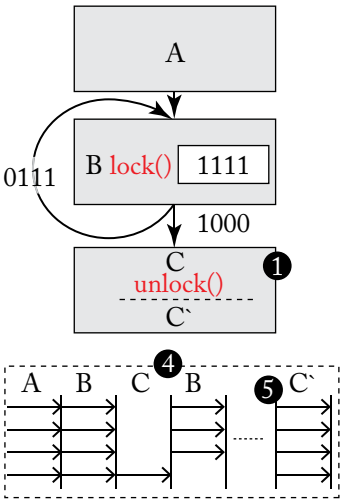


图 3.11：示例显示了与图 3.5 中的自旋锁代码类似的收敛障碍的学术机制行为（基于 El Tantawy 和 Aamodt [2016] 的图 6(a)）。

在第一个指令执行完成之前，不会发出另一个指令。我们将在本章稍后重新审视这个假设。

如果内存系统是“ 理想的 ”，并且在某个固定延迟内响应内存请求，那么理论上可以设计核心以支持足够的warp，以使用精细粒度的多线程来隐藏该延迟。在这种情况下，可以认为我们可以通过以“ 轮询 ”方式调度warp来减少给定吞吐量下芯片的面积。在轮询中，warp被赋予某种固定顺序，例如按递增的线程标识符排序，调度程序根据这个顺序选择warp。这种调度顺序的一个特性是，它允许每个发出的指令有大致相等的时间来完成执行。如果核心中warp的数量乘以每个warp的发射时间超过内存延迟，核心中的执行单元将始终保持忙碌。因此，原则上将warp的数量增加到这一点可以提高每个核心的吞吐量。

然而，有一个重要的权衡：为了让不同的波浪在每个周期发出指令，必须确保每个线程都有自己的寄存器（这避免了在寄存器和内存之间复制和恢复寄存器状态的需要）。因此，增加每个核心的波浪数量会相应增加芯片区域中用于寄存器文件存储的比例，而相应减少用于执行单元的比例。在固定的芯片面积下，增加每个核心的波浪数量将减少每个芯片的核心总数。

在实践中，内存的响应延迟取决于应用程序的局部性特性以及离芯片内存访问所遇到的争用量。什么

在考虑GPU的内存系统时，调度发挥了什么影响？在过去几年中，这一直是相当多的研究主题，我们将在为我们的GPU微架构模型添加更多关于内存系统的细节后，重新审视这个问题。然而，简而言之，局部性特性可能会有利于或不利于轮转调度：当不同的线程在执行中于相似点共享数据时，例如在图形像素着色器中访问纹理映射，线程平等推进是有益的，因为这可以增加“击中”片上缓存的内存引用数量，而这正是轮转调度所鼓励的[Lindholm等，2015]。类似地，当在时间上接近地访问地址空间中的邻近位置时，访问DRAM更为高效，这也受到轮转调度的鼓励[Narasiman等，2011]。另一方面，当线程主要访问不相交的数据时，这通常发生在更复杂的数据结构中，那么反复调度某个特定线程以最大化局部性可能是有益的[Rogers等，2012]。

3.2 双环路近似

为了帮助减少每个核心必须支持的波的数量以隐藏长期执行延迟，能够在早期指令尚未完成时从波中发出后续指令是非常有帮助的。然而，之前描述的一循环微架构阻止了这一点，因为该设计中的调度逻辑仅访问线程标识符和要发出的下一条指令的地址。具体来说，它不知道要为波发出的下一条指令是否依赖于尚未完成执行的早期指令。为了提供这种依赖信息，有必要先从内存中获取指令，以确定存在哪些数据和/或结构风险。为此，GPU 实现了一个指令缓冲区，指令在缓存访问后被放置在其中。使用单独的调度器来决定在指令缓冲区中应该优先发出哪几条指令到管道的其余部分。

指令存储器被实现为一个一级指令缓存，后面由一个或多个级别的二级（通常是统一的）缓存支持。指令缓冲区还可以通过与指令未命中状态保持寄存器（MSHRs）结合，帮助隐藏指令缓存未命中的延迟 [Kroft, 1981]。在缓存命中或从缓存未命中中填充后，指令信息会被放入指令缓冲区。指令缓冲区的组织可以采用多种形式。一种特别简单的方法是在每个 warp 中为一个或多个指令提供存储空间。

接下来，让我们考虑如何检测同一个warp内指令之间的数据依赖关系。传统CPU架构中有两种检测指令之间依赖关系的传统方法：记分板和保留站。保留站用于消除名称依赖关系，引入了需要代价高昂的关联逻辑，这在面积和能量方面都比较昂贵。记分板可以设计为支持顺序执行或乱序执行。支持乱序执行的记分板，如CDC 6600中使用的，也相当复杂。另一方面，记分板对于一个

单线程顺序CPU非常简单：每个寄存器在得分板中用一个单独的比特来表示，当有指令发出要写入该寄存器时，该比特被置为1。任何想要读取或写入在得分板中其对应比特被置为1的寄存器的指令都会被阻塞，直到写入该寄存器的指令清除该比特。这可以防止读后写和写后写的危险。与顺序指令发出结合时，这个简单的得分板可以防止写后读的危险，前提是寄存器文件的读取被限制为顺序进行，而这通常是顺序CPU设计中的常见情况。考虑到这是最简单的设计，因此将消耗最少的面积和能量，GPU实现了顺序得分板。然而，正如接下来讨论的那样，在支持多个warp时，使用顺序得分板面临挑战。

上述所描述的简单有序记分板设计的第一个关注点是现代GPU中包含的寄存器数量非常庞大。每个波束最多有128个寄存器，每个核心最多有64个波束，因此每个核心需要总共8192位来实现记分板。

另一个关于上述简单的顺序比分设计的担忧是，遇到依赖关系的指令必须反复在比分器中查找其操作数，直到它依赖的先前指令将其结果写入寄存器文件。在单线程设计中，这引入了很少的复杂性。然而，在顺序发射的多线程处理器中，来自多个线程的指令可能在等待早期指令完成。如果所有这些指令都必须查询比分器，则需要额外的读取端口。最近的GPU每个核心支持最多64个warp，并且每个warp最多支持4个操作数，允许所有warp在每个周期查询比分器将需要256个读取端口，这将非常昂贵。一个替代方案是限制每个周期可以查询比分器的warp数量，但这限制了可以考虑调度的warp数量。此外，如果检查的指令没有一个是没有依赖关系的，则可能无法发出任何指令，即使其他无法检查的指令恰好没有依赖关系。

这两个问题可以通过Coon等人[2008]提出的设计来解决。该设计不是每个warp每个寄存器持有一个比特，而是每个warp包含少量（在最近的一项研究中估计约为3或4个条目[Lashgar et al., 2016]）条目，每个条目是一个寄存器的标识符，该寄存器将被已发出但尚未完成执行的指令写入。正常的顺序记分板在指令发出和写回时都进行访问。而Coon等人的记分板则在指令被放入指令缓冲区时和指令将其结果写入寄存器文件时进行访问。

当一条指令从指令缓存中获取并放置在指令缓冲区时，相应的warp的记分板条目会与该指令的源寄存器和目标寄存器进行比较。这会产生一个短的位向量，每个条目在该warp的记分板中占用一个位（例如，3位或4位）。如果记分板中的相应条目与指令的任一操作数匹配，则该位被设置。该位向量随后与指令一起复制到指令缓冲区。指令在所有位被清除之前，不符合被指令调度器考虑的资格，这可以通过给每个

将向量的一位输入到NOR门中。指令缓冲区中的依赖位在指令将其结果写入寄存器文件时被清除。如果给定的warp使用完了所有条目，则所有warp会发生取指暂停，或者该指令会被丢弃并必须重新获取。当已执行的指令准备写入寄存器文件时，它会清除在分数板上分配给它的条目，并清除存储在指令缓冲区中来自同一warp的任何指令的相应依赖位。

在双循环架构中，第一个循环选择在指令缓冲区中有空间的波块，查找其程序计数器并执行指令缓存访问以获取下一条指令。第二个循环选择在指令缓冲区中没有未完成依赖关系的指令，并将其发给执行单元。

3.3 三重循环近似

如前所述，为了隐藏长时间的内存延迟，有必要支持每个核心多个线程束，并且为了支持线程束之间的循环切换，必须拥有一个大型寄存器文件，该文件包含每个正在执行的线程束的单独物理寄存器。例如，在最近的NVIDIA GPU架构（如Kepler、Maxwell和Pascal架构）中，这样的寄存器包含256 KB的容量。现在，SRAM内存的面积与端口数量成正比。寄存器文件的简单实现需要每个周期每个指令发出一个操作数的端口。减少寄存器文件面积的一种方法是使用多个单端口内存银行来模拟大量端口。虽然通过将这些内存银行暴露给指令集架构可以实现这种效果，但在某些GPU设计中，似乎使用了一种称为操作数收集器的结构[Coon et al., 2009, Lindholm et al., 2008b, Lui et al., 2008]以更透明的方式实现这一点。操作数收集器有效地形成了如下面所述的第三个调度循环。

为了更好地理解操作数收集器解决的问题，首先考虑图3.12，它展示了一种为增加寄存器文件带宽而设计的简单微架构。该图显示了GPU指令管道的寄存器读取阶段，其中寄存器文件由四个单端口逻辑寄存器组组成。实际上，由于寄存器文件非常大，每个逻辑寄存器组可能进一步分解为更多的物理寄存器组（未显示）。逻辑寄存器组通过交叉开关连接到暂存寄存器（标记为“管道寄存器”），该寄存器在将源操作数传递给SIMD执行单元之前进行缓冲。仲裁器控制对各个寄存器组的访问，并通过交叉开关将结果路由到适当的暂存寄存器。

图3.13展示了每个warp的寄存器到逻辑银行的简单布局。在这个图中，来自warp 0的寄存器$_{r0}$（$_{w0}$）存储在银行0的第一个位置上，来自warp 0的寄存器$_{r1}$存储在银行1的第一个位置上，依此类推。如果计算所需的寄存器数量大于逻辑银行的数量，则分配会回绕。例如，warp 0的寄存器$_{r4}$存储在银行0的第二个位置上。

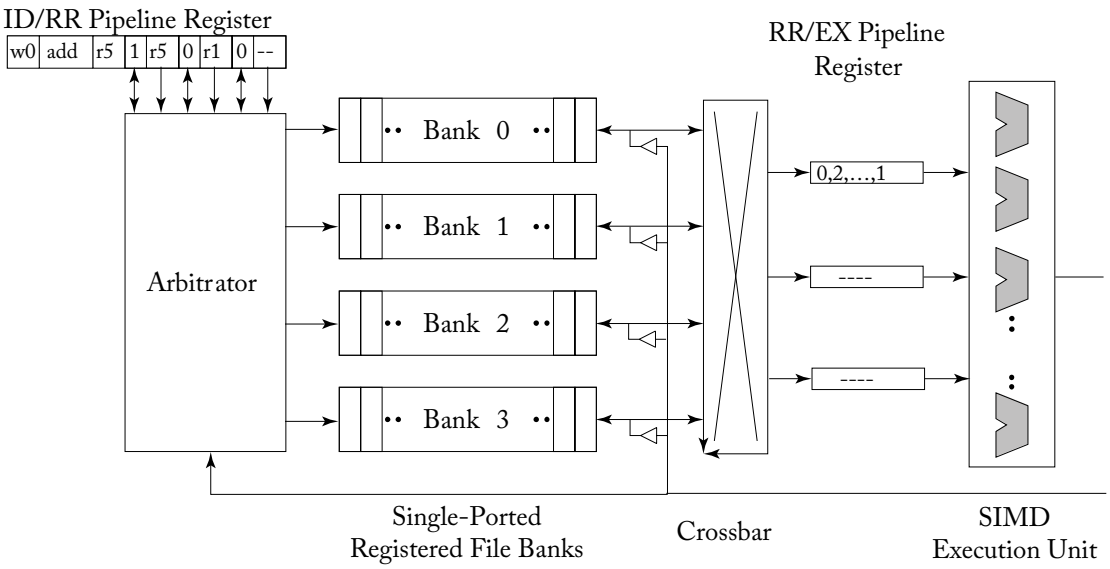


图 3.12：简单的银行寄存器文件微架构。

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

图 3.13：简单的银行寄存器布局。

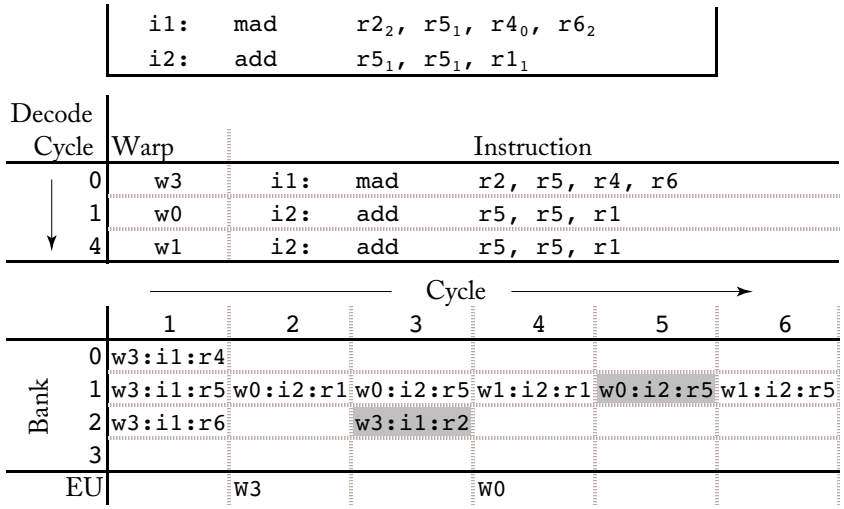


图 3.14：简单的银行寄存器文件的时序。

图3.14展示了一个时序示例，突出显示了该微架构如何影响性能。这个示例涉及顶部显示的两条指令。第一条指令 i1 是一个多重加法操作，读取寄存器 r5、r4 和 r6，它们分别分配在银行1、0和2（在图中通过下标指示）。第二条指令 i2 是一个加法指令，从寄存器 r5 和 r1 读取，这两个寄存器都分配在银行1。图的中间部分显示了指令发出的顺序。在周期0，warp 3发出了指令 i1；在周期1，warp 0发出了指令 i2；在周期4，由于银行冲突的延迟，warp 1发出了指令 i2。图的底部部分展示了不同指令对不同银行的访问时序。在周期1，warp 3的指令 i1 能够在周期1读取到三个源寄存器，因为它们映射到不同的逻辑银行。然而，在周期2，warp 0的指令 i2 只能读取两个源寄存器中的一个，因为这两个寄存器都映射到银行1。在周期3，这条指令的第二个源寄存器与来自warp 3的指令 i1 的写回并行读取。在周期4，warp 1的指令 i2 能够读取它的第一个源操作数，但无法读取第二个，因为这两个寄存器再次都映射到银行1。在周期5，warp 1的指令 i2 的第二个源操作数由于银行已经被warp 0早先发出的指令 i2 的更高优先级写回访问而无法从寄存器文件中读取。最后，在周期6，来自warp 1的指令 i2 的第二个源操作数从寄存器文件中读取。总之，三条指令完成读取其源寄存器的过程耗时六个周期，在此期间许多银行没有被访问。

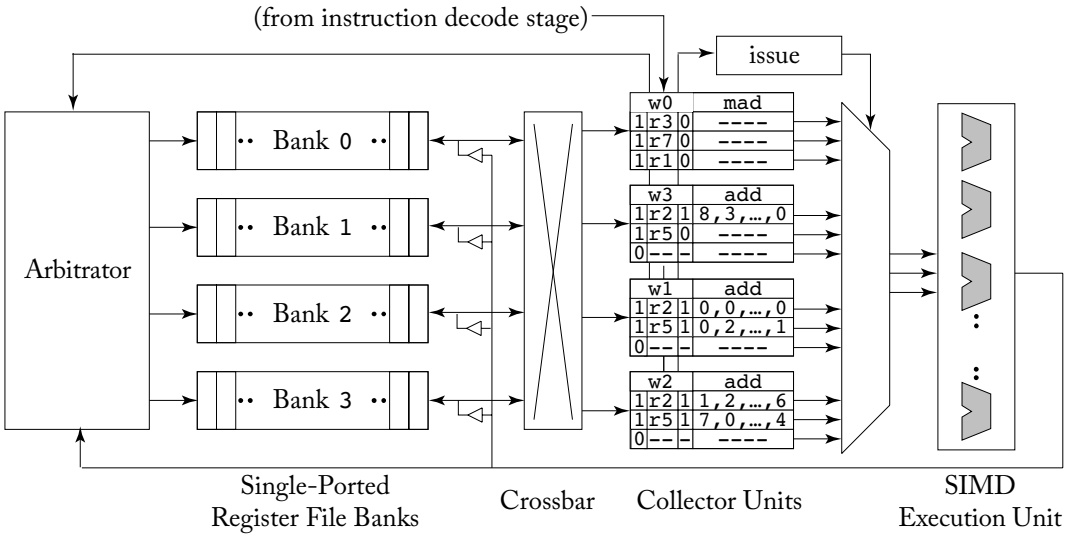


图 3.15：操作数收集器微架构（基于 Tor M. Aamodt 等的图 6）。

3.3.1 操作数收集器

操作数收集器微架构 [Lindholm et al., 2008b] 如图 3.15 所示。主要变化是暂存寄存器已被 *collector units* 替代。每个指令在进入寄存器读取阶段时分配一个收集单元。有多个收集单元，以便多个指令可以重叠读取源操作数，这有助于在单个指令的源操作数之间存在银行冲突的情况下提高吞吐量。每个收集单元包含执行指令所需的所有源操作数的缓冲空间。考虑到多条指令的源操作数数量较大，仲裁器更有可能实现更高的银行级并行性，以允许并行访问多个寄存器文件银行。

操作数收集器使用调度来容忍发生的银行冲突。这使得如何减少银行冲突的数量成为一个开放的问题。图 3.16 展示了 Coon 等人描述的一种修改后的寄存器布局，旨在帮助减少银行冲突。其思想是将等效寄存器从不同的 warp 分配到不同的银行。例如，在图 3.16 中，warp 0 的寄存器 `r0` 被分配到银行 0，而 warp 1 的寄存器 `r0` 被分配到银行 1。这并不能解决单个指令的寄存器操作数之间的银行冲突。然而，它在减少来自不同 warp 的指令之间的银行冲突方面确实有所帮助。特别是，每当 warp 以相对均匀的速度推进时（例如，由于轮询调度或两级调度 [Narasiman et al., 2011]，其中获取组中的各个 warp 以轮询顺序调度）。

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

图 3.16：混合银行寄存器布局。

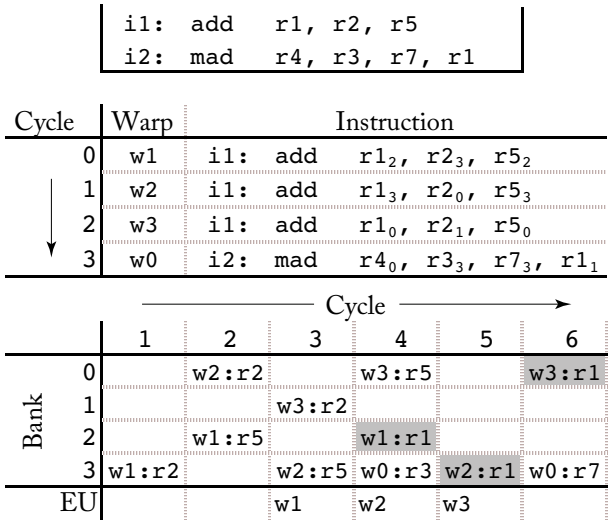


图 3.17：操作数收集器的时序。

图 3.17 显示了一个时序示例，顶部展示了一系列加法和乘加指令。在中间部分显示了发射顺序。从波段 1 到波段 3 的 `i1` 三个实例在周期 0 到 2 之间发射。波段 0 的指令 `i2` 在周期 3 发射。注意，add 指令写入寄存器 `r1`，对于任何给定的波段，它与源寄存器 `r5` 分配在同一个银行中。然而，与使用图 3.13 中的寄存器布局的情况不同，这里不同的波段访问不同的银行，这有助于减少一个波段的写回与其他波段读取源操作数之间的冲突。底部部分显示了由于操作数收集器造成的银行级别访问时序。在周期 1，来自波段 1 的寄存器 `r2` 读取银行 3。在周期 4，注意来自波段 1 的寄存器 `r1` 的写回与来自波段 3 的寄存器 `r5` 的读取以及来自波段 0 的寄存器 `r3` 的读取并行进行。

一个微妙的问题是截止目前为止描述的操作数收集器，因为它不对不同指令准备发出时的顺序施加任何限制，因此可能会允许读后写（WAR）危害[Mishkin等，2016]。如果两条来自不同线程的指令同时准备发出并且存在这种数据依赖关系，可能会发生这种情况。

相同的 warp 存在于一个操作数收集器中，第一条指令读取一个寄存器，而第二条指令将写入该寄存器。如果第一条指令的源操作数访问遇到重复的银行冲突，则第二条指令在第一条寄存器读取（正确的）旧值之前，可能会向寄存器写入一个新值。防止这种WAR危险的一种方法是要求来自同一warp的指令按照程序顺序将操作数从收集器发送到执行单元。Mishkin等人[2016] 探讨了三种低硬件复杂度的潜在解决方案，并评估了它们对性能的影响。第一种，*release-on-commit warpboard*，允许每个warp最多执行一条指令。不出所料，他们发现这对性能产生了负面影响，在某些情况下将性能降低了将近一半。第二个提议是*release-on-read warpboard*，它每次允许每个warp在操作数收集器中仅收集一条指令。这种方案在他们研究的工作负载中最多导致10%的减速。最后，为了允许操作数收集器中的指令级并行性，他们提出了一种使用小的布隆过滤器来跟踪未完成寄存器读取的*bloomboard*机制。这导致的影响比（错误地）允许WAR危险要小几个百分点。此外，Gray进行的分析表明，NVIDIA的Maxwell GPU引入了一个“ 读取依赖障碍 ”，由特殊的“ 控制指令 ”管理，可以用于避免某些指令的WAR危险（见第2.2.1节）。

3.3.2 指令重放：处理结构危害

在GPU流水线中，结构性危害有许多潜在原因。例如，寄存器读取阶段可能会耗尽操作数收集单元。许多结构性危害的来源与内存系统相关，我们将在下一章中详细讨论。通常，多个线程束执行的单个内存指令可能需要拆分为多个单独的操作。这些单独的操作中的每一个都可能在某个周期内充分利用流水线的部分资源。

当一条指令在GPU流水线中遇到结构性冒险时，会发生什么？在单线程顺序CPU流水线中，标准的解决方案是暂时阻塞较年轻的指令，直到遇到阻塞条件的指令可以进一步进展。出于至少两个原因，这种方法在高度多线程的吞吐架构中可以说不太理想。首先，由于寄存器文件的较大规模，以及支持完整图形流水线所需的多个流水线阶段，分发阻塞信号可能会影响关键路径。流水线阻塞周期的分配会导致需要引入额外的缓冲，增加面积。其次，阻塞来自一个工作组的指令可能会导致其他工作组的指令在其后面阻塞。如果那些指令不需要被阻塞指令所需的资源，吞吐量可能会受到影响。

为了避免这些问题，GPU 实现了一种指令重放的形式。指令重放在一些 CPU 设计中被找到，在这些设计中，当推测性地调度对一个具有可变延迟的早期指令的依赖指令时，它被用作一种恢复机制。例如，加载可能在一级缓存中命中或未命中，但在高频率驱动的 CPU 设计中，

频率可以将一级缓存访问管道化，最多延续四个时钟周期。一些CPU的投机机制会根据加载情况唤醒指令，以改善单线程性能。相比之下，GPU避免投机，因为这往往会浪费能量并降低吞吐量。相反，GPU使用指令重放来避免管道堵塞和由停顿导致的电路面积和/或时间开销。

为了实现指令重放，GPU 可以在指令缓冲区中保存指令，直到知道它们已经完成或所有指令的各个部分都已执行 [Lindholm et al., 2015]。

3.4 分支偏差的研究方向

This section is based on Wilson Fung's Ph.D. dissertation [Fung, 2015].

理想情况下，同一个warp中的线程通过相同的控制流路径执行，以便GPU可以在SIMD硬件上同步执行它们。考虑到线程的自主性，当它们在数据依赖的分支上分歧到不同目标时，warp可能会遇到*branch divergence*。现代GPU包含专门的硬件来处理warp中的分支分歧。3.1.1节描述了基线SIMT栈，基线GPU架构在本书中使用该栈。基线SIMT栈通过序列化不同目标的执行来处理warp中的分支分歧。虽然基线SIMT栈对于大多数现有GPU应用程序正确处理了分支分歧，但它存在以下缺陷。

较低的SIMD效率 在分支分歧的情况下，基线SIMT栈串行化每个分支目标的执行。当执行每个目标时，SIMT栈仅激活运行该目标的标量线程的子集。这导致SIMD硬件中的某些通道处于空闲状态，从而降低了整体*SIMD efficiency*。

不必要的序列化 基线SIMT栈对每个分支目标的序列化执行并不是功能正确性的要求。GPU编程模型不会在一个warp内的标量线程之间施加任何隐式数据依赖——它们必须通过共享内存和屏障显式通信。GPU可以交错执行分歧warp的所有分支目标，以利用SIMD硬件中的空闲周期。

不充分的 MIMD 抽象 通过强制分化的波浪在编译器定义的重聚点重新聚合，基线 SIMT 堆栈在每个重聚点隐式地施加了一个波宽同步点。这对于许多现有的 GPU 应用程序有效。然而，这种隐式同步可能与其他用户实现的同步机制（如细粒度锁）发生病理性交互，导致波浪发生死锁。编译器定义的重聚点也没有考虑由系统级构造（如异常和中断）引入的控制流分歧。

区域成本 基线SIMT堆栈每个warp的面积需求仅为32 × 64位（或低至6 × 64位），但面积随着GPU中飞行warp的数量而扩展。在典型的GPU应用中，分支分歧很少，SIMT堆栈占用的面积本可以用于以其他方式提升应用程序吞吐量（例如，较大的缓存、更多的ALU单元等）。

行业和学术界都提出了替代方案来解决上述缺陷。各种提案可以归类为以下几类：波浪压缩、内部波浪分歧路径管理、增加MIMD能力和复杂性降低。有些提案包含了来自多个类别的改进，因此会被提及多次。

3.4.1 WARP 压缩

随着GPU实现细粒度的多线程以容忍长时间内存访问延迟，每个SIMT核心中有许多warp，总共有数百到数千个标量线程。由于这些warp通常在运行相同的计算内核，它们很可能遵循相同的执行路径，并在同一组数据依赖分支上遇到分支分歧。因此，分歧分支的每个目标可能会被大量线程执行，但这些线程分散在多个静态warp中，每个warp单独处理分歧。

在本节中，我们总结了一系列利用这一观察结果来改善遭受分支发散问题的GPU应用性能的研究。这一系列提案都涉及到新颖的硬件机制，将来自不同*static warps*的*compact*线程重新组合成新的*dynamic warps*，以提高这些发散GPU应用的总体SIMD效率。这里，静态束指的是在内核启动时由GPU硬件形成的束。在我们的基准GPU架构中，这一排列在整个束执行过程中是固定的。将标量线程排列成静态束是一个由GPU硬件施加的任意分组，对编程模型几乎是不可见的。

动态扭曲形成。动态扭曲形成（DWF）[Fung et al., 2007, Fung et al., 2009] 利用这个观察，通过将执行相同指令的分散线程重新排列成新的动态扭曲。在一个分歧分支上，DWF 可以通过将散布在多个分歧静态扭曲中的线程压缩成更少的非分歧动态扭曲，从而提高应用程序的整体 SIMD 效率。通过这种方式，DWF 可以捕捉到 MIMD 硬件在 SIMD 硬件上的显著部分收益。然而，DWF 要求扭曲在短时间窗口内遇到相同的分歧分支。这种时间依赖性使得 DWF 对扭曲调度策略非常敏感。

Fung 和 Aamodt [2011] 的后续工作识别了 DWF 的两种主要性能病态：（1）一种贪婪调度策略可能会使某些线程饿死，导致 SIMD

效率降低；并且（2）DWF中的线程重组增加了非合并内存访问和共享内存银行冲突。这些病态导致DWF减缓许多现有的GPU应用程序。此外，依赖于静态波浪中的隐式同步的应用程序在DWF中执行不正确。

上述病理可以通过一种改进的调度策略部分解决，该策略有效地将计算内核分为两组区域：发散区域和非发散（一致）区域。发散区域从 DWF 中显著受益，而一致区域虽然没有分支发散，但容易受到 DWF 病理的影响。我们发现，通过强制 DWF 将标量线程重新排列回一致区域中的静态波段，可以显著减少 DWF 病理的影响。

线程块压缩。线程块压缩（TBC）[Fung and Aamodt, 2011] 基于这一见解，观察到将线程重新排列成新的动态波束并不会带来额外的好处。相反，重新排列，或 *compaction*，只需在分歧分支之后、分歧区域开始时，以及在其重新汇聚点之前发生，即一致区域开始之前。我们注意到现有的每波束SIMT栈（在第3.1.1章中描述）在分歧分支的重新汇聚点隐式地同步了分歧到不同执行路径的线程，在执行一致区域之前将这些分歧的线程合并回一个静态波束。TBC 扩展了 SIMT 栈，以涵盖同一核心中执行的所有波束，强制它们在分歧分支和重新汇聚点处同步和压缩，以实现稳健的DWF性能收益。然而，在每个分歧分支处同步核心内的所有波束进行压缩可能会大大减少可用的线程级并行性（TLP）。GPU架构依赖于丰富的TLP来容忍流水线和存储器延迟。

TBC 在 SIMD 效率和 TLP 可用性之间达成了一项妥协，限制压缩仅在 *thread block* 内进行。GPU 应用通常在单个核心上并发执行多个线程块，以重叠同步和内存延迟。TBC 利用这种软件优化在分歧分支处重叠压缩开销——当一个线程块中的波束在分歧分支处同步以进行压缩时，其他线程块中的波束可以保持硬件繁忙。它扩展了每个波束的 SIMT 栈，以涵盖线程块中的波束。波束调度逻辑使用这个线程块范围的 SIMT 栈来确定线程块中的波束何时同步并被压缩成一组新的波束。其结果是一个更加稳健和简单的机制，捕捉了 DWF 的大部分好处，而没有病态行为。

大扭曲微架构。大扭曲微架构 [Narasiman et al., 2011] 扩展了SIMT堆栈，类似于TBC，以管理一组扭曲的重汇聚。然而，LWM要求组内的扭曲完全同步执行，而不是限制在分支和重汇聚点的压缩，以便可以在每条指令上对该组进行压缩。这比TBC进一步减少了可用的TLP，但允许

LWM 通过有条件的指令和无条件跳转执行压缩。类似于 TBC，LWM 将在同一核心上运行的波浪分成多个组，并限制压缩仅在组内发生。它还选择了一种更复杂的计分板微架构，以线程粒度跟踪寄存器依赖性。这允许组内的一些波浪比其他波浪稍微提前执行，以补偿由于锁步执行而失去的 TLP。

压缩适宜性预测器。Rhu和Erez [2012] 扩展了TBC，提出了一个压缩适宜性预测器（CAPRI）。该预测器识别在每个分支处将线程压缩为少量扭曲的有效性，并仅在预测压缩带来好处的分支处同步线程。这恢复了由于无益的停顿和与TBC的压缩而损失的线程级并行性（TLP）。Rhu和Erez [2012] 还表明，一个类似于单级分支预测器的简单基于历史的预测器就足以实现高准确性。

内部变形压缩。Vaidya 等人 [2013] 提出了一种低复杂度的压缩技术，该技术有利于宽 SIMD 执行组，该执行组在较窄的硬件单元上执行多个周期。他们的基本技术将单个执行组划分为多个与硬件宽度相匹配的子组。遭受分歧的 SIMD 执行组可以通过跳过完全空闲的子组在窄硬件上运行得更快。为了创建更多完全空闲的子组，他们提出了一种混合机制，在发生分歧时将元素压缩到更少的子组中。

同时扭曲交织。Brunie 等人 [2012] 提出了同时分支和扭曲交织（SBI 和 SWI）。他们扩展了 GPU SIMT 前端，以支持每个周期发布两条不同的指令。他们通过将扭曲的宽度扩大到原始大小的两倍来弥补这种增加的复杂性。SWI 从一个遭受分歧的扭曲与另一个分歧扭曲共同发布指令，以填补分支分歧留下的空隙。

寄存器文件微架构的影响

为了避免在SIMT核心之间引入额外的通信流量，硬件压缩提案通常在SIMT核心内部进行局部处理。由于压缩后的线程都位于同一个核心，共享相同的寄存器文件，因此可以通过更灵活的寄存器文件设计在不移动其架构状态的情况下进行压缩 [Fung 等，2007]。

如本章前面讨论的，GPU寄存器文件采用大型单端口SRAM银行实现，以最大限度地提高其面积效率。同一个warp中的线程寄存器存储在同一个SRAM银行中的连续区域中，以便可以通过一个宽端口一起访问。这允许高带宽的寄存器文件访问，同时分摊寄存器文件访问控制硬件的成本。硬件warp压缩创建了可能不符合这种寄存器安排的动态warp。Fung等人[2007]提出了一种更灵活的寄存器文件设计，采用窄端口的SRAM银行。这种设计具有更多的SRAM银行，以维持相同的带宽。

动态微内核。Steffen 和 Zambreno [2010] 改进了 GPU 上光线追踪的 SIMD 效率，使用了 *dynamic micro-kernels*。程序员可以使用原语将数据依赖循环中的迭代分解为连续的微内核启动。这种分解本身并不会提高并行性，因为每次迭代都依赖于前一次迭代的数据。相反，启动机制通过将剩余的活跃线程紧凑成少量的波段，从而改善了同一核心中不同线程之间的负载不平衡。它还不同于其他硬件波段紧凑技术，因为紧凑过程迁移了线程和它们的架构状态，使用每个核心的临时存储器作为中转区域。

3.4.1节总结了一系列在软件中实现扭曲压缩的研究，这些研究不需要更灵活的寄存器文件设计。相反，这些提案引入了额外的内存流量，以将线程从一个SIMT核心重新定位到另一个。

软件中的 Warp 压缩

在现有的GPU上，提高应用程序的SIMD效率的一种方法是通过软件波压缩——使用软件根据其控制流程行为对线程/工作项进行分组。重新分组涉及在内存中移动线程及其私有数据，这可能会引入显著的内存带宽开销。以下是我们重点介绍的几项关于软件压缩技术的研究。

条件流 [Kapasi et al., 2000] 将这一概念应用于流计算。它将具有潜在分歧控制流的流处理器的计算内核拆分为多个内核。在分歧分支处，内核根据每个数据元素的分支结果将其数据流拆分为多个流。然后，每个流由一个单独的内核处理，并在控制流分歧的末尾合并。

Billeter 等人 [2009] 提出了使用并行前缀和来实现 SIMD *stream compaction*。流压缩将具有不同任务的元素流重新组织为相同任务的紧凑子流。该实现利用了 GPU 片上暂存器的访问灵活性，以实现高效率。Hoberock 等人 [2009] 提出了一种用于光线追踪的延迟阴影技术，该技术使用流压缩来提高在具有多种材质类别的复杂场景中像素阴影的 SIMD 效率。每种材质类别都需要其独特的计算。一个同时处理每种材质类别计算的像素着色器在 GPU 上运行效率低下。流压缩将命中相似材质类别的光线分组，从而使 GPU SIMD 硬件能够高效地为这些像素执行着色器。

Zhang 等 [2010] 提出了一个运行时系统，该系统动态地将线程重新映射到不同的 warp 以提高 SIMD 效率以及内存访问的空间局部性。该运行时系统具有管道化的特点，CPU 执行动态重新映射，而 GPU 则对重新映射的数据/线程进行计算。

Khorasani 等人 [2015] 提出了 *Collective Context Collection* (CCC)，一种编译器技术，用于转换给定的 GPU 计算内核，以应对潜在的分支分歧惩罚。

为了提高其在现有GPU上的SIMD效率。CCC专注于计算核心，在这些核心中，每个线程在每一步执行不规律的计算量，例如在不规则图中的广度优先搜索。CCC首先转换计算核心，使得每个线程处理多个节点，而不再是每个线程分配一个节点（或在其他应用中的任务），节点与warp（注意：不是线程）的分配在内核启动之前就已经确定。然后，CCC将计算核心转换，使得warp中的每个线程能够将任务的上下文卸载到存储在共享内存中的warp特定堆栈中。经历低SIMD效率的warp可以将任务卸载到堆栈中，并使用这些卸载的任务来填补空闲线程，处理后来的任务集合。实际上，CCC通过将来自多个warp的任务分组到更少的warp中，执行“ warp压缩 ”，然后通过存储在快速的片上共享内存中的warp特定堆栈，将不同的任务压缩到每个warp中的更少迭代中。

在一个扭曲中的线程分配影响
在本书研究的基线GPU架构中，具有连续线程ID的线程被静态合并在一起形成波（warps）。关于线程静态分配到波或波中的通道的学术研究较少。这种默认的顺序映射对于大多数工作负载有效，因为相邻线程倾向于访问相邻数据，从而提高内存合并效率。然而，一些研究已探讨过替代方案。

SIMD Lane 置换。Rhu 和 Erez [2013b] 观察到，将线程 ID 顺序映射到一个 warp 中的连续线程对于本节前面描述的 warp 压缩技术是次优的。大多数 warp 压缩和形成工作的一个关键限制是，当线程被分配到一个新的 warp 时，它们不能被分配到不同的 lane，否则它们的寄存器文件状态将不得不移动到向量寄存器中的不同 lane。Rhu 和 Erez 观察到程序的结构会将某些控制流路径偏向于某些 SIMD lane。这种偏见使得实现压缩变得更加困难，因为走相同路径的线程往往位于相同的 lane，从而阻止这些线程合并在一起。Rhu 和 Erez 提出了几种不同的线程映射置换，以消除这些程序性偏见，并显著提高压缩率。

内部波段循环压缩。 Vaidya 等人 [2013] 利用 SIMD 数据路径的宽度并不总是等于波段宽度这一事实。 例如，在 NVI [2009] 中，SIMD 宽度为 16，但波段大小为 32。 这意味着一个 32 线程的波段在 2 个核心周期内执行。 Vaidya 等人 [2013] 观察到，当出现分歧时，如果顺序 SIMD 线程被屏蔽以执行某条指令，则该指令可以只在一个周期内发出，跳过被屏蔽的车道。 他们将这一技术称为循环压缩。 然而，如果被屏蔽的线程不连续，则基本技术不会带来任何性能提升。 为了解决这个问题，他们提出了一种混合循环压缩技术，重新安排线程在各个车道中的分布，以创造更多循环压缩的机会。

扭曲标量化。其他研究，例如[Yang et al., 2014]的工作，认为当一个warp中的线程操作相同数据时，SIMT编程模型效率低下。一些解决方案建议在管道中包含一个标量单元，以处理编译器或程序员可以事先识别为标量的工作。AMD的图形核心下一代（GCN）架构为此目的包含一个标量管道。有关更多细节，请参见第3.5节。

3.4.2 内部扭曲发散路径管理

虽然具有即时后主导重合点的SIMT栈可以处理任意控制流下的分支分歧，但在各个方面仍然可以进一步改进。

1. 脱离的线程可以在不同的分支目标上交错执行，从而利用SIMD硬件中的空闲周期。
2. 在一个分支的直接后主导者是确定的收敛点时，分支到不同目标的线程可能能够在分支的直接后主导者之前收敛。

以下小节强调了几项旨在改善SIMT堆栈在这两个方面的工作。

多路径并行性

当一个Warp在分支处发生分歧时，线程被分成多个组，称为*warp-splits*。每个Warp分裂由跟随相同分支目标的线程组成。在基线中，*single path*，SIMT堆栈，来自同一Warp的Warp分裂是按顺序执行的，直到Warp分裂达到其重新汇聚点。这种序列化适合于相对简单的硬件实现，但对于功能正确性并不是必需的。Warp中的线程具有独立的寄存器，并通过内存操作和诸如屏障的同步操作进行显式通信。换句话说，来自同一Warp的每个Warp分裂可以在*parallel*中执行。我们称这种执行模式为*multi-path execution mode*。

尽管不同的warp-split可能无法在相同的硬件上以相同的周期执行（毕竟，它们运行不同的指令），它们可以像多个warp在相同的数据路径上交错执行一样，在相同的硬件上交错执行。通过这种方式，多路径执行模式提高了应用程序中的线程级并行性（TLP），以容忍内存访问延迟。即使SIMD效率未提高，多路径执行也提升了内存受限应用程序的整体性能，而SIMT核心有大量空闲周期可以用于填充有用的工作。

例子 3.1 展示了一个短的计算内核，可能从多路径执行中受益。在这个例子中，两个分支目标中的代码路径都包含从内存加载。在单路径 SIMT 堆栈中，块 B 和 C 的每个都被串行执行，直到相应的 warp-split 到达块 D（重汇聚点），即使当 warp-split 在等待来自内存的数据时停滞。这使整个 warp 停滞，导致数据通路中出现空闲周期，如下所示。

在图3.18中，这必须由其他warp的工作填充。通过多路径执行，块B和C的warp拆分可以交错执行，从而消除由内存访问引入的这些空闲周期。

Algorithm 3.1 Example of multi-path parallelism with branch divergence.

```
X = data[i];          // block A
if( X > 3 )
  result = Y[i] * i;  // block B
else
  result = Z[i] + i;  // block C
return result;        // block D
```

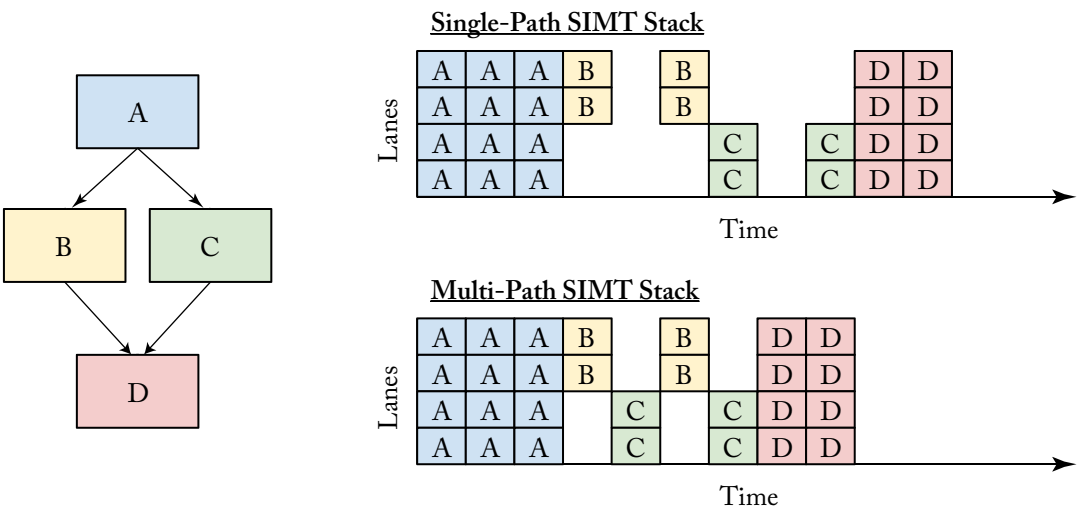


图 3.18：分支分歧时的多路径执行。

动态波形细分。孟等人 [2010] 提出 *dynamic warp subdivision* (DWS) 是首个利用多路径执行中的 TLP 增益的提案。DWS 通过使用波形分割表扩展了 SIMT 栈，以将分歧波形细分为并发波形分割。每个执行分歧分支目标的波形分割可以并行执行，以恢复因内存访问而导致的硬件空闲。波形分割还在内存分歧处创建——当波形中的只有一部分线程在 L1 数据缓存中命中时。DWS 不再等待所有线程获取他们的数据，而是分割波形并允许命中的波形分割先行执行，可能为那些未命中的线程预取数据。

双路径执行。Rhu 和 Erez [2013a] 提出了双路径 SIMT 堆栈 (DPS)，它通过限制每个波动同时执行仅两个并发的波动拆分，解决了 DWS 的一些实现缺陷。虽然这种限制使得 DPS 能够捕获 DWS 优势的很大一部分，但它导致了更简单的硬件设计。DPS 只需将基础 SIMT 堆栈扩展为额外的一组程序计数器 (PC) 和活动掩码，以编码额外的波动拆分。仅在波动的堆栈顶部条目处执行这两个波动拆分；同一波动中的其他波动拆分将暂停，直到其条目达到堆栈顶部。DPS 还伴随着评分板的扩展，以独立跟踪每个波动拆分的寄存器依赖。这允许双路径执行模型在基线评分板下实现比 DWS 更大的线程级并行性 (TLP)。

多路径执行。ElTantaway等人[2014]通过多路径执行模型（MPM）消除了双路径限制。MPM用两个表替代了SIMT堆栈：一个维护来自分叉波中的波分裂集合的波分裂表，以及一个与相同重新汇聚点同步所有波分裂的重新汇聚表。

在一个分歧分支处，重聚表中创建一个新条目，该条目包含分歧分支的重聚点（其直接后主导者）。在warp-split表中创建多个（通常是两个）条目，每个条目对应一个warp-split。每个warp-split条目维护当前warp-split的程序计数器（PC）、其活动掩码、重聚PC（RPC）以及指向重聚表中相应条目的R-index。warp-split表中的每个warp-split在其PC == RPC之前都可用于执行。此时，相应的重聚表条目被更新，以反映来自该warp-split的线程已到达重聚点。当所有待处理线程到达重聚点时，重聚表条目被解除分配，并创建一个新的warp-split条目，其中重聚的线程处于活动状态，从RPC开始。

MPM 还扩展了计分板，以跟踪每个线程的寄存器依赖性，而无需为每个线程完全复制计分板（这样做将使 MPM 不切实际，因为这样会显著增加面积开销）。这是一个关键扩展，使得 warp-split 能够以真正独立的方式执行——没有这个扩展，一个 warp-split 的寄存器依赖性可能会被误认为是来自同一 warp 的另一个 warp-split 的依赖性。

MPM 进一步扩展了机会性早期重新汇聚，提升了非结构化控制流的 SIMD 效率（见第 3.4.2 节）。

DWS以及本节讨论的其他技术与第3.4.1节中讨论的变换压缩技术是正交的。例如，TBC中的块宽SIMT堆栈可以通过DWS进行扩展，以提升可用的TLP。

更好的收敛性

后支配（PDOM）基于栈的重新汇聚机制 [Fung et al., 2007, Fung et al., 2009] 使用通过统一算法识别的重新汇聚点，而不是通过将源代码中的控制流习语转换为指令来实现 [AMD, 2009, Coon and

Lindholm, 2008, Levinthal 和 Porter, 1984]。作为重聚点选择的分歧分支的直接后继是程序中分歧线程 *guaranteed* 重新聚合的最早点。在某些情况下，线程可以在 *earlier point* 处重新聚合，如果硬件能够利用这一点，将提高 SIMD 效率。我们认为这一观察动机促使近年来 NVIDIA GPU 中包含 `break` 指令 [Coon 和 Lindholm, 2008]。

示例 3.2 中的代码（来自[Fung 和 Aamodt，2011]）展示了这种早期的重新汇聚。它导致了图 3.19 中的控制流图，其中边缘标记了单个标量线程沿该路径的概率。块 `F` 是 `A` 和 `C` 的直接后支配者，因为 `F` 是 *all* 从 `A`（或 `C`）开始的路径相交的第一个位置。在基线机制中，当一个 warp 在 `A` 处分歧时，重新汇聚点被设置为 `F`。然而，从 `C` 到 `D` 的路径很少被遵循，因此在 *most* 的情况下，线程可以更早地在 `E` 处重新汇聚。

Algorithm 3.2 Example for branch reconvergence earlier than immediate post-dominator.

```
while (i < K) {
  X = data[i];       // block A
  if( X == 0 )
    result[i] = Y;   // block B
  else if ( X == 1 ) // block C
    break;           // block D
  i++;               // block E
}
return result[i];    // block F
```

可能收敛点。Fung 和 Aamodt [2011] 提出了用 *likely convergence points* 扩展 SIMT 栈。这一扩展在每个 SIMT 栈条目中添加了两个新字段：一个是可能收敛点 (LPC) 的程序计数器 (PC)，另一个是 LPos，这是一个指针，用于记录当分支具有与直接后支配者不同的可能收敛点时创建的特殊可能收敛条目的栈位置。可以通过控制流分析或配置文件信息（可能在运行时收集）来识别每个分支的可能收敛点。Fung 和 Aamodt [2011] 的提案将可能收敛点限制为最靠近的封闭向后分支，以捕捉循环内“ `break` ”语句的影响 [Coon 和 Lindholm, 2008]。

当一个波浪在一个可能收敛点的分支处发散时，三个条目被推入SIMT栈中。第一个条目是LPC条目，为分支的可能收敛点创建。与基线机制相同，分支的取值和落空情况各自创建两个其他条目。每个其他条目的LPC字段被填充为发散分支的可能收敛点，LPos字段被填充为LPC条目的栈位置。LPC条目的RPC设置为直接后支配者，即确定性

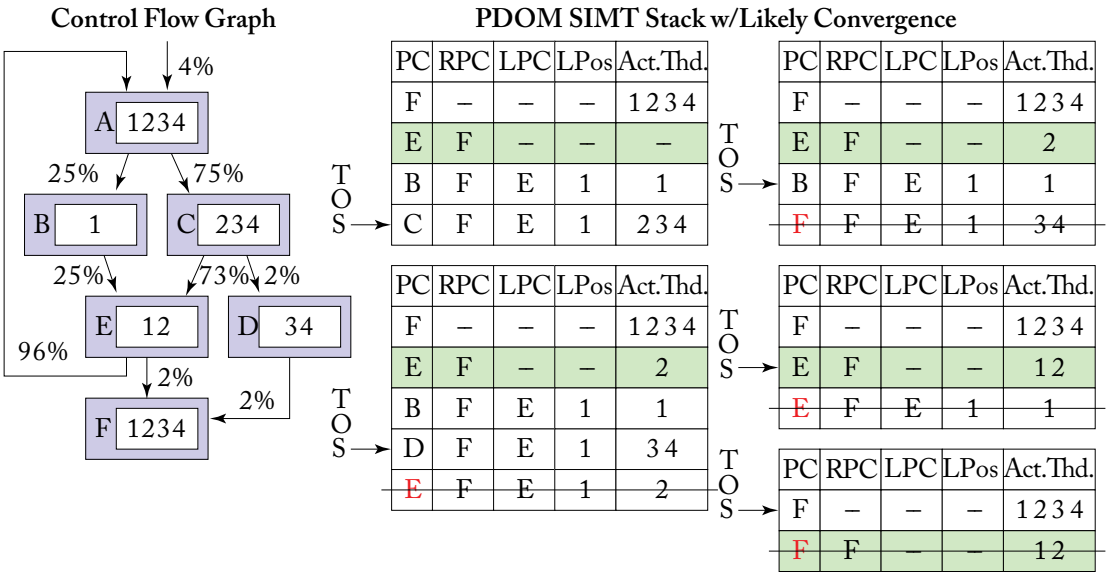


图 3.19：早期重新汇聚点 *before*，直接后支配者。可能的汇聚点捕获了在 E 的早期重新汇聚。

重聚点，发散分支的，使得此条目中的线程将重新聚合到确定的重聚点。

当一个warp在SIMT栈的顶部条目中执行时，它比较其程序计数器(PC)与RPC字段（就像它在基线SIMT栈中一样）以及LPC字段。如果PC == LPC，SIMT栈就会被弹出，并且这个弹出的条目中的线程会合并到LPC条目中。否则，如果PC == RPC，SIMT栈则简单地弹出——RPC条目已经在其活动掩码中记录了这些线程。当LPC条目到达SIMT栈的顶部时，它的执行方式与任何其他SIMT栈条目相同，或者如果其活动掩码为空，则直接弹出。
线程前沿。Diamos 等人 [2011] 完全离开了SIMT堆栈，而是提出在分歧后通过 *thread frontiers* 重新合并线程。一种支持线程前沿的编译器根据其拓扑顺序对内核中的基本块进行排序。通过这种方式，在更高PC处执行的指令的线程永远无法跳转到更低PC的指令。循环通过将循环出口放在循环体的末尾来处理。通过这种排序的代码布局，一个分歧的波速最终将通过优先处理具有较低PC的线程（允许它们赶上）来重新合并。

与具有直接后支配重聚的SIMT栈相比，通过线程边界的重聚为具有非结构化控制流的应用程序带来了更高的SIMD效率。多表达式条件语句的评估语义和异常的使用都可以生成具有非结构化控制流的代码。扩展了可能性的SIMT栈

收敛点可以在具有非结构化控制流的应用程序上产生类似的SIMD效率提升；然而，SIMT堆栈中的每个条目可能只有有限数量的可能收敛点，而线程前沿方法则没有这样的限制。

机会主义早期重聚合。ElTantaway et al. [2014] 提出了 *opportunistic early reconvergence* (OREC)，提升了GPU应用中无结构控制流的SIMD效率，而无需任何额外的编译器分析。OREC 基于同一文献中引入的多路径 (MP) SIMT 栈（参见第3.4.2节）。MP SIMT 栈使用一个单独的warp-split表，保存当前可供执行的warp-splits集合。在一个分叉分支处，使用分支目标PC和分叉分支的重聚合PC创建新的warp-splits。通过OREC，硬件不是简单地将这些新的warp-splits插入warp-split表中，而是搜索warp-split表，寻找具有相同起始PC和RPC的现有warp-split。如果存在这样的warp-split，硬件将在重聚合表中创建一个早期重聚合点，以在原始RPC之前重聚这两个warp-split。早期重聚合点在特定的PC上同步这两个warp-split，以便即使现有warp-split已经通过分叉路径推进，也可以合并。在 ElTantaway et al. [2014] 中，早期重聚合点是现有warp-split的下一个PC。

3.4.3 添加MIMD能力

以下提案通过引入一定程度的MIMD能力来改善GPU对发散控制流的兼容性。所有这些提案都提供了两种操作模式：

- 一种SIMD模式，在这种模式下，前端向一个warp中的所有线程发出一条指令以供执行；或者
- 一种 MIMD 模式，其中前端为每个线程在分歧的波浪中发出不同的指令。

当一个波束没有分歧时，它以SIMD模式执行，以捕获波束中线程所展现的控制流局部性，其能量效率与传统的SIMD架构相当。当波束发生分歧时，它切换到MIMD模式。在这种模式下，波束的运行效率较低，但性能损失低于传统SIMD架构的损失。

向量线程架构。向量线程（VT）架构 [Krashinsky et al., 2004] 结合了 SIMD 和 MIMD 架构的多个方面，旨在捕捉两种方法的最佳特性。VT 架构具有一组通道，这些通道连接到一个公共的 L1 指令缓存。在 SIMD 模式下，所有通道从 L1 指令缓存直接接收指令以进行同步执行，但每个通道可以切换到 MIMD 模式，以其自己的速率运行。

与来自其 L0 缓存的指令保持同步。Lee 等人 [2011] 最近与传统的 SIMT 架构（例如 GPUs）进行的比较表明，VT 架构在常规并行应用中具有相当的效率，而在不规则并行应用中表现得更加高效。

时间SIMT。时间SIMT [Keckler et al., 2011, Krashinsky, 2011] 允许每个通道以MIMD方式执行，类似于VT架构。然而，它不是在所有通道中以同步方式运行一个warp，而是通过单个通道时间复用warp的执行，每个通道运行一组独立的warp。时间SIMT通过仅为整个warp提取每条指令一次，实现了SIMD硬件的效率。这将控制流的开销在时间上进行摊销，而传统的SIMD架构则在多个通道中在空间上摊销相同的开销。

可变扭曲大小架构。可变扭曲大小（VWS）架构 [Rogers 等，2015] 包含多个（例如，8）切片，每个切片都包含一个取指和译码单元，以便每个切片可以同时执行不同的指令，类似于 VT 和时间 SIMT。与通过窄数据路径时间复用大型扭曲不同，VWS 中的每个切片由窄（4宽）扭曲组成。这些窄扭曲随后被分组为更大的执行实体，称为 *gangs*。每个 gang 包含来自每个切片的一个扭曲。

在没有分支分歧的应用中，一组中的各种执行是锁步进行的，从共享提取单元和共享 L1 指令缓存中获取指令。当遇到分支分歧（或内存分歧）时，该组会分裂成多个子组。这些新子组可能会进一步分裂，直到每个波浪（warp）都在自己的子组中。在那时，这些单波浪子组通过切片的提取单元和私有 L0 指令缓存各自独立地执行。这些分裂的子组通过硬件进行比较个别子组的程序计数器（PC），并有机会合并回原始组。如果它们都匹配，原始组将被重新创建。Rogers 等人 [2015] 还建议在第一个分歧分支的直接后支配点插入一个组级同步屏障。

这本书还评估了每个切片中L0指令缓存的容量对性能的影响，以及与共享的L1指令缓存带宽的关系。在非并联模式下，各个切片中的L0缓存可能同时从L1缓存请求指令，从而造成带宽瓶颈。他们的评估表明，即使对于不同的应用程序，256字节的L0缓存也能够过滤掉大部分对共享L1缓存的请求。因此，L1缓存可以仅凭基线SIMT架构的2×带宽来覆盖大部分带宽不足的问题。

同时分支交错。Brunie 等人 [2012] 在线程块压缩发布后提出了同时分支和波动交错（SBI 和 SWI）。他们扩展了 GPU 的 SIMT 前端，以支持每个周期发出两条不同的指令。当 SBI 在遇到分支分歧时同时发出来自同一波的指令。与此同时执行分歧分支的两个目标显著消除了其性能惩罚。

基线SIMT栈对于每个warp的面积需求仅为32 × 64位（或使用AMD GCN中的优化时低至6 × 64位）。虽然与SIMT核心中的寄存器文件相比，这个面积很小，但该面积与GPU中在飞行中的warp数量以及每个warp的线程数量直接相关。此外，在分支发散罕见的典型GPU应用中，SIMT栈占用了原本可以用于通过其他方式提升应用吞吐量的面积。一些提案用替代机制替换SIMT栈，这些机制在warp未遇到任何分支发散时可以共享资源并以其他方式使用。

SIMT堆栈在标量寄存器文件中。AMD GCN [AMD, 2012] 具有一个由工作组中的所有线程共享的标量寄存器文件。它的寄存器可以用作选择寄存器，以控制工作组中每个线程的活动。当编译器检测到计算内核中可能存在的分歧分支时，它使用这个标量寄存器文件在软件中模拟SIMT堆栈。GCN架构具有特殊指令，以加速SIMT堆栈的模拟。

一种优化方法是最小化支持最坏情况下发散所需的标量寄存器数量，优先执行活跃线程较少的目标。这允许以$\log_2(\#threads\ per\ warp)$个标量寄存器支持最坏情况发散，这比基线SIMT栈所需的条目少得多。此外，当计算内核没有潜在的发散分支时，编译器可以将为SIMT栈保留的标量寄存器用于其他标量计算。

线程前沿。如3.4.2节所述，Diamos等人[2011]用*thread frontiers*替代了SIMT堆栈。通过线程前沿，每个线程在寄存器文件中保持自己的程序计数器(PC)，并且代码被拓扑排序，从而保证分支的重汇点始终具有更高的PC。当一个warp发生分叉时，它总是优先选择在所有线程中PC最低的线程。这组线程被称为warp的线程前沿。优先执行前沿中的线程隐含地迫使程序中更前面的所有线程在重汇点等待以便被合并。

由于每线程的程序计数器（PC）仅在计算内核包含潜在的分歧分支时需要，因此编译器只需在这些计算内核中分配一个PC寄存器。在其他计算内核中，额外的寄存器存储有助于提高warp占用率，增加每个SIMT核心能够承载的warp数量，从而更好地容忍内存延迟。

无栈SIMT。Asanovic等人[2013]提议扩展时间SIMT架构，增加一个`syncwarp`指令。在这个提议中，warp中的线程在计算内核的*convergent regions*中以锁步的方式执行，编译器保证warp永远不会发生分歧。在一个分歧分支处，warp中的每个线程都根据其私有PC遵循自己的控制流路径，利用时间SIMT架构中的MIMD能力。

编译器在分支重合点放置一个 `syncwarp` 指令。这迫使所有在分歧的波中线程在进入计算内核的另一个收敛区域之前，在重合点同步。

尽管该机制无法捕捉到嵌套分歧分支可能出现的重新汇聚，但它仍然是一个更便宜的替代方案，可以为很少出现分支分歧的GPU应用程序提供可与基线SIMT栈相媲美的性能。本文介绍了一种结合的汇聚和变体分析，允许编译器确定在任意计算内核中符合*scalarization*和/或*affine transfor- mation*的操作。在无栈SIMT的背景下，相同的分析使编译器能够确定任意计算内核中的汇聚和分歧区域。

编译器首先假设所有基本块为 *thread-invariant*。

2. 它将所有依赖于线程 ID 的指令、原子指令以及在易失性内存上的内存指令标记为 *thread-variant*。

3. 它还会迭代地将所有依赖于线程变体指令的指令标记为线程变体。

4. 所有在线程变体分支指令中标记为 *control dependent* 的基本块中的指令也都是线程变体的。本质上，只要不因其他条件被标记为线程变体，线程变体分支的立即后支配者之外的指令可以保持线程不变。

这种分析允许编译器检测到每个warp中所有线程均匀选择的分支。由于这些分支不导致warp分离，编译器不需要插入代码来检测这些分支的动态分离，也不需要在它们的直接后支配点插入`syncwarp`指令以强制重新汇聚。

预测。在将完整的SIMT栈纳入架构之前，具有可编程着色器的GPU已经通过*predications*支持着色器程序中的有限控制流构造，就像传统的矢量处理器一样。预测在现代GPU中仍然是一种低开销的方式来处理简单的if分支，避免了推送和弹出SIMT栈的开销。在NVIDIA的实现中，每条指令都扩展了一个额外的操作数字段，以指定其预测寄存器。预测寄存器本质上是用于控制流的标量寄存器。

Lee et al. [2014b] 提出了一个 *thread-aware prediction algorithm*，将谓词应用扩展到任意控制流，其性能与 NVIDIA 的 SIMT 堆栈相媲美。线程感知的谓词算法在每个分支处扩展了标准控制流依赖图 (CDG)，在每个基本块的预测可以基于其控制流依赖关系进行计算，并进一步严格优化而不破坏功能行为。论文随后描述了两个

基于这种线程感知CDG的优化，以及他们之前工作的收敛性和方差分析 [Asanovic et al., 2013]。

• *Static branch-uniformity optimization* 在编译器可以保证整块波浪 (warp) 中分支能够一致地采取时应用，如收敛分析所推导的那样。在这种情况下，编译器可以用统一的分支指令替换预测生成。 • *Runtime branch-uniformity optimization* 在其他情况下应用。编译器发出共识分支 (`cbranch.ifnone`)，仅在给定空谓词时（即所有线程禁用）被采取。这允许波浪跳过带有空谓词的代码——这是 SIMT 堆栈提供的一个关键优势。这种方法与之前针对向量处理器的努力（如 BOSCC）不同，因为它依赖于结构分析来确定此优化的候选者。

虽然预测和SIMT堆栈在功能上基本上提供相同的功能，并且在能耗和面积成本上相似，但Lee等人 [2014b] 强调了两种方法之间的以下权衡。

• 由于不同的分支目标由不同的预测寄存器保护，编译器可以调度来自不同分支目标的指令，交替执行不同分支目标，以利用线程级并行性（TLP），否则需要更高级的硬件分支分歧管理。 • 预测往往会增加寄存器压力，这反过来又降低了 warp 占用率，并对整体性能造成惩罚。这是因为保守的寄存器分配无法同时为分支的两个侧面重用寄存器。它无法可靠地证明来自不同分支目标的指令在寄存器中操作的不会是独占的通道集合。两个提出的优化中插入的统一和一致性分支指令缓解了这个问题。 • 预测可能通过多种方式影响动态指令计数。在某些情况下，检查统一分支的开销显著增加了动态指令计数。另一个选择是不执行检查，这意味着某些路径会使用空的预测掩码执行。在其他情况下，它移除了维护 SIMT 栈所需的推送/弹出指令。

最后，论文提出了新的指令来减少预测的开销。

• 对于函数调用和间接分支，他们提出了一种新的`find_unique`指令，通过循环串行执行每个分支目标/函数。 • 除了现有的一致性分支指令`cbranch.ifnone`和`cbranch.ifall`)外，`cbranch.ifany`(将有助于减少动态统一分支检测引入的指令计数开销。

3.5. 标量化和仿射执行的研究方向 57 3.5 标量化和仿射执行的研究方向

如第2章所述，GPU计算API，如CUDA和OpenCL，具有类似MIMD的编程模型，允许程序员在GPU上启动大量的标量线程。虽然这些标量线程中的每一个都可以遵循其独特的执行路径，并且可以访问任意内存位置，但在常见情况下，它们都遵循一小组执行路径并执行类似的操作。大多数现代GPU（如果不是全部）通过SIMT执行模型利用了GPU线程之间的汇聚控制流，其中标量线程被分组到在SIMD硬件上运行的波中（详见3.1.1节）。

本节总结了一系列研究，这些研究通过*scalarization*和*affine execution*进一步利用了这些标量线程的相似性。这些研究的关键见解在于观察到*value structure* [Kim et al., 2013]在执行相同计算内核的线程之间的关系。两种类型的值结构*uniform*和*affine*在示例3.3中的计算内核中进行了说明。

统一变量 在计算内核中，对于每个线程具有相同常量值的变量。在算法 3.3 中，变量 `a` 以及字面量 `THRESHOLD` 和 `Y_MAX_VALUE` 在计算内核中的所有线程中都具有统一值。统一变量可以存储在一个标量寄存器中，并被计算内核中的所有线程重用。

仿射变量 一个变量，其值是计算内核中每个线程的线程 ID 的线性函数。在算法 3.3 中，变量 `y[idx]` 的内存地址可以表示为线程 ID `threadIdx.x` 的 *affine* 变换：

```
&(y[idx]) = &(y[0]) + size(int) * threadIdx.x;
```

这种仿射表示可以存储为一对标量值，一个 *base* 和一个 *stride*，这比完全展开的向量要紧凑得多。

在GPU中有多项研究提案关于如何 *detect* 和 *exploit* 均匀或仿射变量。本节的其余部分总结了这些提案的这两个方面。

3.5.1 均匀或仿射变量的检测

在GPU计算内核中检测均匀或仿射变量的主要有两种方法：编译器驱动检测和硬件检测。

编译器驱动的检测

在GPU计算内核中检测均匀或仿射变量存在的一个方法是通过特殊的编译器分析来实现。这是可能的，因为现有的GPU编程模型，CUDA和OpenCL，已经为程序员提供了声明变量的手段。

算法 3.3 计算内核中的标量和仿射操作示例（来自 [Kim et al., 2013]）。

```
__global__ void vsadd( int y[], int a )
{
  int idx = threadIdx.x;
  y[idx] = y[idx] + a;
  if ( y[idx] > THRESHOLD )
    y[idx] = Y_MAX_VALUE;
}
```

在计算内核中始终是常量，并提供线程 ID 的特殊变量。编译器可以执行控制依赖性分析，以检测完全依赖于常量和线程 ID 的变量，并将它们标记为统一/仿射。仅针对统一/仿射变量的操作则是 *scalarization* 的候选项。

AMD GCN [AMD, 2012] 依赖编译器来检测可以由专用标量处理器存储和处理的统一变量和标量操作。

Asanovic 等人 [2013] 引入了一种组合的收敛和变体分析，使编译器能够确定的任意计算内核中适合 *scalariza-*、*tion* 和/或 *affine transformation* 的操作。在计算内核的收敛区域内的指令可以被转换为标量/仿射指令。在计算内核的分歧区域到收敛区域的任何转换中，编译器插入一个 `syncwarp` 指令，以处理两个区域之间由于控制流引起的寄存器依赖。Asanovic 等人 [2013] 采用了这种分析来为时序SIMT架构生成标量操作 [Keckler et al., 2011; Krashinsky, 2011]。

解耦仿射计算 (DAC) [Wang 和 Lin, 2017] 依赖于类似的编译器分析，以提取标量和仿射候选项，从而解耦成一个单独的 warp。Wang 和 Lin [2017] 通过引入发散的仿射分析增强了这个过程，旨在提取从计算内核开始就已经是仿射的指令串。这些仿射指令串从主内核中解耦，形成一个仿射内核，该仿射内核通过硬件队列将数据传递给主内核。

硬件检测
在硬件中检测均匀/仿射变量相较于编译器驱动的检测提供了两个潜在的优势。

1. 这使得可以在原始GPU指令集架构上应用标量化和仿射执行。这节省了与硬件一起共同开发专用标量化编译器的工作。

2. 硬件检测发生在计算内核执行期间。因此，它能够检测到动态出现的均匀/仿射变量，而这些变量在静态分析中被遗漏。

基于标签的检测。Collange 等人 [2010] 引入了一种基于标签的检测系统。在这个系统中，每个 GPU 寄存器都被扩展了一个标签，表明寄存器是否包含均匀、仿射或通用向量值。在计算内核启动时，包含线程 ID 的寄存器的标签被设置为仿射状态。从常量或共享内存中的单一位置广播值的指令会将目标寄存器的标签设置为均匀状态。在内核执行期间，寄存器的状态根据表 3.1 中的简单规则从源操作数传播到目标操作数。虽然这种基于标签的检测几乎没有硬件开销，但它往往是保守的——例如，它保守地将均匀和仿射变量之间的乘法输出视为向量变量。

表3.1：Collange等人 [2010] 提出的均匀和仿射状态传播规则示例。对于每个操作，第一行和第一列显示输入操作数的状态，其余条目显示每个输入操作数状态排列下输出操作数的状态（U = 均匀，A = 仿射，= 向量）。

+	U	A	V
U	U	A	V
A	A	V	V
V	V	V	V

×	U	A	V
U	U	V	V
A	V	V	V
V	V	V	V

<<	U	A	V
U	U	A	V
A	V	V	V
V	V	V	V

FG-SIMT架构 [Kim et al., 2013] 从Collange et al. [2010] 扩展了基于标签的检测机制，更好地支持分支。仿射分支，或比较仿射操作数的分支，如果其中一个操作数是统一的，则通过标量数据通路解决。Kim et al. [2013] 还引入了一个 *lazy expansion* 方案，在此方案中，仿射寄存器在发生分歧分支或条件指令后懒惰地扩展为完整的向量寄存器。这个扩展是必要的，以允许分歧波束中的一部分线程更新目标寄存器中的槽，同时保持其他槽不变——这维护了SIMT执行语义。与在第一次分歧分支后扩展每个仿射寄存器的更天真的、急切的扩展方案相比，懒惰扩展方案消除了许多不必要的扩展。

写回时的比较。Gilani 等人 [2013] 引入了一种更激进的机制，通过在每次向量指令的写回时比较一个波中的所有线程序号的寄存器值来检测均匀变量。当检测到均匀变量时，检测逻辑将写回重定向到标量寄存器文件，并更新内部表以记住寄存器的状态。随后对该寄存器的使用将被重定向到标量寄存器文件。所有操作数来自标量寄存器文件的指令将在单独的标量管道上执行。

Lee 等人 [2015] 使用了类似的检测方案。他们不是使用简单的统一检测器，而是通过注册器值压缩器增强了寄存器写回阶段，该压缩器使用由 Pekhimenko 等人 [2012] 介绍的算法，将传入的值向量转换为一个元组 $< base, delta, immediate >$ (BDI)。

Wong等人[2016]提出了*Warp Approximation*，一个在warp内利用近似计算的框架，此外还在寄存器写回时进行了检测。检测器计算在写回到寄存器文件的所有值中，共享*d*-MSBs的两个给定值中最小的*d-similarity*。具有高于阈值d相似度的寄存器被标记为*similar*，然后用于确定后续依赖指令中近似执行的资格。

像 Lee 等人 [2015] 的提议，G-Scalar [Liu et al., 2017] 在寄存器写回阶段具有一个寄存器值压缩器，但该压缩器采用了一种更简单的算法，只提取在 warp 中所有通道的所有值中使用的公共字节。如果所有字节都是公共的，寄存器包含一个统一变量。任何仅在统一变量上操作的指令都可以被标量化。

G-Scalar还扩展了寄存器值压缩器，以检测在分支分歧下符合标量执行条件的操作。所有之前的提案在warp分歧后都会回退到向量执行。Liu等人[2017]观察到，在许多分支分歧下的指令中，活动通道的操作数值是均匀的。使用这些部分均匀寄存器的指令实际上符合标量执行的条件。然后，他们扩展寄存器值压缩器，以使用特殊逻辑仅检查来自活动通道的值。这大大增加了各种GPU计算工作负载中的标量指令数量。请注意，在分歧下，写入的寄存器没有被压缩。

3.5.2 在 GPU 中利用均匀或仿射变量

GPU的设计可以通过多种方式利用计算内核中的值结构的存在。

压缩寄存器存储

均匀和仿射变量的紧凑表示允许它们在寄存器文件中以更少的位数存储。回收的存储可以用来维持更多的在途变换，从而在使用相同的寄存器文件资源的情况下提高 GPU 对内存延迟的容忍度。

标量寄存器文件。许多提案/设计利用GPU特性中的均匀或仿射变量，为标量/仿射值提供专用寄存器文件。

- AMD GCN架构具有一个标量寄存器文件，可被标量和向量流水线访问。
- FG-SIMT架构[Kim et al., 2013]将统一/仿射值存储在一个单独的仿射SIMT寄存器文件（ASRF）中。ASRF记录每个状态（仿射/统一/向量）。

寄存器，允许控制逻辑检测适合在控制处理器上进行方向执行的操作。

- Gilani等人[2013]提出的动态均匀检测方案将动态检测到的均匀值存储到专用的标量寄存器文件中。

部分寄存器文件访问。Lee 等人 [2015] 将基本、增量和立即数 (BDI) 压缩应用于写回寄存器文件的寄存器。压缩后的寄存器在作为源操作数读回时会被解压缩回正常向量。在该方案中，每个压缩寄存器仍占用与未压缩寄存器相同的存储槽，但仅占用寄存器银行的一部分，因此读取寄存器的压缩表示所需的能量更少。

Warp Approximate架构 [Wong et al., 2016] 通过仅访问与通过相似性检测选择的代表线程对应的通道，从而减少了寄存器读取/写入的能量使用。

类似地，G-Scalar [Liu et al., 2017] 具有压缩寄存器，这些寄存器仅占用分配的部分存储银行或未压缩寄存器，以减少寄存器读取的能耗。

专用仿射变换。解耦仿射计算（DAC）[Wang and Lin, 2017] 将所有编译器提取的仿射变量缓存在专用仿射变换的寄存器中。这个仿射变换与其他非仿射变换共享相同的向量寄存器文件存储，但仿射变换使用每个寄存器条目的单独通道来存储基值以及不同非仿射变换的增量。

标量化操作

除了高效存储，使用均匀或仿射变量的操作可以是 *scalarized*。与其通过 SIMD 数据路径在一个波束中的所有线程中重复相同的操作，不如在单一标量数据路径中执行一次标量操作，从而在此过程中消耗更少的能量。一般来说，如果算术操作的输入操作数仅由均匀或仿射变量组成，则可以将其标量化。

专用标量流水线。AMD 的 GCN 架构具有专用的标量流水线，能够执行编译器生成的标量指令。FG-SIMT 架构 [Kim et al., 2013] 具有一个控制处理器，能够直接执行动态检测到的仿射操作，而无需调用 SIMD 数据通路。

在这两种实现中，标量流水线还处理 SIMD 流水线的控制流和预测。解耦意味着许多与系统相关的特性（例如，与主处理器的通信）也可以卸载到标量流水线，从而使 SIMD 数据通路摆脱实现完整指令集的负担。

时钟门控 SIMD 数据通路。Warp Approximate 架构 [Wong 等, 2016] 和 G-Scalar [Liu 等, 2017] 都在动态检测到的标量指令上执行。

在SIMD数据通路中的通道。当发生这种情况时，其他通道会进行时钟门控以降低动态功耗。

这种方法消除了在专用标量数据路径上支持完整指令集的重复工作，或不得不对要在标量数据路径上实现的子集进行优先排序的需要。例如，G-Scalar [Liu et al., 2017] 可以以相对较低的开销将特殊功能单元支持的指令标量化。

聚合为仿射扭曲。解耦仿射计算（DAC）[Wang 和 Lin, 2017] 将来自多个扭曲的仿射操作聚合为每个SIMT核心的单个仿射扭曲。这个仿射扭曲在SIMD数据通路上执行，与其他扭曲一样，但每个执行的指令同时在多个扭曲的仿射表示上操作。

内存访问加速
当使用均匀或仿射变量表示内存操作（加载/存储）的地址时，内存操作所触及的内存位置是高度可预测的——每个连续的位置由一个已知的步幅分隔。这为各种优化提供了可能。例如，具有已知步幅的内存位置的内存合并远比任意随机位置的合并简单。仿射变量还可以用于通过单个指令而不是通过加载/存储指令的循环来表示大容量传输。

FG-SIMT 架构 [Kim 等，2013] 在控制过程中具有一个特殊的地址生成单元，能够将带有仿射地址的内存访问扩展为实际地址。由于仿射地址在线程之间具有固定的步幅，因此可以使用更简单的硬件将这些仿射内存访问合并到缓存行中。

解耦仿射计算（DAC）[Wang and Lin, 2017] 还具有类似的优化，以利用仿射内存访问中的固定步幅。此外，它使用仿射扭曲在其他非仿射扭曲之前执行，为这些扭曲预取数据。预取的数据存储在 L1 缓存中，稍后通过特殊的出队指令由相应的非仿射扭曲检索。

3.6 寄存器文件架构的研究方向

现代GPU使用大量硬件线程（warp），将它们的执行多路复用到更少（仍然很大数量）的ALU上，以容忍管道和内存访问延迟。为了实现warp之间快速高效的切换，GPU使用硬件warp调度器，并将所有硬件线程的寄存器存储在片上寄存器文件中。在许多GPU架构中，这些寄存器文件的容量相当可观，有时甚至超过最后一级缓存的容量，这归因于GPU中使用的宽SIMD数据通路，以及需要的warp数量，以容忍数百个周期的内存访问延迟。例如，

NVIDIA的Fermi GPU可以支持超过20,000个在飞线程，并具有2 MB的总寄存器容量。

为了最小化寄存器文件存储占用的面积，GPU上的寄存器文件通常通过低端口数的SRAM银行实现。这些SRAM银行被并行访问，以提供支持在宽SIMD管道以峰值吞吐量运行的指令所需的操作数带宽。如本章前面所述，一些GPU使用操作数收集器来协调来自多条指令的操作数访问，以最小化银行冲突惩罚。

访问这些大型寄存器文件在每次访问时会消耗大量的动态能量，而它们的巨大尺寸也导致了高静态功耗。在NVIDIA GTX280 GPU上，寄存器文件消耗了近10%的总GPU功耗。这为在GPU寄存器文件架构上进行创新以减少能耗提供了明确的动力。因此，近年来在这个主题上有大量的研究论文。 本节的其余部分总结了几项旨在实现这一目标的研究提案。

3.6.1 分层寄存器文件

Gebhart 等人 [2011b] 观察到，在一组真实世界的图形和计算工作负载中，最多有 70% 的值仅被读取一次，而仅有 10% 的值被读取超过两次。为了捕捉大多数寄存器值的短生命期，他们提议在 GPU 上扩展主寄存器文件，使用 *register file cache* (RFC)。这形成了寄存器文件的层次结构，并显著减少了对主寄存器文件的访问频率。

在这项工作中，RFC通过FIFO替换策略为每条指令的目标操作数分配一个新的条目。未命中的源操作数不会加载到RFC中，以减少已经很小的RFC的污染。默认情况下，从RFC驱逐的每个值都会写回主寄存器文件。然而，由于这些值中的许多永远不会再次被读取，Gebhart等人[2011b]通过编译时生成的静态活跃信息扩展了仅硬件实现的RFC。指令编码中增加了一个额外位，以指示消耗寄存器值的最后一条指令。最后一次被读取的寄存器在RFC中标记为无效。在驱逐时，它不会被写回主寄存器文件。

为了进一步减少RFC的大小，Gebhart等人[2011b]将其与二级波warp调度器结合。这个二级波warp调度器将执行限制在一个包含*active*个warp的池中，该池仅由每个SIMT核心中一小部分warp组成。这项工作考虑了一个包含4-8个warp的活动warp池，而每个SIMT核心总共有32个warp。RFC仅保留来自活动warp的值，因此更小。波warp在长延迟操作（如全局内存加载或纹理获取）时会被移出活动池。当这种情况发生时，该warp的RFC条目会被刷新，为通过二级调度器激活的其他warp腾出空间。

编译时管理寄存器文件层次结构。Gebhart et al. [2011a] 进一步扩展了这个寄存器文件层次结构，增加了一个最后结果文件（LRF），它缓冲了每个活动warp的最后一条指令产生的寄存器值。该工作还用编译时管理的操作数寄存器文件（ORF）替代了硬件管理的RFC。ORF中值的进出由编译器显式管理。这消除了RFC所需的标签查找。编译器对大多数GPU工作负载中寄存器使用模式也有更全面的视角，从而能够做出更优的决策。该工作还扩展了两级warp调度器，以便编译器指示何时可以将一个warp切换出活动池。这是为了协调ORF的内容与warp的活动性，在warp被切换出之前，将所有活动数据从ORF移动回主寄存器文件。

3.6.2 昏昏欲睡状态寄存器文件

Abdel-Majeed 和 Annavaram [2013] 提出了一种三模式注册文件设计，旨在降低大型 GPU 注册文件的泄漏功率。三模式注册文件中的每个条目可以在开启、关闭和嗜睡模式之间切换。开启模式是正常操作模式；关闭模式不保留寄存器的值；嗜睡模式保留寄存器的值，但在访问之前需要将条目唤醒到开启模式。在本研究中，所有未分配的寄存器处于关闭模式，而所有已分配的寄存器在每次访问后立即进入嗜睡状态。这种策略利用了 GPU 上对同一寄存器进行连续访问之间的长延迟，由于 GPU 上的细粒度多线程，使寄存器在注册文件中有更多时间处于嗜睡模式。GPU 中的长流水线也意味着，从嗜睡状态唤醒寄存器的额外延迟不会导致显著的性能损失。

3.6.3 寄存器文件虚拟化

Tarjan 和 Skadron [2011] 观察到，在等待内存操作时，GPU 线程中的活跃寄存器数量往往较少。他们声称，在某些 GPU 应用中，多达 60% 的寄存器未被使用。他们建议通过寄存器重命名虚拟化物理寄存器，将物理寄存器文件的大小减少最多 50%，或者将同时执行的线程数量翻倍。在所提议的机制中，线程开始执行时没有分配寄存器，物理寄存器会随着指令解码而分配给目标寄存器。Tarjan 和 Skadron 进一步建议，通过采用编译器分析来确定寄存器的最后读取，可以增强物理寄存器的回收。他们提出了“ 最终读取注释 ”，并建议为每个操作数增加“ 一个位以指示它是否是最后读取 ”，并指出这可能需要在指令编码中增加额外的位。

Jeon 等人 [2015] 量化了通过将寄存器溢出到内存来减少 GPU 寄存器文件大小的影响。他们发现，通过采用溢出将寄存器文件的大小减少 50% 会导致执行时间平均增加 73%。他们回顾了在采用乱序执行的 CPU 上进行寄存器重命名时，早期回收物理寄存器的旧提案。

执行。该提案通过添加“元数据指令”来解决所需额外位的问题，以有效编码何时可以回收物理寄存器，并通过寄存器寿命活跃性分析生成这些指令。他们的重要观察是，在确定何时安全回收物理寄存器时，必须考虑分支发散（Kloosterman 等人 [2017] 进一步阐述）。对于一个 128 KB 的寄存器文件，Jeon 等人的重命名技术的直接实现需要 3.8 KB 的重命名硬件。他们表明，通过不重命名寿命较长的寄存器，这一开销可以减少到 1 KB。为了利用这一机会，他们建议仅对逻辑寄存器编号大于编译器确定的阈值的寄存器进行重命名。Jeon 等人进一步建议使用重命名来启用寄存器文件子数组的电源门控。他们评估了其支持寄存器文件虚拟化的详细提案的有效性，通过寄存器重命名显示，确实可以在没有性能损失的情况下减少寄存器文件的大小达到 50%。

3.6.4 分区寄存器文件

Abdel-Majeed 等人 [2017] 引入了 *Pilot Register File*，它将 GPU 寄存器文件分为快速寄存器文件 (FRF) 和慢速寄存器文件 (SRF)。FRF 使用常规 SRAM 实现，而 SRF 则使用接近阈值电压 (NTV) SRAM 实现。与常规 SRAM 相比，NTV SRAM 具有更低的访问能量和更低的泄漏功耗。作为交换，访问 NTV SRAM 的延迟大大变慢，通常需要多个周期（而不是常规 SRAM 的一个周期）。在这项工作中，SRF 的大小明显大于 FRF。每个 warp 在 FRF 中有 4 个条目。关键是使用 FRF 来处理大部分访问，以弥补 SRF 的缓慢。访问 SRF 的额外延迟由操作数收集器处理。FRF 通过使用 FinFET 的背栅控制进一步增强了低功耗模式。这使得非活动 warp 的 FRF 可以切换到低功耗模式。这使得 FRF 在不显式调度 warp 进出活动池的情况下，能够享受到两级调度器的好处。

这项工作不同于层次注册文件，因为不同的分区持有一组独占的寄存器，并且分区在整个warp的生命周期内保持不变。Abdel-Majeed等人[2017]在每次内核启动时使用一个主控CTA来分析最常用的寄存器，而不是使用编译器来确定要放置在FRF中的寄存器集合。这组高使用率的寄存器被记录在一个查找表中，供每个后续的warp从内核启动时访问。

3.6.5 无规列

Kloosterman 等人 [2017] 引入了 *RegLess*，旨在消除寄存器文件，并用操作数暂存缓冲区代替。论文观察到，在相对较短的时间段内，访问的寄存器数量是总寄存器文件容量的一小部分。例如，在 100 个周期内，他们评估的许多应用程序访问的寄存器少于

当使用 GTO 或两级波束调度器时，2048 KB 寄存器文件的 10%。为了利用这一观察，RegLess 使用编译器算法将内核执行划分为区域。区域是在单个基本块内的连续指令。区域之间的边界被选择，以限制活寄存器的数量。使用区域注释，容量管理器（CM）确定哪些波束有资格进行调度。当一个波束开始执行来自新区域的指令时，该区域中使用的寄存器会从分配在全局内存中的后备存储区带入操作数暂存单元（OSU），并可能在 L1 数据缓存中缓存。OSU 本质上是一个由八个存储体组成的缓存，提供足够的带宽以支持每个周期处理两条指令。为了避免在访问 OSU 中数据时出现停滞，CM 在发出区域中的第一条指令之前预加载寄存器。为了管理预加载过程，CM 为每个波束维护一个状态机，以指示下一个区域所需的寄存器是否存在于 OSU 中。为了减少 OSU 与内存层之间产生的内存流量，RegLess 采用了利用仿射值的寄存器压缩技术（见第 3.5 节）。

Kloosterman 等人对他们的提案进行了详细评估，包括 Verilog 合成和提取 RegLess 引入的硬件单元的寄生电容和电阻值。他们的评估显示，512 项目的 OSU 与 2048 KB 寄存器文件相比，性能略有提升，同时仅占用 25% 的空间，并将整体 GPU 能耗降低了 11%。

CHAPTER 4

记忆系统

本章探讨了GPU的内存系统。GPU计算内核通过加载和存储指令与内存系统进行交互。传统的图形应用程序与多个内存空间（如纹理、常量和渲染表面）进行交互。虽然在CUDA等GPGPU编程API中可以访问这些内存空间，但我们将重点讨论本章中GPGPU编程使用的内存空间，特别是实现它们所需的微架构支持。

CPU 通常包括两个独立的内存空间：寄存器文件和内存。现代 GPU 在逻辑上进一步将内存细分为局部内存和全局内存空间。局部内存空间是每个线程私有的，通常用于寄存器溢出，而全局内存则用于多个线程共享的数据结构。此外，现代 GPU 通常实现由程序员管理的共享进程内存，线程在协同线程数组中共同执行时可以共享访问。包括共享地址空间的一个动机是，在许多应用中，程序员知道在计算的某一步骤需要访问哪些数据。通过一次性将所有这些数据加载到共享内存中，他们可以重叠长延迟的离芯片内存访问，并在对这些数据进行计算时避免长延迟的内存访问。更重要的是，在给定时间内可以在 GPU 和离芯片内存之间传输的字节数（DRAM 带宽）相对于可以在同样时间内执行的指令数量是很小的。此外，在离芯片内存和 GPU 之间传输数据所消耗的能量比从片上内存访问数据所消耗的能量高出几个数量级。因此，从片上内存访问数据可以获得更高的性能并节省能源。

我们将对内存系统的讨论分为两个部分，反映出内存的划分：一部分驻留在GPU核心内，另一部分位于连接到离芯片DRAM芯片的内存分区内。

4.1 第一层次内存结构

本节描述了在 GPU 上发现的一级缓存结构，重点关注统一的 L1 数据缓存和“ 共享内存 ”暂存器，以及它们与核心流水线的交互。我们还包括了对 L1 纹理缓存典型微架构的简要讨论。我们讨论了纹理缓存，尽管在 GPU 计算应用中使用有限，但它提供了一些关于 GPU 如何与 CPU 区别的见解和直觉。一项近期的专利描述了如何将纹理缓存和 L1 数据统一起来（例如，如 NVIDIA 所示）。

麦克斯韦和帕斯卡尔 GPU）[Heinrich et al., 2017]。我们将推迟讨论这个设计，直到首先考虑纹理缓存是如何组织的。GPU 中一级内存结构的一个有趣方面是它们在遇到冒险时如何与核心流水线交互。如第 3 章所述，流水线冒险可以通过重新执行指令来处理。我们将在本章中扩展之前关于重放的讨论，重点关注内存系统中的冒险。

4.1.1 临时存储器和L1数据缓存

在CUDA编程模型中，“ 共享内存 ”指的是一个相对较小的内存空间，预计具有低延迟，但可以被给定CTA内的所有线程访问。在其他体系结构中，这种内存空间有时被称为擦除板内存 [Hofstee, 2005]。访问此内存空间的延迟通常可与寄存器文件访问延迟相媲美。实际上，早期的NVIDIA专利将CUDA “ 共享内存 ”称为全局寄存器文件 [Acocella和Goudy, 2010]。在OpenCL中，这个内存空间被称为 “ 局部内存 ”。从程序员的角度来看，使用共享内存时需要考虑的一个关键方面是其有限的容量之外的 *bank conflicts* 潜力。共享内存被实现为静态随机存取内存（SRAM），并在一些专利中 [Minkin等, 2012] 被描述为每条通道一个银行，每个银行具有一个读端口和一个写端口。每个线程可以访问所有银行。当多个线程在同一周期内访问同一银行，并且线程希望访问该银行中的不同位置时，会出现 *bank conflict* 。在详细考虑共享内存如何实现之前，我们首先看一下L1数据缓存。

L1 数据缓存在缓存中维护全局内存地址空间的一个子集。在某些架构中，L1 缓存仅包含未被内核修改的位置，这有助于避免由于 GPU 上缺乏缓存一致性而导致的复杂性。从程序员的角度来看，访问全局内存时一个关键考虑因素是，在给定的 warp 中，不同线程访问的内存位置之间的关系。如果 warp 中的所有线程都访问落在单一 L1 数据缓存块中的位置，并且该块不在缓存中，则只需向下级缓存发送一个请求。这种访问被称为 “ 合并 ”的。如果 warp 中的线程访问不同的缓存块，则需要生成多个内存访问。这种访问被称为未合并的。程序员会尽量避免银行冲突和未合并的访问，但为了简化编程，硬件允许这两种情况。

图4.1展示了一种如Minkin等人[2012]所描述的GPU缓存组织。该设计实现了统一共享内存和L1数据缓存，这是在NVIDIA的Fermi架构中引入的特性，也出现在Kepler架构中。在图的中心是一个SRAM数据阵列5，可以配置[Minkin等人，2013]，部分用于共享内存的直接映射访问，部分作为集合关联缓存。该设计通过在处理银行冲突和L1数据缓存未命中时使用重播机制，支持与指令流水线的无阻塞接口。为了帮助解释该操作的工作原理，

缓存架构，我们首先考虑共享内存访问是如何处理的，然后考虑合并缓存命中，最后考虑缓存未命中和未合并访问。在所有情况下，内存访问请求首先从指令流水线中的加载/存储单元发送到 L1 缓存 1 。内存访问请求由一组内存地址组成，每个线程在一个组中对应一个地址，以及操作类型。

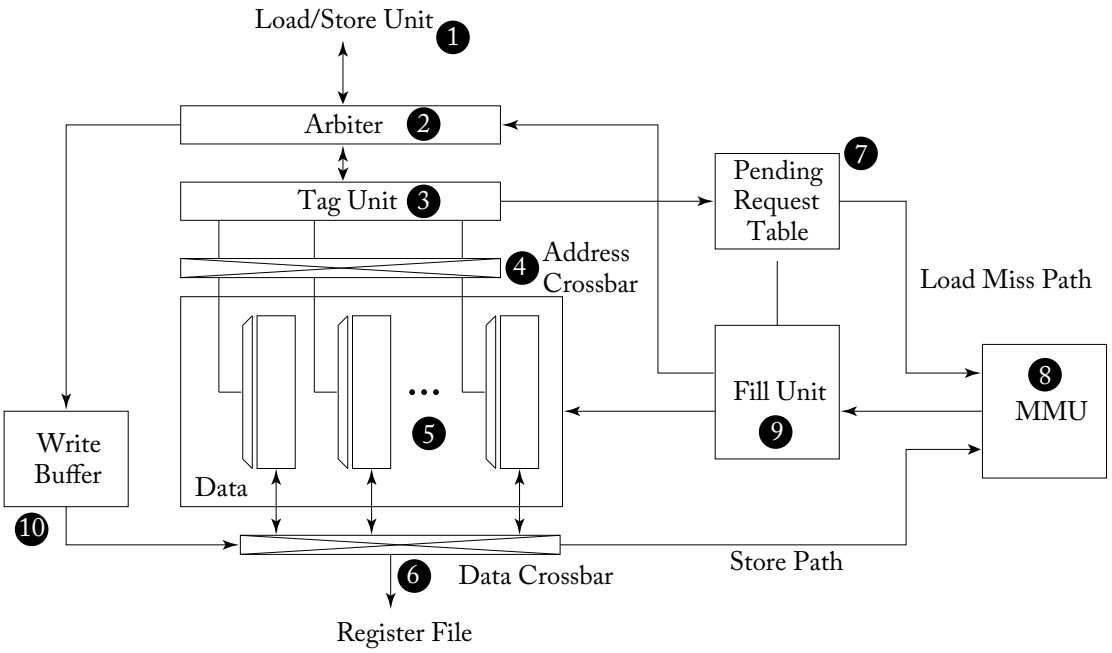


图 4.1：统一的 L1 数据缓存和共享内存 [Minkin et al., 2012]。

共享内存访问操作

对于共享内存访问，仲裁器确定所请求的地址在波束内是否会导致银行冲突。如果所请求的地址会导致一个或多个银行冲突，仲裁器将请求拆分成两部分。第一部分包括波束中一部分线程的地址，这些地址没有银行冲突。这部分原始请求被仲裁器接受以便由缓存进一步处理。第二部分包含那些与第一部分地址发生银行冲突的地址。原始请求的这一部分被返回到指令流水线，必须重新执行。这种后续执行被称为“重放”。原始共享内存请求的重放部分存储存在权衡。虽然通过从指令缓冲区重放内存访问指令可以节省区域，但这会消耗访问大型寄存器文件的能量。为了提高能效，更好的替代方案可能是为重放内存访问指令提供有限的缓冲。

加载/存储单元，并避免在该缓冲区的空闲空间开始耗尽时调度指令缓冲区中的内存访问操作。在考虑重放请求发生什么之前，让我们考虑内存请求的接受部分是如何处理的。

共享内存请求的被接受部分绕过了标签单元3中的标签查找，因为共享内存是直接映射的。在接受共享内存加载请求时，仲裁器安排一个写回事件到指令管道内的寄存器文件，因为在没有银行冲突的情况下，直接映射内存查找的延迟是恒定的。标签单元确定每个线程的请求映射到哪个银行，以控制地址交叉栏4，该栏将地址分配给数据数组内的各个银行。数据数组5中的每个银行宽度为32位，并且拥有自己的解码器，允许独立访问每个银行中的不同行。数据通过数据交叉栏6返回到适当线程的通道中以存储在寄存器文件中。只有对应于warp中活跃线程的通道才会向寄存器文件写入值。

假设共享内存查找的单周期延迟，重放的共享内存请求部分可以在上一个被接受部分之后的一个周期内访问 L1 缓存仲裁器。如果这个重放的部分遇到银行冲突，它会进一步细分为已接受部分和重放部分。

缓存读取操作

接下来，让我们考虑如何处理对全局内存空间的加载。由于只有一部分全局内存空间被缓存到L1，标签单元需要检查数据是否存在于缓存中。尽管数据数组高度银行化，以便各个warp灵活访问共享内存，但对全局内存的访问每个周期仅限于一个缓存块。这个限制有助于减少相对于缓存数据量的标签存储开销，也是在标准DRAM芯片的标准接口的结果。L1缓存块的大小在Fermi和Kepler中为128字节，而在Maxwell和Pascal中则进一步划分为四个32字节的扇区[Liptay, 1968][NVIDIA Corp.]。32字节的扇区大小对应于可以从最近的图形DRAM芯片中以一次访问读取的最小数据大小（例如，GDDR5）。每个128字节的缓存块由在32个银行中每个相同行的32位条目组成。

加载/存储单元 1 计算内存地址，并应用合并规则将一个 Warp 的内存访问分解为单个合并访问，然后将其提供给仲裁器 2。如果可用资源不足，仲裁器可能会拒绝一个请求。例如，如果访问映射到的缓存集中的所有路由均繁忙，或挂起请求表 7 中没有空闲条目，如下所述。假设有足够的资源来处理未命中，仲裁器请求指令流水线在未来的固定周期内安排对寄存器文件的写回，相应于缓存命中。同时，仲裁器还请求标签单元 3 检查该访问是否实际上导致缓存命中或未命中。在发生缓存命中的情况下，所有存储器银行中数据数组 5 的适当行被访问。

数据在指令流水线中返回到寄存器文件中的 6。与共享内存访问的情况一样，仅更新对应于活动线程的寄存器通道。

访问标签单元时，如果确定请求触发了缓存未命中，仲裁器会通知加载/存储单元必须重放请求，并且同时将请求信息发送到待处理请求表（PRT）7。待处理请求表提供的功能与传统的未命中状态保持寄存器 [Kroft, 1981] 在 CPU 缓存内存系统中支持的功能并没有太大不同。NVIDIA 专利中描述的待处理请求表至少有两个版本 [Minkin et al., 2012, Nyland et al., 2011]。与图 4.1 中所示的 L1 缓存架构相关的版本与传统 MSHR 略有相似。数据缓存的传统 MSHR 包含缓存未命中的块地址，以及块偏移量和需要在块填充到缓存时写入的相关寄存器的信息。通过记录多个块偏移量和寄存器来支持对同一块的多个未命中。图 4.1 中的 PRT 支持合并两个对同一块的请求，并记录所需的信息以通知指令流水线重放哪个延迟的内存访问。

图4.1所示的L1数据缓存是虚拟索引和虚拟标签的。与现代CPU微架构对比时，这可能令人感到惊讶，因为现代CPU大多采用虚拟索引/物理标签的L1数据缓存。CPU使用这种组织方式来避免在上下文切换时刷新L1数据缓存的开销[Hennessy和Patterson，2011]。虽然GPU在一个warp发出指令的每个周期都有效地执行上下文切换，但这些warp属于同一个应用程序。即使在GPU中仅限于同时运行一个操作系统应用程序时，基于页面的虚拟内存仍然具有优势，因为它有助于简化内存分配，并减少内存碎片。在PRT中分配一个条目后，内存请求被转发到内存管理单元（MMU）8进行虚拟到物理地址的转换，然后通过交叉开关互连到适当的内存分区单元。正如第4.3节将详细展开的那样，内存分区单元包含一组L2缓存和一个内存访问调度器。内存请求包含有关要访问的物理内存地址和要读取的字节数的信息，以及一个可以在内存请求返回到核心时用于查找PRT中请求相关信息的“ subid ”。

一旦加载的内存请求响应返回给核心，它将通过MMU传递给填充单元9。填充单元反过来使用内存请求中的subid字段在PRT中查找关于请求的信息。这包括填充单元可以通过仲裁器2传递给加载/存储单元的信息，以重新调度加载，并通过在其被放入数据阵列5后锁定缓存中的行来确保其命中缓存。

缓存写操作

图4.1中的L1数据缓存可以支持直写和回写策略。因此，对全局内存的存储指令（写入）可以通过多种方式处理。具体的内存

空间的写入决定写入是被视为直写还是回写。在许多 GPGPU 应用中，访问全局内存通常会表现出非常差的时间局部性，因为内核通常以这样的方式编写：线程在退出之前会将数据写入一个大型数组。对于这样的访问，采用无写分配的直写策略[Hennessy 和 Patterson, 2011]可能是合理的。相反，溢出寄存器至栈的局部内存写入可能会表现出良好的时间局部性，后续的加载操作证明了采用写分配的回写策略是合理的[Hennessy 和 Patterson, 2011]。

要写入共享内存或全局内存的数据首先放置在写入数据缓冲区 (WDB) 10 中。对于未合并的访问或某些线程被遮蔽时，仅写入缓存块的部分内容。如果该块存在于缓存中，则可以通过数据交叉开关 6 将数据写入数据数组。如果数据不在缓存中，则必须首先从 L2 缓存或 DRAM 内存中读取该块。完全填充缓存块的合并写入在使缓存中任何过期数据的标签失效时可能会绕过缓存。

请注意，图4.1中描述的缓存组织不支持缓存一致性。例如，假设在SM 1上执行的线程读取内存位置A，并且该值存储在SM 1的L1数据缓存中，然后在SM 2上执行的另一个线程写入内存位置A。如果SM 1上的任何线程在从SM 1的L1数据缓存中驱逐之前随后读取内存位置A，它将获得旧值而不是新值。为了避免这个问题，从Kepler开始的NVIDIA GPU只允许局部内存访问用于寄存器溢出和堆栈数据，或只读全局内存数据被放置在L1数据缓存中。最近的研究探讨了如何在GPU上启用一致的L1数据缓存[Ren and Lis, 2017，Singh等，2013]以及对明确GPU内存一致性模型的需求[Alglave等，2015]。

4.1.2 L1 纹理缓存

最近，NVIDIA 的 GPU 架构将 L1 数据缓存和纹理缓存结合在一起，以节省面积。为了更好地理解这样的缓存是如何工作的，首先有必要了解一些独立纹理缓存的设计细节。这里涵盖的细节应该有助于提供关于如何为吞吐量处理器开发微架构的额外直觉。这里的讨论大部分基于 Igehy 等人 [1998] 的一篇论文，该论文旨在填补关于工业纹理缓存设计如何容忍缓存未命中导致的长外部延迟的文献空白。最近的行业 GPU 专利 [Minken 等，2010，Minken 和 Rubinstein，2003] 描述了密切相关的设计。由于本书的重点不在于图形，我们仅提供一个简要总结，介绍激励纹理缓存包含的纹理操作。

在3D图形中，希望场景看起来尽可能真实。为了在实时渲染所需的高帧率下实现这种真实感，图形API采用了一种称为纹理映射的技术[Catmull, 1974]。在纹理映射中，一个称为纹理的图像被应用到3D模型的表面上，以使该表面看起来更加真实。例如，纹理可以用来赋予场景中桌子自然木材的外观。为了实现

纹理映射渲染管线首先确定纹理图像中一个或多个样本的坐标。这些样本称为纹素。然后使用这些坐标找到包含纹素的内存位置的地址。由于屏幕上的相邻像素映射到相邻的纹素，并且通常会对附近纹素的值进行平均，因此在纹理内存访问中存在显著的局部性，这可以被缓存利用 [Hakura 和 Gupta, 1997]。

图4.2展示了Igehy等人[1998]描述的L1纹理缓存的微架构。与第4.1.1节中描述的L1数据缓存相比，标签数组2和数据数组5通过FIFO缓冲区3分开。引入这个FIFO的动机是为了隐藏可能需要从DRAM处理的未命中请求的延迟。实际上，纹理缓存的设计是基于缓存未命中将是频繁且工作集合大小相对较小的假设。为了保持标签和数据数组的大小较小，标签阵列基本上在数据阵列之前运行。标签数组的内容反映了数据数组在大约等于未命中请求往返内存的时间后的未来状态。虽然相对于容量有限和未命中处理资源的常规CPU设计，吞吐量有所提高，但缓存命中和未命中在延迟上都经历了大致相同的情况。

详细来说，图4.2所示的纹理缓存的操作如下。加载/存储单元1将计算得到的纹理元素地址发送到标签数组2进行查找。如果访问命中，则数据数组中的数据位置指针将与完成纹理操作所需的其他信息一起放入片段FIFO 3的尾部。当该条目到达片段FIFO的头部时，控制器4使用该指针从数据数组5中查找纹理元素数据并将其返回给纹理过滤单元6。虽然没有详细说明，但对于双线性和三线性过滤（mipmap过滤）等操作，实际上每个片段（即屏幕像素）会进行四次或八次纹理元素查找。纹理过滤单元将纹理元素组合以产生单一颜色值，该颜色值通过寄存器文件返回到指令管道。

在标签查找期间发生缓存缺失的情况下，标签数组通过缺失请求FIFO 8 发送内存请求。缺失请求FIFO向内存系统的更低级别发送请求 9 。通过使用内存访问调度技术 [Eckert, 2008, 2015]，可以提高GPU内存系统中的DRAM带宽利用率，这些技术可能会以无序方式处理内存请求，以减少行切换延迟。为了确保数据数组5的内容反映标签数组2的时间延迟状态，数据必须按顺序从内存系统返回。这是通过使用重排缓冲区10来实现的。

4.1.3 统一纹理和数据缓存

在最近的NVIDIA和AMD GPU架构中，数据和纹理值的缓存采用统一的L1缓存结构。为了以最简单的方式实现这一点，仅缓存那些可以保证为只读的数据值。对于符合这一限制的数据，纹理缓存硬件几乎可以不做修改地使用，除了对寻址逻辑的更改。这样的设计在最近的一项专利中有描述 [Heinrich et al.,

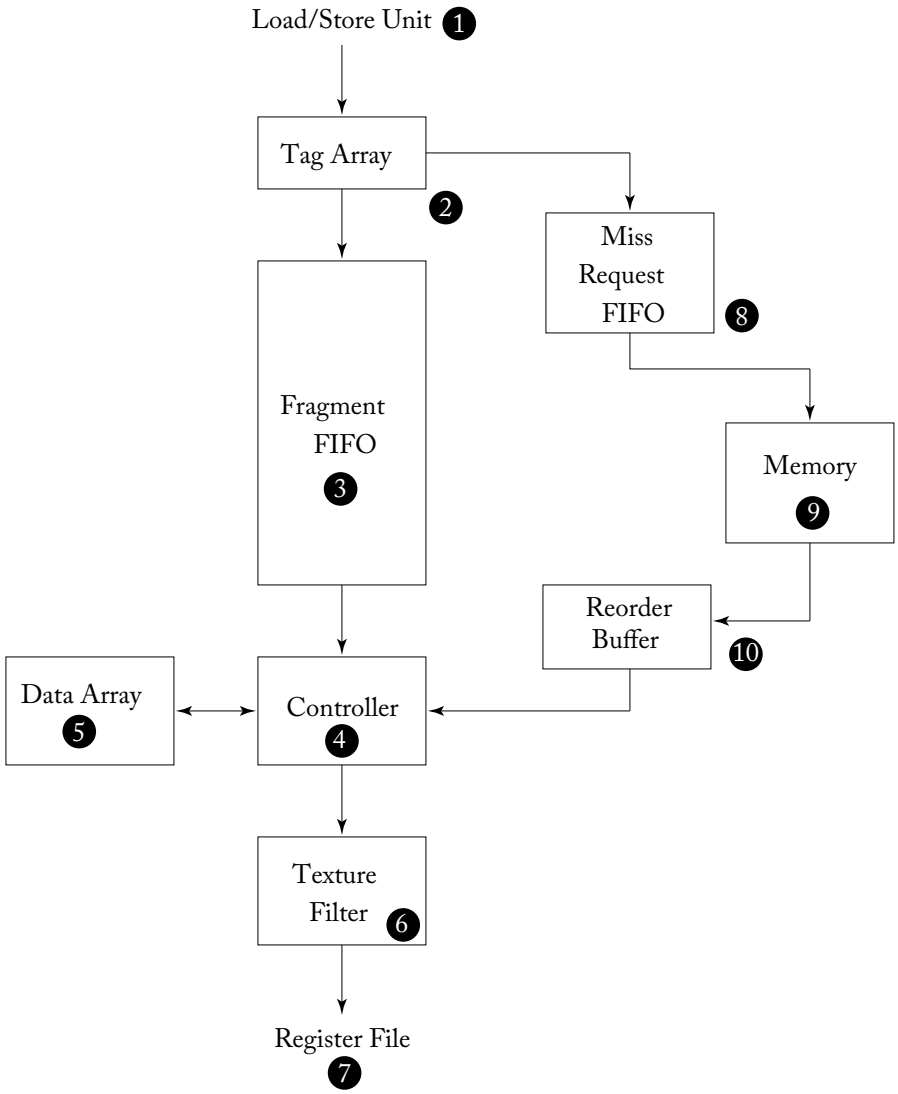


图 4.2: L1 纹理缓存（部分基于 [Igehy et al., 1998] 中的图 2）。

2017]. In AMD's GCN GPU architecture all vector memory operations are processed through the texture cache [AMD, 2012].

为了提供所需的大量内存带宽以支持SIMT核心，高性能GPU通过内存分区单元（在4.3节中描述）与多个DRAM芯片并行连接。内存流量通过地址交错在内存分区单元之间分配。NVIDIA的一项专利描述了在256字节或1,024字节粒度下，平衡最多6个内存分区之间流量的地址交错方案【Edmondson和Van Dyke，2011】。

SIMT 核心通过片上互连网络连接到内存分区单元。最近关于 NVIDIA 的专利中描述的片上互连网络是交叉开关 [Glasco 等，2013，Treichler 等，2015]。AMD 的 GPU 有时被描述为使用环形网络 [Shrout，2007]。

4.3 内存分区单位

下面，我们描述了与几项近期 NVIDIA 专利对应的内存分区单元的微架构。在历史背景方面，这些专利是在 NVIDIA 的 Fermi GPU 架构发布前大约一年提交的。如图 4.3 所示，每个内存分区单元包含一部分第二级 (L2) 缓存，以及一个或多个内存访问调度器，也称为“ 帧缓冲区 ”（frame buffer，FB），和一个光栅操作 (ROP) 单元。L2 缓存包含图形和计算数据。内存访问调度器会重新排序内存读写操作，以减少访问 DRAM 的开销。ROP 单元主要用于图形操作，例如 alpha 混合，并支持图形表面的压缩。ROP 单元还支持类似于 CUDA 编程模型中的原子操作。所有三个单元紧密耦合，下面将对此进行一些详细描述。

4.3.1 L2 缓存

L2缓存设计包含若干优化，旨在提高GPU的单位面积整体吞吐量。每个内存分区内部的L2缓存部分由两个切片组成[Edmondson et al., 2013]。每个切片包含独立的标签和数据数组，并按顺序处理到达的请求[Roberts et al., 2012]。为了匹配GDDR5中32字节的DRAM原子大小，每个切片内的缓存行有四个32字节的扇区。缓存行被分配供存储指令或加载指令使用。在常见的联合写入情况下，为了优化吞吐量，在写失误时完全覆盖每个扇区时不会首先从内存中读取数据。这与CPU缓存在标准计算机体系结构教材中的常见描述有很大不同。我们审查的专利中没有描述如何处理未联合的写入，这种写入未完全覆盖扇区，但有两种解决方案是存储字节级有效位和完全绕过L2。为了减少内存访问调度器的面积，写入内存的数据在L2的缓存行中进行缓冲，同时等待调度的写入。

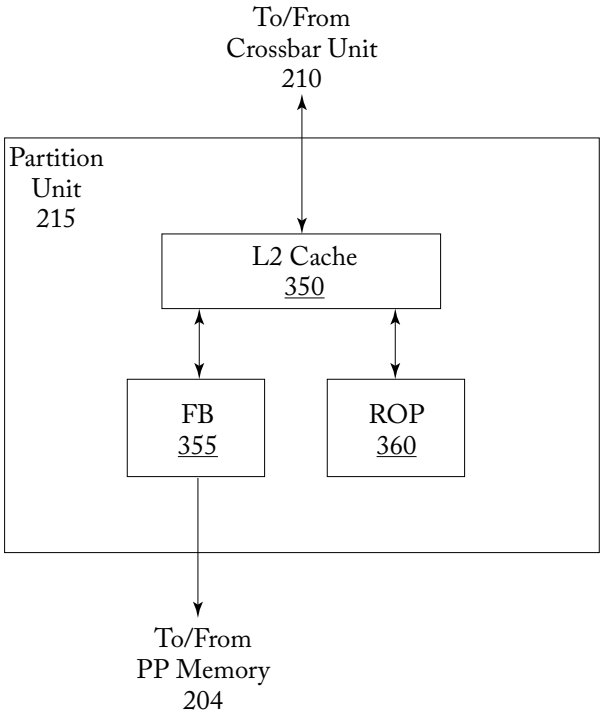


图4.3：内存分区单元概览（基于Edmondson等人[2013]中的图3B）。

4.3.2 原子操作

如Glasco等人所述[2012]，ROP单元包括用于执行原子和归约操作的功能单元。一系列访问同一内存位置的原子操作可以通过ROP单元中的本地ROP缓存进行流水线处理。原子操作可用于在不同线程块中运行的线程之间实现同步。

4.3.3 内存访问调度程序

为了存储大量数据，GPU采用了特殊的动态随机存取存储器（DRAM），如GDDR5 gdd。DRAM将单个比特存储在小电容器中。为了读取这些电容器中的值，首先将一行比特，称为页面，读取到称为行缓冲区的小内存结构中。为了完成此操作，连接到行缓冲区的单个存储电容器的比特线（它们本身也具有电容）必须首先预充电到在0和供电电压之间的一半的电压。当在激活操作期间，电容器通过访问晶体管连接到比特线时，随着电荷在存储单元和比特线之间流动，比特线的电压会稍微向上或向下拉动。一个传感器

放大器然后放大这个小变化，直到读取到一个干净的逻辑0或1。将值读取到行缓冲区的过程会刷新存储在电容器中的值。预充电和激活操作引入了延迟，在此期间无法读取或写入DRAM阵列的数据。为了减少这些开销，DRAM包含多个银行，每个银行都有自己的行缓冲区。然而，即使有多个DRAM银行，在访问数据时，完全隐藏在行之间切换的延迟往往也是不可能的。这导致了内存访问调度器的使用[Rixner等，2000，Zuravleff和Robinson，1997]，它们重新排序DRAM内存访问请求，以减少数据在行缓冲区和DRAM单元之间移动的次数。

为了启用对 DRAM 的访问，GPU 中的每个内存分区可能包含多个内存访问调度器 [Keil 和 Edmondson, 2012]，将其包含的 L2 缓存部分连接到外部 DRAM。实现这一目标的最简单方法是每个 L2 缓存切片都有自己的内存访问调度器。每个内存访问调度器包含单独的逻辑，用于排序从 L2 缓存发送的读请求和写请求（“ 脏数据通知 ”） [Keil 等, 2012]。为了将对 DRAM 存储银行中同一行的读取请求分组，采用了两个独立的表。第一个称为读取请求排序器，属于一个通过内存地址访问的集合关联结构，将所有对给定存储银行中同一行的读取请求映射到一个单一的指针。该指针用于查找名为读取请求存储的第二个表中的个别读取请求列表。

4.4 GPU内存系统的研究方向

4.4.1 内存访问调度与互连网络设计

Yuan 等 [2009] 研究了为运行 CUDA 编写的 GPU 计算应用程序的 GPU 内存访问调度器设计。他们观察到，单个流处理器 (SM) 生成的请求具有行缓冲区局部性。如果对给定内存分区的记忆请求序列中，邻近出现的请求访问同一 DRAM 行且同一 DRAM 银行，则该序列被认为具有行缓冲区局部性。然而，当来自一个 SM 的内存请求被发送到内存分区时，它们与来自其他 SM 发送到同一内存分区的请求混合在一起。结果是，进入内存分区的请求序列的行缓冲区局部性降低。Yuan 等 [2009] 提出通过修改互连网络以保持行缓冲区局部性来减少内存访问调度的复杂性。他们通过引入优先授权来自同一 SM 或具有相似行银行地址的内存请求的数据包的仲裁策略来实现这一点。

Bakhoda等人[2010，2013]探讨了用于GPU的片上互连网络的设计。该互连将流式多处理器连接到内存分区。他们认为，随着SM数量的增加，采用可扩展拓扑（如网格）将变得必要。他们研究了片上网络设计对系统吞吐量的影响，并发现

许多CUDA应用的吞吐量对互连延迟相对不敏感。他们分析了互连流量，发现其具有许多对少对多的模式。他们提出了一种更为严格的可扩展拓扑，由“ 半路由器 ”组成，通过利用这种流量模式来降低路由器的面积成本。

4.4.2 缓存有效性

Bakhoda 等人 [2009] 研究了在使用他们的 GPGPU-Sim 模拟器模拟的 CUDA 兼容 GPU 上为全局内存访问添加 L1 和/或 L2 缓存的影响，结果显示虽然一些应用受益，其他则没有。

随后，贾等（2012）的工作通过在NVIDIA Fermi GPU硬件上启用或禁用缓存来表征缓存的有效性，发现了类似的结果。观察到通过L1缓存将数据读入临时共享内存的应用程序在启用L1缓存时并没有收益。即便排除这些应用程序，贾等（2012）也观察到，仅仅依靠缓存命中率不足以预测缓存是否会改善性能。他们发现，需要考虑缓存对L2缓存请求流量的影响（例如，内存分区）。在他们研究的Fermi GPU上，L1缓存没有分区，因此启用缓存可能会导致在发生未命中时产生更大的128字节外部内存访问。在受内存带宽限制的应用程序上，这额外的外部内存流量可能导致性能下降。贾等（2012）提出了三种局部性的分类法：内部波形（within-warp）、块内（within-block）和跨指令（cross-instruction）。内部波形局部性发生在单个波形内不同线程执行的单个负载的内存读取访问同一缓存块时。块内局部性发生在同一个线程块中来自不同波形的线程执行的单个负载的内存读取访问同一缓存块时。跨指令局部性发生在同一个线程块中不同负载指令执行的线程对同一缓存块进行的内存读取访问时。贾等（2012）引入了一种利用这一分类法的编译时算法，以帮助推断何时启用缓存对单个负载指令有帮助。

4.4.3 内存请求优先级和缓存绕过

跟随上述表征研究[Jia et al., 2012]和Rogers等人的工作[2012]，该工作表明了扭曲调度可以提高缓存有效性（见第5.1.2节），Jia等人[2014]提出了针对GPU的内存请求优先级和缓存绕过技术。与线程数量相比，低关联性的缓存可能会遭受显著的冲突失效[Chen and Aamodt, 2009]。Jia等人[2014]指出，几个用CUDA编写的GPGPU应用程序包含数组索引，这会导致单个warp的单个内存请求在使用标准的模缓存集索引函数时映射到同一缓存集，从而导致失效[Hennessy and Patterson, 2011，Patterson和Hennessy, 2013]。Jia等人[2014]称之为warp内部争用。假设缓存中的空间在某个时刻分配，当一个

当检测到未命中[1]时，有限数量的未命中状态保持寄存器[2]的线程内部争用可能导致内存管线停滞[3]。为了解决线程内部争用，Jia等（2014）提出在未命中发生且由于关联性停滞而无法分配缓存块时，绕过L1数据缓存。当缓存集中的所有块被保留以为待处理的缓存未命中提供数据空间时，就会发生关联性停滞。

Jia et al. [2014] 还研究了他们所称的跨 Warp 争用。这种形式的缓存争用发生在一个 Warp 驱逐了另一个 Warp 带入的数据。为了应对这种争用，Jia et al. [2014] 建议采用他们称之为 “ 内存请求优先化缓冲区 ”（MRPB）的结构。与 CCWS [Rogers et al., 2012] 类似，MRPB 通过修改对缓存的访问顺序来增加局部性，从而减少容量未命中。然而 ，与 CCWS 通过线程调度间接实现这一点不同，MRPB 试图在线程被调度后，通过改变单个内存访问的顺序来增加局部性。

MRPB在一级数据缓存之前实现了内存请求重排序。MRPB的输入是内存请求合并后在指令发出流水线阶段生成的内存请求。MRPB的输出是将内存请求馈送到一级缓存。内部，MRPB包含多个并行的先进先出（FIFO）队列。缓存请求使用 “ 签名 ” 分发到这些FIFO队列。在他们评估的多个选项中，他们发现最有效的签名是使用 “ warp ID ”（一个介于0到可以在流式多处理器上运行的最大warp数量之间的数字）。MRPB采用 “ 排水策略 ” 来确定从哪个FIFO选择内存请求以便下一个访问缓存。在探索的几个选项中，最佳版本是一个简单的固定优先级方案，其中每个队列被分配一个静态优先级 ，包含请求的最高优先级队列优先服务。

详细评估显示，采用MRPB的绕过和重排序的结合机制，在64路16 KB上实现了几何平均加速4%。Jia等人[2014]也进行了一些与CCWS的比较，显示出更大的改进。我们顺便提到，Rogers等人[2012]的评估采用了基线架构，使用更复杂的集合索引哈希函数[4]来减少关联性停顿的影响。此外，Nugteren等人[2014]的后续工作旨在逆向工程NVIDIA Fermi架构中实际使用的集合索引哈希函数的细节，并发现它使用XOR运算（这也有助于减少此类冲突）。

与Rogers等人[2013]类似，Jia等人[2014]表明，他们的程序员透明方法可以提高性能，从而缩小使用缓存的简单代码与使用划痕共享内存的高度优化代码之间的差距。

[1] The default in GPGPU-Sim where it is used to avoid protocol deadlock.
[2] Consistent with a limited set of pending request table entries—see Section 4.1.1.
[3] GPGPU-Sim version 3.2.0, used by Jia et al. [2014], does not model instruction replay described in Sections 3.3.2 and 4.1.
[4] See `cache_config::set_index_hashed` in `https://github.com/tgrogers/ccws-2012/blob/master/simulator/ccws_gpgpu-sim/distribution/src/gpgpu-sim/gpu-cache.cc`

Arunkumar等人[2016]探讨了绕过和变化缓存行大小的影响，这基于静态指令中存在的内存分歧程度。他们利用观察到的重用距离模式和内存分歧度来预测绕过和最佳缓存行大小。

李和吴[2016]提出了一种基于控制环的缓存绕过方法，该方法试图在运行时逐条指令预测重用行为。缓存行的重用行为被监控。如果由特定程序计数器加载的缓存行没有经历足够的重用，则该指令的访问将被绕过。

4.4.4 利用跨波动异质性

Ausavarungnirun 等人 [2015] 提出了一系列改进，针对 GPU 的共享 L2 和内存控制器，以减轻不规则 GPU 应用程序中的内存延迟偏差。这些技术统称为内存偏差修正（MeDiC），利用了在同一内核的各个 warp 中内存延迟偏差程度的异质性的观察。根据它们与共享 L2 缓存的交互方式，内核中的每个 warp 可以被特征化为全命中/大多命中、全未命中/大多未命中或平衡。作者展示了，拥有并非全命中的 warp 并没有太大好处，因为大多命中的 warp 必须等待最慢的访问返回后才能继续处理。他们还展示了 L2 缓存的排队延迟可能会对性能产生非微不足道的影响，并且通过为所有非全命中的 warp 的所有请求（即使那些可能命中的请求）绕过 L2 缓存，可以减轻这一影响。这通过减少排队延迟来降低全命中 warp 的访问延迟。除了自适应绕过技术外，他们还提出了对缓存替换策略和内存控制器调度程序的修改，以尝试最小化被检测为全命中 warp 的延迟。他们还展示了，即使对于全命中的 warp，L2 缓存银行之间的排队延迟差异也可能导致额外的潜在可避免的排队延迟，因为 L2 银行之间的排队延迟存在不平衡。

作者提出的微架构机制由四个组件组成：（1）一个波动类型检测模块——将 GPU 中的波动分类为五种潜在类型之一：全失效、主要失效、平衡、主要命中或全命中；（2）一个波动类型感知旁路逻辑模块，决定请求是否应该绕过 L2 缓存；（3）一个波动类型感知插入策略，确定在 L2 的 LRU 栈中放置插入的位置；（4）一个波动类型感知内存调度器，控制 L2 缺失/旁路如何发送到 DRAM。

检测机制通过间隔性地采样每个warp的命中率（总命中数/访问数）来操作。基于该比率，warp会采用上述五种分类之一。确定这些分类边界的精确命中率会根据每个工作负载动态调整。在分类间隔期间，没有请求可以绕过缓存，以便对每个warp的L2特性中的相位变化做出反应。

旁路机制位于 L2 缓存前面，接收带有生成它们的工作类型标签的内存请求。该机制试图消除来自所有未命中的工作和将主要未命中的工作转化为所有未命中的工作。该块简单地将标记为来自所有未命中和主要未命中的工作的所有请求直接发送到内存调度器。

MeDiC 的缓存管理策略通过改变来自 DRAM 的请求在 L2 的 LRU 栈中的位置来运行。主要由未命中的 warp 请求的缓存行被插入到 LRU 位置，而所有其他请求则被插入到传统的 MRU 位置。

最后，MeDic 修改了基线内存请求调度器，使其包含两个内存访问队列：一个用于全命中和大多数命中的 Warp 的高优先级队列，以及一个用于平衡、大多数未命中和全未命中的 Warp 的低优先级队列。内存调度器简单地优先处理高优先级队列中的所有请求，而低优先级队列中的任何请求则优先级较低。

4.4.5 协调缓存绕过

Xie 等人 [2015] 探讨了有选择地启用缓存绕过以提高缓存命中率的潜力。他们采用剖析方法来确定在 GPGPU 应用中每个静态加载指令是否具有良好的局部性、差的局部性或适度的局部性。根据结果，他们相应地标记指令。标记为具有良好局部性的加载操作被允许使用 L1 数据缓存。标记为具有差的局部性的加载操作始终被绕过。标记为具有适度局部性的加载指令采用一种自适应机制，其工作原理如下。自适应机制在线程块粒度上运行。对于给定的线程块，所有执行的适度局部性加载操作被统筹处理。它们要么使用 L1，要么绕过。行为在启动线程块时根据一个阈值决定，该阈值使用考虑 L1 缓存命中和流水线资源冲突的性能指标在线调整。他们的评估显示，这种方法相比静态波限制显著提高了缓存命中率。

4.4.6 自适应缓存管理

陈等人 [2014b] 提出了协调缓存绕过和波束节流的方案，该方案利用波束节流和缓存绕过来提高对高度缓存敏感应用的性能。所提机制在运行时检测缓存争用和内存资源争用，并相应地协调节流和绕过策略。该机制通过现有的 CPU 缓存绕过技术实现缓存绕过，该技术使用保护距离防止在多个访问中缓存行被驱逐。在插入缓存时，该行被分配一个保护距离，计数器跟踪行的剩余保护距离。一旦剩余保护距离达到 0，该行将不再受到保护，可以被驱逐。当新的内存请求尝试将新行插入没有未受保护行的集合时，该内存请求将绕过缓存。

保护距离是全局设置的，最优值在不同工作负载之间有所不同。在这项工作中，Chen 等人 [2014b] 扫描了静态保护距离，并证明 GPU 工作负载对保护距离值相对不敏感。

4.4.7 缓存优先级排序

Li等人[2015]观察到，栅格节流优化了L1缓存命中率，但可能导致其他资源，如离芯带宽和L2缓存，显著未被充分利用。他们提出了一种为栅格分配令牌的机制，以确定哪些栅格可以将行分配到L1缓存中。额外的“ 非污染栅格 ”没有获得令牌，因此虽然它们可以执行，但不允许从L1中驱逐数据。这导致了一个优化空间，在这个空间中，可以调度的栅格数量（W）和拥有令牌的栅格数量（T）都可以设置为小于可以执行的最大栅格数量。他们显示，静态选择W和T的最佳值使得相较于静态栅格限制的CCWS有了17%的改善。

基于这一观察，Li等人[2015]探讨了两种机制，以学习W和T的最佳值。第一种方法基于保持高线程级并行性的理念，同时提高缓存命中率。在这种方法中，称为dynPCALMTLP，采样期间运行一个内核，W设置为最大的warp数量，然后在不同的SIMT核心上改变T。然后选择实现最大性能的T值。这导致与CCWS可比的性能，同时显著减少面积开销。第二种方法称为dynPCALCCWS，最初使用CCWS设置W，然后使用dynPCALMTLP确定T。接着，它监视共享结构的资源使用情况，以动态增加或减少W。这导致与CCWS相比性能提高11%。

4.4.8 虚拟内存页面放置

Agarwal等人[2015]考虑了在包括容量优化和带宽优化内存的异构系统中支持缓存一致性的多种物理内存类型的影响。由于优化带宽的DRAM在成本和能量上比优化容量的DRAM更昂贵，未来的系统很可能包含两者。Agarwal等人[2015]观察到，当前Linux等操作系统的页面放置策略并未考虑内存带宽的非均匀性。他们研究了一种未来系统，其中GPU可以以低延迟访问低带宽/高容量的CPU内存——这种延迟为100个核心周期。他们的实验使用了经过修改的GPGPU-Sim 3.2.2，配置了额外的MSHR资源以模拟更现代的GPU。

通过这种设置，他们首先发现，对于受内存带宽限制的应用，利用 CPU 和 GPU 内存来增加总内存带宽有显著的性能提升机会。他们发现，对于一些内存延迟限制较小的 GPGPU 应用情况则并非如此。在假设页面均匀访问并且带宽优化内存的容量不是限制的情况下，他们证明了分配

将页面分配到内存区域的比例与区域的可用内存带宽成正比是最优的。假设带宽受限内存的容量不是问题，他们发现用概率随机将页面分配到带宽或容量优化的内存，按照内存带宽的比例，在实际的GPGPU程序中有效。然而，当带宽优化内存的容量不足以满足应用需求时，他们发现有必要细化页面放置，考虑访问频率。

为优化页面放置，他们提出了一种系统，该系统涉及使用修改版的NVIDIA开发工具nvcc和ptxas实现的分析传递，以及对现有CUDA API的扩展，以包括页面放置提示。使用基于分析的页面放置提示可获得约90% oracle页面放置算法的好处。他们将页面迁移策略留待未来工作。

4.4.9 数据放置

Chen et al. [2014a] 提出了 PORPLE，一种可移植的数据放置策略，包含规范语言、一个源到源编译器和一个自适应运行时数据放置器。他们抓住了这样一个观察：在 GPU 上存在多种不同类型的内存，选择数据应放置在何处对于程序员来说是困难的，并且往往无法在不同的 GPU 架构之间移植。PORPLE 的目标是可扩展、输入自适应，并且通常适用于常规和不规则的数据访问。他们的方法依赖于三种解决方案。

第一个解决方案是一个内存规范语言，旨在帮助扩展性和可移植性。内存规范语言描述了GPU上各种形式的内存，基于对这些空间的访问在何种条件下被串行化。例如，对相邻全局数据的访问是合并的，因此是并发访问，但对同一个共享内存的银行的访问必须被串行化。

第二个解决方案是一个名为 PORPLE-C 的源到源编译器，它将原始的 GPU 程序转换为一个与布局无关的版本。编译器在访问内存时插入保护代码，选择与预测的数据最佳布局相对应的访问。

最后，为了预测哪个数据放置最为优化，他们使用PORPLE-C通过代码分析找到静态访问模式。当静态分析无法确定访问模式时，编译器会生成一个函数来追踪运行时访问模式并尝试进行预测。该函数在CPU上运行一段短时间，帮助确定在启动内核之前最佳的基于GPU的数据放置。在本工作范围内，系统仅处理数组的放置，因为数组是GPU内核中最常用的数据结构。

在PORPLE中用于进行放置预测的轻量级模型根据内存的序列化条件生成生成的事务数量的估计。对于具有缓存层次结构的内存，它使用缓存的重用距离估计。

命中率。当多个数组共享一个缓存时，每个数组所占用的缓存量的估计是基于根据数组大小的线性分区。

4.4.10 多芯片模块GPU

Arunkumar et al. [2017] 指出，摩尔定律的放缓将导致 GPU 性能增长速度的减缓。他们建议通过在多芯片模块上构建大型 GPU 来扩展性能缩放，采用较小的 GPU 模块（见图 4.4）。他们展示了通过结合远程数据的本地缓存、考虑局部性和首次接触页面分配的模块 CTA 调度，可以在保持 10% 性能的情况下达到单个大型（且无法实现的）单片 GPU 的性能。根据他们的分析，这比在相同工艺技术下使用最大可实现的单片 GPU 的性能提高了 45%。

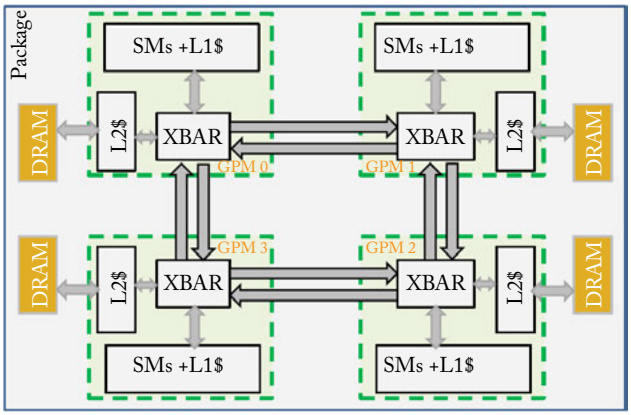


图 4.4：一个多芯片模块GPU（基于Arunkumar等人[2017]的图3）。

CHAPTER 5

跨领域研究 GPU 计算架构

本章节详细介绍了GPGPU架构中几个研究方向，这些方向并不完全符合之前专注于GPU架构特定部分的章节。第5.1节探讨了GPU中线程调度的相关工作。第5.2节关注于替代编程方法论，第5.4节则审视了异构CPU/GPU计算的相关研究。

5.1 线程调度

当代GPU与CPU在本质上是不同的，因为它们依赖于大量的并行性。无论程序是如何指定的（例如，使用OpenCL、CUDA、OpenACC等），没有广泛软件定义的并行性的工作负载都不适合GPU加速。GPU采用几种机制来聚合和调度所有这些线程。GPU上线程的组织和调度主要有三种方式。

线程分配给波 Since GPUs 使用 SIMD 单元以 MIMD 编程模型定义的线程执行，因此线程必须以波的形式组合在一起，以便进行锁步执行。在本书研究的基线 GPU 架构中，具有连续线程 ID 的线程被静态组合在一起以形成波。第 3.4.1 节总结了关于在波内进行替代线程排列以实现更好波压缩的研究提案。

动态线程块分配给核心 与 CPU 中逐个分配线程到硬件线程不同，在 GPU 中，工作是大批量分配给 GPU 核心的。这项工作单位由多个以线程块形式存在的 warp 组成。在我们的基准 GPU 中，线程块按照轮询顺序分配给核心。核心的资源（如 warp 插槽、寄存器文件和共享内存空间）是以线程块为粒度进行分配的。由于与每个线程块相关联的状态量大，因此当前的 GPU 不会抢占其执行。线程块中的线程在其资源可以分配给另一个线程块之前会执行到完成。

逐周期调度决策 在一个线程块被分配给一个GPU核心后，一组细粒度的硬件调度器在每个周期决定选择哪一组波形。

获取指令，它会扭曲以发出执行指令，以及何时读取/写入每个发出指令的操作数。

调度多个内核 线程块级别和逐周期调度决策可以在一个内核内以及在同一GPU上并发运行的不同内核之间进行。传统的内核调度仅允许在同一时间内活动的内核运行在GPU上。然而，NVIDIA的流和HyperQ调度机制的引入使得并发内核的运行成为可能。这种情况在某些方面类似于CPU上的多程序设计。

5.1.1 线程块分配给核心的研究

当内核启动时，每个内核启动中的线程被分组到线程块中。一个全局的线程块调度机制将每个线程块分配给一个SIMT核心，依据资源的可用性。每个核心都有固定量的局部存储器（在CUDA中称为共享内存，在OpenCL中称为局部内存）、寄存器数量、warp槽位和线程块槽位。在内核启动时，这些参数对于每个线程块来说都是已知的。最明显的线程块调度算法是以轮询的方式将线程块分配给核心，以最大化参与的核心数量。线程块会不断地调度，直到每个核心中至少有一个资源耗尽。请注意，一个内核可能由比GPU一次可运行的线程块更多的线程块组成。因此，内核中的某些线程块在其他线程块执行时甚至可能没有在GPU上运行。有几种研究技术已考虑在线程块调度空间中的权衡。

在线程块级别的节流。Kayiran等人[2013]提出节流分配给每个核心的线程块数量，以减少由于线程过度分配引起的内存系统竞争。他们开发了一种算法来监控核心空闲周期和内存延迟周期。该算法首先将每个核心的最大线程块数量的一半分配给它。然后监控空闲和内存延迟周期。如果一个核心主要在等待内存，则不再分配更多的线程块，现有的线程块可能被暂停以阻止它们发出指令。这项技术实现了一种粗粒度的并行节流机制，限制了内存系统的干扰，并提高了整体应用性能，即使同时处于活动状态的CTA较少。

动态调节GPU资源。Sethia和Mahlke [2014] 提出了Equalizer，一个硬件运行时系统，动态监控资源争用并调整线程数量、核心频率和内存频率，以改善能耗和性能。该系统的决策基于四个参数：（1）SM中活跃的warp数量；（2）等待从内存获取数据的warp数量；（3）准备执行算术指令的warp数量；以及（4）准备执行内存指令的warp数量。基于这些参数，它首先决定在SM中保持活跃的warp数量，然后根据该值和其他三个计数器的值（作为内存的代理）进行调整。

竞争、计算强度和内存强度) 它决定如何最好地调整核心和内存系统的频率。

Equalizer 有两种工作模式：节能模式和性能增强模式。在节能模式下，它通过缩减未充分利用的资源来节省能量，从而最小化能耗，同时减少对性能的影响。在性能增强模式下，Equalizer 通过提高瓶颈资源来提升性能，并以节能的方式进行。

它们通过检查与改变内存频率、计算频率和同时运行的线程数量相关的性能和能量权衡，将来自Rodinia和Parboil的一组工作负载特征化为计算密集型、内存密集型、缓存敏感型或未饱和型。如果目标是最小化能量（而不牺牲性能），那么计算密集型内核应该在较低的内存频率下运行，而内存内核则应该在较低的SIMT核心频率下运行。这有助于减少在基线速率下未被充分利用的系统中不必要的能量消耗。

均衡器以区间为基础对频率和并发性做出决策。该技术在每个SIMT核心上添加监控硬件，基于之前列出的四个计数器进行本地决策。它在每个SIMT核心本地决定该纪元的三个输出参数（CTA数量、内存频率和计算频率）应该是多少。它向全局工作分配引擎通知该SM应使用的CTA数量，如果SIMT核心需要更多工作，则发出新的块。如果SM应以较少的CTA运行，它会暂停核心上的一些CTA。在决定运行的CTA数量后，每个SIMT核心向全局频率管理器提交一个内存/计算电压目标，该管理器根据多数函数设置芯片范围内的频率。

局部决策是通过观察等待执行内存指令的 warp 数量和等待执行 ALU 指令的 warp 数量来做出的。如果试图等待内存的 warp 数量大于一个 CTA 中的 warp 数量，则在这个 SIMT 核心上运行的 CTA 数量会减少，这可能有助于缓存敏感工作负载的性能。如果准备发出内存（或 ALU）的 warp 数量大于一个 CTA 中的 warp 数量，则认为 SIMT 核心是内存（或计算）密集型的。如果等待内存（或计算）的 warp 数量少于一个 CTA 中的 warp 数量，当活跃的 warp 中有超过一半在等待且在内存上等待的 warp 不超过两个时，SIMT 核心仍然可以被认为是 ALU 或内存受限。如果是这种情况，则核心上活跃的 CTA 数量增加一个，并根据等待的计算 warp 或等待的内存 warp 数量确定 SIMT 核心是计算受限还是内存受限。

一旦SIMT核心做出了本地决策，内存和核心的频率将根据均衡器的工作模式缩放±15%。

循环调度的早期特征。Lakshminarayana 和 Kim [2010] 探讨了早期 GPU 中多种 warp 调度策略，该 GPU 没有硬件管理的缓存，并表明，对于每个 warp 执行对称（平衡）动态指令计数的应用，基于公平性的 warp 和 DRAM 访问调度策略能够提高性能。这一策略在他们研究中使用的常规 GPU 工作负载上表现良好，因为 warp 之间的常规内存请求在核心内被合并，并更好地利用了 DRAM 行缓冲器的局部性。该论文还描述了其他几种 warp 调度策略，包括由 Tullsen 等人 [1996] 首次提出的 ICOUNT，它针对同时多线程的 CPU 设计。ICOUNT 的设计目的是通过优先处理进展最快的 warp（或线程）来提升系统吞吐量。Lakshminarayana 和 Kim [2010] 表明，在早期的无缓存 GPU 上，仅优先处理少数几个 warp 对于早期的常规工作负载通常并不会提高性能。

双层调度。Gebhart 等人 [2011c] 引入了使用双层调度器来提高能效。他们的双层调度器将核心中的战斗组分为两个池：一个活跃池，包含将在下一个周期中被考虑调度的战斗组，和一个非活跃池，包含不被考虑调度的战斗组。当战斗组遇到编译器识别的全局或纹理内存依赖时，它会从活跃池转移出去，并以轮询方式从非活跃池重新进入活跃池。每个周期从较小的战斗组池中进行选择可以减少战斗组选择逻辑的大小和能耗。

Naraisman等人[2011]提出的两级调度器关注于通过允许线程组在不同时间访问相同的长延迟操作来提高性能。这有助于确保在提取组内维持缓存和行缓冲的局部性。系统可以通过在提取组之间切换来隐藏长延迟操作。相比之下，Cache-Conscious Warp Scheduling（见下文）则侧重于通过自适应限制系统可以维持的多线程数量来提高性能，具体取决于失去的内波局部性有多少。

缓存意识波前调度。Rogers 等 [2012] 将存在于 GPU 内核中的内存局部性分类为 *intra-warp*，其中一个波束加载然后引用其自己的数据，或 *inter-warp*，其中一个波束与其他波束共享数据。他们展示了波束内部局部性是发生在对缓存敏感的工作负载中最常见的局部性形式。基于这一观察，他们提出了一种缓存意识波前调度（CCWS）机制，通过根据内存系统反馈限制主动调度的波束数量来利用这种局部性。

积极地在更少的warp之间调度，使得每个warp能够消耗更多的缓存空间，并减少L1数据缓存争用。特别是，当具有局部性的工作负载正在翻转缓存时，会发生节流。为了检测这种翻转，CCWS引入了一种基于L1数据缓存的替换牺牲者标签的丢失局部性检测机制。

图 5.1 显示了 CCWS 的高级微架构。在每次从缓存中逐出时，受害者的标签会被写入线束私有的受害者标签数组。每个线束都有自己的受害者标签数组，因为 CCWS 只关注检测线束内部的局部性。在每次后续的缓存缺失中，将对缺失线束的受害者标签数组进行探测。如果在受害者标签中找到该标签，则表示某些线束内部的局部性已经丢失。CCWS 假设如果该线束能够对 L1 数据缓存有更多的独占访问权限，那么它可能可以命中这一行，因此可能通过节流获益。

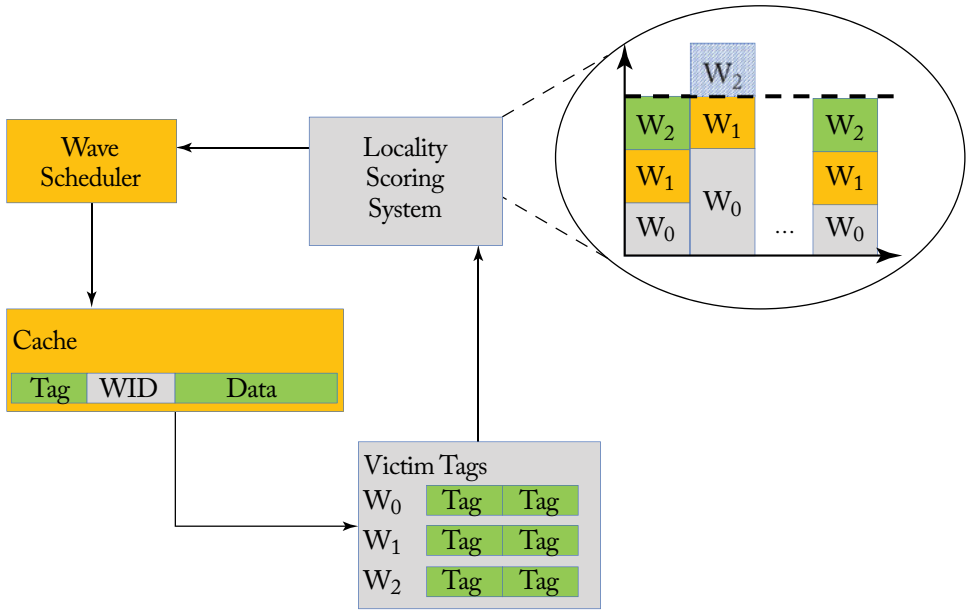


图 5.1：关注缓存的波前调度微架构。

为了反映这种局部性损失，系统向调度系统发送信号。问题调度器使用局部性评分系统来近似系统中每个warp丢失了多少局部性，该评分系统近似表示每个warp所需的额外缓存容量。局部性评分系统中的所有warp都被分配了初始分数，假设所有warp需要相同的缓存容量且没有出现节流（如图5.1中的叠加条形图所示）。随着时间的推移，丢失的局部性被检测到，单个warp的分数会增加。在图5.1的示例中，warp 0经历了局部性损失，其分数已增加。其分数的增加使warp 3超过了一个阈值，从而阻止其发出L1数据缓存请求，有效地限制了在核心上主动调度的warp数量。随着时间的推移，如果没有丢失局部性，那么warp 0的分数会减少，直到warp 2能够下降到阈值以下，并能够再次发出内存请求。

CCWS 进一步证明了缓存命中率对调度决策的敏感性，通过将各种调度机制与缓存替换策略进行比较。论文表明，warp 调度器可用的决策空间远大于替换策略所提供的相对受限的决策空间。论文还进一步证明，CCWS 方案使用 LRU 替换策略可以比之前的调度机制更好地提高缓存命中率，即使它们使用的是 Belady 最优缓存替换策略。

Rogers 等人 [2013] 提出了意识到差异的 Warp 调度 (DAWS)，它通过对每个 warp 的缓存占用进行更准确的估计来扩展 CCWS。DAWS 利用这样一个事实：在 GPU 工作负载中，大多数 intra-warp 局部性发生在循环中。DAWS 为循环中的 warp 创建了每个 warp 的缓存占用估计。DAWS 根据每个 warp 预测的循环占用预先限制循环中的 warp 数量。DAWS 还根据每个 warp 经历的控制流分歧程度调整其缓存占用估计。已经离开循环的 warp 中的线程不再对占用估计做出贡献。DAWS 进一步探索了 GPU 的可编程性方面，证明了通过更智能的 warp 调度程序，即使在内存传输上没有优化的基准（例如，使用共享内存而不是缓存），也可以非常接近于同一基准的 GPU 优化版本。

预取感知的波束调度。Jog等人[2013b]在GPU上探索了一种预取感知的波束调度器。他们基于二级调度机制，但从非连续的波束中形成提取组。该策略增加了DRAM中的银行级并行性，因为预取器不会对连续的访问查询同一个DRAM银行。他们进一步扩展了这一理念，根据波束组分配来操控预取器。通过为其他组中的波束预取数据，他们可以提高行缓冲区的局部性，并在预取请求和对数据的需求之间提供间隔。

CTA感知调度。Jog等人[2013a]提出了一种CTA感知的warp调度器，该调度器还基于两级调度器，通过选择性组合CTAs形成获取组。他们利用多种基于CTA的特性来提高性能。他们采用了一种节流优先级技术，该技术限制核心中活动warp的数量，这与其他节流调度器类似。结合节流，他们利用不同核心间的CTA页面局部性。在仅支持局部性感知的CTA调度器中，连续的CTAs通常会在同一时间访问相同的DRAM银行，从而降低银行级并行性。他们将此与预取机制结合，以提高DRAM行局部性。

调度对分支分歧缓解技术的影响。Meng et al. [2010] 引入了动态波形划分（DWS），该技术在某些线程访问缓存而某些线程未命中时拆分波形。该方案允许在缓存命中的每个标量线程即使其波形同伴未命中也能继续执行。DWS 通过允许执行进程来提高性能。

超前线程以更早地启动它们的未命中，并为那些滞后的线程创建了预取效果。DWS 试图通过提高数据加载到缓存中的速率来改善内战局部性。

Fung 等人 [2007] 探讨了变形调度策略对其动态变形形成 (DWF) 技术有效性的影响。DWF 试图通过在同一 warp 中的标量线程在分支指令上采取不同路径时动态创建新 warp 来缓解控制流分歧。他们提出了五种调度器，并评估了它们对 DWF 的影响。

Fung 和 Aamodt [2011] 还提出了三种线程块优先级机制，以补充他们的线程块压缩（TBC）技术。这些优先级机制试图将同一个 CTA 内的线程一起调度。他们的方法类似于 Narasiman 等人 [2011] 提出的两级调度的并发工作，除了线程块是一起调度而不是获取组。

第3.4节包含DWS、DWF和TBC的更详细摘要。

调度和缓存重新执行。Sethia 等人 [2015] 引入了 Mascar，它试图在内存密集型工作负载中更好地重叠计算与内存访问。Mascar 由两种交织的机制组成。

- 一种内存感知的波束调度器（MAS），在核心中的 MSHR 和 L1 未命中队列条目被过度订阅时，优先执行单个波束。 这种优先级有助于提高性能，即使工作负载不包含数据局部性，通过使在顺序核心上执行的波束更快地到达其计算操作，从而实现优先波束的计算与其他波束的内存访问的重叠。

- 一种缓存访问重执行（CAR）机制，可以帮助避免 L1 数据缓存抖动，通过使数据在缓存中的 warp 被低局部性访问阻止发出而造成内存管道阻塞时实现 L1 数据缓存命中在失效中。

MAS 有两种操作模式：等优先级 (EP) 和内存访问优先级 (MAP) 模式。系统根据 L1 MSHR 和内存缺失队列的填充程度在 EP 和 MP 之间切换。一旦这些结构几乎满了，系统就会切换到 MP 模式。MAS 包含两个队列，一个用于内存迁移（尝试发出内存指令的迁移），另一个用于计算迁移（尝试发出其他类型指令的迁移）。在每个队列中，迁移的调度采用贪婪优先然后最旧的顺序。通过增强记分板来跟踪依赖于内存的指令，以指示何时基于加载填充输出寄存器。当观察到负载均衡且内存系统没有过度订阅时，调度器在 EP 模式下运行。在 EP 模式下，调度机制优先考虑内存迁移。由于内存系统未过度订阅，因此预测提前发起内存访问将提高性能。当在 MAP 模式下运行时，调度器优先考虑计算迁移，以更好地使可用计算与受瓶颈影响的内存系统重叠。只有一个内存迁移，即“ 拥有迁移 ”，被允许发出内存指令，直到它到达依赖于待处理内存请求的操作。

除了调度机制，Sethia 等 [2015] 显示，内存密集型内核的峰值 IPC 性能远低于计算密集型内核。他们强调，在内存密集型应用中，大量周期因过度内存访问而导致 SIMT 核心的负载存储单元因内存压力而停滞。由于 LSU 被阻塞，存在相当一部分时间，准备好执行的波（warp）数据在 L1 数据缓存中，但由于 LSU 被其他波的内存请求阻塞，导致这些波无法发出指令。缓存访问重执行（CAR）机制旨在通过在 LSU 管道的一侧提供一个缓冲区来解决此行为，该缓冲区存储被阻塞的内存指令，并允许其他指令进入 LSU。只有当 LSU 未被阻塞且没有新的请求需要发出时，重执行队列中的请求才会被处理，除非重执行队列已满，在这种情况下，重执行队列中的访问将被优先处理，直到队列中有空间释放。

当重新执行队列与内存感知调度器结合时，需要特别注意，因为重新执行队列中的请求可能来自于优先拥有的 warp 之外的其他 warp。在 MAP 模式下，当非拥有者 warp 从重新执行队列发送的请求未命中 L1 时，这些请求会进一步延迟。特别是，当非拥有者 warp 的请求未命中 L1 时，该请求不会转发到 L2 缓存，而是重新插入到重新执行队列的末尾。

5.1.3 多内核调度研究

在GPU上支持抢占。Park等人[2015]解决了在GPU上支持抢占式多任务处理的挑战。它采用了更宽松的幂等性定义，以便在一个线程块内实现计算的刷新。更宽松的幂等性定义涉及检测从线程执行开始时是否已执行为幂等。他们的提议Chimera动态选择三种方法中的一种，以实现每个线程块的上下文切换：

- 完整的上下文保存/存储；
- 等待线程块完成; 并且
- 如果由于幂等性线程块可以安全地从头开始重新启动，则可以简单地停止线程块而不保存任何上下文。

每种上下文切换技术在切换的延迟和对系统吞吐量的影响之间提供了不同的权衡。为了实现Chimera，一个算法估计当前正在运行的线程块的子集，这些线程块可以在对系统吞吐量影响最小的情况下停止，同时满足用户指定的上下文切换延迟目标。

ElTantawy 和 Aamodt [2018] 探讨了在运行涉及细粒度同步的代码时，扭曲调度的影响。他们使用真实的 GPU 硬件，展示了当线程在等待锁时会出现显著的开销。他们指出，单纯地撤回执行包含未能获取锁的线程的扭曲，可能会阻碍或减缓同一扭曲中已经持有锁的其他线程的进展。他们提出了一种硬件结构，用于动态识别哪些循环涉及自旋锁，而这因使用基于栈的重归并而变得更加具有挑战性 [ElTantawy 和 Aamodt, 2016]。该结构使用包含程序计数器最低有效位的路径历史和单独的谓词寄存器更新历史，以准确检测在锁上自旋的循环。为了减少竞争并提高性能，他们建议在扭曲中执行自旋循环的向后分支时降低其优先级，前提是该扭曲中持有锁的线程已释放这些锁。他们发现，这种方法相比于 Lee 和 Wu [2014]，可以将性能和能耗分别提高 1.5× 和 1.6×。

5.2 表达平行主义的替代方式

细粒度工作队列。金和巴滕 [2014] 提出了在每个 SIMT 核心中添加一个细粒度硬件工作列表。他们利用了这样的观察：不规则的 GPGPU 程序通常在以数据驱动的方式实现时表现最佳，其中工作是动态生成并在线程之间平衡，而不是采用拓扑方法，其中启动了一定数量的线程——而这些线程中的许多并没有有效工作。数据驱动的方法有潜力提高工作效率和负载平衡，但在没有广泛软件优化的情况下，可能会遭遇性能低下。本文提出了一种片上硬件工作列表，支持核心内部和核心之间的负载平衡。他们使用线程等待机制，并按时间间隔重新平衡线程生成的任务。他们在 lonestar GPU 基准套件中对各种不规则应用的实施评估了他们的硬件机制，这些应用同时利用了拓扑和数据驱动的工作分配。

核心硬件工作列表解决了数据驱动软件工作列表的两个主要问题：(1) 当线程推送生成的工作时，内存系统中的争用和 (2) 由于基于线程 ID 的静态分区而导致的负载不平衡。依赖于静态分区的软件实现在推送和拉取时都会遭受内存争用。静态分区工作解决了拉取争用。硬件工作列表分布在多个结构中，减少了争用。它通过在线程变得空闲之前动态重新分配生成的工作来改善负载平衡。作者在指令集架构 (ISA) 中添加了用于从硬件队列推送和拉取的特定指令。核心中的每个执行单元被分配一个小型单端口 SRAM，用作特定执行单元使用和生成的工作 ID 的存储。

本文提出了一种基于区间和需求驱动（仅在推/拉请求上重新分配）的工作重分配方法，并对前者进行了深入评估。基于区间的方法在简单的阈值基础上或更复杂的排序基础上重新分配工作。阈值方法将工作量超过阈值的通道分类为贪婪（工作过多），将工作量少于阈值的通道分类为贫困（工作不足）。然后，排序过程将工作从贪婪银行重新分配给贫困银行。基于排序的技术更复杂，但实现了更好的负载均衡，因为所有贫困银行也可以向其他贫困银行捐赠工作。他们的技术还包括一个全局排序机制，可用于在核心之间分配工作。此外，架构还支持虚拟化硬件工作列表，使其能够扩展到生成比硬件结构可用容量更多动态工作的工作负载。

基于嵌套并行模式的编程。Lee 等人 [2014a] 提出了在 GPU 上对嵌套并行模式的局部性感知映射，利用了没有通用的最优映射将嵌套并行计算映射到 GPU 线程的观察。具有嵌套并行性的算法（例如 map/reduce 操作）可以在不同层次上将其并行性暴露给 GPU，具体取决于 GPU 程序的编写方式。作者利用了 GPU 上嵌套并行映射的三个概括。

- 一维映射，平行化顺序程序的外循环；
- 线程块/线程映射，将顺序程序的外循环的每次迭代分配给一个线程块，并在一个线程块中并行化内部模式；并
- 基于Warp的映射，将外循环的每次迭代分配给一个Warp，并在Warp中并行化内层模式。

该工作提出了一个自动编译框架，该框架根据局部性和嵌套模式中暴露的并行度生成预测性能分数，以选择最适合一组常见嵌套并行模式的映射。这些模式包括集合操作，如 map、reduce、foreach、filter 等。该框架试图将线程映射到集合中每个元素上的操作。框架通过首先将应用程序中的每个嵌套层级分配给一个维度（x、y、z 等）来处理模式的嵌套。一个双重嵌套模式（即一个包含 reduce 的 map）具有两个维度。然后，映射确定 CUDA 线程块中给定维度的线程数量。在设置线程块的维度和大小之后，框架进一步通过使用线程跨越和拆分的概念，将多个元素分配给每个线程来控制内核中的并行度。在一个二维内核中（即两个层级的模式嵌套），如果每个维度被分配为 span(1)，那么在内核中启动的每个线程仅负责操作集合中的一个元素。这种映射暴露了最大的并行度。相反，span(all) 表示每个线程操作集合中的所有元素。跨度可以是 (1) 和 (all) 之间的任何数字。使用 span(all)

在两种特殊情况下：当维度的大小在内核启动后才知道（例如，当内模式中操作的元素数量动态确定时）以及当模式需要同步时（例如，归约操作）。

由于 span(all) 可能严重限制暴露的并行性并导致 GPU 使用不足，因此框架还提供了 *split* 的概念。split(2) 表示每个线程在给定维度上操作一半元素（可以理解为 span(all)/2）。使用 split 时，框架会启动第二个内核（称为合并内核）来聚合跨分割的结果，产生与内核使用 span(all) 分区时相同的结果。

为了选择每个维度的块大小以及每个维度的划分/跨度，框架使用基于硬约束和软约束的评分算法。该算法遍历整个搜索空间，包括所有可能的维度、块大小和跨度。搜索空间相对于循环嵌套的级别呈指数增长。然而，指数的底数小于100，典型的内核包含少于3个级别。因此，该空间在几秒钟内就可以完全搜索。搜索剪枝了违反硬约束的配置——即那些导致执行不正确的配置，例如块中的线程数过高。它为软约束分配加权评分，例如确保顺序内存访问模式被分配到x维度以改善内存合并。

该框架还执行了两个常见的GPU优化：预分配内存，而不是在嵌套内核中动态分配全局内存，以及在确定将数据预取到共享内存对于嵌套模式有益时利用共享内存。结果表明，自动生成的代码与经过专家调优的代码具有竞争力的性能。

动态并行性。Wang 和 Yalamanchili [2014] 描述了在 Kepler GPU 硬件上使用 CUDA 动态并行性所需的开销，并发现这些开销可能是相当大的。具体来说，他们确定了几个限制他们研究中的工作负载效率的关键问题。首先，应用程序使用了非常大量的设备启动内核。其次，每个内核通常只有 40 个线程（稍微多于一个 warp）。第三，尽管每个动态内核中执行的代码相似，但启动配置不同，导致内核配置信息的存储开销很大。最后，为了实现并发，设备启动的内核被放置在单独的流中，以利用 Kepler 上支持的 32 个并行硬件队列（Hyper-Q）。他们发现这些因素结合起来导致了非常差的利用率。

王等人 [2016a] 随后提出动态线程块启动 (DTBL)，它修改了 CUDA 编程模型，使设备启动的内核能够共享硬件队列资源，从而实现更大的并行性和更好的 GPU 硬件利用率。他们提案的关键在于使动态启动的内核能够与正在运行相同代码的现有内核聚合在一起。这通过维护一个聚合线程块的链表来支持，该链表在启动内核时由修改后的硬件使用。

通过修改 GPGPU-Sim 评估 DTBL，并发现 DTBL 相较于 CDP 提高了 1.4× 的性能，相较于不采用 CDP 的高度优化的 CUDA 版本提高了 1.2×。

Wang等人[2016b]随后探讨了动态启动线程块被调度到哪个SM的影响。他们发现，通过鼓励子线程块与父SM在同一SM上调度，并考虑SM之间的工作负载分配，他们能够将性能提高27%，与简单的轮询分配机制相比。

5.3 对事务性内存的支持

本节总结了对在GPU架构上支持事务内存（TM）[Harris et al., 2010, Herlihy and Moss, 1993]编程模型的各种提议。

这些提议的动机在于事务内存（TM）编程模型能够减轻在具有丰富不规则并行性的GPU应用中管理线程之间不规则、细粒度通信的挑战。在现代GPU上，应用开发者可以通过屏障来粗化线程之间的同步，或者尝试使用在许多现代GPU上可用的单字原子操作来实现这些通信的细粒度锁。前者可能涉及对基础算法的重大修改，而后者则涉及细粒度锁定的开发工作不确定性，对实际的市场驱动软件开发而言风险太大（尽管有几个例外）。在GPU上启用TM简化了同步，并提供了一种强大的编程模型，促进了细粒度通信和并行工作负载的强扩展性。TM的这一承诺希望能鼓励软件开发者探索这些不规则应用的GPU加速。

在GPU上支持事务内存的独特挑战。GPU的高度多线程特性给事务内存系统设计提出了一系列新的挑战。GPU上的事务内存系统旨在扩展到数万个小的并发事务，而不是运行几十个占用相对较大资源的并发事务，这也是近期针对多核处理器的事务内存研究的重点。这反映了GPU的高度多线程特性，有数万个线程协同工作，每个线程执行一个小任务以实现共同目标。这些小事务以字级粒度进行跟踪，使得冲突检测的分辨率比缓存块更高。此外，GPU中每个核心的私有缓存被数百个GPU线程共享。这极大地减少了利用缓存一致性协议检测冲突的好处，而该技术通常应用于针对传统大型CPU核心的硬件事务内存。

5.3.1 KILO TM

Kilo TM [Fung et al., 2011] 是首个针对 GPU 架构的已发布硬件传输机提案。

Kilo TM 采用基于价值的冲突检测 [Dalessandro et al., 2010, Olszewski et al., 2007] 来消除对全局元数据进行冲突检测的需求。每个事务

简单地读取全局内存中的现有数据以进行验证——以确定它是否与另一个已提交的事务存在冲突。这种验证形式利用了GPU内存子系统的高度并行特性，避免了冲突事务之间的直接交互，并以最精细的粒度检测冲突。

然而，本地实现基于值的冲突检测要求事务串行提交。为了提升提交并行性，Kilo TM 吸收了现有事务管理系统的理念 [Chafi et al., 2007, Spear et al., 2008]，并通过创新解决方案进行了扩展。特别是，Fung 等人 [2011] 引入了 *recency bloom filter*，一种新颖的数据结构，通过时间和顺序的概念来压缩大量小项目集。Kilo TM 使用该结构压缩所有提交事务的写集。每个提交事务查询最近性布隆过滤器，以获取一组近似的冲突事务——在这个冲突集中的某些事务是伪阳性。Kilo TM 利用这些近似信息为数百个不冲突事务安排验证和并行提交。最近性布隆过滤器的这种近似特性使其保持小型，大小在几个千字节的数量级，从而可以驻留在芯片上以便快速访问。使用最近性布隆过滤器来提高事务提交并行性是 Kilo TM 的一个重要组成部分。

分支分歧与事务内存。事务内存编程模型引入了一种新的分支分歧类型。当一个波束完成一个事务时，它的每个活动线程都会尝试提交。一些线程可能会中止，并需要重新执行它们的事务，而其他线程可能会通过验证并提交它们的事务。由于这一结果在整个波束中可能并不一致，波束在验证后可能会发生分歧。Fung 等人 [2011] 提出了一个简单的扩展，以使 SIMT 硬件处理这种由于事务中止引入的特定类型的分支分歧。这个扩展独立于 Kilo TM 的其他设计方面，但这是支持 GPU 上事务内存的必要组成部分。

图5.2显示了SIMT栈如何扩展以处理由于事务中止引起的控制流分歧。当一个warp进入事务（在B行，`tx_begin`）时，它会在SIMT栈上推送两个特殊条目。第一个类型为R的条目存储重新启动事务的信息。它的活动掩码最初为空，并且其PC字段指向`tx_begin`之后的指令。第二个类型为T的条目跟踪当前的事务尝试。在`tx_commit` (的F)行，任何未通过验证的线程都会在R条目中设置其掩码位。当warp完成提交过程（即，其活动线程已提交或中止）时，T条目被弹出。然后将使用R条目的活动掩码和PC推送一个新的T条目到栈中，以重新启动已中止的线程。接着，R条目的活动掩码被清除。如果R条目的活动掩码为空，则T和R条目都会被弹出，显示原始的N条目。然后将其PC修改为指向`tx_commit`之后的指令，warp恢复正常执行。事务内warp的分支分歧处理与非事务性分歧的方式相同。

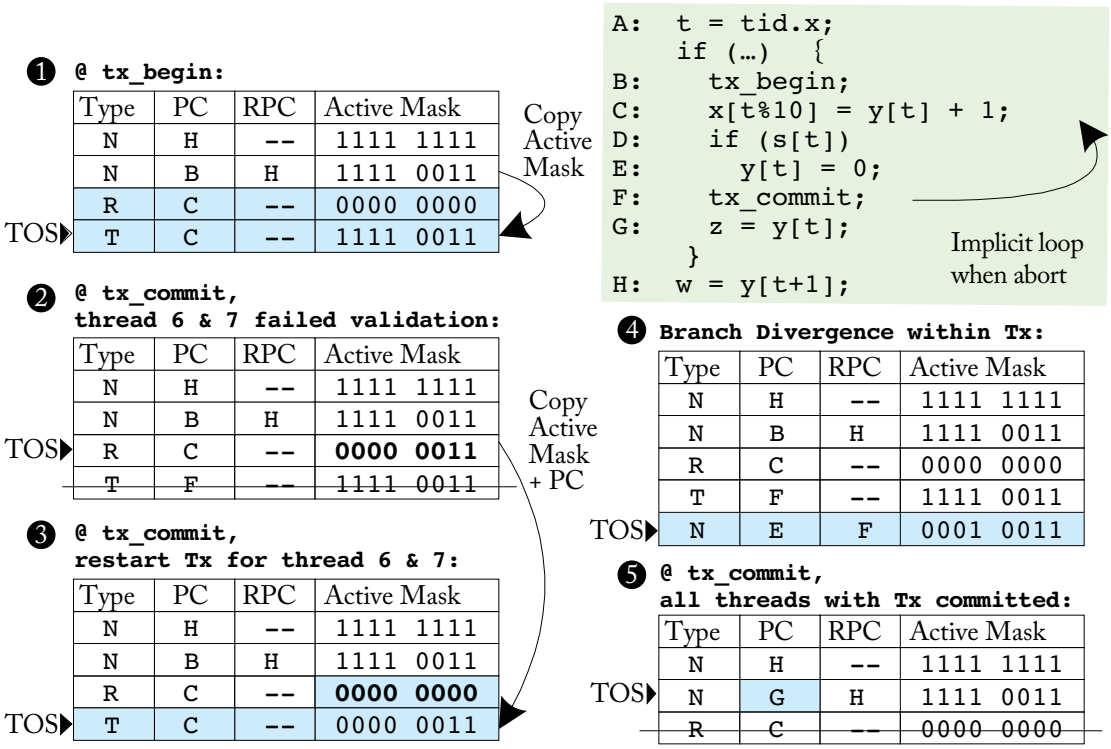


图5.2：SIMT堆栈扩展以处理由于事务中止（验证失败）导致的分歧。线程6和7验证失败并被重新启动。堆栈条目类型：正常（N），事务重试（R），事务顶部（T）。对于每种情况，新增条目或修改的字段已被阴影标记。

5.3.2 WARP TM 和时间冲突检测

在他们的后续论文中，Fung 和 Aamodt [2013] 提出了两种不同的增强措施，以提高 Kilo TM 的性能和能效：warp级事务管理（WarpTM）和时间冲突检测（TCD）。

扭曲级事务管理利用了GPU编程模型中的线程层次结构——线程间的空间局部性——来提高Kilo TM的效率。特别地，WarpTM摊销了Kilo TM的控制开销，并增强了GPU内存子系统的效用。这些优化只在可以高效地解决warp内的冲突时才有可能，因此，低开销的warp内冲突解决机制对于维持WarpTM的效益至关重要。为此，Fung和Aamodt [2013] 提出了一个两阶段的并行warp内冲突解决方案，这个方案可以高效地并行解决warp内的冲突。随着所有warp内冲突的解决，Kilo TM可以合并标量内存。

或对同一波中的多个事务进行验证和提交的访问转化为更宽的访问。这种优化称为 *validation and commit coalescing*，是使 Kilo TM 能够充分利用针对向量宽访问优化的宽 GPU 内存系统的关键。

时间冲突检测是一种低开销机制，使用一组全球同步的片上定时器来检测只读事务的冲突。一旦初始化，这些片上定时器在其微架构模块中本地运行，并且不会与其他定时器通信。这种没有通信的隐式同步使得TCD与各种软件事务内存系统中使用的现有基于时间戳的冲突检测方法 [Dalesandro等，2010，Spear等，2006，Xu等，2014] 区分开来。TCD使用这些定时器捕获的时间戳来推断事务的内存读取相对于其他事务更新的顺序。Kilo TM结合TCD来检测无需基于值的冲突检测即可直接提交的无冲突只读事务。这样做显著减少了这些事务的内存带宽开销，这在使用事务进行数据结构遍历的GPU-TM应用中可能会频繁发生。

5.4 异质系统

异构系统中的并发管理。Kayiran等人[2014]提出了一种限制并发的方案，用于调节GPU多线程，从而减少多程序运行的CPU/GPU系统中的内存和网络竞争。在异构系统中，GPU的干扰可能导致同时执行的CPU应用程序显著性能下降。他们提出的线程级并行（TLP）限制方案监测共享CPU/GPU内存控制器和互联网络中的拥塞指标，以估算每个GPU核心上应主动调度的GPU warps数量。他们提出了两种方案，一种专注于仅提高CPU性能，另一种旨在通过平衡由于多线程受限而导致的GPU性能下降与CPU干扰来优化整体系统吞吐量（包括CPU和GPU）。作者评估了在一个GPU核心与CPU核心比例为2:1的平铺异构架构中，warp调度对性能的影响，这一比例的合理性源于NVIDIA GPU SM的面积大约是现代超标量Intel芯片的一半，使用相同的工艺技术。基线配置完全共享CPU和GPU之间的网络带宽和内存控制器，以最大化资源利用率。通过使用这个方案，作者观察到限制GPU TLP可以对GPU性能产生正面和负面的影响，但对于CPU性能从未造成伤害。

为了提升CPU性能，作者提出了一种以CPU为中心的并发管理技术，该技术监控全局内存控制器中的停滞情况。这种技术分别计算由于内存控制器输入队列满而导致的内存请求停滞的数量，以及由于从内存控制器到核心的回复网络满而导致的内存请求停滞的数量。这些指标在每个内存控制器处本地监控。

聚合在一个集中单元中，该单元将信息发送到 GPU 核心。启发式驱动方案为这些值设置了高低阈值。如果两个请求停顿计数的总和较低（基于阈值），则在 GPU 上积极调度的 warp 数量会增加。如果两个计数的总和较高，则会减少活跃 warp 的数量，希望 CPU 性能由于更少的 GPU 内存流量而提高。

一种更平衡的技术，旨在最大化整体系统吞吐量，增强了以CPU为中心的方法，以考虑到从warp节流对GPU性能的影响。这种平衡技术监控在并发重新平衡间隔（他们的工作中为1,024个周期）内，GPU不能发出指令的周期数。当前多线程限制级别的GPU停顿的移动平均值存储在每个GPU核心上，并用于确定多线程级别是否应该增加或减少。这种平衡技术在两个阶段内调节GPU的TLP。在第一阶段，其操作与以CPU为中心的解决方案相同，此时不考虑GPU停顿，GPU TLP仅根据内存争用进行限制。在第二阶段（当GPU TLP节流开始导致GPU性能下降，因为GPU的延迟容忍度已降低时开始），系统停止节流GPU并发，如果它预测这样做会损害GPU性能。这个预测是通过查找在目标多线程级别的GPU停顿的移动平均数来完成的，这个数值是从之前在该级别执行时记录下来的。如果在目标多线程级别观察到的GPU停顿与当前多线程级别之间的差异超过阈值k，则TLP级别不会降低。这个k值可以由用户设置，是指定GPU性能优先级的一个代理。

异构系统一致性。Power 等人 [2013a] 提出了一种硬件机制，以高效支持集成系统中 CPU 和 GPU 之间的缓存一致性。他们发现，由于 GPU 产生的内存流量增加，目录带宽成为显著的瓶颈。他们采用粗粒度区域一致性 [Cantin et al., 2005] 来减少传统缓存块基础一致性目录中产生的过多目录流量。一旦获得粗粒度区域的权限，大多数请求将不必访问目录，一致性流量可以转移到不一致的直接访问总线上，而不是低带宽的一致性互连网络。

异构 TLP 感知缓存管理用于 CPU-GPU 架构。Lee 和 Kim [2012] 评估了在异构环境中管理 CPU 核心和 GPU 核心之间共享最后一级缓存 (LLC) 的效果。他们证明，虽然缓存命中率是 CPU 工作负载的一个关键性能指标，但许多 GPU 工作负载对缓存命中率不敏感，因为内存延迟可以通过线程级并行隐藏。为了确定 GPU 应用程序是否对缓存敏感，他们开发了一种按核心性能采样技术，其中一些核心绕过共享的 LLC，而一些核心则插入到最近最少使用的位置。根据这些核心的相对性能，他们可以为其余的 GPU 核心设置绕过策略，如果性能提高则插入 LLC，如果不敏感则绕过。

其次，他们观察到之前以CPU为中心的缓存管理偏向于访问频率较高的核心。GPU核心在LLC处的流量显示出是CPU核心的五到十倍。这增加了对GPU的偏置缓存容量，从而降低了CPU应用的性能。他们提出扩展之前提出的基于效用的缓存分区工作[Qureshi 和 Patt, 2006]，以考虑LLC访问的相对比率。当GPU核心对缓存敏感时，CPU核心的缓存访问方式分配超出了基于效用的缓存分区所提供的，以考虑CPU和GPU之间访问幅度和延迟敏感性的差异。

参考文献

照片分析。http://vlsiarch.eecs.harvard.edu/accelerators/die-photo-analysis 1

https://en.wikipedia.org/wiki/GDDR5_SDRAM 76

Top500.org 2

托尔·M·阿莫德特，威尔逊·W·L·冯，英德普里特·辛格，艾哈迈德·艾尔-沙菲，吉米·夸，泰勒·赫瑟lington，阿尤布·古布兰，安德鲁·博克托，蒂姆·罗杰斯，阿里·巴霍达，哈迪·朱伊巴。*GPGPU-Sim 3.x Manual*。16, 38

M. Abdel-Majeed 和 M. Annavaram. 扭曲寄存器文件：一种用于 GPGPU 的高效能寄存器文件. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013年2月. DOI: 10.1109/hpca.2013.6522337. 64

M. Abdel-Majeed, A. Shafaei, H. Jeon, M. Pedram, 和 M. Annavaram. 初步寄存器文件：针对 GPU 的节能分区寄存器文件。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2017 年 2 月。DOI: 10.1109/h-pca.2017.47. 65

多米尼克·阿科塞拉和马克·R·高迪。美国专利 #7,750,915: 在共享内存资源中对存储在多个银行中的数据元素的并发访问（受让方：NVIDIA公司），2010年7月。68

Neha Agarwal, David Nellans, Mark Stephenson, Mike O’Connor 和 Stephen W. Keckler. 在异构内存系统中针对 GPU 的页面放置策略. 在 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. DOI: 10.1145/2694344.2694381. 82

Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, 和 John Wickerson. GPU 并发：弱行为和编程假设。在 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*，页577 – 591，2015。DOI: 10.1145/2694344.2694391。72

约翰·R·艾伦，肯·肯尼迪，凯莉·波特菲尔德和乔·沃伦。将控制依赖转换为数据依赖。在 *Proc. of the ACM Symposium on Principles and Practices of Parallel Programming (PPoPP)*，第177 – 189页，1983年。 DOI: 10.1145/567067.567085。 16

罗伯特·阿尔弗森，大卫·卡拉汉，丹尼尔·卡ummings，布赖恩·科布伦茨，艾伦·波特菲尔德和伯顿·史密斯。《tera计算机系统》。在*Proc. of the ACM International Conference on Supercomputing (ICS)*，第1-6页，1990年。DOI：10.1145/77726.255132。16

R700-Family Instruction Set Architecture. AMD, 2009年3月。49

AMD Southern Islands Series Instruction Set Architecture. AMD, 第1.1版，2012年12月。13、17、19、20、26、54、58、74

A. Arunkumar, S. Y. Lee 和 C. J. Wu. ID-cache：基于指令和内存分歧的GPU缓存管理。在 *Proc. of the IEEE International Symposium on Workload Char- acterization (IISWC)*，2016。DOI: 10.1109/iiswc.2016.7581276。79

阿基尔·阿鲁库马尔、叶戈尼·博洛丁、本杰明·乔、乌戈耶萨·米利奇、艾曼·埃布拉希米、奥雷斯特·维拉、阿默尔·贾利尔、卡罗尔-简·吴和大卫·内兰斯。MCM-GPU：用于持续性能扩展的多芯片模块GPU。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第320 – 332页，2017年。DOI：10.1145/3079856.3080231。84

Krste Asanovic, Stephen W. Keckler, Yunsup Lee, Ronny Krashinsky, 和 Vinod Grover. 数据并行架构的收敛与标量化. In *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 2013. DOI: 10.1109/cgo.2013.6494995. 54, 56, 58

Rachata Ausavarungnirun, Saugata Ghose, Onur Kayran, Gabriel H. Loh, Chita R. Das, Mahmut T. Kandemir, 和 Onur Mutlu. 利用跨块异构性提高 GPGPU 性能. 在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2015. DOI: 10.1109/pact.2015.38. 80

Ali Bakhoda、George L. Yuan、Wilson W. L. Fung、Henry Wong 和 Tor M. Aamodt。使用详细的 GPU 模拟器分析 CUDA 工作负载。在 *Proc. of the IEEE Symposium of Performance and Analysis of Systems and Software, (ISPASS'09)*，第 163 – 174 页，2009 年。DOI: 10.1109/ispass.2009.4919648。6、14、78

Ali Bakhoda, John Kim和Tor M. Aamodt。针对多核加速器的高效能片上网络。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第421 – 432页，2010年。DOI: 10.1109/micro.2010.50。77

阿里·巴霍达、约翰·金和托尔·M阿莫特。为吞吐量加速器设计片上网络。*ACM Transactions on Architecture and Code Optimization (TACO)*, 10(3):21, 2013。 DOI: 10.1145/2509420.2512429。 77

Markus Billeter、Ola Olsson 和 Ulf Assarsson。高效流压缩在宽 SIMD 多核架构上。在 *Proc. of the ACM Conference on High Performance Graphics*，第 159 – 166 页，2009 年。DOI: 10.1145/1572769.1572795。 45

尼古拉斯·布鲁尼、希尔万·科朗热和格雷戈里·迪阿莫斯。持续GPU性能的同时分支和波浪交错。在*Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第49 – 60页，2012年。DOI: 10.1109/isca.2012.6237005。44，53

伊恩·巴克、蒂姆·福尔、丹尼尔·霍恩、杰瑞米·苏格曼、凯文·法塔哈利安、迈克·休斯顿和帕特·汉拉汉。《用于GPU的Brook：图形硬件上的流计算》。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)* 中，页面 777 – 786，2004年。DOI: 10.1145/1186562.1015800。6

布赖恩·卡布拉尔。“ SASS ”代表什么？ `https://stackoverflow.com/questions/9798258/what-is-sass-short-for`，2016年11月。14

J. F. Cantin, M. H. Lipasti，和 J. E. Smith. 通过粗粒度一致性跟踪提高多处理器性能. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2005年6月. DOI: 10.1109/isca.2005.31. 100

埃德温·卡特穆尔。用于计算机显示曲面的一种细分算法。*Technical Report*，DTIC 文档，1974年。72

哈桑·查非、贾里德·卡斯珀、布赖恩·D·卡尔斯特罗姆、奥斯汀·麦克唐纳、蒯驰、闵·吴京·白克、基督斯·科齐拉基斯和昆勒·奥卢科顿。一个可扩展的非阻塞事务内存方法。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第97 – 108页，2007年。DOI: 10.1109/hpca.2007.346189。97

郭扬陈、博吴、东李和西鹏沈。《PORPLE：一种用于GPU可移植数据放置的可扩展优化器》。在*Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014a。DOI: 10.1109/micro.2014.20. 83

Xi E. Chen 和 Tor M. Aamodt. 一种一阶细粒度多线程吞吐量模型. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)* 中，页码 329 – 340，2009 年。DOI: 10.1109/hpca.2009.4798270. 78

徐浩 陈，李文 蒋，克里斯托弗 I. 罗德里格斯，崔杰，王智颖，和吴文梅。用于节能 GPU 计算的自适应缓存管理。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014b。DOI: 10.1109/micro.2014.11. 81, 82

Sylvain Collange, David Defour 和 Yao Zhang. GPGPU 计算中均匀和仿射向量的动态检测. 在 *Proc. of the European Conference on Parallel Processing (Euro-Par)*, 2010. DOI: 10.1007/978-3-642-14122-5_8. 59

布雷特·W·库恩和约翰·埃里克·林霍尔姆。美国专利 #7,353,369：在 SIMD 体系结构中管理发散线程的系统和方法（受让方 NVIDIA 公司），2008年4月。26, 49, 50

布雷特·W·库恩，彼得·C·米尔斯，斯图尔特·F·奥伯曼和孟·Y·秀。美国专利 #7,434,032：使用具有独立内存区域的记分板进行多线程处理时跟踪寄存器使用情况，并存储顺序寄存器大小指示符（受让人为NVIDIA公司），2008年10月。34

布雷特·W·库恩，约翰·埃里克·林德霍尔姆，盖瑞·塔罗利，斯维托斯拉夫·D·茨维特科夫，约翰·R·尼科尔斯，和许明·Y·肖。美国专利 #7,634,621：寄存器文件分配（受让人 NVIDIA 公司），2009年12月。35

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman 和 F. Kenneth Zadeck。高效计算静态单赋值形式和控制依赖图。 *ACM Transactions on Programming Languages and Systems (TOPLAS)*，13(4)：451 – 490，1991年。 DOI：10.1145/115372.115320。 16

卢克·达莱桑德罗，迈克尔·F斯皮尔和迈克尔·L斯科特。NOrec：通过废除所有权记录来简化STM。在*Proc. of the ACM Symposium on Principles and Practices of Parallel Programming (PPoPP)*，第67 – 78页，2010。DOI：10.1145/1693453.1693464。96

R. H. Dennard, F. H. Gaensslen 和 K. Mai。设计具有非常小物理尺寸的离子注入 MOSFET。如果 *IEEE Journal of Solid-State Circuits*，1974 年 10 月。 DOI: 10.1109/jssc.1974.1050511。 1

格雷戈里·迪亚莫斯、本杰明·阿什鲍、苏布拉马尼亚姆·迈尤兰、安德鲁·克尔、怀城·吴和苏达卡尔·亚拉曼奇利。线程边界的SIMD重聚。在*Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第477 – 488页，2011年。DOI: 10.1145/2155620.2155676。26，51，54

格雷戈里·弗雷德里克·迪阿莫斯，理查德·克雷格·约翰逊，维诺德·格罗弗，奥利维尔·吉鲁，杰克·H·肖凯特，迈克尔·艾伦·费特曼，阿杰·S·提鲁马拉，彼得·尼尔森，以及罗尼·梅尔·克拉辛斯基。使用收敛屏障执行发散线程，2015年7月13日。27，28，29，30

罗杰·艾克特。美国专利 #7,376,803：用于DRAM系统的页面流排序器（受让方：NVIDIA公司），2008年5月。73

罗杰·埃克特。 美国专利 #9,195,618：调度内存请求的方法和系统（受让人：NVIDIA公司），2015年11月。 73

约翰·H·埃德蒙森和詹姆斯·M·范·戴克。美国专利 #7872657：使用分区跨度的内存寻址方案，2011年1月。75

约翰·H·爱德蒙森等人 美国专利 #8,464,001：缓存及与帧缓冲管理脏数据拉取和高优先级清理机制相关的方法，2013年6月。75, 76

Ahmed ElTantaway, Jessica Wenjie Ma, Mike O'Connor, 和 Tor M. Aamodt. 一种可扩展的多路径微架构用于高效的GPU控制流. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2014. DOI: 10.1109/h-pca.2014.6835936. 26, 28, 30, 31, 49, 52

Ahmed ElTantawy 和 Tor M. Aamodt. 在 SIMT 架构上进行 MIMD 同步。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 1 – 14 页，2016 年。 DOI: 10.1109/micro.2016.7783714。 26, 27, 30, 32, 93

Ahmed ElTantawy 和 Tor M. Aamodt. 为精细粒度同步的扭曲调度. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)* 中，页码 375 – 388, 2018. DOI: 10.1109/hpca.2018.00040. 93

亚历山大·L敏肯等，美国专利 #7,649,538：具有先进过滤的可重构高性能纹理管道（受让方：NVIDIA 公司），2010年1月。 72

Wilson W. L. Fung. *GPU Computing Architecture for Irregular Parallelism*. 博士论文，英属哥伦比亚大学，2015年1月。DOI：10.14288/1.0167110。41

Wilson W. L. Fung 和 Tor M. Aamodt。线程块压缩以实现高效的 SIMT 控制流。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 25 – 36 页，2011 年。DOI: 10.1109/hpca.2011.5749714。26，28，42，43，50，91

威尔逊 W. L. 冯 和 托尔 M. Aamodt. 通过时空优化实现能效 GPU 事务内存. 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 408 – 420 页，2013 年. DOI: 10.1145/2540708.2540743. 98

Wilson W. L. Fung, Ivan Sham, George Yuan 和 Tor M. Aamodt. 动态波形变换和调度以实现高效的 GPU 控制流。载于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 407 – 420 页，2007 年。DOI: 10.1109/micro.2007.4408272。14, 23, 25, 42, 44, 49, 91

Wilson W. L. Fung，Inderpreet Singh，Andrew Brownsword 和 Tor M. Aamodt。GPU 架构的硬件事务内存。载于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 296 – 307 页，2011 年。DOI: 10.1145/2155620.2155655。96，97

Wilson Fung et al. 动态变形：在 SIMD 图形硬件上有效的 MIMD 控制流。*ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):7:1 – 7:37, 2009. DOI: 10.1145/1543753.1543756. 42, 49

M. Gebhart, S. W. Keckler 和 W. J. Dally. 一个编译时管理的多级寄存器文件层次结构. 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2011年12月。DOI: 10.1145/2155620.2155675. 63

Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, 和 Kevin Skadron. 在流处理器中管理线程上下文的节能机制. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2011b. DOI : 10.1145/2000064.2000093. 63 Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, 和 Kevin Skadron. 在流处理器中管理线程上下文的节能机制. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 235 – 246, 2011c. DOI: 10.1145/2000064.2000093. 88 Isaac Gelado , John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, 和 Wen-mei W. Hwu. 用于异构并行系统的非对称分布式共享内存模型. 在 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 页码 347 – 358, 2010. DOI: 10.1145/1736020.1736059. 3 Syed Zohaib Gilani, Nam Sung Kim, 和 Michael J. Schulte. 针对计算密集型 GPGPU 应用的节能计算. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2013. DOI: 10.1109/hpca.2013.6522330. 59, 61 David B. Glasco 等. 美国专利 #8,135,926: 结合外部 ALU 块的基于缓存的原子操作控制, 2012 年 3 月. 76 David B. Glasco 等. 美国专利 #8,539,130: 有效数据包传输的虚拟通道, 2013 年 9 月. 75 Scott Gray. NVIDIA Maxwell 架构的汇编器. `https://github.com/NervanaSyst ems/maxas` 16, 40 Zvika Guz, Evgeny Bolotin, Idit Keidar, Avinoam Kolodny, Avi Mendelson, 和 Uri C. Weiser. 多核与多线程机器: 远离山谷. *IEEE Computer Architecture Letters*, 8(1):25 – 28, 2009. DOI: 10.1109/l-ca.2009.4. 5 Ziyad S. Hakura 和 Anoop Gupta. 纹理映射缓存架构的设计与分析. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 108 – 120, 1997. DOI: 10.1145/264107.264152. 73 Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, 和 Mark Horowitz. 理解通用芯片中的低效来源. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 37 – 47 , 2010. DOI: 10.1145/1815961.1815968. 1 Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, 和 William J. Dally. EIE: 在压缩深度神经网络上的高效推理引擎. 在

Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)，第243 – 254页，2016年。DOI: 10.1109/isca.2016.30. 6

马克·哈里斯。*An Easy Introduction to CUDA C and C++*。https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/，2012年。11

Tim Harris, James Larus 和 Ravi Rajwar. *Transactional Memory*, 第2版. Morgan & Claypool, 2010. DOI: 10.1201/b11417-16. 96

史蒂文·J·海因里希等人。美国专利 #9,595,075：纹理硬件中的加载/存储操作（受让方：NVIDIA 公司），2017年3月。68, 73

约翰·亨尼西和大卫·帕特森。 *Computer Architecture—A Quantitative Approach*，第5版。摩根·考夫曼，2011年。1，10，71，72，78

莫里斯·赫利希和J·艾略特·B·莫斯。事务内存：对无锁数据结构的架构支持。在 *Proc. of the ACM/IEEE International Symposium on Computer Architec- ture (ISCA)*，第289 – 300页，1993年。DOI：10.1109/isca.1993.698569。96

贾里德·霍伯罗克，维克多·陆，尹涛·贾和约翰·C·哈特。延迟着色的流压缩。在 *Proc. of the ACM Conference on High Performance Graphics*，第173 – 180页，2009年。DOI: 10.1145/1572769.1572797。45

H. Peter Hofstee. 节能处理器架构与 Cell 处理器。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 258-262 页，2005。DOI: 10.1109/hpca.2005.26. 68

马克·霍洛维茨，埃拉德·阿隆，迪尼什·帕蒂尔，塞缪尔·纳夫齐格，拉杰什·库马尔，和凯里·伯恩斯坦。CMOS的扩展、功耗与未来。在 *IEEE International Electron Devices Meeting*，2005年。DOI: 10.1109/iedm.2005.1609253。5

霍曼·伊盖希、马修·艾尔德里奇和克科阿·普劳德福特。《纹理缓存架构中的预取》。在 *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*，第 133 页及后续，1998年。DOI：10.1145/285305.285321。72, 73, 74

Hyeran Jeon, Gokul Subramanian Ravi, Nam Sung Kim 和 Murali Annavaram。GPU 寄存器文件虚拟化。载于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 420 – 432 页，2015年。DOI: 10.1145/2830772.2830784。64

W. Jia, K. A. Shaw, 和 M. Martonosi. MRPB：大规模并行处理器的内存请求优先级排序。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2014. DOI: 10.1109/hpca.2014.6835938. 78, 79

Wenhao Jia, Kelly A Shaw 和 Margaret Martonosi。描述和改进GPU中需求获取缓存的使用。在*Proc. of the ACM International Conference on Supercom- puting (ICS)*，第15 – 24页，2012年。DOI: 10.1145/2304576.2304582。78

Adwait Jog、Onur Kayiran、Nachiappan Chidambaram Nachiappan、Asit K. Mishra、Mah mut T. Kandemir、Onur Mutlu、Ravishankar Iyer 和 Chita R. Das。OWL：用于提高 GPG PU 性能的协作线程阵列感知调度技术。在 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*，2013a。DOI：10.1145/2451116.2451158。90

Adwait Jog, Onur Kayiran, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar I yer, 和 Chita R. Das. GPGPU 的协调调度和预取. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013b. DOI: 10. 1145/2508148.2485951. 90

诺曼·P·乔皮，克里夫·扬，尼尚特·帕提尔，戴维·帕特森，古拉夫·阿格拉瓦尔，拉敏德·巴哈，莎拉·贝茨，苏雷什·巴蒂亚，南·博登，阿尔·博彻斯等。在数据中心内对张量处理单元的性能分析。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2017。DOI：10.1145/3079856.3080246。2

大卫·R凯利，佩哈德·米斯特里，达娜·沙阿，和董平·张。*Heterogeneous Computing with OpenCL 2.0*。摩根·考夫曼出版社，2015。10

Ujval J. Kapasi 等人在数据并行架构上的高效条件操作。发表于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 第 159 – 170 页, 2000 年。DOI: 10.1109/micro.2000.898067. 45

O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. M utlu, 和 C. R. Das. 在异构架构中管理 GPU 并发。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014。DOI: 10.1109/micro.2014.62。99

Onur Kayiran, Adwait Jog, Mahmut T. Kandemir, 和 Chita R. Das。既不偏多也不少：为 G PGPU 优化线程级并行性。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2013。86

S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, 和 D. Glasco. GPU 和并行计算的未来. *Micro, IEEE*, 31(5):7 – 17, 2011年9月. DOI: 10.1109/mm.2011.89. 53, 58

香农·凯尔和约翰·H·爱德蒙森。美国专利#8,195,858：在共享L2总线上管理冲突，2012年6月。77

Shane Keil 等人。美国专利 #8,307,165：高页面局部性的 DRAM 请求排序，2012年11月。77

Farzad Khorasani, Rajiv Gupta 和 Laxmi N. Bhuyan。 在存在分歧的情况下高效执行波段与协作上下文收集。 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2015。 DOI: 10.1145/2830772.2830796。 45

J. Y. Kim 和 C. Batten. 使用细粒度硬件任务列表加速 GPGPU 上的不规则算法. 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MI- CRO)*，2014 年。DOI: 10.1109/micro.2014.24. 93

Ji Kim、Christopher Torng、Shreesha Srinath、Derek Lockhart 和 Christopher Batten。利用 SIMT 架构中的价值结构的微架构机制。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2013 年。DOI: 10.1145/2508148.2485934。57, 58, 59, 60, 61, 62

Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, 和 Mark Silberstein. GPUnet: GPU 程序的网络抽象. 在 *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, 第 6 – 8 页, 2014. DOI: 10.1145/2963098. 2

大卫·B·柯克和W·黄文梅。*Programming Massively Parallel Processors: A Hands-on Approach*。摩根·考夫曼，2016年。DOI: 10.1016/c2011-0-04129-7。9

约翰·克鲁斯特曼、乔纳森·博蒙特、D·阿努谢·贾姆希迪、乔纳森·贝利、特雷弗·马奇和斯科特·马尔基。Regless：基于GPU的即时操作数暂存。在*Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第151 – 164页，2017年。DOI: 10.1145/3123939.3123974。65

R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper 和 K. Asanovic. 向量线程架构. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 52 – 63 页，2004年6月。 DOI: 10.1109/isca.2004.1310763. 52

Ronny M. Krashinsky. 美国专利申请 #20130042090 A1：时间 SIMT 执行优化，2011 年 8 月。53，58

大卫·克罗夫特。无锁定指令获取/预取缓存组织。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 81 – 87 页，1981年。doi: 10.1145/285930.285979。33，71

延斯·克鲁格和鲁迪格·韦斯特曼。线性代数运算符用于数值算法的GPU实现。在 *ACM Transactions on Graphics (TOG)*，第22卷，页面908 – 916，2003年。DOI: 10.1145/882262.882363。6

Junjie Lai 和 André Seznec. Fermi 和 Kepler GPU 上 SGEMM 的性能上限分析与优化。在 *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*，第 1 – 10 页，2013。DOI: 10.1109/cgo.2013.6494986。16

Nagesh B. Lakshminarayana 和 Hyesoon Kim. 指令获取和内存调度对 GPU 性能的影响. 在 *Workshop on Language, Compiler, and Architecture Support for GPGPU*，2010。88

Ahmad Lashgar、Ebad Salehi 和 Amirali Baniasadi。关于反向工程 GPG-PU 的案例研究：优秀的内存处理资源。*ACM SIGARCH Computer Architecture News*，43(4)：15 – 21，2016。DOI：10.1145/2927964.2927968。34 C. L. Lawson、R. J. Hanson、D. R. Kincaid 和 F. T. Krogh。Fortran 使用的基本线性代数子程序。*ACM Transactions on Mathematical Software*，5(3)：308 – 323，1979 年 9 月。DOI：10.1145/355841.355848。10

HyoukJoong Lee, Kevin J. Brown, Arvind K. Sujeeth, Tiark Rompf, 和 Kunle Olukotun. 基于局部性的嵌套并行模式在GPU上的映射. 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2014a. DOI: 10.1109/micro.2014.23. 94

J. Lee 和 H. Kim. TAP：一种针对 CPU-GPU 异构架构的 TLP 感知缓存管理策略。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2012。DOI: 10.1109/hpca.2012.6168947。100 S. Y. Lee 和 C. J. Wu. Ctrl-C：一种基于指令感知控制循环的自适应缓存绕过策略，用于 GPU。在 *Proc. of the IEEE International Conference on Computer Design (ICCD)*，第 133 – 140 页，2016。DOI: 10.1109/iccd.2016.7753271。80 Sangpil Lee, Keunsoo Kim, Gunjae Koo, Hyeran Jeon, Won Woo Ro 和 Murali Annavaram. Warped-compression：通过寄存器压缩实现高效能 GPU 的能量使用。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2015。DOI: 10.1145/2749469.2750417。59, 60, 61 Shin-Ying Lee 和 Carole-Jean Wu. CAWS：针对 GP GPU 工作负载的关键性感知 warp 排程。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2014。DOI: 10.1145/2628071.2628107。93 Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund 等. 揭穿 GPU 与 CPU 之间的 100X 神话：对 CPU 和 GPU 上吞吐量计算的评估。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 451 – 460 页，2010。DOI: 10.1145/1815961.1816021。2

李允燮、瑞玛斯·阿维兹伊尼斯、亚历克斯·比沙拉、理查德·夏、德里克·洛克哈特、克里斯托弗·巴滕和克尔斯特·阿萨诺维奇。探索数据并行加速器中可编程性与效率之间的权衡。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 129 – 140页，2011年。DOI: 10.1145/2000064.2000080。53

Yunsup Lee, Vinod Grover, Ronny Krashinsky, Mark Stephenson, Stephen W. Keckler 和 Krste Asanović。探索数据上的SPMD分歧管理设计空间-

并行架构。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014b。DOI: 10.1109/micro.2014.48。55, 56 Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt 和 Vijay Janapa Reddi。GPUWattch: 在 GPG-PUs 中实现能量优化。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 487 – 498 页，2013。DOI: 10.1145/2508148.2485964。6 Adam Levinthal 和 Thomas Porter。章节——一个 SIMD 图形处理器。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，第 77 – 82 页，1984。DOI: 10.1145/800031.808581。50 Dong Li, Minsoo Rhu, Daniel R. Johnson, Mike O'Connor, Mattan Erez, Doug Burger, Donald S. Fussell 和 Stephen W. Redder。在吞吐量处理器中基于优先级的缓存分配。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2015。DOI: 10.1109/hpca.2015.7056024。82 E. Lindholm, J. Nickolls, S. Oberman 和 J. Montrym。NVIDIA Tesla: 一种统一的图形和计算架构。*Micro, IEEE*，28(2):39 – 55，2008年3月 – 4月。DOI: 10.1109/mm.2008.31。9 Erik Lindholm, Mark J. Kilgard 和 Henry Moreton。用户可编程的顶点引擎。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，第 149 – 158 页，2001。DOI: 10.1145/383259.383274。6, 21 John Erik Lindholm, Ming Y. Siu, Simon S. Moy, Samuel Liu 和 John R. Nickolls。美国专利 #7,339,592: 使用较低端口数内存模拟多端口内存（受让方 NVIDIA 公司），2008年3月。35, 38 Erik Lindholm 等。美国专利 #9,189,242: 基于信用的流处理器 Warp 调度（受让方 NVIDIA 公司），2015年11月。33, 41 John S. Liptay。System/360 模型 85 的结构方面 II：缓存。在 *IBM Systems Journal*，7(1):15 – 21，1968。DOI: 10.1147/sj.71.0015。70 Z. Liu, S. Gilani, M. Annavaram 和 N. S. Kim。G-Scalar: 一种成本效益高的通用标量执行架构，用于节能 GPU。 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2017。DOI: 10.1109/hpca.2017.51。60, 61, 62 Samuel Lui, John Erik Lindholm, Ming Y. Siu, Brett W. Coon 和 Stuart F. Oberman。美国专利申请 11/555,649: 操作数收集器架构（受让方 NVIDIA 公司），2008年5月。35

迈克尔·D·麦库尔、阿奇·D·罗宾逊和詹姆斯·雷因德斯。*Structured Parallel Programming: Patterns for Efficient Computation*。爱思唯尔，2012。10

介元梦、戴维·塔尔扬和凯文·斯卡德隆。集成分支和内存分歧容忍的动态波段细分。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 235 – 246 页，2010 年。doi: 10.1145/1815961.1815992。30, 48, 90

亚历山大·L·敏肯和奥伦·鲁宾斯坦。美国专利 #6,629,188：用于纹理缓存的数据预取电路和方法（受让人：NVIDIA公司），2003年9月。72

亚历山大·L·明金等。美国专利 #8,266,383：使用延迟/重放机制的缓存未命中处理（受让方：英伟达公司），2012年9月。68, 69, 71

亚历山大·L·敏金等。美国专利 #8,595,425: 可配置缓存用于多个客户端（受让方：英伟达公司），2013年11月。 68

迈克尔·米什金、南成金和米科·里帕斯蒂。在 GPGPUSIM 中的读后写危害预防。在 *Workshop on Deplicating, Deconstructing, and Debunking (WDDD)*，2016 年 6 月。39，40

约翰·蒙特里姆和亨利·莫顿。GeForce 6800。 *IEEE Micro*, 25(2):41 – 51, 2005。DOI: 10.1109/mm.2005.37. 1

Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu 和 Yale N. Patt。通过大型 Warp 和两级 Warp 调度提高 GPU 性能。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 308 – 317 页，2011 年。DOI: 10.1145/2155620.2155656。 33, 38, 43, 88, 91

John R. Nickolls 和 Jochen Reusch. MP-1 和 MP-2 中的自主 SIMD 灵活性. 在 *Proc. of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*，第 98 – 99 页, 1993. DOI: 10.1145/165231.165244. 9

Cedric Nugteren, Gert-Jan Van den Braak, Henk Corporaal, 和 Henri Bal. 基于重用距离理论的详细GPU缓存模型. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第37-48页，2014年. DOI: 10.1109/hpca.2014.6835955. 79

NVIDIA's Next Generation CUDA Compute Architecture: Fermi. NVIDIA，2009。16，46

Nvidia. *NVIDIA tesla V100 GPU architecture*. 2017. 27, 31

NVIDIA公司 Pascal l1 缓存。 `https://devtalk.nvidia.com/default/topic/1006066/pascal-l1-cache/?offset=6` 70

NVIDIA 公司。Inside volta：全球最先进的数据中心 GPU。https://devblogs.nvidia.com/parallelforall/inside-volta/，2017年5月。1, 17, 26 NVIDIA 公司。*NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110*，a. 13 NVIDIA 公司。*NVIDIA GeForce GTX 680*，b. 16 NVIDIA 公司。*CUDA Binary Utilities*，c. 16 *Parallel Thread Execution ISA (Version 6.1)*。NVIDIA 公司，CUDA Toolkit 9.1 版，2017年11月。14 Lars Nyland 等。美国专利 #8,086,806：用于合并并行线程内存访问的系统和方法（受让人：NVIDIA 公司），2011年12月。71 Marek Olszewski、Jeremy Cutler 和 J. Gregory Steffan。JudoSTM：一种动态二进制重写的软件事务内存方法。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，第 365 – 375 页，2007 年。DOI: 10.1109/pact.2007.4336226。96 Jason Jong Kyu Park、Yongjun Park 和 Scott Mahlke。Chimera：在共享 GPU 上进行多任务的协作抢占。在 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*，2015 年。DOI: 10.1145/2694344.2694346。92 David A. Patterson 和 John L. Hennessy。*Computer Organization and Design: The Hardware/-Software Interface*。2013。78 Gennady Pekhimenko、Vivek Seshadri、Onur Mutlu、Phillip B. Gibbons、Michael A. Kozuch 和 Todd C. Mowry。基于增量的立即压缩：针对片上缓存的实用数据压缩。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2012 年。DOI: 10.1145/2370816.2370870。60 J. Power、A. Basu、J. Gu、S. Puthoor、B. M. Beckmann、M. D. Hill、S. K. Reinhardt 和 D. A. Wood。针对集成 CPU-GPU 系统的异构系统一致性。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2013 年 12 月。DOI: 10.1145/2540708.2540747。100 Jason Power、Arkaprava Basu、Junli Gu、Sooraj Puthoor、Bradford M. Beckmann、Mark D. Hill、Steven K. Reinhardt 和 David A. Wood。针对集成 CPU-GPU 系统的异构系统一致性。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 457 – 467 页，2013 年。DOI: 10.1145/2540708.2540747。4

M. K. Qureshi 和 Y. N. Patt. 基于效用的缓存分区：一种低开销、高性能的运行时机制，用于分区共享缓存。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 423 – 432 页，2006。DOI: 10.1109/micro.2006.49. 101

任小伟和米耶兹科·利斯。通过相对论缓存一致性在GPU中实现高效的顺序一致性。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第625 – 636页，2017年。DOI: 10.1109/hpca.2017.40. 72

Minsoo Rhu 和 Mattan Erez. CAPRI: 对 GPGPU 架构中控制分歧处理的压实适应性预测。在 *Proc. of the ACM/IEEE International Sympo- sium on Computer Architecture (ISCA)*，第 61 – 71 页，2012年。 DOI: 10.1109/isca.2012.6237006。 44

Minsoo Rhu 和 Mattan Erez. 高效 GPU 控制流的双路径执行模型. 收录于 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 页码 591 – 602, 2013a. DOI: 10.1109/hpca.2013.6522352. 49

Minsoo Rhu 和 Mattan Erez. 在 GPGPU 中通过 SIMD 通道排列最大化 SIMD 资源利用率. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013b. DOI: 10.1145/2485922.2485953. 46 Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson 和 John D. Owens. 内存访问调度. 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 128 – 138, 2000. DOI: 10.1109/isca.2000.854384. 77 James Roberts 等. 美国专利 #8,234,478: 使用数据缓存阵列作为 DRAM 加载/存储缓冲区, 2012 年 7 月. 75

蒂莫西·G·罗杰斯, 迈克·奥康纳, 和托尔·M·阿莫德特. 缓存意识的波前调度. 在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*中, 2012年. DOI: 10.1109/micro.2012.16. 33, 78, 79, 88

蒂莫西·G·罗杰斯、迈克·奥康纳和托尔·M·阿莫德特。考虑差异的波动调度。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2013。 DOI: 10.1145/2540708.2540718。79, 90

蒂莫西·G·罗杰斯，丹尼尔·R·约翰逊，迈克·奥康纳和斯蒂芬·W·凯克勒。一个可变的波形大小架构。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2015。DOI: 10.1145/2749469.2750410. 53

Sangmin Seo, Gangwon Jo 和 Jaejin Lee. 在 OpenCL 中对 NAS 并行基准的性能特征分析. 在 *Proc. of the IEEE International Symposium on Workload Charac- terization (IISWC)*，第 137 – 148 页，2011 年. DOI: 10.1109/iiswc.2011.6114174. 10

A. Sethia, D. A. Jamshidi 和 S. Mahlke. Mascar: 通过减少内存停顿来加速 GPU 线程组. 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 第 174 – 185 页, 2015. DOI: 10.1109/hpca.2015.7056031. 91, 92

安基特·塞西亚和斯科特·马赫尔克。均衡器：动态调节GPU资源以实现高效执行。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014。DOI: 10.1109/micro.2014.16。86

瑞安·施劳特。AMD ATI Radeon HD 2900 XT 评测：R600 到来。*PC Perspective*，2007年5月。75

Mark Silberstein, Bryan Ford, Idit Keidar 和 Emmett Witchel。GPUfs：将文件系统与GPU集成。见 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*，第 485 – 498 页，2013 年。DOI: 10.1145/2451116.2451169。2 Inderpreet Singh, Arrvindh Shriraman, Wilson W. L. Fung, Mike O ’Connor 和 Tor M. Aamodt。GPU体系结构的缓存一致性。见 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 578 – 590 页，2013 年。DOI: 10.1109/hpca.2013.6522351。72 Michael F. Spear, Virendra J. Marathe, William N. Scherer 和 Michael L. Scott。软件事务内存的冲突检测与验证策略。见 *Proc. of the EATCS International Symposium on Distributed Computing*，第 179 – 193 页，Springer-Verlag，2006 年。DOI: 10.1007/11864219_13。99 Michael F. Spear, Maged M. Michael 和 Christoph Von Praun。RingSTM：使用单个原子指令的可扩展事务。见 *Proc. of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*，第 275 – 284 页，2008 年。DOI: 10.1145/1378533.1378583。97 Michael Steffen 和 Joseph Zambreno。通过架构支持动态微内核来提高全局渲染算法的SIMT效率。见 *Proc. of the ACM/IEEE Inter- national Symposium on Microarchitecture (MICRO)*，第 237 – 248 页，2010 年。DOI: 10.1109/micro.2010.45。45 Ivan E. Sutherland。*Sketchpad a Man-machine Graphical Communication System*。博士论文，1963 年。DOI: 10.1145/62882.62943。6 David Tarjan 和 Kevin Skadron。针对多线程处理器的按需寄存器分配与回收，2011 年 6 月 30 日。美国专利申请 12/649,238。64 Sean J. Treichler 等。美国专利 #9,098,383：支持多种流量类型的综合交叉开关，2015 年 8 月。75

Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo 和 Rebecca L. Stamm. 利用选择：在可实现的同时多线程处理器上的指令提取和发射。 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，1996。 DOI: 10.1145/232973.232993。 88 Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa 和 David Kaeli. Multi2Sim：用于 CPU-GPU 计算的仿真框架。 在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，第 335 – 344 页，2012。 DOI: 10.1145/2370816.2370865。 17 Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy 和 Mani Az- imi. 通过内部 warp 压缩进行 SIMD 拆分优化。 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 368 – 379 页，2013。 DOI: 10.1145/2485922.2485954。 44, 46 Wladimir J. van der Lann. Decuda。 `http://wiki.github.com/laanwj/decuda/` 14 Jin Wang 和 Sudhakar Yalamanchili. 无结构 GPU 应用中的动态并行性特征与分析。 在 *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*，第 51 – 60 页，2014。 DOI: 10.1109/iiswc.2014.6983039。 95 Jin Wang, Norm Rubin, Albert Sidelnik 和 Sudhakar Yalamanchili. 动态线程块发射：支持 GPU 上不规则应用的轻量级执行机制。 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 528 – 540 页，2016a。 DOI: 10.1145/2749469.2750393。 95 Jin Wang, Norm Rubin, Albert Sidelnik 和 Sudhakar Yalamanchili. Laperm：用于 GPU 上动态并行性的局部性感知调度器。 在 *Proc. of the ACM/IEEE International Sympo- sium on Computer Architecture (ISCA)*，2016b。 DOI: 10.1109/isca.2016.57。 96 Kai Wang 和 Calvin Lin. SIMT GPU 的解耦仿射计算。 在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2017。 DOI: 10.1145/3079856.3080205。 58, 61, 62 D. Wong, N. S. Kim 和 M. Annavaram. 通过内部 warp 操作数值相似性近似 warp。 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2016。 DOI: 10.1109/hpca.2016.7446063。 60, 61 X. Xie, Y. Liang, Y. Wang, G. Sun 和 T. Wang. 针对 GPU 的协调静态和动态缓存绕过。 在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2015。 DOI: 10.1109/hpca.2015.7056023。 81

云龙 Xu, 瑞 Wang, Nilanjan Goswami, Tao Li, Lan Gao, 和 德配 Qian. 面向 GPU 架构的软件事务内存. 在 *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*, 页面 1:1 – 1:10, 2014. DOI: 10.1145/2581122.2544139. 99

Y. Yang, P. Xiang, M. Mantor, N. Rubin, L. Hsu, Q. Dong, 和 H. Zhou. 在 SIMT 架构中支持灵活标量单元的案例研究。 在 *Proc. of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014年。 DOI: 10.1109/ipdps.2014.21. 47

George L. Yuan, Ali Bakhoda和Tor M. Aamodt. 针对多核加速器架构的复杂性有效内存访问调度. 在*Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 第34 – 44页, 2009年. DOI: 10.1145/1669112.1669119. 77

侯云青。NVIDIA FERMI的汇编器。`https://github.com/hyqneuron/asfermi` 16 张艾迪·Z，蒋云廉，郭子宇和沈希朋。实时精简GPU应用：通过运行时线程-数据重新映射消除线程发散。出现在*Proc. of the ACM International Conference on Supercomputing (ICS)*，第115 – 126页，2010年。DOI: 10.1145/1810085.1810104。45 威廉·K·祖拉夫和蒂莫西·罗宾逊。美国专利 #5,630,096：一种控制器，用于同步DRAM，通过允许内存请求和命令无序发出以最大化吞吐量，1997年5月13日。77

作者简介

TOR M. AAMODT

托尔·M·阿莫德特是英属哥伦比亚大学电气和计算机工程系的教授，自2006年以来一直担任该系的教职员工。他目前的研究重点是通用GPU的架构和节能计算，最近包括机器学习的加速器。与他研究小组的学生一起，他开发了广泛使用的GPGPU-Sim模拟器。他的三篇论文被*IEEE Micro Magazine*选为“最佳推荐”，第四篇被选为“最佳推荐”荣誉提名。他的一篇论文还被选为*Communications of the ACM*.的“研究亮点”。他被列入MICRO名人堂。他曾于2012年至2015年担任*IEEE Computer Architecture Letters*的副编辑，2012年至2016年担任*International Journal of High Performance Computing Applications*的副编辑，2013年担任ISPASS的程序主席，2014年担任ISPASS的总主席，并曾在多个程序委员会任职。他于2012年至2013年在斯坦福大学计算机科学系担任访问副教授。他在2010年获得了NVIDIA学术合作奖，2016至2019年获得了NSERC发现加速器，2016年获得谷歌教员研究奖。

Tor 在多伦多大学获得了他的工学学士（BASc）、硕士（MASc）和博士学位（Ph.D.）。他的博士研究大部分是在他担任英特尔微架构研究实验室实习生期间完成的。随后，他在 NVIDIA 工作，负责 GeForce 8 系列 GPU 的内存系统架构（“帧缓存”）——这是第一款支持 CUDA 的 NVIDIA GPU。

托尔在不列颠哥伦比亚省注册为专业工程师。

威尔逊·韦伦·冯

Wilson Wai Lun Fung 是三星电子在三星奥斯汀研发中心 (SARC) 的先进计算实验室 (ACL) 的一名建筑师，他致力于下一代 GPU IP 的开发。他对计算机架构的理论和实践方面都很感兴趣。Wilson 是 NVIDIA 研究生奖学金、NSERC 研究生奖学金和 NSERC 加拿大研究生奖学金的获得者。Wilson 曾是广泛使用的 GPGPU-Sim 模拟器的主要贡献者之一。他的两篇论文被评选为计算机架构领域的“顶尖之作”由 *IEEE Micro Magazine.* Wilson 获得不列颠哥伦比亚大学的 BASc（计算机工程）、MASc 和博士学位。在攻读博士学位期间，Wilson 在 NVIDIA 实习。

蒂莫西·G·罗杰斯

蒂莫西·G·罗杰斯是普渡大学电气与计算机工程系的助理教授，他的研究专注于大规模多线程处理器设计。他对探索能够提高程序员生产力和能源效率的计算机系统和架构感兴趣。蒂莫西是NVIDIA研究生奖学金和NSERC亚历山大·格雷厄姆·贝尔加拿大研究生奖学金的获得者。他的工作被*IEEE Micro Magazine*评选为计算机架构的“最佳选择”，并在*Communications of the ACM*.中被选为“研究亮点”。在攻读博士学位期间，蒂莫西曾在NVIDIA研究院和AMD研究院实习。在上研究生之前，蒂莫西曾在艺电担任软件工程师，并获得麦吉尔大学的电气工程学士学位。