MORGAN & CLAYPOOL PUBLISHERS

通用图形处理器架构

Tor M. Aamodt Wilson
Wai Lun Fung Timoth
y G. Rogers

SYNTHESIS LECTURES ON
COMPUTER ARCHITECTURE

Margaret Martonosi, *Series Editor*

通用图形处理器架构

计算机体系结构综合讲座

编辑
玛格丽特·马托诺西, ***Princeton University***

名誉创始编辑 Mark D. Hill,

University of Wisconsin, Madison

Synthesis Lectures on Computer Architecture 出版 50 到 100 页的出版物，主题涉及设计、分析、选择和互连硬件组件的科学和艺术，以创建满足功能、性能和成本目标的计算机。范围将主要遵循顶级计算机架构会议的范围，例如 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

异构系统编译算法 Steven Bell、Jing Pu、James Hegarty 和 Mark Horowitz 2018

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

计算机架构师的深度学习 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

使用时间神经网络进行时空计算 James E. Smith 2017

四

虚拟化的硬件和软件支持 Edouard Bugnion、Ja
son Nieh 和 Dan Tsafrir 2017

数据中心设计和管理：计算机架构师的视角 Benjamin C. Lee 2016

内存层次结构压缩入门 Somayeh Sardashti、Angelos Arelakis、
Per Stenström 和 David A. Wood 2015

硬件加速器研究基础设施 Yakun Sophia Shao 和
David Brooks 2015

分析 Rajesh Bordawekar、Bob Blainey 和 R
uchir Puri 2015

Customizable Computing Yu-Ting Chen, Jason Cong, Michael Gill, Gl
enn Reinman, and Bingjun Xiao 2015

Die-stacking Architecture
Yuan Xie and Jishen Zha
o 2015
单指令多数据执行 Christopher J. Hughes 2
015

节能计算机架构：最新进展 Magnus Själander、Margare
t Martonosi 和 Stefanos Kaxiras 2014

FPGA 加速计算机系统仿真 Hari Angepat、Derek Chio
u、Eric S. Chung 和 James C. Hoe 2014

硬件预取入门 Babak Falsafi 和 T
homas F. Wenisch 2014

片上光子互连：计算机架构师的视角 Christopher J. Nitta、Matthe
w K. Farrens 和 Venkatesh Akella 2013 年

计算机架构中的优化和数学建模 Tony Nowatzki、Michael Ferris、Karthikeyan S
ankaralingam、Cristian Estan、Nilay Vaish 和 David Wood 2013

计算机架构师的安全基础知识 Ruby
B. Lee 2013

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

共享内存同步 Michael L. Scott
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 Mario Nemirovsky 和
Dean M. Tullsen 2013

通用图形处理的性能分析与调优 单位
(GPGPU) Hyesoon Kim、Richard Vuduc、Sara Baghsorkhi、Jee Cho
i 和 Wen-mei Hwu 2012

自动并行化：基本编译器技术概述 Samuel P. Midkiffi 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 Daniel J. Sorin、Mark D. Hill 和 David A. Wood 2011

动态二进制修改：工具、技术和应用 Kim Hazelwood 2011

计算机架构师的量子计算，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 Dennis Abts 和 John Kim 2011

处理器微架构：实施视角 Antonio González、Fernando Latorre 和 Grigorios Magklis 2010

事务内存，第二版 Tim Harris、James Larus 和 Ravi Rajwar 2010

计算机体系结构性能评估方法 Lieven Eeckhout 2010

可重构超级计算简介 Marco Lanzagorta、Stephen Bique 和 Robert Rosenberg 2009

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

记忆系统：你无法回避它、无法忽略它、无法伪造它 Bruce Jacob 2009

容错计算机架构 Daniel J. Sorin 2009

数据中心作为计算机：仓库规模机器的设计简介

路易斯·安德烈·巴罗佐和乌尔斯·
霍尔泽尔 2009

计算机架构技术在节能方面的应用 Stefanos Kaxiras
和 Margaret Martonosi 2008

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

事务性记忆 James R. Larus
和 Ravi Rajwar 2006

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

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

www.morganclaypool.com

ISBN：9781627059237 平装本 ISBN：9781627056182 电子书 ISBN：9781681733586 精装本

DOI 10.2200/S00848ED1V01Y201804CAC044

Morgan & Claypool 出版社系列出版物
SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

第44讲
系列编辑：Margaret Martonosi，*Princeton University* 创始编辑 名誉编辑：Mark D. Hill，*University of Wisconsin, Madison* 系列 ISSN 印刷版 1935-3235 电子版 1935-3243

通用图形处理器架构

Tor M. Aamodt
University of British Columbia
冯伟麟
Samsung Electronics
蒂莫西·G·罗杰斯
Purdue University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #44

MORGAN & CLAYPOOL PUBLISHERS

抽象的

图形处理器 (GPU) 最初是为支持视频游戏而开发的，现在越来越多地用于从机器学习到加密货币挖掘等通用（非图形）应用。与中央处理单元 (CPU) 相比，GPU 可以通过将更大比例的硬件资源用于计算来实现更高的性能和效率。此外，与领域特定加速器相比，当代 GPU 的通用可编程性使软件开发人员对它更具吸引力。本书为那些有兴趣研究支持通用计算的 GPU 架构的读者提供了入门知识。它收集了目前仅在各种不同来源中找到的信息。作者领导了 GPGPU-Sim 模拟器的开发，该模拟器广泛用于 GPU 架构的学术研究。

本书第一章介绍了 GPU 的基本硬件结构并简要概述了其历史。第 2 章总结了与本书其余部分相关的 GPU 编程模型。第 3 章探讨了 GPU 计算核心的架构。第 4 章探讨了 GPU 内存系统的架构。在描述现有系统的架构之后，第 3 章和第 4 章概述了相关研究。第 5 章总结了影响计算核心和内存系统的跨领域研究。

本书为希望了解用于通用应用程序加速的图形处理器单元 (GPU) 架构的人士，以及想要了解快速发展的有关如何改进这些 GPU 架构的研究的人士提供了宝贵的资源。

关键词

GPGPU、计算机架构

内容

Preface

本书面向希望了解图形处理器单元 (GPU) 架构并了解日益增多的改进设计研究的读者。本书假定读者熟悉计算机架构概念（例如流水线和缓存），并有兴趣进行与 GPU 架构相关的研究和/或开发。此类工作往往侧重于不同设计之间的权衡，因此本书旨在提供对此类权衡的见解，以便读者避免通过反复试验来学习经验丰富的设计师已知的知识。

为了实现这一目标，本书将目前在专利、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己的研究的学生或从业者取得成果所需的时间。

本书虽然涵盖了当前 GPU 设计的各个方面，但也试图“综合”已发表的研究成果。这部分是出于必要，因为供应商很少谈论特定 GPU 产品的微架构。在描述“基线”GPGPU 架构时，本书既依赖已发布的产品描述（期刊论文、白皮书、手册），有时也依赖专利中的描述。专利中的细节可能与实际产品的微架构大不相同。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，因为我们的重点是了解已经研究过的或可能在未来研究中值得探索的架构权衡。

本书的几个部分重点总结了最近关于改进 GPU 架构的许多研究论文。由于这个主题近年来越来越受欢迎，本书要涵盖的内容太多了。因此，我们不得不做出艰难的选择，决定要涵盖什么，要省略什么。

Tor M. Aamodt、Wilson Wai Lun Fung 和 Timothy G. Rogers 2
018 年 4 月

致谢

我们要感谢家人在撰写本书期间给予的支持。此外，我们还要感谢我们的出版商 Michael Morgan 和编辑 Margaret Martonosi，感谢他们在本书完成过程中表现出的极大耐心。我们还感谢 Carole-Jean Wu、Andreas Moshovos、Yash Ukidave、Aamir Raihan 和 Amruth Sandhupatla 对本书初稿提供的详细反馈。最后，我们感谢 Mark Hill 分享他对撰写综合讲座的策略的想法以及对本书的具体建议。

Tor M. Aamodt、Wilson Wai Lun Fung 和 Timothy G. Rogers 2
018 年 4 月

CHAPTER 1

介绍

本书探讨了图形处理器单元 (GPU) 的硬件设计。GPU 最初是为了实现实时渲染而引入的，主要应用于视频游戏。如今，从智能手机、笔记本电脑、数据中心到超级计算机，GPU 随处可见。事实上，对 Apple A8 应用处理器的分析表明，它为集成 GPU 分配的芯片面积比中央处理器单元 (CPU) 内核的芯片面积更大 [A8H]。对更逼真的图形渲染的需求是 GPU 创新的最初驱动力 [Montrym and Moreton，2005]。虽然图形加速仍然是其主要用途，但 GPU 越来越多地支持非图形计算。如今，一个备受关注的突出例子是越来越多地使用 GPU 来开发和部署机器学习系统 [NVIDIA Corp.，2017]。因此，本书的重点是与提高非图形应用程序的性能和能源效率相关的功能。

本章为入门章节，简要概述了 GPU。我们从第 1.1 节开始，探讨了更广泛类别的计算加速器的发展动机，以了解 GPU 与其他选项的比较。然后，在第 1.2 节中，我们简要概述了当代 GPU 硬件。最后，第 1.4 节提供了本书其余部分的路线图。

1.1 计算加速器的概况

几十年来，一代又一代的计算系统都表现出了价格比呈指数级增长的趋势。其根本原因是晶体管尺寸减小、硬件架构改进、编译器技术和算法改进等因素的共同作用。据估计，这些性能提升中有一半是由于晶体管尺寸减小，从而导致设备运行速度更快 [Hennessy and Patterson，2011]。然而，自 2005 年左右以来，晶体管的缩放比例已不再遵循现在称为 Dennard Scaling 的经典规则 [Dennard et al.，1974]。一个关键的后果是，随着设备尺寸的减小，时钟频率的提高速度现在要慢得多。要提高性能，就需要找到更高效的硬件架构。

通过利用硬件专业化，可以将能源效率提高多达 500× [Hameed 等人，2010]。正如 Hameed 等人所指出的，实现这种效率提升有几个关键方面。转向矢量硬件（例如 GPU 中的矢量硬件）通过消除指令处理的开销，可使效率提高约 10×。硬件专业化其余收益的很大一部分是最小化数据移动的结果，这

可以通过引入执行多个算术运算的复杂操作来实现，同时避免访问大型存储器阵列（例如寄存器文件）。

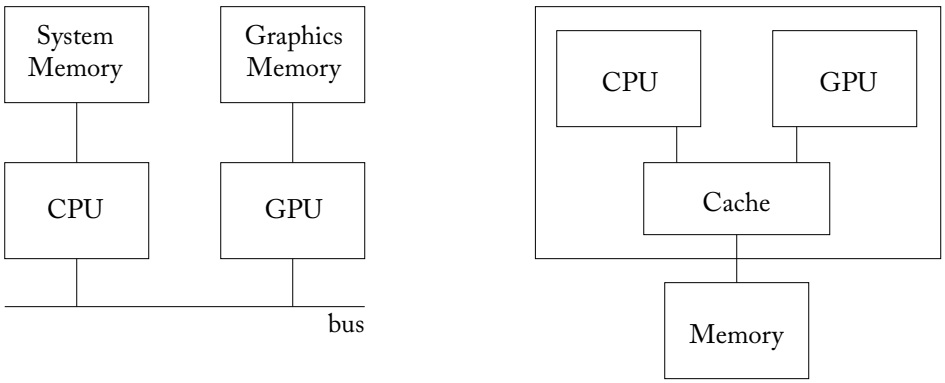
当今计算机架构师面临的一个关键挑战是找到更好的方法来平衡使用专用硬件所能获得的效率提升与支持各种程序所需的灵活性需求。在没有架构的情况下，只有可用于大量应用程序的算法才能高效运行。一个新兴的例子是专门用于支持深度神经网络的硬件，例如 Google 的张量处理单元 [Jouppi et al., 2017]。虽然机器学习似乎可能会占用很大一部分计算硬件资源，并且这些资源可能会迁移到专用硬件，但我们认为，仍然需要有效地支持以传统编程语言编写的软件来表示的计算。

除了将 GPU 用于机器学习之外，人们对 GPU 计算产生浓厚兴趣的原因之一是现代 GPU 支持图灵完备编程模型。图灵完备的意思是，只要有足够的时间和内存，就可以运行任何计算。相对于专用加速器，现代 GPU 非常灵活。对于可以充分利用 GPU 硬件的软件，GPU 的效率可以比 CPU 高出一个数量级 [Lee et al., 2010]。这种灵活性和效率的结合非常可取。因此，许多顶级超级计算机（无论是峰值性能还是能效）现在都采用了 GPU [top]。在后续几代产品中，GPU 制造商已经改进了 GPU 架构和编程模型，以提高灵活性，同时提高能效。

1.2 GPU 硬件基础知识

那些第一次接触 GPU 的人经常会问，GPU 最终是否会完全取代 CPU。这似乎不太可能。在目前的系统中，GPU 并不是独立的计算设备。相反，它们与 CPU 组合在单个芯片上，或者通过将仅包含 GPU 的附加卡插入包含 CPU 的系统中来实现。CPU 负责在 GPU 上启动计算并将数据传输到 GPU 和从 GPU 传输数据。CPU 和 GPU 之间进行这种分工的原因之一是计算的开始和结束通常需要访问输入/输出 (I/O) 设备。虽然人们一直在努力开发应用程序编程接口 (API)，以直接在 GPU 上提供 I/O 服务，但到目前为止，这些都假设存在附近的 CPU [Kim et al., 2014, Silberstein et al., 2013]。这些 API 的功能是提供方便的接口，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？用于访问 I/O 设备并以其他方式提供操作系统服务的软件似乎缺乏适合在 GPU 上运行的功能（例如大规模并行性）。因此，我们首先考虑 CPU 和 GPU 之间的交互。

图 1.1 显示了包含 CPU 和 GPU 的典型系统的抽象图。左侧是典型的独立 GPU 设置，包括连接 CPU 和 GPU 的总线（例如 PCIe），用于 NVIDIA 的 Volta GPU 等架构，右侧是典型的集成 CPU 和 GPU 的逻辑图，例如 AMD 的 Bristol Ridge APU 或移动 GPU。请注意，包含独立 GPU 的系统为 CPU（通常称为系统内存）和 GPU（通常称为设备内存）提供了单独的 DRAM 内存空间。用于这些内存的 DRAM 技术通常不同（CPU 使用 DDR，GPU 使用 GDDR）。CPU DRAM 通常针对低延迟访问进行了优化，而 GPU DRAM 针对高吞吐量进行了优化。相比之下，具有集成 GPU 的系统具有单个 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 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。

GPU 计算应用程序开始在 CPU 上运行。通常，应用程序的 CPU 部分将分配并初始化一些数据结构。在 NVIDIA 和 AMD 的旧款独立 GPU 上，GPU 计算应用程序的 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，应用程序的 CPU 部分必须协调数据从 CPU 内存到 GPU 内存的移动。较新的独立 GPU（例如 NVIDIA 的 Pascal 架构）具有软件和硬件支持，可自动将数据从 CPU 内存传输到 GPU 内存。这可以通过利用 CPU 和 GPU 上的虚拟内存支持 [Gelado 等，2010] 来实现。NVIDIA 称之为“统一内存”。在 CPU 和 GPU 集成在同一芯片上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。但是，由于 CPU 和 GPU 使用缓存和

其中一些缓存可能是私有的，可能存在缓存一致性问题，硬件开发人员需要解决这个问题[Power et al., 2013b]。

在某个时刻，CPU 必须在 GPU 上启动计算。在当前系统中，这是在 CPU 上运行的驱动程序的帮助下完成的。在 GPU 上启动计算之前，GPU 计算应用程序会指定应在 GPU 上运行哪些代码。此代码通常称为内核（第 2 章中有更多详细信息）。同时，GPU 计算应用程序的 CPU 部分还指定应运行多少个线程以及这些线程应在何处查找输入数据。要运行的内核、线程数和数据位置通过 CPU 上运行的驱动程序传达给 GPU 硬件。驱动程序将转换信息并将其放置在 GPU 可访问的内存中，GPU 已配置为查找它的位置。然后，驱动程序向 GPU 发出信号，告知它有新的计算需要运行。

现代 GPU 由许多核心组成，如图 1.2 所示。NVIDIA 将这些核心称为 *streaming multiprocessors*，AMD 将它们称为 *compute units*。每个 GPU 核心执行与已启动以在 GPU 上运行的内核相对应的单指令多线程 (SIMT) 程序。GPU 上的每个核心通常可以运行大约一千个线程。在单个核心上执行的线程可以通过暂存器进行通信并使用快速屏障操作进行同步。每个核心通常还包含第一级指令和数据缓存。它们充当带宽过滤器，以减少发送到内存系统较低级别的流量。当在第一级缓存中找不到数据时，核心上运行的大量线程用于隐藏访问内存的延迟。

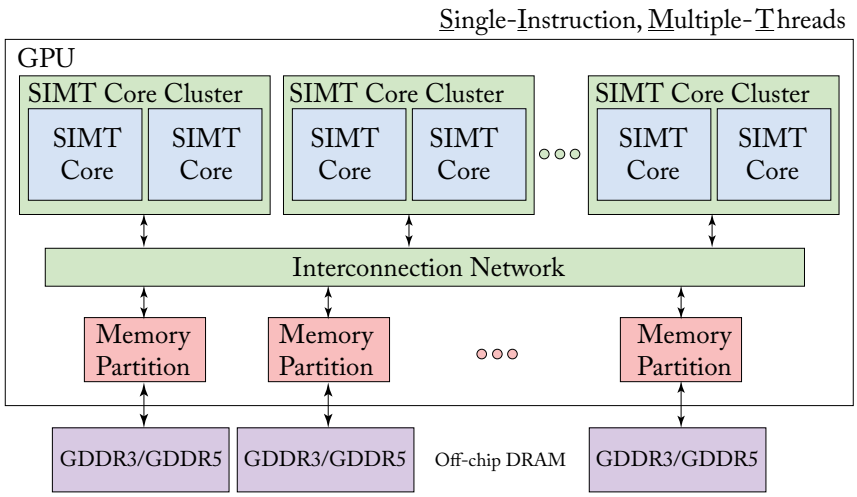


图 1.2：通用的现代 GPU 架构。

为了维持高计算吞吐量，必须在高计算吞吐量和高内存带宽之间取得平衡。这又要求内存系统中具有并行性。

tem。在 GPU 中，这种并行性是通过包含多个内存通道来实现的。通常，每个内存通道都与内存分区中的最后一级缓存的一部分相关联。GPU 内核和内存分区通过片上互连网络（如交叉开关）连接。其他组织方式也是可能的。例如，在超级计算市场上直接与 GPU 竞争的 Intel Xeon Phi 将最后一级缓存与内核一起分配。

在高度并行的工作负载上，GPU 可以通过将更大比例的芯片面积分配给算术逻辑单元并相应地减少控制逻辑面积来获得比超标量乱序 CPU 更高的单位面积性能。为了直观地了解 CPU 和 GPU 架构之间的权衡，Guz 等人 [2009] 开发了一个富有洞察力的分析模型，展示了性能如何随线程数而变化。为了使模型简单化，他们假设了一个简单的缓存模型，其中线程不共享数据和无限的片外内存带宽。图 1.3 重现了他们论文中的一张图，说明了他们在模型中发现的一个有趣的权衡。当少数线程共享一个大缓存时（多核 CPU 就是这种情况），性能会随着线程数量的增加而提高。但是，如果线程数量增加到缓存无法容纳整个工作集的程度，性能就会下降。随着线程数进一步增加，性能也随之提高，多线程能够隐藏较长的片外延迟。GPU 架构由该图的右侧表示。GPU 旨在通过采用多线程来容忍频繁的缓存未命中。

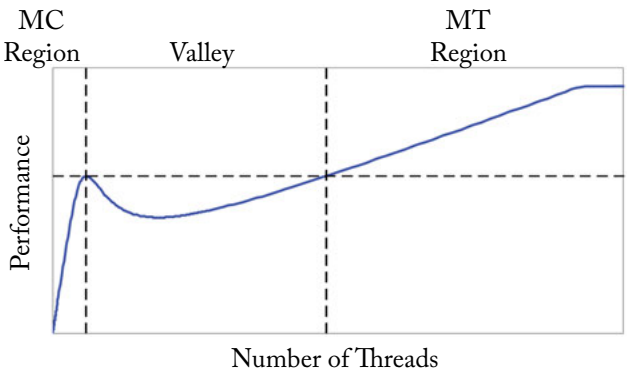


图 1.3：基于分析模型对多核 (MC) CPU 架构和多线程 (MT) 架构（如 GPU）之间的性能权衡的分析表明，如果线程数量不足以覆盖片外内存访问延迟，则可能出现“ 性能低谷 ”（基于 Guz 等人 [2009] 的图 1）。

随着 Dennard Scaling [Horowitz 等，2005] 的终结，提高能源效率已成为计算机架构研究创新的主要驱动力。一个关键的观察结果是，访问大型内存结构所消耗的能量可能与计算一样多或更多。

例如，表 1.1 提供了 45 nm 工艺技术中各种操作的能量数据 [Han et al., 2016]。在提出新颖的 GPU 架构设计时，重要的是要考虑能耗。为了帮助实现这一点，最近的 GPGPU 架构模拟器（如 GPGPU-Sim [Bakhoda et al., 2009]）采用了能量模型 [Leng et al., 2013]。

表 1.1：45 nm 工艺技术各项操作的能耗（基于 Han et al. [2016] 中的表 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 简史

本节简要介绍图形处理单元的历史。计算机图形学出现于 20 世纪 60 年代，当时出现了 Ivan Sutherland 的 Sketchpad [Sutherland, 1963] 等项目。从早期开始，计算机图形学就成为电影动画离线渲染不可或缺的一部分，同时还用于视频游戏的实时渲染开发。早期的视频卡始于 1981 年的 IBM 单色显示适配器 (MDA)，它仅支持文本。后来，视频卡引入了 2D 和 3D 加速。除了视频游戏之外，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）功能相对固定。 NVIDIA 在 2001 年推出的 GeForce 3 中以顶点着色器 [Lindholm 等人，2001] 和像素着色器的形式为 GPU 引入了可编程性。研究人员很快学会了如何使用这些早期的 GPU 通过将矩阵数据映射到纹理中并应用着色器 [Krüger 和 Westermann，2003] 来实现线性代数，而将通用计算映射到 GPU 上以使程序员不需要了解图形的学术工作也随之而来 [Buck 等人，2004]。这些努力启发了 GPU 制造商除了支持图形之外，还直接支持通用计算。第一款这样做的商业产品是 NVIDIA GeForce 8 系列。GeForce 8 系列引入了多项创新，包括从着色器和暂存器内存写入任意内存地址的能力，以限制片外带宽，而这些是早期 GPU 所缺乏的。下一个创新是使用 NVIDIA 的 Fermi 架构实现读写数据缓存。后续改进包括 AMD 的 Fusion 架构，该架构将 CPU 和 GPU 集成在同一芯片上，并实现动态并行性，从而实现

从 GPU 本身启动线程。最近，NVIDIA 的 Volta 推出了 Tensor Cores 等功能，专门用于机器学习加速。

1.4 本书大纲

本书的其余部分组织如下。

在设计硬件时，考虑硬件支持的软件非常重要。因此，在第 2 章中，我们简要概述了编程模型、代码开发过程和编译流程。

在第 3 章中，我们探讨了支持执行数千个线程的单个 GPU 核心的架构。我们逐渐建立了对支持高吞吐量和灵活编程模型所涉及的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，以帮助新进入该领域的人员快速掌握最新知识。

在第 4 章中，我们将探讨内存系统，包括 GPU 核心内的一级缓存以及内存分区的内部组织。了解 GPU 的内存系统非常重要，因为在 GPU 上运行的计算通常受到片外内存带宽的限制。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 5 章概述了有关 GPU 计算架构的额外研究，这些研究并不完全适合第 3 章或第 4 章。

CHAPTER 2

编程模型

本章的目的是提供足够的背景信息，说明如何对 GPU 进行非图形计算编程，以便那些之前没有使用过 GPU 的人能够理解后面章节的讨论。我们在这里重点介绍基本材料，将更深入的介绍留给其他参考资料（例如，[Kirk and Wen-Mei, 2016]）。有许多 GPU 计算基准套件可用于架构研究。了解 GPU 的编程方式对于对 GPU 计算感兴趣的计算机架构师来说很重要，这样可以更好地理解硬件/软件接口，但如果您想在研究中探索对硬件/软件接口进行更改，这一点就变得至关重要。在后一种情况下，现有的基准可能不存在，因此可能需要创建基准，也许可以通过修改现有 GPU 计算应用程序的源代码来创建。例如，探索在 GPU 上引入事务内存 (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 硬件来利用 GPU 应用程序中的数据级并行。GPU 计算 API（例如 CUDA 和 OpenCL）不直接向程序员公开这种 SIMD 硬件，而是采用类似 MIMD 的编程模型，允许程序员在 GPU 上启动大量标量线程。这些标量线程中的每一个都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 硬件在 SIMD 硬件上同步执行标量线程组（在 AMD 术语中称为 *warps*（或 *wavefronts*，即））），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 等，2008a，Nickolls 和 Reusch，1993]。

本章的其余部分将在此讨论的基础上展开，内容安排如下。在第 2.1 节中，我们探讨了最近的 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 计算应用程序的编译过程，并简要介绍了 GPU 指令集架构。

2.1 执行模型

GPU 计算应用程序开始在 CPU 上执行。对于独立 GPU，应用程序的 CPU 部分通常会分配用于 GPU 计算的内存，然后启动将输入数据传输到 GPU 内存的传输，最后在 GPU 上启动计算内核。对于集成 GPU，只需要最后一步。计算内核是

通常由数千个线程组成。每个线程执行相同的程序，但根据计算结果，可能遵循该程序的不同控制流。下面我们使用用 CUDA 编写的特定代码示例详细考虑此流程。在下一节中，我们将从汇编级别研究执行模型。我们的讨论并不涉及 GPU 编程模型的性能方面。然而，Seo 等人 [2011] 在 OpenCL（一种类似于 CUDA 的编程模型，可以编译为多种架构）的背景下提出了一个有趣的观察，即针对一种架构（例如 GPU）精心优化的代码可能在另一种架构（例如 CPU）上表现不佳。

图 2.1 提供了众所周知的 *single-precision scalar value A times vector value X plus vector value Y* 运算的 CPU 实现的 C 代码，即 SAXPY。SAXPY 是著名的基本线性代数软件 (BLAS) 库 [Lawson 等，1979] 的一部分，可用于实现高斯消元法等更高级的矩阵运算 [McCool 等，2012]。鉴于其简单性和实用性，它经常在教授计算机体系结构时用作示例 [Hennessy 和 Patterson，2011]。图 2.2 提供了相应的 CUDA 版本的 SAXPY，它将执行分为 CPU 和 GPU。

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 GPU 上的计算特定细节，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`。此函数将参数 `n` 中向量 `x` 和 `y` 中的元素数量、参数 `a` 中的标量值以及用于表示向量 `x` 和 `y` 的数组指针作为输入参数。该函数对数组 `x` 和 `y` 的每个元素进行迭代。在每次迭代中，第 4 行的代码使用循环变量 `i` 读取值 `x[i]` 和 `y[i]`，将 `x[i]` 乘以 `a`，然后加上 `y[i]`，最后用结果更新 `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 CUDA 版本的 SAXPY。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。我们不会逐行介绍此代码，而是首先重点介绍 GPU 执行的特定方面。

在 GPU 上执行的线程是函数指定的计算 *kernel* 的一部分。在图 2.2 所示的 CUDA 版本的 SAXPY 中，第 1 行上的 CUDA 关键字 `__global__` 表示核函数 `saxpy` 将在 GPU 上运行。在图 2.2 的示例中，我们已将图 2.1 中的“ `for` ”循环并行化。具体而言，图 2.1 中原始的仅 CPU C 代码中第 4 行上的“ `for` ”循环的每次迭代都转换为运行图 2.2 中第 3-5 行代码的单独线程。

计算内核通常由数千个线程组成，每个线程都通过运行相同的函数来启动。在我们的示例中，CPU 使用 CUDA 的内核配置语法在第 17 行开始在 GPU 上进行计算。内核配置语法看起来很像 C 中的函数调用，但包含一些附加信息来指定所包含的线程数

```
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 Harris [2012]）。

在三角括号 (<<<>>>) 之间。组成计算内核的线程被组织成一个层次结构，该层次结构由 *grid* 和 *thread blocks* 组成，后者由 *warps* 组成。在 CUDA 编程模型中，各个线程执行操作数为标量值（例如 32 位浮点数）的指令。为了提高效率，典型的 GPU 硬件会同步执行线程组。NVIDIA 将这些组称为 *warps*，AMD 将这些组称为 *wavefronts*。NVIDIA 的 Warp 由 32 个线程组成，而 AMD 的 Wavefront 由 64 个线程组成。Warp 被分组为一个更大的单元，NVIDIA 将其称为协作线程阵列 (CTA) 或线程块。第 17 行表示计算内核应启动由 `nblocks` 线程块组成的单个网格，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分发给 GPU 上正在运行的线程的每个实例。

当今许多移动设备片上系统都将 CPU 和 GPU 集成到单个芯片中，就像当今笔记本电脑和台式电脑上的处理器一样。但是，传统上，GPU 有自己的 DRAM 内存，而如今用于机器学习的数据中心内的 GPU 仍然保留着这种做法。我们注意到，NVIDIA 推出了统一内存，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。鉴于人们对机器学习的兴趣日益浓厚，并且本书的目标是了解硬件，在我们的示例中，我们考虑程序员管理单独的 GPU 和 CPU 内存的一般情况。

按照许多 NVIDIA CUDA 示例中使用的样式，我们使用前缀 `h_` 来命名在 CPU 内存中分配的内存的指针变量，使用 `d_` 来命名在 GPU 内存中分配的内存的指针。在第 13 行，CPU 调用 CUDA 库函数 `cudaMalloc`。此函数调用 GPU 驱动程序并要求它在 GPU 上分配内存供程序使用。对 `cudaMalloc` 的调用将 `d_x` 设置为指向 GPU 内存的某个区域，该区域包含足够的空间来保存 `n` 32 位浮点值。在第 15 行，CPU 调用 CUDA 库函数 `cudaMemcpy`。此函数调用 GPU 驱动程序并要求它将 `h_x` 指向的 CPU 内存中数组的内容复制到 `d_x` 指向的 GPU 内存中数组。

最后，让我们关注 GPU 上的线程执行。并行编程中采用的一种常见策略是为每个线程分配一部分数据。为了促进这一策略，GPU 上的每个线程都可以在线程块网格中查找自己的身份。在 CUDA 中执行此操作的机制采用网格、块和线程标识符。在 CUDA 中，网格和线程块具有 x、y 和 z 维度。在执行时，每个线程在网格和线程块内都有一个固定的、唯一的非负整数 x、y 和 z 坐标组合。每个线程块在网格内都有 x、y 和 z 坐标。同样，每个线程在线程块内都有 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 维度未指定，因此所有线程的 y 和 z 线程块和线程坐标都为零值。在第 3 行中，`threadIdx.x` 的值标识了线程块内线程的 x 坐标，而 `blockIdx.x`

表示线程块在其网格中的 *x* 坐标。值 `blockDim.x` 表示 *x* 维度中的最大线程数。在我们的示例中，`blockDim.x` 的计算结果为 256，因为这是第 17 行指定的值。表达式 `blockIdx.x*blockDim.x + threadIdx.x` 用于计算偏移量 `i`，以便在访问数组 `x` 和 `y` 时使用。我们将看到，使用索引 `i`，我们为每个线程分配了唯一的元素 `x` 和 `y`。

在很大程度上，编译器和硬件的组合使程序员能够忽略 warp 中线程执行的锁步特性。编译器和硬件使得 warp 中的每个线程看起来都是独立执行的。在图 2.2 中的第 4 行，我们将索引 `i` 的值与 `n`、数组 `x` 和 `y` 的大小进行比较。`i` 小于 `n` 的线程执行第 5 行。图 2.2 中的第 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 行之后将控制权返回给 CPU。在第 18 行，CPU 调用 GPU 驱动程序将 `d_y` 指向的数组从 GPU 内存复制回 CPU 内存。

SAXPY 示例未说明的 CUDA 编程模型的一些其他细节，但我们将在后面讨论，如下所示。

CTA 内的线程可以通过每个计算核心的暂存器有效地相互通信。NVIDIA 将此暂存器称为 *shared memory*。每个流式多处理器 (SM) 包含一个共享内存。共享内存中的空间由在该 SM 上运行的所有 CTA 分配。AMD 的 Graphics Core Next (GCN) 架构 [AMD, 2012] 包含一个类似的暂存器，AMD 将其称为 *local data store* (LDS)。这些暂存器很小，每个 SM 的范围为 16-64 KB，并作为不同的内存空间暴露给程序员。程序员使用源代码中的特殊关键字（例如 CUDA 中的“ `__shared__` ”）将内存分配到暂存器中。暂存器充当软件控制的缓存。虽然 GPU 也包含硬件管理的缓存，但通过此类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经常以可预测的方式重复使用的数据时，应用程序可以从使用暂存器中受益。与 NVIDIA 的 GPU 不同，AMD 的 GCN GPU 还包括一个由 GPU 上的所有核心共享的 *global data store* (GDS) 暂存器。暂存器用于图形应用程序在不同的图形着色器之间传递结果。例如，LDS 用于在 GCN 中的顶点和像素着色器之间传递参数值 [AMD，2012]。

CTA 中的线程可以使用硬件支持的屏障指令高效同步。不同 CTA 中的线程可以通信，但需要通过所有线程都可以访问的全局地址空间进行通信。访问此全局地址空间通常比访问共享内存更耗时，无论是在时间上还是在能源上。

NVIDIA 在 Kepler 一代 GPU 中引入了 CUDA 动态并行 (CDP) [NVIDIA Corporation, 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 GPU 上运行的线程之间的负载不平衡，从而导致

GPU 硬件未得到充分利用。在许多方面，其动机与动态扭曲形成 (DWF) [Fung et al., 2007] 以及第 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，我们简要讨论了计算内核从 CUDA 和 OpenCL 等高级语言到 GPU 硬件执行的汇编语言的转换，以及当前 GPU 指令集的形式。GPU 架构与 CPU 架构略有不同的一个有趣方面是 GPU 生态系统的发展方式，以支持指令集的发展。例如，x86 微处理器向后兼容 1976 年发布的 Intel 8086。向后兼容性意味着为上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，40 年前为 Intel 8086 编译的软件理论上可以在当今的任何 x86 处理器上运行。

2.2.1 NVIDIA GPU 指令集架构

鉴于提供 GPU 硬件的供应商数量众多（每家都有自己的硬件设计），随着早期 GPU 变得可编程，通过 OpenGL 着色语言 (OGSL) 和 Microsoft 的高级着色语言 (HLSL) 进行一定程度的指令集虚拟化变得十分普遍。当 NVIDIA 于 2007 年初推出 CUDA 时，他们决定走一条类似的道路，并推出了自己的用于 GPU 计算的高级虚拟指令集架构，称为并行线程执行 ISA 或 PTX [NVI，2017]。NVIDIA 会在每次发布 CUDA 时完整记录此虚拟指令集架构，以至于本书的作者可以轻松开发 GPGPU-Sim 模拟器来支持 PTX [Bakhoda et al.，2009]。PTX 在许多方面类似于标准精简指令集计算机 (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。其中一个例子是使用一组无限的虚拟寄存器。图 2.3 展示了图 2.2 中的 SAXPY 程序的 PTX 版本。

在 GPU 上运行 PTX 代码之前，需要将 PTX 编译为硬件支持的实际指令集架构。NVIDIA 将此级别称为 SASS，是“ Streaming ASSembler ”的缩写 [Cabral, 2016]。从 PTX 到 SASS 的转换过程可以通过 GPU 驱动程序或 NVIDIA CUDA 工具包提供的独立程序 `ptxas` 来完成。NVIDIA 没有完整记录 SASS。虽然这使得学术研究人员更难开发出捕捉所有编译器优化效果的架构模拟器，但它使 NVIDIA 摆脱了客户需求，即在硬件级别提供向后兼容性，从而能够从一代到下一代完全重新设计指令集架构。不可避免的是，希望从低层次了解性能的开发人员开始创建自己的工具来反汇编 SASS。第一个此类成果由 Wladimir Jasper van der Laan 完成，名为“ decuda ”[van der Lann]，于 2007 年底为 NVIDIA 的 GeForce 8 系列 (G80) 推出，距首次发布支持 CUDA 的硬件不到一年时间。

```
.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：图 2.2 中计算内核对应的 PTX 代码（使用 CUDA 8.0 编译）。

decuda 项目对 SASS 指令集有了足够详细的了解，因此可以开发汇编程序。这有助于在 GPGPU-Sim 3.2.2 [Tor M. Aamodt 等] 中开发对 NVIDIA GT200 架构以下 SASS 的支持。NVIDIA 最终推出了一款名为 `cuobjdump` 的工具，并开始部分记录 SASS。NVIDIA 的 SASS 文档 [NVIDIA Corporation, c] 目前（2018 年 4 月）仅提供汇编操作码名称列表，但没有提供操作数格式或 SASS 指令语义的详细信息。最近，随着 GPU 在机器学习中的使用呈爆炸式增长以及对性能优化代码的需求，其他人为后续架构（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 展示了针对 NVIDIA 的 Fermi 架构 [NVI, 2009] 编译并使用 NVIDIA 的 CUDA Toolkit) 的 `cuobjdump` (part 提取的 SAXPY 内核的 SASS 代码。图 2.4 中的第一列是指令的地址。第二列是汇编代码，第三列是编码指令。如上所述，NVIDIA 仅部分记录了其硬件汇编代码。比较图 2.3 和图 2.4，可以注意到虚拟和硬件 ISA 级别之间的相似之处和不同之处。在高层次上，它们具有重要的相似之处，例如两者都是 RISC（都使用加载和存储来访问内存）并且都使用谓词 [Allen et al., 1983]。更细微的差别包括：（1）PTX 版本具有一组本质上无限的可用寄存器，因此每个定义通常使用一个新寄存器，很像静态单一分配 [Cytron et al., 1991]，而 SASS 使用一组有限的寄存器；（2）内核参数通过分组常量内存传递，可通过 SASS 中的非加载/存储指令访问，而 PTX 中的参数被分配到自己单独的“参数”地址空间中。

图 2.5 展示了由同一版本的 CUDA 为 NVIDIA 的 Pascal 架构生成并使用 NVIDIA 的 `cuobjdump` 提取的 SAXPY 的 SASS 代码。将图 2.5 与图 2.4 进行比较，很明显 NVIDIA 的 ISA 发生了显著变化，包括指令编码方面。图 2.5 包含一些没有反汇编指令的行（例如，第 3 行的地址 0x0000）。这些是 NVIDIA Kepler 架构中引入的特殊“控制指令”，以消除使用记分板进行显式依赖性检查的需要 [NVIDIA Corporation，b]。Lai 和 Seznec [2013] 探索了 Kepler 架构的控制指令编码。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 [Alverson et al., 1990] 上的显式依赖前瞻。Gray 描述了他们能够推断出的 NVIDIA 的 Maxwell 架构的控制指令编码的大量细节。据 Gray 说，Maxwell 中每三条常规指令就有一条控制指令。NVIDIA 的 Pascal 架构似乎也是如此，如图 2.5 所示。据 Gray 说，Maxwell 上的 64 位控制指令包含三组 21 位，为以下三条指令中的每一条编码以下信息：停顿计数；yield 提示标志；以及写入、读取和等待依赖屏障。Gray 还描述了常规指令上寄存器重用标志的使用，这也可以在图 2.5 中看到（例如，用于第一个源的 `R0.reuse`

```
1  Address     Dissassembly                                    Encoded Instruction
2  ========    ============================================    ======================
3  /*0000*/         MOV R1, c[0x1][0x100];                     /* 0x2800440400005de4 */
4  /*0008*/         S2R R0, SR_CTAID.X;                        /* 0x2c00000094001c04 */
5  /*0010*/         S2R R2, SR_TID.X;                          /* 0x2c00000084009c04 */
6  /*0018*/         IMAD R0, R0, c[0x0][0x8], R2;              /* 0x2004400020001ca3 */
7  /*0020*/         ISETP.GE.AND P0, PT, R0, c[0x0][0x20], PT; /* 0x1b0e40008001dc23 */
8  /*0028*/     @P0 BRA.U 0x78;                                /* 0x40000001200081e7 */
9  /*0030*/    @!P0 MOV32I R5, 0x4;                            /* 0x18000000100161e2 */
10 /*0038*/    @!P0 IMAD R2.CC, R0, R5, c[0x0][0x28];          /* 0x200b8000a000a0a3 */
11 /*0040*/    @!P0 IMAD.HI.X R3, R0, R5, c[0x0][0x2c];        /* 0x208a8000b000e0e3 */
12 /*0048*/    @!P0 IMAD R4.CC, R0, R5, c[0x0][0x30];          /* 0x200b8000c00120a3 */
13 /*0050*/    @!P0 LD.E R2, [R2];                             /* 0x840000000020a085 */
14 /*0058*/    @!P0 IMAD.HI.X R5, R0, R5, c[0x0][0x34];        /* 0x208a8000d00160e3 */
15 /*0060*/    @!P0 LD.E R0, [R4];                             /* 0x8400000000402085 */
16 /*0068*/    @!P0 FFMA R0, R2, c[0x0][0x24], R0;             /* 0x3000400090202000 */
17 /*0070*/    @!P0 ST.E [R4], R0;                             /* 0x9400000000402085 */
18 /*0078*/         EXIT;                                      /* 0x8000000000001de7 */
```

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

操作数在整数短乘法加法指令 XMAD 的第 7 行中）。这似乎表明从 Maxwell 开始，NVIDIA GPU 中就添加了“ 操作数重用缓存 ”（请参阅第 3.6.1 节中的相关研究）。此操作数重用缓存似乎允许在每次主寄存器文件访问时多次读取寄存器值，从而降低能耗和/或提高性能。

2.2.2 AMD 图形核心 Next 指令集架构

与 NVIDIA 不同的是，AMD 在推出其 Southern Islands 架构时发布了完整的硬件级 ISA 规范 [AMD，2012]。Southern Islands 是 AMD 的 Graphics Core Next (GCN) 架构的第一代产品。AMD 硬件 ISA 文档的可用性帮助学术研究人员开发了在较低级别工作的模拟器 [Ubal et al.，2012]。AMD 的编译流程还包括一个虚拟指令集架构，称为 HSAIL，是异构系统架构 (HSA) 的一部分。

AMD 的 GCN 架构与 NVIDIA GPU（包括 NVIDIA 最新的 Volta 架构 [NVIDIA Corp.，2017]）之间的一个关键区别是标量和矢量指令是分开的。图 2.6 和 2.7 重现了 AMD [2012] 的高级 OpenCL（类似于 CUDA）代码示例以及 AMD South-

```
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 中的计算内核相对应的低级 SASS 代码（使用 CUDA 8.0 为 NVIDIA Pascal 架构 sm_60 编译）。

ern Islands 架构。在图 2.7 中，标量指令以 s_ 开头，矢量指令以 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矢量单元。矢量指令在矢量单元上执行，并为波前中的每个单独线程计算不同的 32 位值。相反，在标量单元上执行的标量指令计算波前中所有线程共享的单个 32 位值。在图 2.7 所示的示例中，标量指令与控制流处理有关。具体而言，exec 是一个特殊寄存器，用于预测 SIMT 执行的各个矢量通道的执行。第 3.1.1 节更详细地描述了在 GPU 上使用掩码进行控制流处理的更多细节。 GCN 架构中标量单元的另一个潜在好处是，SIMT 程序中计算的某些部分通常将计算出与线程 ID 无关的相同结果（参见第 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：Southern Islands（下一个图形核心）微码（基于 AMD [2012] 中的图 2.2）。

AMD 的 GCN 硬件指令集手册 [AMD，2012] 提供了许多有关 AMD GPU 硬件的有趣见解。例如，为了实现长延迟操作的数据依赖性解析，AMD 的 GCN 架构包含 `S_WAITCNT` 指令。每个波前都有三个计数器：矢量内存计数、本地/全局数据存储计数和寄存器导出计数。每个计数器都指示给定类型的未完成操作的数量。编译器或程序员插入 `S_WAITCNT` 指令，让波前等待，直到未完成操作的数量降至指定阈值以下。

CHAPTER 3

SIMT 核心：指令和寄存器数据流

在本章和下一章中，我们将研究现代 GPU 的架构和微架构。我们将 GPU 架构的讨论分为两部分：(1) 本章中研究实现计算的 SIMT 核心，然后 (2) 下一章中研究内存系统。

在传统的图形渲染角色中，GPU 会访问详细纹理贴图等数据集，而这些数据集太大以致无法完全缓存在片上。为了实现高性能可编程性（这在图形处理中很有必要），有必要采用一种能够维持大量片外带宽的架构，这在图形处理中很有必要，因为随着图形模式数量的增加，这既可以降低验证成本，也可以使游戏开发人员更轻松地区分他们的产品 [Lindholm et al., 2001]。因此，当今的 GPU 会同时执行数万个线程。虽然每个线程的片上内存存储空间很小，但缓存仍可有效减少大量片外内存访问。例如，在图形工作负载中，相邻像素操作之间存在明显的空间局部性，这些局部性可以被片上缓存捕获。

图 3.1 说明了本章讨论的 GPU 流水线的微架构。该图说明了图 1.2 中所示的单个 SIMT 核心的内部组织。流水线可分为 SIMT 前端和 SIMD 后端。流水线由三个调度“循环”组成，它们在单个流水线中共同作用：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。在本章的其余部分中，我们将通过考虑依赖于每个循环的架构的关键方面来帮助您全面了解此图中的各个块。

由于要完全理解这个组织涉及许多细节，我们将讨论分为几个部分。我们按顺序进行讨论，目的是开发越来越详细的核心微体系结构视图。我们从整个 GPU 管道的高级视图开始，然后填写详细信息。我们将这些越来越准确的描述称为“近似值”，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 的中心组织原则是多线程，我们将这些“近似值”组织起来

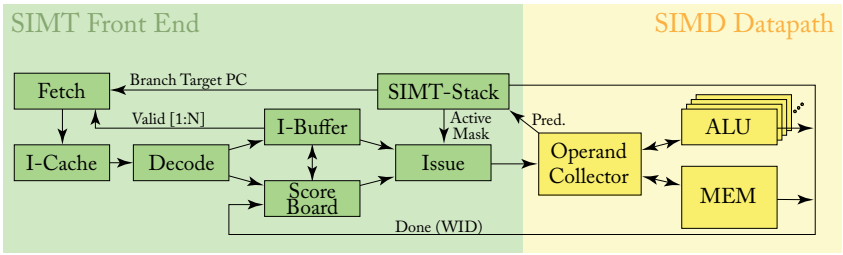


图 3.1：通用 GPGPU 核心的微架构。

围绕上述三个调度循环。我们发现，通过考虑三个越来越精确的“ 近似循环 ”来组织本章会很方便，这些循环会逐步考虑到这些调度程序循环的细节。

3.1 单圈近似

我们首先考虑具有单个调度程序的 GPU。这种对硬件的简化看法与人们仅阅读 CUDA 编程手册中的硬件描述时对硬件的预期并无不同。

为了提高效率，NVIDIA 将线程组织成“ warp ”组，AMD 将其称为“ wavefront ”。因此，调度的单位是 warp。在每个周期中，硬件选择一个 warp 进行调度。在一次循环近似中，warp 的程序计数器用于访问指令内存，以查找 warp 要执行的下一条指令。获取指令后，将解码该指令并从寄存器文件中获取源操作数寄存器。在从寄存器文件中获取源操作数的同时，确定 SIMT 执行掩码值。以下小节介绍了如何确定 SIMT 执行掩码值，并将它们与现代 GPU 中也采用的预测进行了对比。

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。如果设置了 SIMT 执行掩码，则每个线程都在与通道关联的功能单元上执行。与现代 CPU 设计一样，功能单元通常是异构的，这意味着给定的功能单元仅支持指令子集。例如，NVIDIA GPU 包含 *special function unit* (SFU)、*load/store unit*、*floating-point function unit*、*integer function unit*，以及 Volta 中的 *Tensor Core*。

所有功能单元名义上都包含与 warp 中的线程数一样多的通道。但是，一些 GPU 使用了不同的实现方式，即在几个时钟周期内执行单个 warp 或波前。这是通过以更高的频率对功能单元进行计时来实现的，这可以实现更高的单位面积性能，但代价是增加能耗。实现功能单元更高时钟频率的一种方法是将其执行流水线化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（而非性能）的角度来看，它为程序员提供了各个线程完全独立执行的抽象。这种编程模型可能仅通过预测就能实现。然而，在当前的 GPU 中，它是通过传统预测与我们将称为 *SIMT stack* 的谓词掩码堆栈的组合来实现的。

SIMT 堆栈有助于有效处理所有线程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，一个分支的控制依赖于另一个分支。第二个问题是完全跳过计算，而 warp 中的所有线程都避开控制流路径。对于复杂的控制流，这可以节省大量成本。传统上，支持谓词的 CPU 通过使用多个谓词寄存器来处理嵌套控制流，文献中提出了支持跨通道谓词测试。

GPU 使用的 SIMT 堆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 堆栈至少部分由专用于此目的的特殊指令管理。相反，我们将描述学术著作中引入的稍微简化的版本，该版本假设硬件负责管理 SIMT 堆栈。

为了描述 SIMT 堆栈，我们使用了一个示例。图 3.2 说明了包含嵌套在 do-while 循环中的两个分支的 CUDA C 代码，图 3.3 说明了相应的 PTX 程序集。图 3.4 重现了 Fung 等人的图 5。[Fung et al., 2007]，说明了此代码如何与 SIMT 堆栈交互，假设 GPU 每个 warp 有四个线程。

图 3.4a 给出了与图 3.2 和 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 至第 6 行以及图 3.3 中第 1 至第 6 行的代码。在执行图 3.3 中第 6 行的分支之后，这四个线程遵循不同的（发散的）控制流，这对应于图 3.2 中第 6 行的“ if ”语句。具体而言，如图 3.4a 中的标签“ `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 中的标签“ `F/0001` ”所示，执行分支后，第四个线程跳转到基本块 F，对应图 3.3 中的第 14 行（图 3.2 中的第 14 行）。

类似地，当在基本块 B 中执行的三个线程到达图 3.3 中第 9 行的分支时，第一个线程会分叉到基本块 C，而第二个和第三个线程会分叉到基本块 D。然后，所有三个线程都会到达基本块 E 并一起执行，如图 3.4a 中的标签“ `E/1110` ”所示。在基本块 G 中，所有四个线程都会一起执行。

GPU 硬件如何使 warp 中的线程能够遵循代码中的不同路径，同时采用每个周期只允许执行一条指令的 SIMD 数据路径？

```
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：用于说明 SIMT 堆栈操作的示例 CUDA C 源代码。

```
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：用于说明 SIMT 堆栈操作的示例 PTX 汇编代码。

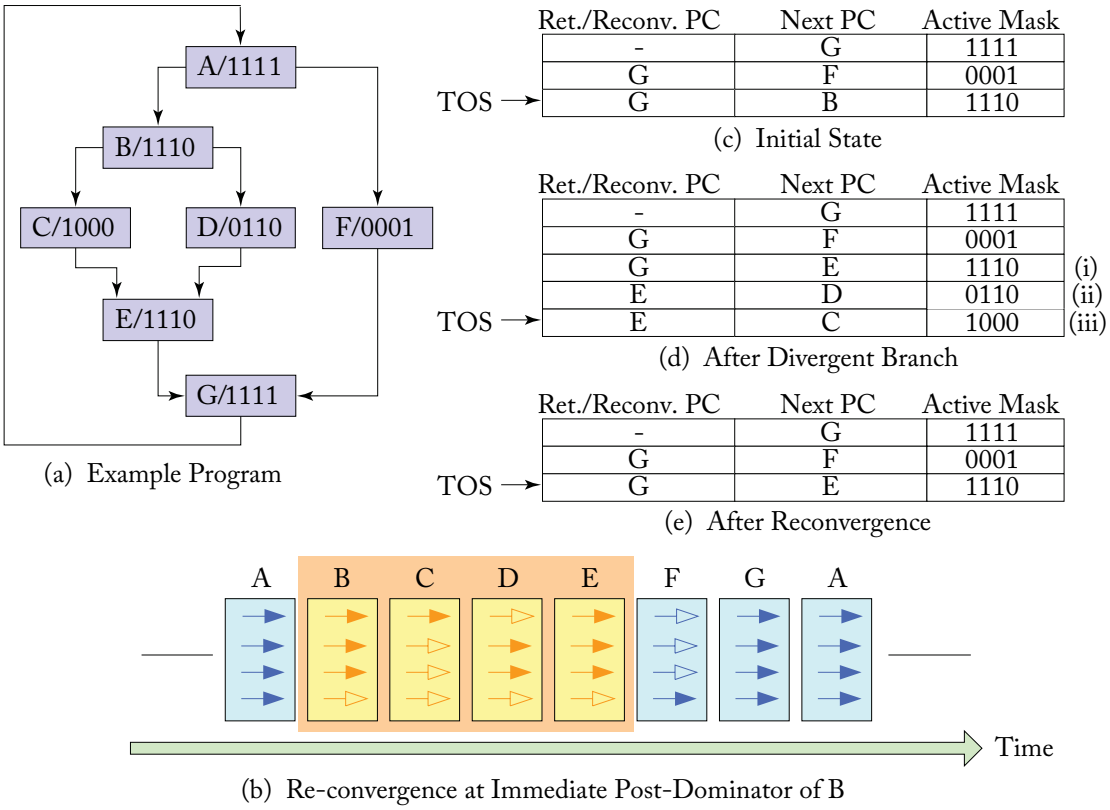


图 3.4：SIMT 堆栈操作示例（基于 Fung 等人 [2007] 的图 5）。

当前 GPU 中使用的方法是序列化在给定 warp 内遵循不同路径的线程的执行。这在图 3.4b 中进行了说明，其中箭头代表线程。实心箭头表示线程正在执行相应基本块中的代码（由每个矩形顶部的字母表示）。空心箭头表示线程已被屏蔽。时间在图中向右推进，如下方箭头所示。最初，每个线程都在基本块 B 中执行。然后，在分支之后，前三个线程执行基本块 B 中的代码。请注意，此时线程四已被屏蔽。为了保持 SIMD 执行，第四个线程在不同时间（在此示例中为几个周期后）通过基本块 F 执行备用代码路径。

为了实现这种不同代码路径的序列化，一种方法是使用如图 3.4c-e 所示的堆栈。此堆栈上的每个条目包含三个条目：一个重新收敛程序计数器 (RPC)、下一个要执行的指令的地址 (Next PC) 和一个活动掩码。

图 3.4c 显示了 Warp 执行完图 3.3 中第 6 行的分支后堆栈的状态。由于三个线程分支到基本块 B，一个线程分支到基本块 F，因此堆栈顶部 (TOS) 中添加了两个新条目。Warp 执行的下一条指令由堆栈顶部 (TOS) 条目中的下一个 PC 值确定。在图 3.4c 中，这个下一个 PC 值为 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 Warp 中只有前三个线程应该执行此指令。

线程束中的前三个线程继续执行基本块 B 中的指令，直到它们到达图 3.3 中第 9 行的分支。执行此分支后，它们会发散，如前所述。此分支发散会导致堆栈发生三处变化。首先，执行分支之前的 TOS 条目的下一个 PC 条目（图 3.4d 中标记为 (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，添加两个条目（图 3.4d 中标记为 (ii) 和 (iii)），每个条目对应于执行分支后线程束中的每条路径。

重新收敛点是程序中可以强制分叉的线程继续以锁步方式执行的位置。通常首选最近的重新收敛点。在给定程序执行中，编译时可以保证分叉的线程可以再次以锁步方式执行的最早点是导致分支分叉的分支的直接后支配者。在运行时，有时可以在程序的较早点重新收敛 [Coon 和 Lindholm，2008 年，Diamos 等，2011 年，Fung 和 Aamodt，2011 年]。

一个有趣的问题是“ 应该使用什么顺序在发散分支之后将条目添加到堆栈中？ ”为了将重新收敛堆栈的最大深度降低到与 Warp 中的线程数成对数关系，最好先将活动线程最多的条目放在堆栈上，然后再将活动线程较少的条目放在堆栈上 [AMD，2012]。在图 3.4 的部分 (d) 中，我们遵循此顺序，而在部分 (c) 中，我们使用了相反的顺序。

3.1.2 SIMT 死锁和无堆栈 SIMT 架构

最近，NVIDIA 披露了即将推出的 Volta GPU 架构的细节 [NVIDIA Corp.，2017]。他们强调的一项变化是发散下的掩蔽行为及其与同步的交互方式。基于堆栈的 SIMT 实现可能导致死锁情况，ElTantawy 和 Aamodt [2016] 将其称为“ SIMT 死锁 ”。学术工作描述了用于 SIMT 执行的替代硬件 [ElTantaway 等，2014]，经过微小更改 [ElTantawy 和 Aamodt，2016] 可以避免 SIMT 死锁。NVIDIA 将他们的新线程发散管理方法称为独立线程调度。独立线程调度的描述表明它们实现了与上述学术提案类似的行为。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 对独立线程调度的描述一致，并在最近的 NVIDIA 专利申请 [Diamos et al., 2015] 中进行了披露。

图 3.5 的左侧部分给出了一个说明 SIMT 死锁问题的 CUDA 示例，中间部分显示了相应的控制流图。行 A 将共享变量 mutex 初始化为零，以指示锁是空闲的。在行 B 上，warp 中的每个线程执行 `atomicCAS` 操作，该操作对包含 mutex 的内存位置执行比较和交换操作。`atomicCAS` 操作是编译器内在函数，它被转换为 `atom.global.cas` PTX 指令。从逻辑上讲，比较和交换首先读取 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，则比较和交换操作会将 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都以原子方式执行上述一系列逻辑操作。因此，同一 warp 内不同线程对任何单个位置的多次访问（`atomicCAS`）都是串行的。由于图 3.5 中的所有线程都访问相同的内存位置，因此只有一个线程会看到互斥锁的值为 0，其余线程会看到该值为 1。接下来，在考虑 SIMT 堆栈的同时，考虑在`atomicCAS` 返回后，B 行上的`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，一个线程想要退出循环，而其余线程想要留在循环中。退出循环的线程将到达重新收敛点，因此将不再在 SIMT 堆栈上处于活动状态，因此无法执行`atomicExch` 操作来释放 C 行上的锁。留在循环中的线程将在 SIMT 堆栈的顶部处于活动状态，并将无限期地旋转。线程之间产生的循环依赖引入了一种新形式的死锁，ElTantawy 和 Aamodt [2016] 将其称为 SIMT 死锁，如果线程在 MIMD 架构上执行，这种死锁就不会出现。

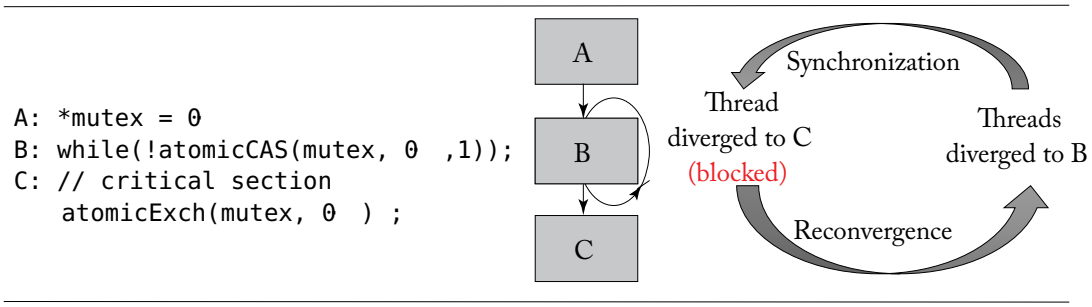


图 3.5：SIMT 死锁示例（基于 ElTantawy 和 Aamodt [2016] 中的图 1）。

接下来，我们总结了一种无堆栈分支重收敛机制，类似于 NVIDIA 最近的美国专利申请 [Diamos et al., 2015]。该机制与 NVIDIA 迄今为止对 Volta 重收敛处理机制的描述一致 [Nvidia, 2017]。关键思想是用每个 warp 收敛屏障取代堆栈。图 3.6 显示了 NVIDIA 专利申请中所述的每个 warp 维护的各种字段，图 3.8

提供了一个相应的示例来说明收敛屏障的操作。实际上，该提案提供了多路径 IPDOM 的替代实现 [ElTantaway 等，2014]，这将在第 3.4.2 节与早期的学术著作一起描述。收敛屏障机制与 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屏障机制，我们考虑在图 3.8 中的代码上执行单个 warp，它显示了由图 3.7 中所示的 CUDA 代码产生的控制流图。

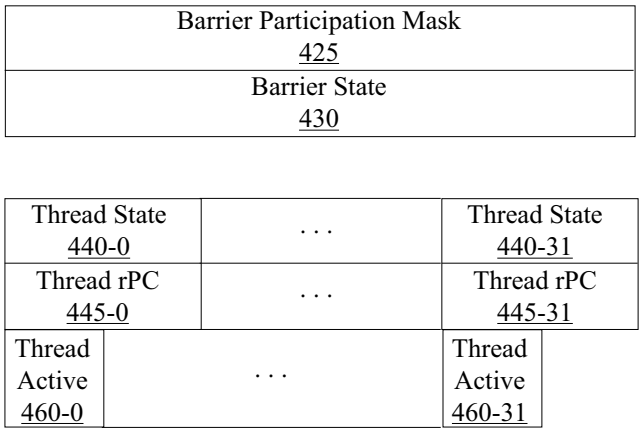


图 3.6：NVIDIA 最近描述的替代无堆栈收敛屏障的基于分支发散处理机制（基于 Diamos 等人 [2015] 的图 4B）。

```
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））。

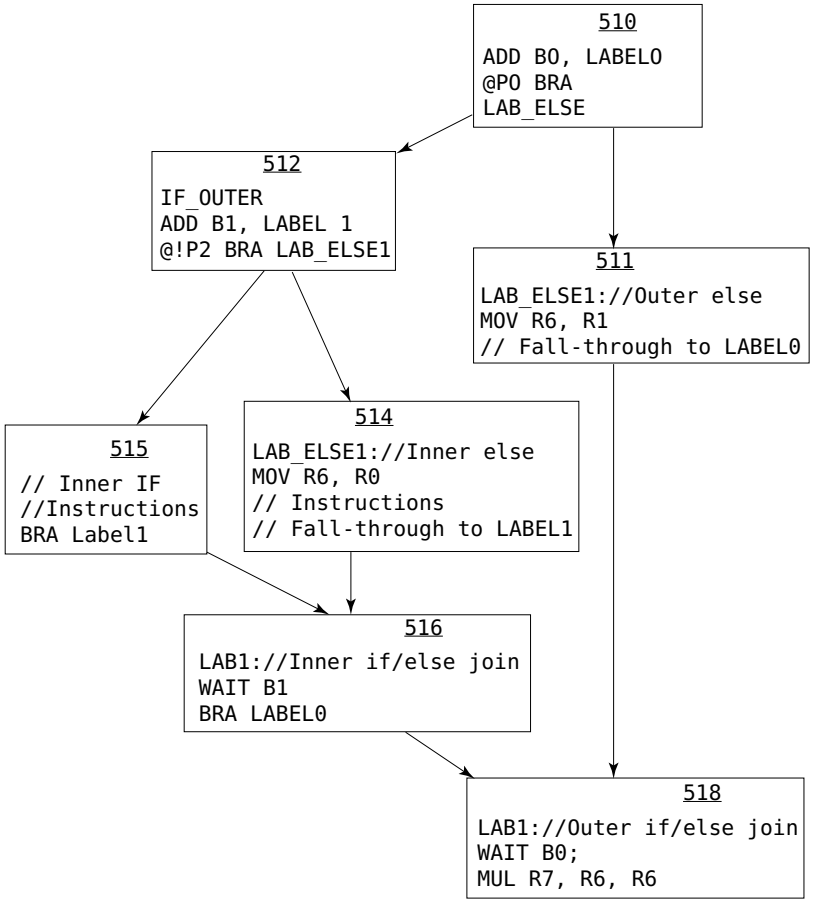


图 3.8：NVIDIA 最近描述的收敛屏障分支发散处理机制的代码示例（基于 Diamos 等人 [2015] 的图 5B）。

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，由硬件 Warp 调度程序使用。每个 *Barrier Participation Mask* 用于跟踪给定 Warp 中的哪些线程参与给定的收敛屏障。给定 Warp 可能有多个屏障参与掩码。在常见情况下，给定屏障参与掩码跟踪的线程将等待彼此在分叉分支后到达程序中的公共点，从而重新收敛在一起。为了支持这一点，*Barrier State* 字段用于跟踪哪些线程已到达给定的收敛屏障。*Thread State* 跟踪 Warp 中每个线程是否已准备好执行、是否在收敛屏障处受阻（如果是，则跟踪哪个屏障）或是否已让步。似乎让步状态可用于使 Warp 中的其他线程在以下情况下越过收敛屏障向前推进：

否则会导致 SIMT 死锁。Thread rPC 字段跟踪每个非活动线程的下一条要执行的指令的地址。Thread Active 字段是一个位，用于指示 warp 中的相应线程是否处于活动状态。

假设一个 Warp 包含 32 个线程，屏障参与掩码为 32 位宽。如果设置了某个位，则表示 Warp 中的相应线程参与此收敛屏障。线程在执行分支指令（例如图 3.8 中基本块 510 和 512 末尾的指令）时会发散。这些分支对应于图 3.7 中的两个“ if ”语句。屏障参与掩码由 Warp 调度程序用于将线程停止在特定的收敛屏障位置，该位置可以是分支的直接后支配者或其他位置。在任何给定时间，每个 Warp 可能需要多个屏障参与掩码来支持嵌套控制流构造（例如图 3.7 中的嵌套 if 语句）。图 3.6 中的寄存器可以使用通用寄存器或专用寄存器或两者的某种组合来实现（专利申请未说明）。鉴于屏障参与掩码只有 32 位宽，如果每个线程都拥有屏障参与掩码的副本，那么这将是多余的，就像单纯使用通用寄存器文件来存储它一样。但是，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，这使得掩码的软件管理成为可取之道。

为了初始化收敛屏障参与掩码，需要使用特殊的“ ADD ”指令。在 Warp 执行此 ADD 指令时，所有处于活动状态的线程都会在 ADD 指令指示的收敛屏障中设置其位。执行分支后，某些线程可能会发散，这意味着要执行的下一条指令（即 PC）的地址将有所不同。发生这种情况时，调度程序将选择具有共同 PC 的线程子集并更新线程活动字段以启用 Warp 中这些线程的执行。学术提案将这样的线程子集称为“ Warp 分裂 ”[ElTantaway 等，2014 年，ElTantawy 和 Aamodt，2016 年，Meng 等，2010 年]。与基于堆栈的 SIMT 实现相比，使用收敛屏障实现，调度程序可以自由地在发散线程组之间切换。当某些线程已获得锁而其他线程尚未获得锁时，这可以使 Warp 中的线程之间向前推进。

“ WAIT ”指令用于在 Warp Split 达到收敛屏障时停止 Warp Split。如 NVIDIA 专利申请中所述，WAIT 指令包含一个操作数，用于指示收敛屏障的身份。WAIT 指令的作用是将 Warp Split 中的线程添加到屏障的屏障状态寄存器中，并将线程的状态更改为阻塞。一旦屏障参与掩码中的所有线程都执行了相应的 WAIT 指令，线程调度程序就可以将所有线程从原始 Warp Split 切换到活动状态，同时保持 SIMD 效率。图 3.8 中的示例有两个收敛屏障，B1 和 B2，基本块 516 和 518 中有 WAIT 指令。为了实现 Warp Split 之间的切换，NVIDIA 描述了使用 YIELD 指令以及其他细节，例如对间接分支的支持，我们在此讨论中省略了这些细节 [Diamos et al., 2015]。

图 3.9 显示了基于堆栈的重新收敛的时序示例，图 3.10 说明了使用 NVIDIA Volta 白皮书中所述的独立线程调度的潜在时序。在图 3.10 中，我们可以看到 Volta 将语句 A 和 B 与语句 X 和 Y 交错，这与图 3.9 中的行为形成对比。此行为与上面描述的收敛屏障机制一致（以及多路径 IPDOM [ElTantaway 等，2014]）。最后，图 3.11 说明了无堆栈架构如何执行图 3.5 中的自旋外观代码以避免 SIMT 死锁。

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```

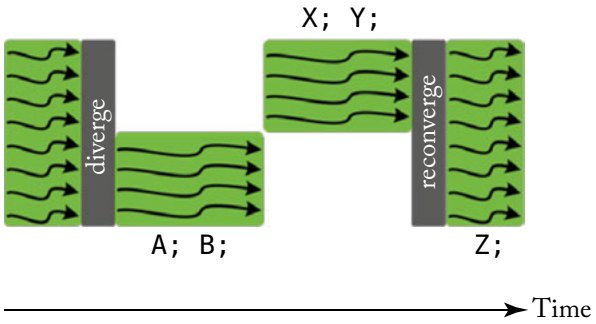


图 3.9：基于堆栈的重新收敛行为的示例（基于 Nvidia [2017] 的图 20）。

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
__syncwarp()
```

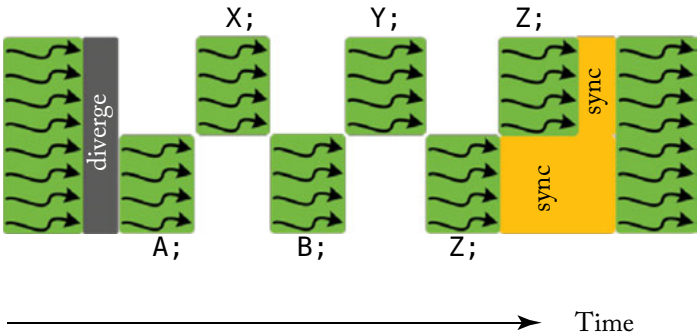


图 3.10：展示 Volta 重新收敛行为的示例（基于 Nvidia [2017] 的图 23）。

3.1.3 WARP 调度

GPU 主机中的每个核心都包含许多 Warp。一个非常有趣的问题是这些 Warp 应该按什么顺序进行调度。为了简化讨论，我们假设每个 Warp 在调度时只发出一条指令，而且 Warp 不符合条件

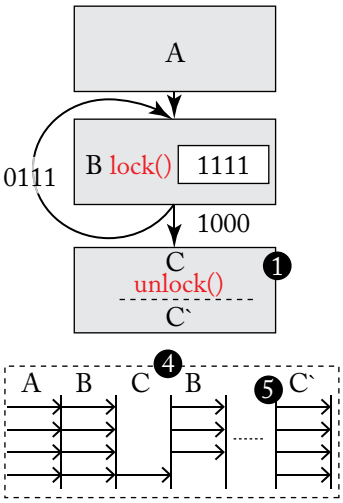


图 3.11：展示与图 3.5 中的自旋锁代码上的收敛屏障类似的学术机制的行为的示例（基于 ElTantawy 和 Aamodt [2016] 中的图 6（a））。

发出另一条指令，直到第一条指令执行完毕。我们将在本章后面重新讨论这一假设。

如果内存系统是“ 理想的 ”，能以某个固定的延迟响应内存请求，那么理论上可以设计内核来支持足够多的 warp，从而使用细粒度多线程来隐藏这种延迟。在这种情况下，可以说我们可以通过“ 循环 ”顺序调度 warp 来减少给定吞吐量的芯片面积。在循环中，warp 被赋予某种固定的顺序，例如按线程标识符递增的顺序排列，并且调度程序按照此顺序选择 warp。此调度顺序的一个特性是，它允许每个发出的指令在大致相同的时间内完成执行。如果内核中的 warp 数乘以每个 warp 的发出时间超过了内存延迟，那么内核中的执行单元将始终保持繁忙。因此，将 warp 数增加到这个程度原则上可以增加每个内核的吞吐量。

但是，有一个重要的权衡：要使不同的 Warp 能够在每个周期发出指令，每个线程必须有自己的寄存器（这样就无需在寄存器和内存之间复制和恢复寄存器状态）。因此，增加每个核心的 Warp 数量会增加用于寄存器文件存储的芯片面积相对于专用于执行单元的面积的比例。对于固定的芯片面积，增加每个核心的 Warp 数量将减少每个芯片的核心总数。

实际上，内存的响应延迟取决于应用程序的局部性属性以及片外内存访问所遇到的争用量。

在考虑 GPU 的内存系统时，调度会产生什么影响？这在过去几年中一直是大量研究的主题，在将有关内存系统的更多细节添加到我们的 GPU 微架构模型后，我们将回到这个问题。然而，简而言之，局部性可以支持或阻碍循环调度：当不同的线程在其执行的相似点共享数据时，例如在访问图形像素着色器中的纹理贴图时，线程取得同等进展是有益的，因为这可以增加在片上缓存中“命中”的内存引用数量，这是循环调度所鼓励的 [Lindholm 等人，2015]。同样，当在时间上就近访问地址空间中的邻近位置时，访问 DRAM 会更有效率，这也是循环调度所鼓励的 [Narasiman 等人，2011]。另一方面，当线程主要访问不相交的数据时（这在更复杂的数据结构中往往会发生），重复调度给定的线程以最大化局部性是有益的 [Rogers et al., 2012]。

3.2 双环路近似

为了减少每个核心必须支持的 Warp 数量以隐藏较长的执行延迟，能够在先前的指令尚未完成时从 Warp 发出后续指令会很有帮助。但是，前面描述的单循环微架构阻止了这一点，因为该设计中的调度逻辑只能访问线程标识符和要发出的下一个指令的地址。具体来说，它不知道 Warp 发出的下一个指令是否依赖于尚未完成执行的先前指令。要提供此类依赖关系信息，必须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 实现了指令缓冲区，指令放在缓存访问之后。使用单独的调度程序来决定指令缓冲区中的哪些指令应该在其余管道之后发出。

指令存储器实现为第一级指令缓存，由一个或多个级别的二级（通常是统一的）缓存支持。指令缓冲区还可以与指令未命中状态保持寄存器 (MSHR) 结合使用，帮助隐藏指令缓存未命中延迟 [Kroft，1981]。缓存命中或缓存未命中填充后，指令信息将放入指令缓冲区。指令缓冲区的组织形式多种多样。一种特别直接的方法是每个 warp 存储一个或多个指令。

接下来，让我们考虑如何检测同一 warp 中指令之间的数据依赖关系。在传统 CPU 架构中，检测指令之间的依赖关系有两种传统方法：记分板和保留站。保留站用于消除名称依赖关系，并引入了对关联逻辑的需求，而关联逻辑在面积和能量方面都很昂贵。记分板可以设计为支持按序执行或无序执行。支持无序执行的记分板（如 CDC 6600 中使用的记分板）也相当复杂。另一方面，

单线程有序 CPU 非常简单：每个寄存器在记分板中都用一个位表示，只要发出要写入该寄存器的指令，该位就会被设置。任何想要读取或写入在记分板中设置了相应位的寄存器的指令都会被阻止，直到写入寄存器的指令清除该位。这可以防止写后读和写后写危险。当与有序指令发出相结合时，这个简单的记分板可以防止读后写危险，前提是寄存器文件的读取被限制为按顺序进行，这在有序 CPU 设计中通常就是这种情况。鉴于它是最简单的设计，因此将消耗最少的面积和能量，GPU 实现了有序记分板。但是，如下所述，在支持多个 warp 时使用有序记分板存在挑战。

上述简单有序记分板设计的第一个问题是现代 GPU 中包含的寄存器数量非常多。每个 Warp 最多有 128 个寄存器，每个核心最多有 64 个 Warp，因此实现记分板需要每个核心总共 8192 位。

上述简单按序记分板设计的另一个问题是，遇到依赖关系的指令必须反复在记分板中查找其操作数，直到它所依赖的上一条指令将其结果写入寄存器文件。对于单线程设计，这几乎不会带来复杂性。但是，在按序发出中，来自多个线程的多线程处理器指令可能正在等待早期指令完成。如果所有这些指令都必须探测记分板，则需要额外的读取端口。最近的 GPU 支持每个核心最多 64 个 Warp，最多 4 个操作数允许所有 Warp 每个周期探测记分板，这将需要 256 个读取端口，这将非常昂贵。一种替代方法是限制每个周期可以探测记分板的 Warp 数量，但这会限制可以考虑进行调度的 Warp 数量。此外，如果所检查的指令都不存在依赖关系，则即使其他无法检查的指令恰好不存在依赖关系，也不会发出任何指令。

这两个问题都可以使用 Coon 等人 [2008] 提出的设计来解决。该设计不是在每个 warp 中为每个寄存器保存一个位，而是在每个 warp 中包含少量条目（在最近的一项研究中估计约为 3 或 4 个 [Lashgar 等人，2016]），其中每个条目都是将由已发出但尚未完成执行的指令写入的寄存器的标识符。常规的按顺序记分板在指令发出和写回时都会被访问。相反，Coon 等人的记分板是在将指令放入指令缓冲区时以及指令将其结果写入寄存器文件时访问的。

当从指令缓存中取出一条指令并将其放入指令缓冲区时，相应 Warp 的记分板条目将与该指令的源寄存器和目标寄存器进行比较。这会产生一个短位向量，该 Warp 的记分板中每个条目对应一个位（例如 3 位或 4 位）。如果记分板中的相应条目与该指令的任何操作数匹配，则设置一个位。然后，将此位向量与指令一起复制到指令缓冲区中。直到所有位都被清除后，指令调度程序才会考虑该指令，这可以通过将每个位输入到寄存器中来确定

将向量的位放入 NOR 门。当指令将其结果写入寄存器文件时，指令缓冲区中的依赖位将被清除。如果给定 warp 的所有条目都已用完，则所有 warp 的提取都会停止，或者指令将被丢弃并必须再次提取。当已执行的指令准备好写入寄存器文件时，它会清除记分板中分配给它的条目，并清除指令缓冲区中存储的来自同一 warp 的任何指令的相应依赖位。

在双循环架构中，第一个循环选择指令缓冲区中有空间的 Warp，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。第二个循环选择指令缓冲区中没有未完成依赖项的指令并将其发送到执行单元。

3.3 三环路近似

如前所述，为了隐藏较长的内存延迟，需要支持每个核心的多个 Warp，并且为了支持 Warp 之间的逐周期切换，需要有一个大型寄存器文件，其中包含每个正在执行的 Warp 的单独物理寄存器。例如，在 NVIDIA 的最新 GPU 架构（例如 Kepler、Maxwell 和 Pascal 架构）上，此类寄存器包含 256 KB。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要求每个操作数每个周期发出的每条指令都有一个端口。减少寄存器文件面积的一种方法是使用多个单端口存储器组来模拟大量端口。虽然可以通过将这些库暴露给指令集架构来实现这种效果，但在某些 GPU 设计中，似乎使用一种称为操作数收集器的结构 [Coon 等人，2009 年，Lindholm 等人，2008b 年，Lui 等人，2008 年] 以更透明的方式实现这一点。操作数收集器有效形成第三个调度循环，如下所述。

为了更好地理解操作数收集器所解决的问题，首先考虑图 3.12，它展示了一种用于提供增加的寄存器文件带宽的简单微架构。该图显示了 GPU 指令流水线的寄存器读取阶段，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以进一步分解为大量物理寄存器组（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“流水线寄存器”），暂存寄存器在将源操作数传递给 SIMD 执行单元之前对其进行缓冲。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 显示了每个 Warp 的寄存器到逻辑组的简单布局。在该图中，Warp 0 (`w0`) 中的寄存器 `r0` 存储在组 0 的第一个位置，Warp 0 中的寄存器 `r1` 存储在组 1 的第一个位置，依此类推。如果计算所需的寄存器数量大于逻辑组的数量，则分配会回绕。例如，Warp 0 的寄存器 `r4` 存储在组 0 的第二个位置。

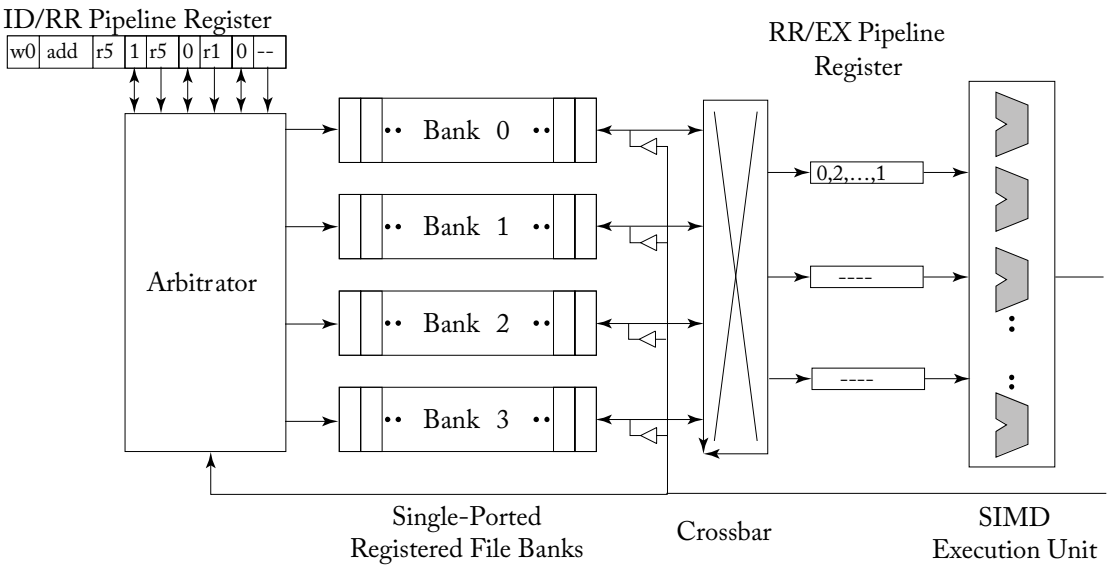


图 3.12：简单的分组寄存器文件微架构。

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：简单的分组寄存器布局。

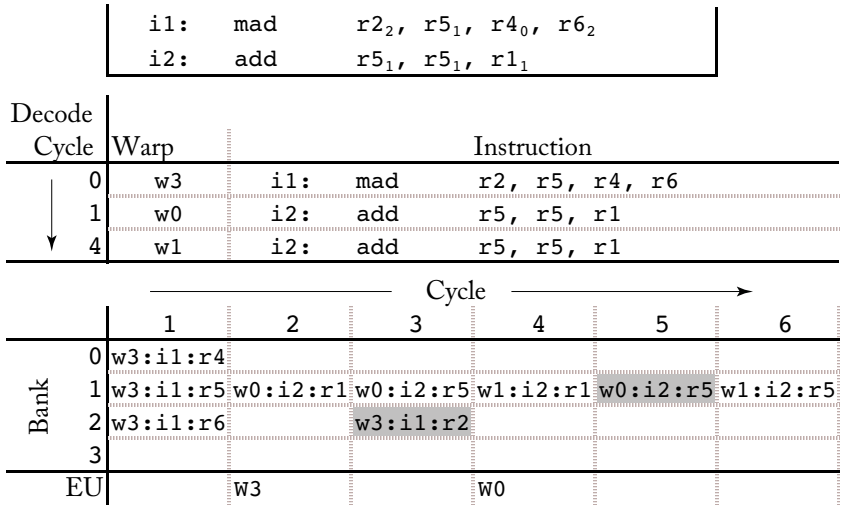


图 3.14：简单寄存器组文件的时间安排。

图 3.14 说明了一个时序示例，突出显示了此微架构如何导致性能下降。该示例涉及顶部显示的两条指令。第一条指令 `i1` 是一个多重加法运算，它从分配在存储体 1、0 和 2 中的寄存器 `r5`、`r4` 和 `r6` 中读取（在图中用下标表示）。第二条指令 `i2` 是一个加法指令，它从分配在存储体 1 中的寄存器 `r5` 和 `r1` 中读取。图的中间部分显示了指令的发出顺序。在周期 0，warp 3 发出指令 `i1`，在周期 1，warp 0 发出指令 `i2`，在周期 4，warp 1 在因存储体冲突而延迟之后发出指令 `i2`，如下所述。图的底部说明了不同指令访问不同存储体的时序。在周期 1 上，来自 warp 3 的指令 `i1` 能够在周期 1 上读取其所有三个源寄存器，因为它们映射到不同的逻辑存储体。但是，在第 2 个周期，来自 Warp 0 的指令 `i2` 只能读取两个源寄存器中的一个，因为两个源寄存器都映射到存储体 1。在第 3 个周期，此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。在第 4 个周期，来自 Warp 1 的指令 `i2` 能够读取其第一个源操作数，但不能读取第二个，因为两个源操作数也都映射到存储体 1。在第 5 个周期，来自 Warp 1 的指令 `i2` 的第二个源操作数无法从寄存器文件中读取，因为该存储体已经被 Warp 0 先前发出的高优先级指令 `i2` 的写回访问。最后，在第 6 个周期，来自 Warp 1 的第二个源操作数 `i2` 将从寄存器文件中读取。总之，三条指令需要六个周期才能完成读取其源寄存器，在此期间许多存储体都未被访问。

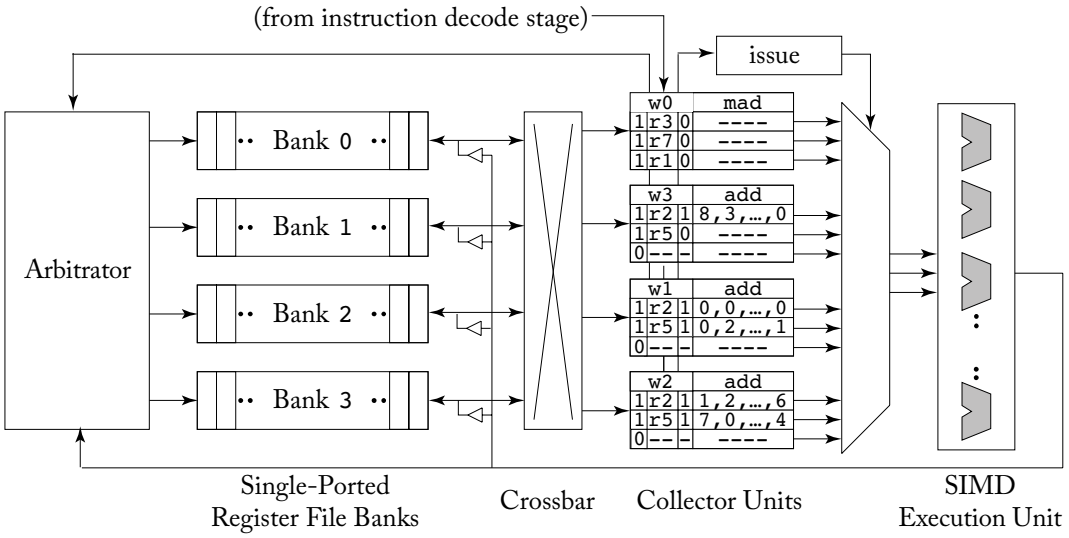


图 3.15：操作数收集器微架构（基于 Tor M. Aamodt 等人的图 6）。

3.3.1 操作数收集器

图 3.15 显示了操作数收集器微架构 [Lindholm et al., 2008b]。关键变化是暂存寄存器已被 *collector units* 取代。每条指令在进入寄存器读取阶段时都会分配一个收集器单元。有多个收集器单元，因此多条指令可以重叠读取源操作数，这有助于在各个指令的源操作数之间存在存储体冲突的情况下提高吞吐量。每个收集器单元都包含执行一条指令所需的所有源操作数的缓冲空间。鉴于多条指令的源操作数数量较多，仲裁器更有可能实现更高的存储体级并行性，以允许并行访问多个寄存器文件存储体。

操作数收集器使用调度来容忍发生的库冲突。这留下了一个如何减少库冲突的问题。图 3.16 展示了 Coon 等人为帮助减少库冲突而描述的经过修改的寄存器布局。其思想是将来自不同 warp 的等效寄存器分配到不同的库中。例如，在图 3.16 中，warp 0 的寄存器 `r0` 分配给库 0，而 warp 1 的寄存器 `r0` 分配给库 1。这并不能解决单个指令的寄存器操作数之间的库冲突。但是，它确实有助于减少来自不同 warp 的指令之间的库冲突。特别地，每当 warp 取得相对均匀的进展时（例如，由于循环调度或两级调度 [Narasiman et al., 2011]，其中获取组中的各个 warp 按循环顺序进行调度）。

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： 混合寄存器布局。

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle →

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

图3.17：操作数收集器时序。

图 3.17 显示了时序示例，其中顶部显示了加法和乘加指令序列。中间显示了发出顺序。来自 warp 1 至 3 的三个 `i1` 实例在第 0 至 2 个周期发出。来自 warp 0 的指令 `i2` 的一个实例在第 3 个周期发出。请注意，加法指令写入寄存器 `r1`，对于任何给定 warp，该寄存器与源寄存器 `r5` 分配在同一个存储体中。但是，与使用图 3.13 中的寄存器布局的情况不同，这里不同的 warp 访问不同的存储体，这有助于减少一个 warp 的写回和读取其他 warp 中的源操作数之间的冲突。底部显示了由于操作数收集器而导致的存储体级别访问时序。在周期 1 上，来自 Warp 1 的寄存器 `r2` 读取 Bank 3。在周期 4 上，注意到来自 Warp 1 的寄存器 `r1` 的写回与来自 Warp 3 的寄存器 `r5` 的读取以及来自 Warp 0 的寄存器 `r3` 的读取同时进行。

到目前为止描述的操作数收集器的一个微妙问题是，由于它没有在不同指令准备发出时强加任何顺序，因此它可能允许写后读 (WAR) 风险 [Mishkin et al., 2016]。如果来自

同一个 warp 中的指令同时存在于操作数收集器中，第一条指令读取第二条指令将要写入的寄存器。如果第一条指令的源操作数访问遇到重复的存储体冲突，那么第二条指令很可能在第一个寄存器读取到（正确的）旧值之前将新值写入寄存器。防止这种 WAR 风险的一种方法是要求同一个 warp 中的指令按程序顺序离开操作数收集器到执行单元。Mishkin 等人 [2016] 探索了三种具有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。第一个方案是 *release-on-commit warpboard*，它允许每个 warp 最多执行一条指令。不出所料，他们发现这会对性能产生负面影响，在某些情况下性能几乎降低了两倍。他们的第二个方案是 *release-on-read warpboard*，它只允许每个 warp 一次一条指令在操作数收集器中收集操作数。这种方案导致他们研究的工作负载最多减慢 10%。最后，为了在操作数收集器中实现指令级并行，他们提出了一种 *bloomboard* 机制，该机制使用小型布隆过滤器来跟踪未完成的寄存器读取。与（错误地）允许 WAR 风险相比，这导致的影响不到百分之几。另外，Gray 进行的分析表明，NVIDIA 的 Maxwell GPU 引入了一种“读取依赖屏障”，它由特殊的“控制指令”管理，可用于避免某些指令的 WAR 风险（参见第 2.2.1 节）。

3.3.2 指令重放：处理结构性危险

GPU 流水线中存在许多导致结构性危险的潜在原因。例如，寄存器读取阶段可能会用尽操作数收集器单元。许多结构性危险的来源与内存系统有关，我们将在下一章中更详细地讨论这一点。通常，warp 执行的单个内存指令可能需要分解为多个单独的操作。每个单独的操作都可能在给定周期内充分利用流水线的一部分。

当一条指令在 GPU 流水线中遇到结构性危险时会发生什么？在单线程有序 CPU 流水线中，标准解决方案是暂停较新的指令，直到遇到暂停条件的指令可以取得进一步进展。这种方法在高度多线程吞吐量架构中可能不太可取，至少有两个原因。首先，考虑到寄存器文件的大小以及支持完整图形流水线所需的许多流水线阶段，分发暂停信号可能会影响关键路径。流水线暂停周期分布导致需要引入额外的缓冲来增加面积。其次，暂停一个 warp 中的指令可能会导致其他 warp 中的指令在其后暂停。如果这些指令不需要导致暂停的指令所需的资源，则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形式。指令重放出现在一些 CPU 设计中，在推测性地将依赖指令调度到具有可变延迟的较早指令时，它被用作恢复机制。例如，负载可能在第一级缓存中命中或未命中，但以高时钟频率运行的 CPU 设计

频率可以在多达四个时钟周期内流水线一级缓存访问。一些 CPU 根据负载推测唤醒指令，以提高单线程性能。相比之下，GPU 避免推测，因为它往往会浪费能源并降低吞吐量。相反，GPU 中使用指令重放来避免堵塞流水线和电路面积和/或因停顿而导致的时序开销。

为了实现指令重放，GPU 可以将指令保存在指令缓冲区中，直到知道它们已经完成或指令的所有各个部分都已执行 [Lindholm et al., 2015]。

3.4 分支分歧的研究方向

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，这样 GPU 就可以在 SIMD 硬件上同步执行它们。考虑到线程的自主性，当 Warp 的线程在数据相关分支处分叉到不同目标时，该 Warp 可能会遇到 *branch divergence*。现代 GPU 包含特殊硬件来处理 Warp 中的分支分叉。第 3.1.1 节介绍了本书中的基本 GPU 架构所使用的基本 SIMT 堆栈。基本 SIMT 堆栈通过序列化不同目标的执行来处理 Warp 中的分支分叉。虽然基本 SIMT 堆栈可以正确处理大多数现有 GPU 应用程序的分支分叉，但它存在以下缺陷。

较低的 SIMD 效率 在存在分支发散的情况下，基线 SIMT 堆栈会序列化每个分支目标的执行。在执行每个目标时，SIMT 堆栈仅激活运行目标的标量线程子集。这会导致 SIMD 硬件中的某些通道处于空闲状态，从而降低整体 *SIMD efficiency*。

无需序列化 基线 SIMT 堆栈对每个分支目标的序列化执行对于功能正确性而言并非必需的。GPU 编程模型不会在 warp 中的标量线程之间施加任何隐式数据依赖性 - 它们必须通过共享内存和屏障进行显式通信。GPU 可以交错执行分叉 warp 的所有分支目标，以利用 SIMD 硬件中的空闲周期。

MIMD 抽象不足 通过强制发散的 Warp 在编译器定义的重新收敛点重新收敛，基线 SIMT 堆栈在每个重新收敛点隐式地施加了一个 Warp 范围的同步点。这适用于许多现有的 GPU 应用程序。但是，这种隐式同步可能会与其他用户实现的同步机制（例如细粒度锁）发生不良交互，从而导致 Warp 死锁。编译器定义的重新收敛点也没有考虑由系统级构造（例如异常和中断）引入的控制流发散。

面积成本 虽然每个 Warp 的基线 SIMT 堆栈的面积要求仅为 32 × 64 位（或低至 6 × 64 位），但面积会随着 GPU 中正在运行的 Warp 数量而变化。在分支发散很少见的典型 GPU 应用中，SIMT 堆栈占用的面积原本可用于以其他方式提高应用程序吞吐量（例如，大缓存、更多 ALU 单元等）。

业界和学术界都提出了一些替代方案来解决上述缺陷。各种提案可以分为以下几类：warp 压缩、warp 内发散路径管理、添加 MIMD 功能和降低复杂性。一些提案包含涵盖多个类别的改进，因此被多次提及。

3.4.1 经纱压缩

由于 GPU 实现了细粒度多线程来容忍较长的内存访问延迟，因此每个 SIMT 核心中都有许多 Warp，总共有数百到数千个标量线程。由于这些 Warp 通常运行相同的计算内核，因此它们很可能遵循相同的执行路径，并在同一组数据相关分支中遇到分支分歧。因此，分歧分支的每个目标可能由大量线程执行，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受分支发散影响的 GPU 应用程序性能的研究。本系列中的提案都涉及新颖的硬件机制，将来自不同 *static warps* 的 *compact* 线程转换为新的 *dynamic warps*，以提高这些不同 GPU 应用程序的整体 SIMD 效率。此处，静态 warp 是指从内核启动生成标量线程时由 GPU 硬件形成的 warp。在我们的基准 GPU 架构中，这种安排在整个 warp 执行过程中都是固定的。标量线程到静态 warp 的排列是 GPU 硬件强加的任意分组，对于编程模型来说基本上是不可见的。

动态 Warp 形成。动态 Warp 形成 (DWF) [Fung et al., 2007, Fung et al., 2009] 利用这一观察结果，将执行相同指令的分散线程重新排列为新的动态 Warp。在发散分支中，DWF 可以将分散在多个发散静态 Warp 中的线程压缩为更少的非发散动态 Warp，从而提高应用程序的整体 SIMD 效率。通过这种方式，DWF 可以在 SIMD 硬件上捕获 MIMD 硬件的很大一部分优势。但是，DWF 要求 Warp 在短时间窗口内遇到相同的发散分支。DWF 的这种时间依赖性使其对 Warp 调度策略非常敏感。

Fung 和 Aamodt [2011] 的后续工作确定了 DWF 的两种主要性能病症：（1）贪婪的调度策略可能会使某些线程挨饿，从而导致 SIMD

效率降低；（2）DWF 中的线程重组增加了非合并内存访问和共享内存库冲突。这些缺陷导致 DWF 减慢了许多现有的 GPU 应用程序的速度。此外，依赖于静态 warp 中的隐式同步的应用程序在 DWF 中执行不正确。

上述问题可以通过改进的调度策略得到部分解决，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（连贯）区域。发散区域从 DWF 中受益匪浅，而连贯区域没有分支发散，但容易出现 DWF 问题。我们发现，通过强制 DWF 将标量线程重新排列回连贯区域中的静态扭曲，可以显著减少 DWF 问题的影响。

线程块压缩。线程块压缩 (TBC) [Fung and Aamodt, 2011] 建立在此见解之上，其观察结果是不断将线程重新排列到新的动态 Warp 中不会产生额外的好处。相反，重新排列或 *compaction* 只需要在发散分支之后、发散区域的开始处以及其重新收敛点之前、连贯区域的开始处进行。我们注意到，现有的每个 Warp SIMT 堆栈（在第 3.1.1 章中描述）隐式同步发散到不同执行路径的线程，在执行连贯区域之前将这些发散的线程合并回静态 Warp。 TBC 扩展了 SIMT 堆栈以涵盖在同一核心中执行的所有 warp，迫使它们在不同的分支和重新收敛点处同步和压缩，以实现强大的 DWF 性能优势。但是，在每个不同的分支处同步核心内的所有 warp 以进行压缩会大大减少可用的线程级并行 (TLP)。GPU 架构依靠丰富的 TLP 来容忍管道和内存延迟。

TBC 通过将压缩限制在仅发生在 *thread block* 内来在 SIMD 效率和 TLP 可用性之间达成妥协。GPU 应用程序通常在单个核心上同时执行多个线程块以重叠同步和内存延迟。TBC 利用这种软件优化来重叠不同分支处的压缩开销 - 当一个线程块中的 warp 同步以在不同分支处进行压缩时，其他线程块中的 warp 可以让硬件保持繁忙。它扩展了每个 warp 的 SIMT 堆栈以包含线程块中的 warp。warp 调度逻辑使用这个线程块范围的 SIMT 堆栈来确定何时应同步线程块中的 warp 并将其压缩成新的 warp 集。结果是一种更加健壮和简单的机制，它捕获了 DWF 的许多好处，而没有病态行为。

大型 Warp 微架构。大型 Warp 微架构 [Narasiman et al., 2011] 扩展了 SIMT 堆栈，类似于 TBC，用于管理一组 Warp 的重新收敛。但是，LWM 不会限制分支和重新收敛点处的压缩。它要求组内的 Warp 完全同步执行，以便它可以在每条指令中压缩组。这比 TBC 更能减少可用的 TLP，但允许

LWM 使用预测指令和无条件跳转执行压缩。与 TBC 类似，LWM 将在同一核心上运行的 Warp 拆分为多个组，并限制压缩仅在一个组内进行。它还选择了一种更复杂的记分板微架构，以线程粒度跟踪寄存器依赖性。这允许组中的某些 Warp 略微领先于其他 Warp 执行，以补偿由于锁步执行而丢失的 TLP。

压缩充分性预测器。Rhu 和 Erez [2012] 使用压缩充分性预测器 (CAPRI) 扩展了 TBC。该预测器确定将线程压缩为每个分支上的几个 warp 的有效性，并且仅在预测压缩会产生效益的分支上同步线程。这可以回收由于无益的停滞和使用 TBC 压缩而丢失的 TLP。Rhu 和 Erez [2012] 还表明，类似于单级分支预测器的简单基于历史的预测器足以实现高精度。

Warp 内部压缩。Vaidya 等人 [2013] 提出了一种低复杂度压缩技术，该技术有利于在较窄的硬件单元上执行多个周期的宽 SIMD 执行组。他们的基本技术是将单个执行组划分为与硬件宽度匹配的多个子组。通过跳过完全空闲的子组，受发散影响的 SIMD 执行组可以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 swizzle 机制，在发散时将元素压缩成更少的子组。

同时进行 Warp 交织。Brunie 等人 [2012] 提出了同时进行分支和 Warp 交织 (SBI 和 SWI)。他们扩展了 GPU SIMT 前端，以支持每个周期发出两个不同的指令。他们通过将 Warp 加宽到其原始大小的两倍来补偿这种增加的复杂性。SWI 同时发出来自出现发散的 Warp 的指令和来自另一个发散 Warp 的指令，以填补分支发散留下的空白。

对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，硬件压缩方案通常在 SIMT 核心内本地进行。由于压缩线程都位于共享相同寄存器文件的同一核心上，因此可以使用更灵活的寄存器文件设计执行压缩而无需移动其架构状态 [Fung et al., 2007]。

如本章前面所述，GPU 寄存器文件采用大型单端口 SRAM 组来实现，以最大程度地提高其面积效率。同一 Warp 中的线程的寄存器存储在同一 SRAM 组中的连续区域中，因此可以通过单个宽端口一起访问它们。这允许高带宽寄存器文件访问，同时分摊寄存器文件访问控制硬件。硬件 Warp 压缩会创建可能不遵循这种寄存器排列的动态 Warp。Fung 等人 [2007] 提出了一种更灵活的寄存器文件设计，其特点是 SRAM 组具有窄端口。此设计具有更多 SRAM 组来维持相同的带宽。

动态微内核。Steeffen 和 Zambreno [2010] 使用 *dynamic micro-kernels* 提高了 GPU 上光线追踪的 SIMD 效率。程序员可以使用原语将数据相关循环中的迭代分解为连续的微内核启动。这种分解本身并不能提高并行性，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动线程压缩为几个 warp 来改善同一核心中不同线程之间的负载不平衡。它与其他硬件 warp 压缩技术的不同之处在于，压缩使用每个核心的暂存器内存作为暂存区，将线程与其架构状态一起迁移。

3.4.1 节总结了一系列在软件中实现 warp 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来将线程从一个 SIMT 核心重新定位到另一个 SIMT 核心。

软件中的 Warp Compaction

在现有的 GPU 上，提高应用程序 SIMD 效率的一种方法是通过软件 Warp 压缩 - 使用软件根据线程/工作项的控制流行为对其进行分组。重新分组涉及在内存中移动线程及其私有数据，这可能会带来很大的内存带宽开销。下面我们重点介绍几项关于软件压缩技术的工作。

条件流 [Kapasi et al., 2000] 将这一概念应用于流计算。它将具有潜在发散控制流的流处理器的计算内核拆分为多个内核。在发散分支处，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，每个流由单独的内核处理，并在控制流发散结束时合并回来。

Billeter 等人 [2009] 提出使用并行前缀和来实现 SIMD *stream compaction*。流压缩将具有各种任务的元素流重新组织为相同任务的紧凑子流。此实现利用 GPU 片上暂存器的访问灵活性来实现高效率。Hoberock 等人 [2009] 提出了一种用于光线追踪的延迟着色技术，该技术使用流压缩来提高具有许多材质类的复杂场景中像素着色的 SIMD 效率。每种材质类都需要其独特的计算。结合每种材质类的计算的像素着色器在 GPU 上的运行效率低下。流压缩将击中具有相似材质类的对象的光线分组，从而允许 GPU SIMD 硬件高效地执行这些像素的着色器。

Zhang 等人 [2010] 提出了一种运行时系统，该系统可以动态地将线程重新映射到不同的 warp 中，以提高 SIMD 效率以及内存访问空间局部性。该运行时系统采用流水线系统，其中 CPU 执行动态重新映射，GPU 对重新映射的数据/线程执行计算。

Khorasani 等人 [2015] 提出了 *Collective Context Collection* (CCC)，这是一种编译器技术，它转换给定的 GPU 计算内核，并产生潜在的分支发散惩罚

以提高其在现有 GPU 上的 SIMD 效率。CCC 专注于计算内核，其中每个线程在每个步骤中执行不规则量的计算，例如通过不规则图进行广度优先搜索。CCC 不会为每个线程分配一个节点（或其他应用程序中的任务），而是首先转换计算内核，以便每个线程处理多个节点，并在内核启动之前确定节点到 warp（注意：不是线程）的分配。然后，CCC 转换计算内核，以便 warp 中的每个线程都可以将任务的上下文卸载到存储在共享内存中的特定于 warp 的堆栈。在当前任务集中经历低 SIMD 效率的 warp 可以将任务卸载到堆栈，并使用这些卸载的任务来填充空闲的线程，然后处理后面的一组任务。实际上，CCC 通过将多个 warp 中的任务分组为更少的一组 warp，然后通过存储在快速片上共享内存中的特定于 warp 的堆栈将分散的任务压缩为每个 warp 中更少的迭代，从而执行“ warp 压缩 ”。

Warp 中线程分配的影响

在本书研究的基准 GPU 架构中，具有连续线程 ID 的线程会静态融合在一起以形成 Warp。很少有学术研究将线程静态分配给 Warp 或 Warp 中的通道。这种默认顺序映射适用于大多数工作负载，因为相邻线程倾向于访问相邻数据，从而改善内存合并。然而，一些研究已经研究了替代方案。

SIMD 通道排列。Rhu 和 Erez [2013b] 观察到，对于本节前面描述的 Warp 压缩技术，将线程 ID 顺序映射到 Warp 中的连续线程并非最佳选择。大多数 Warp 压缩和形成工作的一个关键限制是，当线程被分配到新的 Warp 时，它们不能被分配到不同的通道，否则它们的寄存器文件状态必须移动到矢量寄存器中的不同通道。Rhu 和 Erez 观察到，程序的结构将某些控制流路径偏向某些 SIMD 通道。这种偏向使得实现压缩变得更加困难，因为采用相同路径的线程往往位于同一通道中，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排列，以消除这些程序偏差并显著提高压缩率。

内部 Warp 周期压缩。Vaidya 等人 [2013] 利用了 SIMD 数据路径的宽度并不总是等于 Warp 宽度这一事实。例如，在 NVI [2009] 中，SIMD 宽度为 16，但 Warp 大小为 32。这意味着 32 线程 Warp 将在 2 个核心周期内执行。Vaidya 等人 [2013] 观察到，当出现分歧时，如果为一条指令屏蔽了连续的 SIMD 线程，则该指令可以在一个周期内发出，跳过屏蔽的关闭通道。他们将这种技术称为周期压缩。但是，如果屏蔽的关闭线程不连续，则基本技术不会带来任何性能改进。为了解决这个问题，他们提出了一种混合周期压缩，重新排列哪些线程位于哪些通道中，以创造更多的周期压缩机会。

Warp 标量化。其他研究（例如 [Yang et al., 2014] 的研究）认为，当具有 Warp 的线程对相同数据进行操作时，SIMT 编程模型效率低下。许多解决方案建议在管道中包含一个标量单元，以便编译器或程序员可以先验地识别为标量。AMD 的 Graphics Core Next (GCN) 架构为此目的包含一个标量管道。有关更多详细信息，请参阅第 3.5 节。

3.4.2 经线内发散路径管理

虽然具有立即后支配器重新收敛点的 SIMT 堆栈可以处理具有任意控制流的分支发散，但它可以在各个方面得到进一步改进。

1. 发散到发散 warp 的不同分支目标的线程可以交错执行，以利用 SIMD 硬件中的空闲周期。
2. 虽然发散分支的直接后支配者是明确的收敛点，但是发散到不同分支目标的线程可能能够在发散分支的直接后支配者之前收敛。

以下小节重点介绍了几项尝试从这两个方面改进 SIMT 堆栈的工作。

多路径并行

当 Warp 在分支处发散时，线程会被分成多个组，称为 *warp-splits*。每个 Warp 分裂都由遵循相同分支目标的线程组成。在基线 *single path*、SIMT 堆栈中，来自同一 Warp 的 Warp 分裂会逐个执行，直到 Warp 分裂达到其重新收敛点。这种序列化适合相对简单的硬件实现，但对于功能正确性而言并非必需。Warp 中的线程具有独立的寄存器，并通过内存操作和同步操作（如屏障）明确地相互通信。换句话说，来自同一 Warp 的每个 Warp 分裂都可以在 *parallel* 中执行。我们将这种执行模式称为 *multi-path execution mode*。

虽然不同的 warp-split 可能无法在同一硬件上以同一周期执行（毕竟它们运行不同的指令），但它们可以在同一硬件上交错执行，就像多个 warp 在同一数据路径上交错执行一样。这样，多路径执行模式可以提高应用程序中可用的线程级并行性 (TLP)，以容忍内存访问延迟。即使 SIMD 效率没有提高，多路径执行也可以提高内存受限应用程序的整体性能，其中 SIMT 核心有大量空闲周期可以用于有用的工作。

示例 3.1 展示了一个可能受益于多路径执行的短计算内核。在此示例中，两个分支目标中的代码路径都包含来自内存的加载。在单路径 SIMT 堆栈中，块 B 和 C 中的每一个都按顺序执行，直到相应的 Warp 拆分到达块 D（重新收敛点），即使 Warp 拆分因等待来自内存的数据而停滞。这会停滞整个 Warp，从而在数据路径中引入空闲周期，如图所示

在图 3.18 中，该位置必须由其他 Warp 中的工作填充。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
```

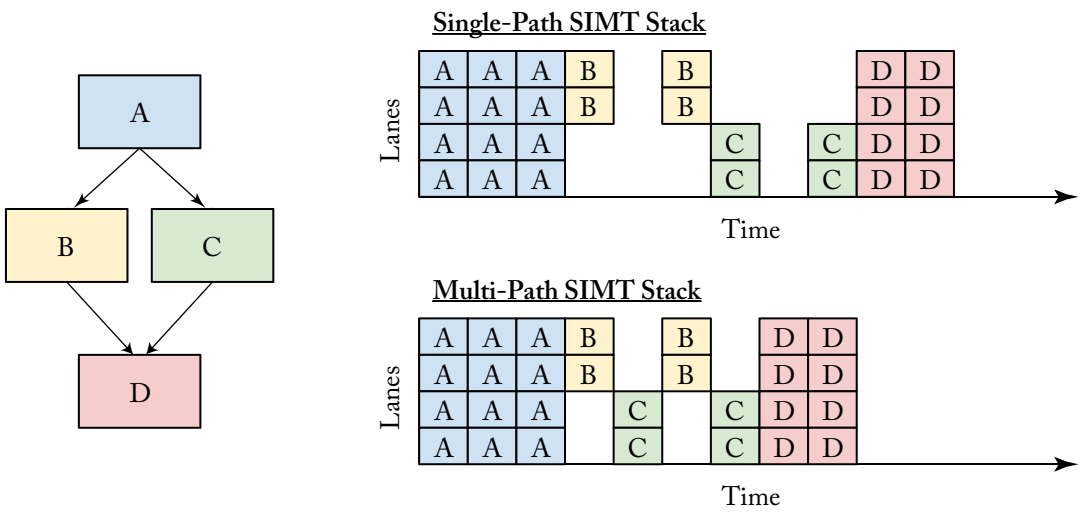


图 3.18：分支分歧处的多路径执行。

动态 Warp 细分。Meng 等人 [2010] 提出 *dynamic warp subdivision* (DWS) 是第一个利用多路径执行带来的 TLP 提升的方案。DWS 使用 Warp 拆分表扩展了 SIMT 堆栈，将发散的 Warp 细分为并发 Warp 拆分。每个 Warp 拆分都执行发散的分支目标，可以并行执行以回收由于内存访问而导致的硬件空闲。当 Warp 中只有一部分线程命中 L1 数据缓存时，内存发散时也会创建 Warp 拆分。DWS 不会等待所有线程获取其数据，而是拆分 Warp 并允许命中缓存的 Warp 拆分先执行，从而可能为未命中缓存的线程预取数据。

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 warp 仅执行两个并发的 warp-split 来解决 DWS 的一些实现缺陷。虽然这种限制使 DPS 能够充分利用 DWS 的大部分优势，但它导致硬件设计变得简单得多。DPS 只需要扩展基线 SIMT 堆栈，添加一组额外的 PC 和活动掩码来编码额外的 warp-split。只有 warp 堆栈顶部条目处的两个 warp-split 是并行执行的；同一 warp 中的每个其他 warp-split 都会暂停，直到其条目到达堆栈顶部。DPS 还附带记分板的扩展，以独立跟踪每个 warp-split 的寄存器依赖性。这使得双路径执行模型能够实现比使用基线记分板的 DWS 更大的 TLP。

多路径执行。ElTantaway 等人 [2014] 使用多路径执行模型 (MPM) 消除了双路径限制。MPM 用两个表替换了 SIMT 堆栈：一个 Warp-split 表，用于维护来自发散 Warp 的 Warp-split 集；一个重收敛表，用于同步所有 Warp-split 到相同的重收敛点。

在发散分支处，在重收敛表中创建一个新条目，其中包含发散分支的重收敛点（其直接后支配者）。在 Warp-split 表中创建多个（通常为两个）条目，每个 Warp-split 一个。每个 Warp-split 条目维护 Warp-split 的当前 PC、其活动掩码、重收敛 PC (RPC) 和指向重收敛表中相应条目的 R 索引。Warp-split 表中的每个 Warp-split 都可供执行，直到其 PC == RPC。此时，相应的重收敛表条目会更新，以反映来自此 Warp-split 的线程已到达重收敛。当所有待处理线程都已到达重新汇聚点时，重新汇聚表条目将被释放，并从 RPC 开始创建一个新的 warp-split 条目，其中重新汇聚的线程处于活动状态。

MPM 还扩展了记分板，以跟踪每个线程的寄存器依赖性，而无需完全复制每个线程的记分板（这会导致 MPM 不切实际，因为这样做会产生很大的面积开销）。这是一个至关重要的扩展，它允许 warp-split 以真正独立的方式执行 - 如果没有此扩展，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 通过机会性早期重新收敛得到进一步扩展，从而提高了非结构化控制流的 SIMD 效率（参见第 3.4.2 节）。

DWS 以及本节讨论的其他技术与 3.4.1 节讨论的 warp 压缩技术是正交的。例如，TBC 中的块宽 SIMT 堆栈可以使用 DWS 进行扩展，以提升可用的 TLP。

更好的融合

基于后支配器（PDOM）堆栈的重新收敛机制[Fung et al.，2007，Fung et al.，2009]使用统一算法确定的重新收敛点，而不是将源代码中的控制流习语转换为指令[AMD，2009，Coon and

Lindholm，2008，Levinthal and Porter，1984]。选择作为重新收敛点的发散分支的直接后支配者是程序中发散线程重新收敛的最早点。在某些情况下，线程可以在 *earlier* 和 *point* 处重新收敛，如果硬件可以利用这一点，它将提高 SIMD 效率。我们相信这一观察促使在最近的 NVIDIA GPU 中加入 `break` 指令 [Coon and Lindholm，2008]。

示例 3.2（来自 [Fung and Aamodt, 2011]）中的代码展示了这种早期的重新收敛。它产生了图 3.19 中的控制流图，其中的边缘标有各个标量线程遵循该路径的概率。块 `F` 是 `A` 和 `C` 的直接后支配者，因为 `F` 是从 `A`（或 `C`）开始的 *all* 路径重合的第一个位置。在基线机制中，当 warp 在 `A` 处发散时，重新收敛点设置为 `F`。但是，从 `C` 到 `D` 的路径很少被遵循，因此在 *most* 情况下，线程可以在 `E` 处更早地重新收敛。

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] 建议使用 *likely convergence points* 扩展 SIMT 堆栈。此扩展为每个 SIMT 堆栈条目添加了两个新字段：一个用于可能收敛点 (LPC) 的 PC，另一个用于 (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堆栈位置。每个分支的可能收敛点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。Fung 和 Aamodt [2011] 的提议将可能收敛点限制为最近的封闭后向分支，以捕获循环内“`break`”语句的影响 [Coon and Lindholm，2008]。

当 Warp 在具有可能收敛点的分支处发散时，三个条目会被推送到 SIMT 堆栈上。第一个条目是 LPC 条目，是为分支的可能收敛点创建的。与基线机制一样，还会为分支的执行和失败创建另外两个条目。这些其他条目中的每个条目中的 LPC 字段都填充了发散分支的可能收敛点，LPos 字段填充了 LPC 条目的堆栈位置。LPC 条目的 RPC 设置为直接后支配者，即确定的

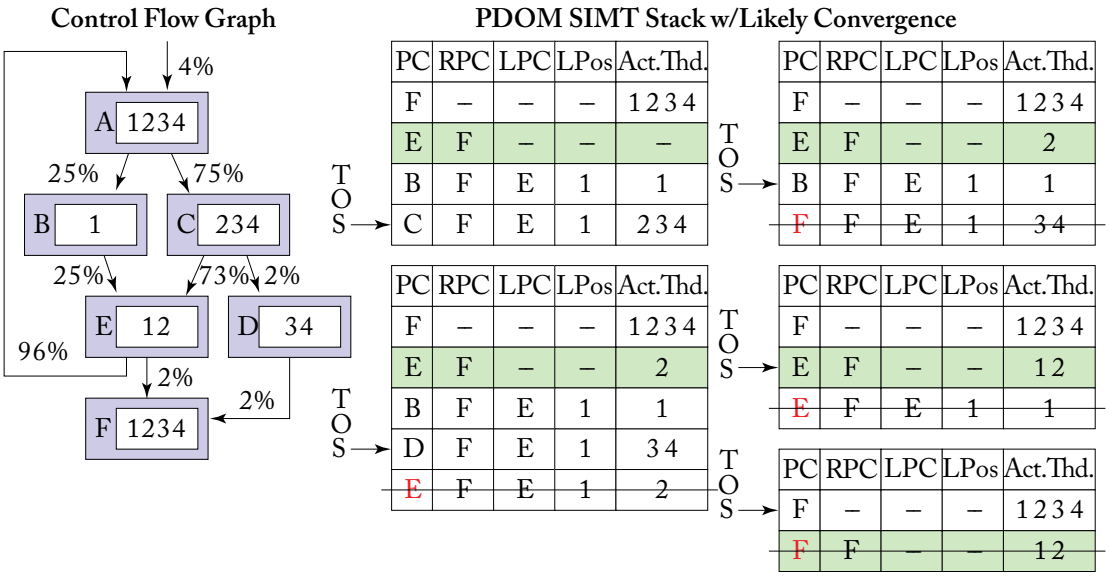


图 3.19：早期的再收敛点 *before* 紧接着后支配点。可能的收敛点捕捉到 E 处的早期再收敛。

重新收敛点，即发散分支的重新收敛点，以便此条目中的线程将重新收敛到确定的重新收敛点。

当一个 warp 使用 SIMT 堆栈中的顶部条目执行时，它会将其 PC 与 RPC 字段（就像它与基线 SIMT 堆栈所做的一样）以及 LPC 字段进行比较。如果 PC == LPC，则弹出 SIMT 堆栈，并且此弹出条目中的线程将合并到 LPC 条目中。否则，如果 PC == RPC，则简单地弹出 SIMT 堆栈 — RPC 条目已将这些线程记录在其活动掩码中。当 LPC 条目到达 SIMT 堆栈的顶部时，它会像任何其他 SIMT 堆栈条目一样执行，或者如果其活动掩码为空，则直接弹出。

线程边界。Diamos 等人 [2011] 完全脱离了 SIMT 堆栈，而是提出通过 *thread frontiers* 在发散后重新收敛线程。支持线程边界的编译器根据内核中基本块的拓扑顺序对其进行排序。这样，在较高 PC 的指令处执行的线程永远不会跳转到较低 PC 的指令。通过将循环出口放在循环体的末尾来处理循环。通过这种排序的代码布局，发散的 warp 最终将通过优先考虑具有较低 PC 的线程（允许它们赶上）来重新收敛。

与具有立即后支配器重新收敛的 SIMT 堆栈相比，通过线程边界重新收敛可为具有非结构化控制流的应用程序带来更高的 SIMD 效率。多表达式条件语句的求值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。扩展的 SIMT 堆栈可能

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 效率改进；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，而线程边界方法没有这样的限制。

机会性早期重收敛。ElTantaway 等人 [2014] 提出了 *opportunistic early reconvergence* (OREC)，无需任何额外的编译器分析，即可提高具有非结构化控制流的 GPU 应用程序的 SIMD 效率。OREC 以同一篇论文中介绍的多路径 (MP) SIMT 堆栈为基础（参见第 3.4.2 节）。MP SIMT 堆栈使用单独的 warp-split 表来保存当前可供执行的 warp-split 集。在发散分支处，将使用分支目标 PC 和发散分支的重收敛 PC 创建新的 warp-split。使用 OREC，硬件不会简单地将这些新的 warp-split 插入 warp-split 表，而是在 warp-split 表中搜索具有相同起始 PC 和 RPC 的现有 warp-split。如果存在这样的 Warp-split，硬件会在重聚表中创建一个早期重聚点，以便在原始 RPC 之前将两个 Warp-split 收敛。早期重聚点将两个 Warp-split 同步到特定的 PC，这样即使现有 Warp-split 已经通过分叉路径前进，它们也可以合并。在 ElTantaway 等人 [2014] 中，早期重聚点是现有 Warp-split 的下一个 PC。

3.4.3 添加 MIMD 功能

以下提案通过整合一定数量的 MIMD 功能来提高 GPU 与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：

- SIMD 模式，其中前端发出一条指令，在 warp 中的所有线程中执行；或者
- MIMD 模式，其中前端为发散 warp 中的每个线程发出不同的指令。

当 Warp 未发散时，它会在 SIMD 模式下执行，以捕获 Warp 中线程所表现出的控制流局部性，其能效可与传统 SIMD 架构相媲美。当 Warp 发散时，它会切换到 MIMD 模式。Warp 在此模式下运行效率较低，但性能损失低于传统 SIMD 架构上的损失。

矢量线程架构。矢量线程 (VT) 架构 [Krashinsky 等，2004] 结合了 SIMD 和 MIMD 架构的特点，目的是兼顾两种方法的优点。VT 架构具有一组连接到公共 L1 指令缓存的通道。在 SIMD 模式下，所有通道都直接从 L1 指令缓存接收指令以进行锁步执行，但每个通道都可以切换到 MIMD 模型，以自己的方式运行

与其 L0 缓存中的指令同步。Lee 等人 [2011] 最近与传统 SIMT 架构（例如 GPU）进行了比较，结果表明 VT 架构与常规并行应用程序具有相当的效率，而与不规则并行应用程序相比，其执行效率要高得多。

时间 SIMT。时间 SIMT [Keckler 等，2011，Krashinsky，2011] 允许每个通道以 MIMD 方式执行，类似于 VT 架构。但是，它不是同步在所有通道上运行 warp，而是通过单个通道对 warp 的执行进行时间复用，并且每个通道运行单独的 warp 集。时间 SIMT 通过为整个 warp 仅获取一次每条指令来实现 SIMD 硬件的效率。这分摊了时间上的控制流开销，而传统的 SIMD 架构则在空间上的多个通道上分摊相同的开销。

可变 Warp 大小架构。可变 Warp 大小 (VWS) 架构 [Rogers et al., 2015] 包含多个（例如 8 个）切片，每个切片包含一个提取和解码单元，因此每个切片可以同时执行不同的指令，类似于 VT 和时间 SIMT。VWS 中的每个切片由窄（4 宽）Warp 组成，而不是通过窄数据路径对大型 Warp 进行时间复用。然后，这些窄 Warp 被分组到称为 *gangs* 的较大执行实体中。每个组包含来自每个切片的 Warp。

在没有分支发散的应用程序中，一个组中的 warp 以锁步执行，从共享提取单元和共享 L1 指令缓存中获取指令。一旦遇到分支发散（或内存发散），该组就会分裂成多个组。新的组可能会进一步分裂，直到每个 warp 都在自己的组中。此时，这些单 warp 组将通过切片的提取单元和私有 L0 指令缓存在自己的切片上单独执行。这些分裂的组会通过硬件比较各个组的 PC 而适时地合并回原始组。如果它们都匹配，则重新创建原始组。Rogers 等人 [2015] 还建议在第一个发散分支的紧邻后支配者处插入组级同步屏障。

本书还评估了每个切片中的 L0 指令缓存容量与共享 L1 指令缓存带宽之间的性能影响。在非联动模式下，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 缓存也可以过滤掉对共享 L1 缓存的大部分请求，因此，L1 缓存仅使用基线 SIMT 架构的 2× 带宽就可以弥补大部分带宽不足。

同时分支交织。Brunie 等人 [2012] 在线程块压缩发布后提出了同时分支和 Warp 交织 (SBI 和 SWI)。他们扩展了 GPU SIMT 前端，以支持每个周期发出两个不同的指令。当 SBI 遇到分支分歧时，它会从同一个 Warp 中同时发出指令。同时执行分歧分支的两个目标可显著消除其性能损失。

每个 Warp 的基准 SIMT 堆栈的面积要求仅为 32×64 位（或使用 AMD GCN 中使用的优化低至 6×64 位）。虽然与 SIMT 核心中的寄存器文件相比，这个面积很小，但这个面积与 GPU 中正在运行的 Warp 数量以及每个 Warp 的线程数量成正比。此外，在分支发散很少见的典型 GPU 应用中，SIMT 堆栈占用的区域原本可用于以其他方式提高应用程序吞吐量。许多提案用替代机制取代 SIMT 堆栈，这些机制共享资源，当 Warp 未遇到任何分支发散时，这些资源可用于其他方式。

标量寄存器文件中的 SIMT 堆栈。AMD GCN [AMD，2012] 具有一个标量寄存器文件，该文件由 warp 中的所有线程共享。其寄存器可用作预测寄存器，以控制 warp 中每个线程的活动。当编译器检测到计算内核中可能存在分歧的分支时，它会使用此标量寄存器文件在软件中模拟 SIMT 堆栈。GCN 架构具有特殊指令来加速 SIMT 堆栈模拟。

最小化支持最坏情况发散所需的标量寄存器数量的一种优化是优先执行活动线程数较少的目标。这样就可以使用 $\log_2(\#threads\ per\ warp)$ 个标量寄存器来支持最坏情况发散，这比基线 SIMT 堆栈所需的条目要少得多。此外，当计算内核没有潜在发散分支时，编译器可以使用为 SIMT 堆栈保留的标量寄存器进行其他标量计算。

线程边界。如第 3.4.2 节所述，Diamos 等人 [2011] 用 *thread frontiers* 替换了 SIMT 堆栈。使用线程边界，每个线程在寄存器文件中维护自己的 PC，并且代码按拓扑排序，以便分支的重新收敛点始终具有更高的 PC。当 Warp 发散时，它始终优先考虑其所有线程中 PC 最低的线程。这组线程称为 Warp 的线程边界。优先执行边界中的线程会隐式强制程序中所有更靠前的线程在要合并的分支的重新收敛点处等待。

由于只有当计算内核包含潜在发散分支时才需要每个线程的 PC，因此编译器只需在这些计算内核中分配一个 PC 寄存器即可。在其他计算内核中，额外的寄存器存储可以提高 warp 占用率，从而增加每个 SIMT 内核可以维持的 warp 数量，以更好地容忍内存延迟。

无堆栈 SIMT。Asanovic 等人 [2013] 提议使用 `syncwarp` 指令扩展时间 SIMT 架构。在此提议中，warp 中的线程在计算内核的 *convergent regions* 中执行时以锁步方式运行，其中编译器保证 warp 永远不会发散。在发散分支处，warp 中的每个线程都遵循其私有 PC 的控制流路径，从而利用时间 SIMT 架构中的 MIMD 功能。

编译器将 `syncwarp` 指令放置在发散分支的重新收敛点。这会强制发散 warp 中的所有线程在进入计算内核的另一个收敛区域之前在重新收敛点同步。

虽然这种机制无法捕捉嵌套发散分支可能出现的重新收敛，但它仍然是一种更便宜的替代方案，可以为很少出现分支发散的 GPU 应用程序提供与基线 SIMT 堆栈相当的性能。本文介绍了一种组合收敛和变体分析，允许编译器确定适用于 *scalarization* 和/或 *affine transfor- mation* 的任意计算内核中的操作。在无堆栈 SIMT 的上下文中，相同的分析允许编译器确定任意计算内核中的收敛和发散区域。

1. 编译器首先假设所有基本块都是*thread-invariant*。

2.它将所有依赖于线程ID的指令，原子指令以及易失性存储器上的内存指令标记为*thread-variant*。

3. 然后，它迭代地将所有依赖于线程变体指令的指令也标记为线程变体。

4. 基本块中所有在线程变量分支指令上为 *control dependent* 的指令也是线程变量。本质上，线程变量分支的直接后支配者之外的指令可以保持线程不变，只要它们没有因其他条件而被标记为线程变量。

此分析允许编译器检测每个 Warp 中所有线程一致执行的分支。由于这些分支不会导致 Warp 发散，因此编译器无需插入代码来检测这些分支的动态发散，也无需在其紧邻的后支配者处插入 `syncwarp` 指令来强制重新收敛。

预测。在将完整的 SIMT 堆栈纳入架构之前，具有可编程着色器的 GPU 一直通过 *predications* 在着色器程序中支持有限的控制流构造，就像传统的矢量处理器一样。预测在现代 GPU 中仍然是一种处理简单 if 分支的低开销方式，可避免推送和弹出 SIMT 堆栈的开销。在 NVIDIA 的实现中，每条指令都扩展了一个额外的操作数字段来指定其预测寄存器。预测寄存器本质上是专用于控制流的标量寄存器。

Lee 等人 [2014b] 提出了一种 *thread-aware prediction algorithm*，将预测的应用扩展到任意控制流，其性能可与 NVIDIA 的 SIMT 堆栈相媲美。线程感知预测算法扩展了标准控制流依赖图 (CDG)，每个分支都有预测节点。然后可以根据每个基本块的控制流依赖关系计算每个基本块所需的预测，并在不破坏功能行为的情况下进一步严格优化。然后，本文描述了两种

基于该线程感知 CDG 的优化，以及他们先前工作中的收敛和方差分析 [Asanovic et al., 2013]。

- 当编译器可以保证分支可以在整个 Warp 中均匀执行（由收敛分析得出）时，将应用 *Static branch-uniformity optimization*。在这种情况下，编译器可以用统一分支指令取代谓词生成。 • 在其他情况下，将应用 *Runtime branch-uniformity optimization*。编译器发出一致分支 (`cbranch.ifnone`)，只有在给定空谓词（即禁用所有线程）时才会执行这些分支。这允许 Warp 跳过带有空谓词的代码 — 这是 SIMT 堆栈提供的一个关键优势。这种方法不同于之前针对矢量处理器（如 BOSCC）的努力，因为它依靠结构分析来确定此优化的候选对象。

虽然预测和 SIMT 堆栈从根本上以相似的能源和面积成本提供相同的功能，但 Le e 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，因此编译器可以调度来自不同分支目标的指令，交错执行不同的分支目标以利用线程级并行 (TLP)，否则将需要更高级的硬件分支发散管理。 • 预测往往会增加寄存器压力，从而降低 warp 占用率并造成整体性能损失。发生这种情况的原因是保守的寄存器分配无法重用分支两侧的寄存器。它无法稳健地证明来自不同分支目标的指令没有在寄存器中操作独占的通道集。两个提议的优化插入的统一和一致分支指令缓解了这个问题。 • 预测可能会以多种方式影响动态指令数。在某些情况下，检查统一分支的开销会显著增加动态指令数。或者，不执行检查意味着某些路径使用空预测掩码执行。在其他情况下，它会删除维护 SIMT 堆栈所需的推送/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，通过循环按顺序执行每个分支目标/函数。 • 除了现有的一致分支指令 `cbranch.ifnone` 和 `cbranch.ifall`) 之外，`cbranch.ifany` (将有助于减少动态统一分支检测引入的指令数开销。

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（例如 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（如果不是全部）现代 GPU 都通过 SIMT 执行模型利用 GPU 线程之间的收敛控制流，其中标量线程被分组为在 SIMD 硬件上运行的 warp（参见第 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对执行相同计算内核的线程之间的 *value structure* [Kim et al., 2013] 的观察。示例 3.3 中的计算内核说明了两种类型的值结构，即 *uniform* 和 *affine*。

统一变量 计算内核中每个线程都具有相同常量值的变量。在算法 3.3 中，变量 `a` 以及文字 `THRESHOLD` 和 `Y_MAX_VALUE` 在计算内核的所有线程中都具有统一的值。统一变量可以存储在单个标量寄存器中，并由计算内核中的所有线程重复使用。

线性变量 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;
```

这种精细表示可以存储为一对标量值，*base* 和 *stride*，这比完全展开的向量紧凑得多。

关于如何在 GPU 中实现 *detect* 和 *exploit* 均匀或仿射变量，有多个研究提案。本节的其余部分从这两个方面总结了这些提案。

3.5.1 均匀或仿射变量的检测

检测 GPU 计算内核中统一或仿射变量的存在主要有两种方法：编译器驱动检测和通过硬件检测。

编译器驱动检测

检测 GPU 计算内核中是否存在统一或仿射变量的一种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将变量声明为

算法 3.3 计算内核中的标量和仿射运算示例（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;
}
```

在整个计算内核中，常量以及为线程 ID 提供特殊变量。编译器可以执行控制依赖性分析，以检测仅依赖于常量和线程 ID 的变量，并将它们标记为统一/仿射。仅对统一/仿射变量起作用的操作将成为 *scalarization* 的候选。

AMD GCN [AMD，2012] 依靠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运算。

Asanovic 等人 [2013] 引入了一种综合收敛和变体分析，允许编译器确定任意计算内核中适用于 *scalariza-*、*tion* 和/或 *affine transformation* 的操作。计算内核收敛区域内的指令可以转换为标量/仿射指令。在计算内核从发散区域到收敛区域的任何转换中，编译器都会插入一条 `syncwarp` 指令来处理两个区域之间控制流引起的寄存器依赖关系。Asanovic 等人 [2013] 采用此分析为 Temporal-SIMT 架构生成标量操作 [Keckler 等人，2011，Krashinsky，2011]。

解耦仿射计算 (DAC) [Wang and Lin, 2017] 依靠类似的编译器分析来提取标量和仿射候选，并将其解耦为单独的 warp。Wang and Lin [2017] 通过发散仿射分析增强了该过程，目标是提取从计算内核开始就已仿射的指令链。这些仿射指令链从主内核解耦为仿射内核，该内核通过硬件队列将数据输入主内核。

硬件检测
在硬件中检测均匀/仿射变量比编译器驱动的检测有两个潜在优势。

1. 这使得标量化和仿射执行可以与原始的 GPU 指令集架构一起应用。它节省了与硬件共同开发专用标量化编译器的努力。

2. 硬件检测发生在计算内核执行期间。因此，它能够检测动态发生的均匀/仿射变量，但静态分析会遗漏这些变量。

基于标签的检测。Collange 等人 [2010] 介绍了一种基于标签的检测系统。在这个系统中，每个 GPU 寄存器都用一个标签进行扩展，指示寄存器是否包含统一、仿射或通用向量值。在启动计算内核时，包含线程 ID 的寄存器的标签设置为仿射状态。从常量或共享内存中的单个位置广播值的指令将目标寄存器的标签设置为统一状态。在内核执行期间，寄存器的状态根据表 3.1 中的简单规则在算术指令中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，但它往往比较保守——例如，它保守地将统一和仿射变量之间的乘法输出规定为向量变量。

表 3.1：来自 Collange 等人的指令间均匀和仿射状态传播规则示例 [2010]。对于每个操作，第一行和第一列显示输入操作数的状态，其余条目显示输入操作数状态每次变换的输出操作数状态（U = 均匀，A = 仿射，= 矢量）。

+	U	A	V	×	U	A	V	<<	U	A	V
U	U	A	V	U	U	V	V	U	U	A	V
A	A	V	V	A	V	V	V	A	V	V	V
V	V	V	V	V	V	V	V	V	V	V	V

FG-SIMT 架构 [Kim et al., 2013] 扩展了 Collange et al. [2010] 的基于标签的检测机制，更好地支持分支。如果其中一个操作数是统一的，则通过标量数据路径解析仿射分支或比较仿射操作数的分支。Kim et al. [2013] 还引入了一种 *lazy expansion* 方案，其中仿射寄存器在发散分支或谓词指令之后被惰性扩展为全矢量寄存器。需要进行此扩展以允许发散 warp 中的线程子集更新其在目标寄存器中的槽，同时保持其他槽不变 - 这保持了 SIMT 执行语义。与在第一个发散分支之后扩展每个仿射寄存器的更简单、更急切的扩展方案相比，惰性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] 引入了一种更积极的机制来检测统一变量，方法是在每次写回矢量指令时比较来自 warp 中所有线程的寄存器值。在检测到统一变量时，检测逻辑将写回重新路由到标量寄存器文件，并更新内部表以记住寄存器的状态。随后将寄存器的使用重定向到标量寄存器文件。所有操作数都来自标量寄存器文件的指令在单独的标量流水线上执行。

Lee 等人 [2015] 使用了类似的检测方案。他们没有使用简单的统一检测器，而是使用寄存器值压缩器增强了寄存器写回阶段，该压缩器使用 Pekhimenko 等人 [2012] 引入的算法将传入的值向量转换为 $< base, delta, immediate >$ (BDI) 的元组。

Wong 等人 [2016] 引入了 *Warp Approximation*，这是一个利用 warp 中的近似计算的框架，它还具有寄存器写回检测功能。检测器计算写回到寄存器文件的向量中的所有值中最小的 *d-similarity*，即两个共享 *d*-MSB 的给定值。d 相似度高于阈值的寄存器被标记为 *similar*，然后用于确定后续相关指令中近似执行的资格。

与 Lee et al. [2015] 的提议一样，G-Scalar [Liu et al., 2017] 在寄存器写回阶段采用了寄存器值压缩器，但该压缩器采用了一种更简单的算法，该算法仅提取所有通道中所有值使用的公共字节。如果所有字节都是公共的，则寄存器包含一个统一变量。任何仅对统一变量进行操作的指令都可以标量化。

G-Scalar 还扩展了寄存器值压缩器，以检测在分支发散下适合标量执行的操作。所有先前的提案都会在 warp 发散后立即恢复为矢量执行。Liu 等人 [2017] 观察到，在分支发散下的许多指令中，活动通道的操作数值是统一的。使用这些部分统一寄存器的指令实际上适合标量执行。然后，他们使用特殊逻辑扩展了寄存器值压缩器，以仅检查来自活动通道的值。这大大增加了跨各种 GPU 计算工作负载的标量指令数量。请注意，在发散下，写入的寄存器不会被压缩。

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价值结构的存在。

压缩寄存器存储

均匀和仿射变量的紧凑表示允许它们以更少的位存储在寄存器文件中。回收的存储空间可用于维持更多的飞行中扭曲，从而提高 GPU 对相同寄存器文件资源的内存延迟的容忍度。

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

- AMD GCN 架构具有标量寄存器文件，可由标量和矢量管道访问。
- FG-SIMT 架构 [Kim et al., 2013] 将均匀/仿射值存储在单独的仿射 SIMT 寄存器文件 (ASRF) 中。 ASRF 记录每个寄存器的状态（仿射/均匀/矢量）

寄存器，允许控制逻辑检测控制处理器上有资格直接执行的操作。

- Gilani 等人 [2013] 提出的动态统一检测提案将动态检测到的统一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] 将基数、增量、立即数 (BDI) 压缩应用于写回寄存器文件的寄存器。压缩寄存器在作为源操作数读回时被解压缩回法向量。在此方案中，每个压缩寄存器仍占用与未压缩寄存器相同的存储槽，但仅占用寄存器组的子集，因此读出寄存器的压缩表示所需的能量较少。

Warp Approximate 架构 [Wong et al., 2016] 通过仅访问与相似性检测选择的代表线程相对应的通道来减少寄存器读/写能耗。

类似地，G-Scalar [Liu et al.， 2017] 具有压缩寄存器，其仅占用分配的寄存器的子集或未压缩的寄存器，以减少寄存器读取的能量。

专用仿射扭曲。解耦仿射计算 (DAC) [Wang and Lin，2017] 在专用仿射扭曲的寄存器中缓冲所有编译器提取的仿射变量。此仿射扭曲与其他非仿射扭曲共享相同的矢量寄存器文件存储，但仿射扭曲使用每个单独寄存器条目的单独通道来存储基数以及不同非仿射扭曲的增量。

标量化操作

除了高效的存储之外，具有均匀或仿射变量的运算可以是 *scalarized*。标量运算可以在单个标量数据路径中执行一次，而不是通过 SIMD 数据路径在 warp 中的所有线程上重复相同的运算，从而在此过程中消耗的能量要少得多。一般来说，如果算术运算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 架构具有专用标量流水线，可执行编译器生成的标量指令。FG-SIMT 架构 [Kim et al., 2013] 具有控制处理器，能够直接执行动态检测到的仿射运算，而无需调用 SIMD 数据路径。

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功能（例如，与主机处理器的通信）也可以卸载到标量流水线，从而使 SIMD 数据路径摆脱实现完整指令集的负担。

时钟门控 SIMD 数据路径。Warp Approximate 架构 [Wong et al., 2016] 和 G-Scalar [Liu et al., 2017] 均在其中一个上执行动态检测的标量指令

SIMD 数据路径中的通道。发生这种情况时，其他通道将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，也消除了必须对要在标量数据路径上实现的子集进行分类的重复工作。例如，G-Scalar [Liu et al., 2017] 可以以相对较低的开销标量化特殊功能单元支持的指令。

聚合到 Affine Warp。解耦 Affine 计算 (DAC) [Wang and Lin, 2017] 将来自多个 Warp 的 Affine 操作聚合到每个 SIMT 核心的单个 Affine Warp 中。此 Affine Warp 与其他 Warp 一样在 SIMD 数据路径上执行，但执行的每条指令同时在多个 Warp 的 Affine 表示上运行。

内存访问加速
当使用统一或仿射变量来表示内存操作（加载/存储）的地址时，内存操作涉及的内存位置是高度可预测的——每个连续位置都由已知步长分隔。这允许进行各种优化。例如，具有已知步长的内存位置的内存合并比任意随机位置的合并简单得多。仿射变量还可用于表示使用单个指令而不是通过加载/存储指令循环的批量传输。

FG-SIMT 架构 [Kim et al., 2013] 在控制过程中采用特殊的地址生成单元，将具有近似地址的内存访问扩展为实际地址。由于近似地址在线程之间具有固定的步长，因此可以使用更简单的硬件将这些近似内存访问合并到缓存行中。

解耦仿射计算 (DAC) [Wang and Lin, 2017] 也具有类似的优化，以利用仿射内存访问中的固定步长。此外，它使用仿射 Warp 在其他非仿射 Warp 之前执行，预取这些 Warp 的数据。预取的数据存储在 L1 缓存中，稍后由相应的非仿射 Warp 通过特殊的出队指令检索。

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），在数量少得多（但数量仍然很大）的 ALU 上多路复用执行，以容忍流水线和内存访问延迟。为了在 warp 之间快速高效地切换，GPU 使用硬件 warp 调度程序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很大，有时会超过最后一级缓存的容量，这是由于 GPU 中使用的宽 SIMD 数据路径以及容忍数百个内存访问延迟周期所需的 warp 数量。例如，

NVIDIA 的 Fermi GPU 可以支持超过 20,000 个运行线程，并且总寄存器容量为 2 MB。

为了最大限度地减少寄存器文件存储所占用的区域，GPU 上的寄存器文件通常通过低端口数 SRAM 组实现。SRAM 组被并行访问，以提供维持在宽 SIMD 流水线上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个指令的操作数访问，以最大限度地减少组冲突惩罚。

访问这些大型寄存器文件每次访问都会消耗大量动态能量，而且它们的大尺寸也会导致高静态功耗。在 NVIDIA GTX280 GPU 上，寄存器文件消耗了近 10% 的 GPU 总功率。这为创新 GPU 寄存器文件架构以降低其能耗提供了明确的激励。因此，近年来出现了大量关于这一主题的研究论文。本节的其余部分总结了旨在实现这一目标的几项研究提案。

3.6.1 分层寄存器文件

Gebhart 等人 [2011b] 观察到，在一组现实世界的图形和计算工作负载中，一条指令产生的值中多达 70% 只被读取一次，只有 10% 只被读取两次以上。为了捕捉大多数寄存器值的这种短暂生命周期，他们建议使用 *register file cache* (RFC) 扩展 GPU 上的主寄存器文件。这形成了寄存器文件的层次结构，并大大降低了对主寄存器文件的访问频率。

在本研究中，RFC 通过 FIFO 替换策略为每条指令的目标操作数分配一个新条目。未命中 RFC 的源操作数不会加载到 RFC 中，以减少对已经很小的 RFC 的污染。默认情况下，从 RFC 中逐出的每个值都会写回主寄存器文件。然而，由于其中许多值再也不会被读取，Gebhart 等人 [2011b] 使用编译时生成的静态活跃信息扩展了纯硬件 RFC。在指令编码中添加了一个额外的位，以指示最后一条使用寄存器值的指令。最后一次读取的寄存器在 RFC 中被标记为死寄存器。在逐出时，它不会写回主寄存器文件。

为了进一步减小 RFC 的大小，Gebhart 等人 [2011b] 将其与两级 Warp 调度程序相结合。这个两级 Warp 调度程序将执行限制在一个 *active* Warp 池中，该池仅由每个 SIMT 核心中的一小部分 Warp 组成。这项工作考虑了 4-8 个 Warp 的活动 Warp 池，每个 SIMT 核心总共有 32 个 Warp。RFC 只保存来自活动 Warp 的值，因此较小。在执行长延迟操作（例如全局内存加载或纹理提取）时，Warp 会从活动池中移除。发生这种情况时，Warp 的 RFC 条目将被刷新，从而为由第二级调度程序激活的另一个 Warp 释放空间。

编译时管理的寄存器文件层次结构。Gebhart 等人 [2011a] 进一步扩展了此寄存器文件层次结构，添加了最后结果文件 (LRF)，该文件仅缓冲每个活动 Warp 的最后一条指令产生的寄存器值。这项工作还用编译时管理的操作数寄存器文件 (ORF) 取代了硬件管理的 RFC。ORF 中值的进出移动由编译器明确管理。这消除了 RFC 所需的标签查找。编译器还可以更全面地了解大多数 GPU 工作负载中的寄存器使用模式，从而做出更优化的决策。这项工作还扩展了两级 Warp 调度程序，以便编译器指示何时可以将 Warp 切换出活动池。这需要协调 ORF 的内容和 Warp 的活跃度，在 Warp 切换出之前将所有实时数据从 ORF 移回主寄存器文件。

3.6.2 瞌睡状态寄存器文件

Abdel-Majeed 和 Annavaram [2013] 提出了一种三模式寄存器文件设计，以降低大型 GPU 寄存器文件的泄漏功率。三模式寄存器文件中的每个条目都可以在 ON、OFF 和 Drowsy 模式之间切换。ON 模式是正常操作模式；OFF 模式不保留寄存器的值；Drowsy 模式保留寄存器的值，但条目需要在访问前唤醒到 ON 模式。在这项工作中，所有未分配的寄存器都处于 OFF 模式，并且所有分配的寄存器在每次访问后立即进入休眠状态。由于 GPU 上的细粒度多线程，此策略利用了 GPU 上对同一寄存器的连续访问之间的长延迟，以允许寄存器文件中的寄存器大部分时间处于休眠模式。GPU 中的长管道还意味着从休眠状态唤醒寄存器的额外延迟不会带来显着的性能损失。

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 线程中的活动寄存器数量往往很少。对于某些 GPU 应用程序，他们声称多达 60% 的寄存器未被使用。他们建议通过使用寄存器重命名来虚拟化物理寄存器，将物理寄存器文件的大小减少多达 50% 或将并发执行的线程数量增加一倍。在提出的机制中，线程开始执行时没有分配任何寄存器，并且在解码指令时将物理寄存器分配给目标寄存器。Tarjan 和 Skadron 进一步建议，可以通过使用编译器分析来确定寄存器的最后一次读取，从而增强物理寄存器的释放。他们提出了“ 最终读取注释 ”，并建议为每个操作数添加“ 一个位来指示它是否是最后一次读取 ”，并指出这可能需要在指令编码中添加额外的位。

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，通过使用溢出将寄存器文件的大小减少 50% 会使执行时间平均增加 73%。他们回顾了在使用无序的 CPU 上使用寄存器重命名时尽早回收物理寄存器的旧提案

执行。他们建议通过添加“元数据指令”来解决添加“最终读取注释”所需的额外位的问题，这些指令可以有效地编码何时可以回收物理寄存器，并使用寄存器生存期活跃度分析生成这些指令。他们提出的一个重要观察是，在确定在哪里可以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 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，通过不重命名具有长生存期的寄存器可以将这个开销减少到 1 KB。为了利用这个机会，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使用重命名。Jeon 等人进一步建议使用重命名来启用寄存器文件子阵列的电源门控。他们评估了通过寄存器重命名支持寄存器文件虚拟化的详细提案的有效性，结果表明确实可以将寄存器文件的大小减少 50%，而性能不会有任何损失。

3.6.4 分区寄存器文件

Abdel-Majeed 等人 [2017] 引入了 *Pilot Register File*，它将 GPU 寄存器文件划分为快速和慢速寄存器文件（FRF 和 SRF）。FRF 使用常规 SRAM 实现，而 SRF 使用近阈值电压 (NTV) SRAM 实现。与常规 SRAM 相比，NTV SRAM 的访问能耗低得多，泄漏功率也低得多。作为交换，NTV SRAM 的访问延迟要慢得多，通常包含几个周期（而不是常规 SRAM 中的一个周期）。在这项工作中，SRF 明显大于 FRF。每个 warp 在 FRF 中都有 4 个条目。关键是使用 FRF 来服务大部分访问以弥补 SRF 的缓慢。访问 SRF 的额外延迟由操作数收集器处理。通过使用 FinFET 的背栅控制，FRF 进一步增强了低功耗模式。这允许非活动 Warp 的 FRF 切换到低功耗模式。这样，FRF 就可以享受两级调度程序的好处，而无需明确调度 Warp 进出活动池。

这项工作与分层寄存器文件不同，因为不同的分区保存一组独有的寄存器，并且分区在 Warp 的整个生命周期中保持不变。Abdel-Majeed 等人 [2017] 不使用编译器来确定要放置在 FRF 中的寄存器集，而是在每次内核启动时使用引导 CTA 来分析最常用的寄存器。这组高使用率的寄存器记录在查找表中，内核启动后的每个后续 Warp 都可以访问该查找表。

3.6.5 调整

Kloosterman 等人 [2017] 引入了 *RegLess*，旨在消除寄存器文件并将其替换为操作数暂存缓冲区。本文观察到，在相对较短的时间内，访问的寄存器数量只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次数少于

使用 GTO 或两级 Warp 调度程序时，约占 2048 KB 寄存器文件的 10%。为了利用这一观察结果，RegLess 使用编译器算法将内核执行划分为多个区域。区域是单个基本块内的连续指令。选择区域之间的边界是为了限制活动寄存器的数量。容量管理器 (CM) 使用区域注释确定哪些 Warp 有资格进行调度。当 Warp 开始执行新区域的指令时，该区域中使用的寄存器将从全局内存中分配的备用存储区域带入操作数暂存单元 (OSU)，并可能缓存在 L1 数据缓存中。OSU 本质上是一个由 8 个存储体组成的缓存，它提供的带宽足以每周期处理两条指令。为了避免在访问 OSU 中的数据时停顿，CM 会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 warp 维护一个状态机，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 和内存层次结构之间产生的内存流量，RegLess 采用了利用近似值的寄存器压缩技术（参见第 3.5 节）。

Kloosterman 等人对其提案进行了详细评估，包括 Verilog 综合和提取 RegLess 引入的硬件单元的寄生电容和电阻值。他们的评估表明，512 个条目的 OSU 可以实现比 2048 KB 寄存器文件略好的性能，同时仅占用 25% 的空间并将整体 GPU 能耗降低 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交互。传统图形应用程序与多个内存空间（如纹理、常量和渲染表面）交互。虽然 CUDA 等 GP GPU 编程 API 可以访问这些内存空间，但本章我们将重点介绍 GPGPU 编程中使用的内存空间，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 在逻辑上将内存进一步细分为本地和全局内存空间。本地内存空间是每个线程私有的，通常用于寄存器溢出，而全局内存用于多个线程共享的数据结构。此外，现代 GPU 通常实现程序员管理的暂存器内存，在协作线程阵列中一起执行的线程之间共享访问权限。包含共享地址空间的一个动机是，在许多应用程序中，程序员知道在计算的给定步骤中需要访问哪些数据。通过一次将所有这些数据加载到共享内存中，它们可以重叠长延迟的片外内存访问，并在对这些数据执行计算时避免对内存的长延迟访问。更重要的是，在给定时间内（DRAM 带宽）在 GPU 和片外内存之间传输的字节数相对于在相同时间内可以执行的指令数较小。此外，在片外存储器和 GPU 之间传输数据所消耗的能量比从片上存储器访问数据所消耗的能量高出几个数量级。因此，从片上存储器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部分，分别反映内存分为驻留在 GPU 核心内的部分和连接到片外 DRAM 芯片的内存分区内的部分。

4.1 一级存储器结构

本节介绍 GPU 上的一级缓存结构，重点介绍统一的 L1 数据缓存和暂存器“ 共享内存 ”，以及它们如何与核心管道交互。我们还简要讨论了 L1 纹理缓存的典型微架构。我们讨论了纹理缓存，它在 GPU 计算应用中的使用有限，因为它提供了一些关于 GPU 与 CPU 不同之处的见解和直觉。最近的一项专利描述了如何统一纹理缓存和 L1 数据（例如，在 NVIDIA 的

Maxwell 和 Pascal GPU）[Heinrich 等人，2017]。我们将这种设计的讨论推迟到首先考虑纹理缓存的组织方式之后。GPU 中第一级内存结构的一个有趣方面是它们在遇到风险时如何与核心管道交互。如第 3 章所述，管道风险可以通过重放指令来处理。我们在本章中扩展了之前关于重放的讨论，重点关注内存系统中的风险。

4.1.1 暂存器和 L1 数据缓存

在 CUDA 编程模型中，“ 共享内存 ” 是指相对较小的内存空间，预期延迟较低，但给定 CTA 内的所有线程都可以访问。在其他架构中，这种内存空间有时被称为暂存器内存 [Hofstee，2005]。访问此内存空间的延迟通常与寄存器文件访问延迟相当。事实上，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，此内存空间称为 “ 本地内存 ”。从程序员的角度来看，使用共享内存时要考虑的一个关键方面是其有限的容量之外的 *bank conflicts* 的潜力。共享内存实现为静态随机存取存储器 (SRAM)，在一些专利 [Minkin 等，2012] 中描述为每通道一个存储体，每个存储体有一个读取端口和一个写入端口。每个线程都可以访问所有存储体。当多个线程在给定周期内访问同一个存储体，并且这些线程希望访问该存储体中的不同位置时，就会出现 *bank conflict*。在详细考虑共享内存的实现方式之前，我们首先来看一下 L1 数据缓存。

L1 数据缓存维护缓存中全局内存地址空间的子集。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，访问全局内存时的一个关键考虑因素是给定 warp 内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单个 L1 数据缓存块内的位置，并且该块不在缓存中，则只需向较低级别的缓存发送单个请求。这种访问被称为 “ 合并 ”。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。程序员试图避免库冲突和未合并访问，但为了简化编程，硬件允许两者。

图 4.1 展示了 Minkin 等人 [2012] 描述的 GPU 缓存组织。图示中的设计实现了统一的共享内存和 L1 数据缓存，这是 NVIDIA Fermi 架构中引入的一项功能，Kepler 架构中也有此功能。图表的中心是 SRAM 数据阵列 5，可以配置 [Minkin et al., 2013] 部分用于共享内存的直接映射访问，部分配置为组相联缓存。该设计在处理存储体冲突和 L1 数据缓存未命中时使用重放机制，支持与指令流水线的非停顿接口。为了帮助解释此操作，

缓存架构我们首先考虑如何处理共享内存访问，然后考虑合并的缓存命中，最后考虑缓存未命中和未合并的访问。对于所有情况，内存访问请求首先从指令管道内的加载/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，每个线程都有一个内存地址以及操作类型。

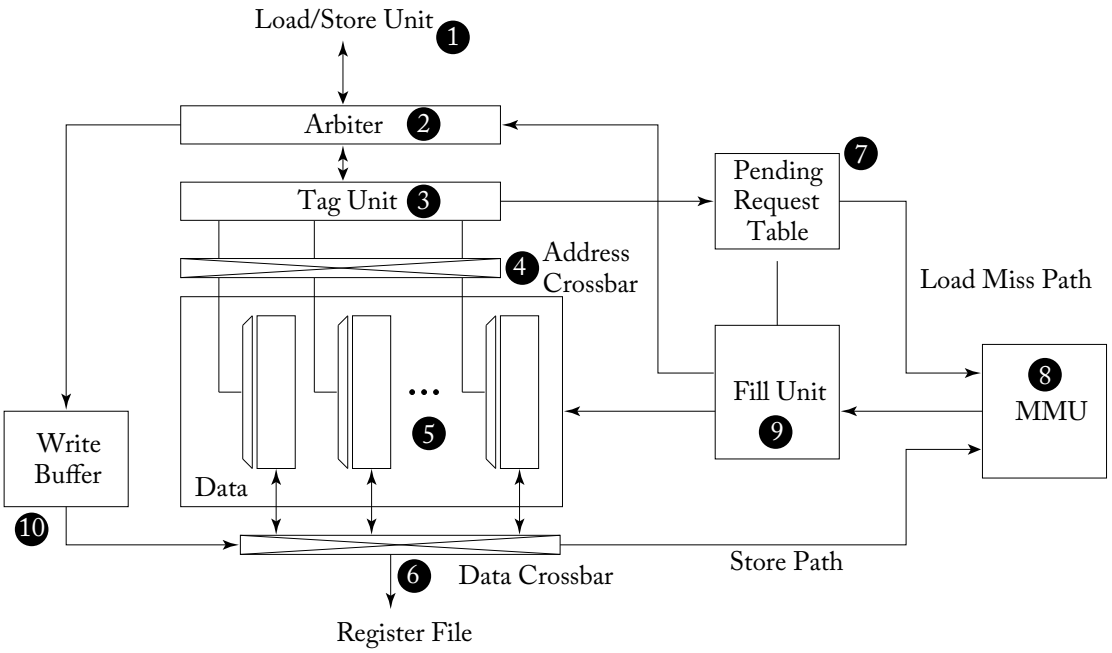


图 4.1：统一的 L1 数据缓存和共享内存 [Minkin et al., 2012]。

共享内存访问操作

对于共享内存访问，仲裁器会确定 Warp 中请求的地址是否会导致存储体冲突。如果请求的地址会导致一个或多个存储体冲突，仲裁器会将请求拆分为两部分。第一部分包括 Warp 中没有存储体冲突的线程子集的地址。仲裁器会接受原始请求的这一部分，以供缓存进一步处理。第二部分包含与第一部分中的地址导致存储体冲突的地址。原始请求的这一部分将返回到指令管道，必须再次执行。此后续执行称为“ 重放 ”。原始共享内存请求的重放部分的存储位置存在权衡。虽然可以通过从指令缓冲区重放内存访问指令来节省面积，但这会消耗访问大型寄存器文件的能量。更好的节能替代方案可能是在 Warp 中提供有限的缓冲来重放内存访问指令

加载/存储单元，并避免在指令缓冲区的可用空间即将用完时从指令缓冲区调度内存访问操作。在考虑重放请求会发生什么之前，让我们先考虑一下如何处理内存请求的已接受部分。

由于共享内存是直接映射的，因此共享内存请求的已接受部分将绕过标签单元 3 内的标签查找。当接受共享内存加载请求时，仲裁器将安排一个写回事件到指令管道内的寄存器文件，因为在没有存储体冲突的情况下，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存储体，以控制地址交叉开关 4，地址交叉开关 4 将地址分配给数据阵列中的各个存储体。数据阵列 5 内的每个存储体都是 32 位宽，并且具有自己的解码器，允许独立访问每个存储体中的不同行。数据通过数据交叉开关 6 返回到相应线程的通道以存储在寄存器文件中。只有与 warp 中的活动线程相对应的通道才会将值写入寄存器文件。

假设共享内存查找的延迟为单周期，则共享内存请求的重放部分可以在前一个接受部分之后的周期访问 L1 缓存仲裁器。如果此重放部分遇到库冲突，则进一步细分为接受和重放部分。

缓存读取操作

接下来，让我们考虑如何处理全局内存空间的加载。由于 L1 中仅缓存了全局内存空间的子集，因此标记单元需要检查数据是否存在于缓存中。虽然数据阵列高度存储，以便各个 warp 能够灵活访问共享内存，但对全局内存的访问仅限于每个周期一个缓存块。此限制有助于减少相对于缓存数据量的标记存储开销，也是标准 DRAM 芯片标准接口的结果。Fermi 和 Kepler 中的 L1 缓存块大小为 128 字节，Maxwell 和 Pascal [NVIDIA Corp.] 中进一步分为四个 32 字节扇区 [Liptay，1968]。32 字节扇区大小对应于单次访问中可从最新图形 DRAM 芯片（例如 GDDR5）读取的最小数据大小。每个 128 字节缓存块由 32 个存储体中同一行的 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 warp 的内存访问分解为单独的合并访问，然后将其输入到仲裁器 2 。如果资源不足，仲裁器可能会拒绝请求。例如，如果访问映射到的缓存集中的所有路径都处于繁忙状态，或者待处理请求表 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固定数量的周期内安排对寄存器文件的写回，以对应缓存命中。同时，仲裁器还请求标记单元 3 检查访问实际上是否导致缓存命中或未命中。如果发生缓存命中，则在所有存储体中访问数据阵列 5 的相应行，并且

数据返回到指令流水线中的寄存器文件。与共享内存访问的情况一样，只有与活动线程相对应的寄存器通道才会更新。

访问标记单元时，如果确定请求触发了缓存未命中，仲裁器将通知加载/存储单元它必须重放该请求，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 CPU 缓存系统中传统未命中状态保持寄存器 [Kroft, 1981] 支持的功能类似。NVIDIA 专利 [Minkin et al., 2012, Nyland et al., 2011] 中描述了至少两个版本的待处理请求表。图 4.1 中显示的与 L1 缓存架构相关的版本看起来与传统的 MSHR 有点相似。数据缓存的传统 MSHR 包含缓存未命中的块地址以及块偏移和相关寄存器的信息，这些信息需要在块填充到缓存中时写入。通过记录多个块偏移和寄存器，可以支持对同一块的多次未命中。图 4.1 中的 PRT 支持将两个请求合并到同一个块，并记录通知指令管道哪些延迟的内存访问需要重放所需的信息。

图 4.1 中显示的 L1 数据缓存是虚拟索引和虚拟标记的。与大多数采用虚拟索引/物理标记 L1 数据缓存的现代 CPU 微架构相比，这可能令人惊讶。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 and Patterson，2011]。虽然 GPU 会在 warp 发出的每个周期有效地执行上下文切换，但 warp 是同一应用程序的一部分。即使 GPU 一次只能运行一个 OS 应用程序，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 中分配条目后，内存请求将转发到内存管理单元 (MMU) 8 进行虚拟到物理地址转换，然后通过交叉互连转发到适当的内存分区单元。如第 4.3 节中所述，内存分区单元包含一组 L2 缓存以及一个内存访问调度程序。除了有关访问哪个物理内存地址以及读取多少字节的信息之外，内存请求还包含一个“ subid ”，当内存请求返回核心时，可以使用该“ subid ”来查找 PRT 中包含有关该请求的信息的条目。

一旦将加载的内存请求响应返回到核心，MMU 就会将其传递给填充单元 9 。填充单元依次使用内存请求中的 subid 字段在 PRT 中查找有关该请求的信息。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，以重新安排加载，然后通过在将加载放入数据阵列 5 后锁定缓存中的行来保证加载命中缓存。

缓存写入操作

图 4.1 中的 L1 数据缓存可以支持直写和回写策略。因此，可以以多种方式处理全局内存的存储指令（写入）。特定内存

写入的空间决定了写入是被视为直写还是回写。许多 GPGPU 应用程序中对全局内存的访问可能具有非常差的时间局部性，因为内核的编写方式通常是线程在退出之前将数据写入大型数组。对于此类访问，直写而不使用写入分配 [Hennessy and Patterson, 2011] 策略可能有意义。相比之下，将寄存器溢出到堆栈的本地内存写入可能表现出良好的时间局部性，后续加载证明使用写入分配策略回写是合理的 [Hennessy and 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[redacted]某些线程被屏蔽时，只会写入缓存块的一部分。如果缓存中存在该块，则可以通过数据交叉开关 6 将数据写入数据阵列。如果缓存中不存在该数据，则必须首先从 L2 缓存或 DRAM 内存中[redacted]取该块。如果合并写入完全填充缓存块，并且它们使缓存中任何陈旧数据的标签无效，则它们可能会绕过缓存。

请注意，图 4.1 中描述的缓存组织确实支持缓存一致性。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 数据缓存中逐出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 仅允许本地内存访问寄存器溢出和堆栈数据或只读全局内存数据放置在 L1 数据缓存中。最近的研究探索了如何在 GPU 上启用一致的 L1 数据缓存 [Ren and Lis，2017，Singh et al.，2013] 以及对明确定义的 GPU 内存一致性模型的需求 [Alglave et al.，2015]。

4.1.2 L1 纹理缓存

NVIDIA 的最新 GPU 架构将 L1 数据缓存和纹理缓存相结合以节省空间。为了更好地理解这种缓存的工作原理，首先需要了解一些独立纹理缓存的设计。这里介绍的细节应该有助于提供有关如何开发吞吐量处理器微架构的更多直觉。这里的大部分讨论都基于 Igehy 等人的一篇论文 [1998]，该论文旨在填补有关工业纹理缓存设计如何容忍缓存未命中的长片外延迟的文献不足。最近的行业 GPU 专利 [Minken et al., 2010, Minken and Rubinstein, 2003] 描述了密切相关的设计。由于本书的重点不是图形，我们仅简要概述了促使包含纹理缓存的纹理操作。

在 3D 图形中，希望使场景看起来尽可能逼真。为了在实时渲染所需的高帧速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，将称为纹理的图像应用于 3D 模型中的表面，以使表面看起来更逼真。例如，纹理可用于使场景中的桌子呈现天然木材的外观。为了实现

纹理映射渲染管道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。然后使用坐标查找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取平均值，因此纹理内存访问中存在明显的局部性，缓存可以利用这一点 [Hakura and Gupta，1997]。

图 4.2 说明了 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 数据缓存不同，标记数组 2 和数据数组 5 由 FIFO 缓冲区 3 分隔。此 FIFO 的动机是隐藏可能需要从 DRAM 提供服务的未命中请求的延迟。本质上，纹理缓存的设计假设缓存未命中会频繁发生，并且工作集大小相对较小。为了保持标记和数据数组的大小较小，标记数组基本上在数据数组之前运行。标记数组的内容反映了在经过大约等于未命中请求到内存并返回的往返时间的一段时间后数据数组的内容。虽然吞吐量相对于容量和未命中处理资源有限的常规 CPU 设计有所提高，但缓存命中和未命中的延迟大致相同。

具体来说，图 4.2 中所示的纹理缓存的操作如下。加载/存储单元 1 发送计算出的纹素地址以在标签数组 2 中执行查找。如果访问命中，则指向数据数组中数据位置的指针将与完成纹理操作所需的任何其他信息一起放置在片段 FIFO 3 尾部的条目中。当条目到达片段 FIFO 的头部时，控制器 4 使用指针从数据数组 5 中查找纹素数据并将其返回到纹理过滤单元 6 。虽然没有详细显示，但对于双线性和三线性过滤（mipmap 过滤）等操作，每个片段（即屏幕像素）实际上有四次或八次纹素查找。纹理过滤单元将纹素组合起来以产生单个颜色值，该颜色值通过寄存器文件返回到指令管道。

如果在标签查找期间发生缓存未命中，标签阵列将通过未命中请求 FIFO 8 发送内存请求。未命中请求 FIFO 将请求发送到内存系统 9 的较低级别。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。这些技术可以无序处理内存请求，以减少行切换惩罚。为了确保数据阵列 5 的内容反映标签阵列 2 的时间延迟状态，必须按顺序从内存系统返回数据。这是使用重新排序缓冲区 10 来实现的。

4.1.3 统一纹理和数据缓存

在 NVIDIA 和 AMD 的最新 GPU 架构中，数据和纹理值的缓存是使用统一的 L1 缓存结构进行的。为了以最直接的方式实现这一点，只有可以保证只读的数据值才会缓存在 L1 中。对于遵循此限制的数据，除了对寻址逻辑进行更改外，几乎无需修改即可使用纹理缓存硬件。最近的一项专利 [Heinrich et al.,

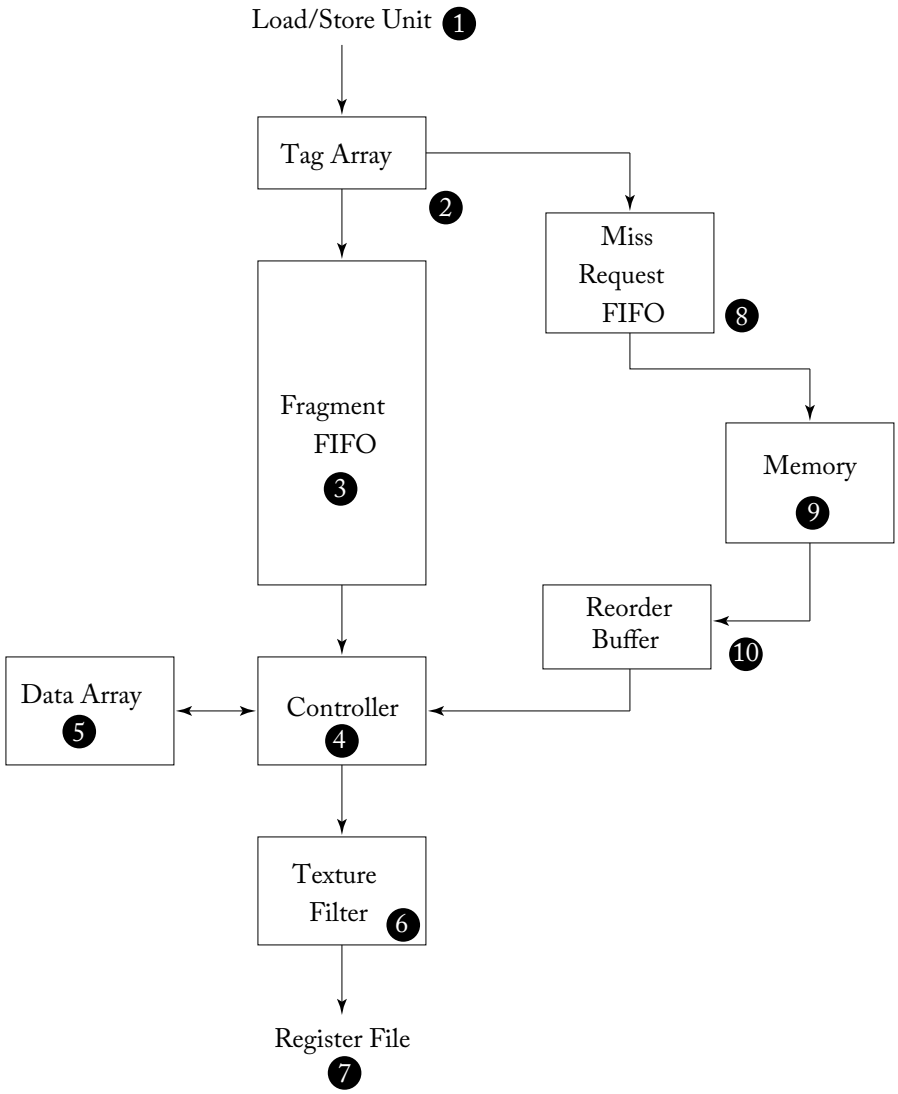


图 4.2：L1 纹理缓存（部分基于 [Igehy et al., 1998] 中的图 2）。

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

为了提供 SIMT 核心所需的大量内存带宽，高性能 GPU 通过内存分区单元并行连接到多个 DRAM 芯片（如第 4.3 节所述）。内存流量使用地址交错分布在内存分区单元中。NVIDIA 的一项专利描述了地址交错方案，用于在粒度为 256 字节或 1,024 字节的最多 6 个内存分区之间平衡流量 [Edmondson and Van Dyke，2011]。

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

4.3 内存分区单元

下面，我们描述了与最近的几项 NVIDIA 专利相对应的内存分区单元的微架构。从历史背景来看，这些专利是在 NVIDIA Fermi GPU 架构发布前一年左右提交的。如图 4.3 所示，每个内存分区单元包含一部分二级 (L2) 缓存以及一个或多个内存访问调度程序（也称为“ 帧缓冲区 ”或 FB）和一个光栅操作 (ROP) 单元。L2 缓存包含图形和计算数据。内存访问调度程序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 单元还支持原子操作，如 CUDA 编程模型中的原子操作。这三个单元紧密耦合，下面将详细介绍。

4.3.1 二级缓存

L2 缓存设计包括多项优化，以提高 GPU 单位面积的总吞吐量。每个内存分区内的 L2 缓存部分由两个切片组成 [Edmondson 等人，2013]。每个切片包含单独的标签和数据阵列，并按顺序处理传入请求 [Roberts 等人，2012]。为了匹配 GDDR5 中 32 字节的 DRAM 原子大小，切片内的每个缓存行都有四个 32 字节扇区。缓存行分配给存储指令或加载指令使用。为了在写入未命中时完全覆盖每个扇区的合并写入的常见情况下优化吞吐量，首先不会从内存中读取任何数据。这与标准计算机架构教科书中通常描述的 CPU 缓存完全不同。我们研究的专利中没有描述如何处理未完全覆盖扇区的未合并写入，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程序的面积，正在写入内存的数据被缓冲在 L2 中的缓存行中，同时写入等待调度。

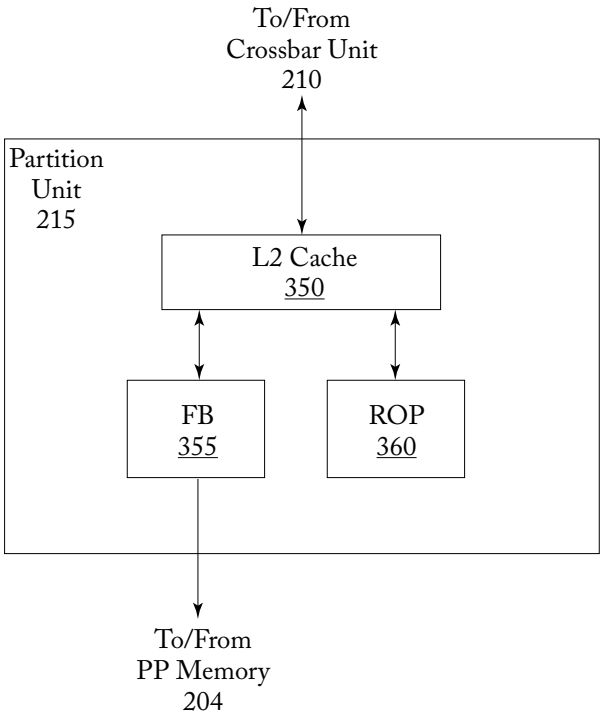


图 4.3：内存分区单元概览（基于 Edmondson 等人 [2013] 中的图 3B）。

4.3.2 原子操作

如 Glasco 等人 [2012] 所述，ROP 单元包括用于执行原子和归约操作的功能单元。由于 ROP 单元包含本地 ROP 缓存，因此可以对访问同一内存位置的一系列原子操作进行流水线处理。原子操作可用于实现在不同线程块中运行的线程之间的同步。

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)，例如 GDDR5 gdd。DRAM 将各个位存储在小型电容器中。例如，为了从这些电容器读取值，首先将一行位（称为页面）读入称为行缓冲器的小型存储器结构中。为了完成此操作，必须首先将连接各个存储电容器和行缓冲器的位线（本身具有电容）预充电到介于 0 和电源电压之间的一半电压。当在激活操作期间通过访问晶体管将电容器连接到位线时，随着电荷从位线流入或流出存储单元，位线的电压会被略微上拉或下拉。感测

然后，放大器放大这个微小的变化，直到读取到干净的逻辑 0 或 1。将值读入行缓冲区的过程会刷新存储在电容器中的值。预充电和激活操作会引入延迟，在此期间无法读取或写入 DRAM 阵列的任何数据。为了减轻这些开销，DRAM 包含多个存储体，每个存储体都有自己的行缓冲区。然而，即使有多个 DRAM 存储体，也常常不可能完全隐藏访问数据时在行之间切换的延迟。这导致使用内存访问调度程序[Rixner et al.，2000；Zuravleffi and Robinson，1997]来重新排序 DRAM 内存访问请求，以减少必须在行缓冲区和 DRAM 单元之间移动数据的次数。

为了能够访问 DRAM，GPU 中的每个内存分区可能包含多个内存访问调度程序 [Keil and Edmondson，2012]，将其包含的 L2 缓存部分连接到片外 DRAM。最简单的方法是让 L2 缓存的每个片段都有自己的内存访问调度程序。每个内存访问调度程序都包含单独的逻辑，用于对从 L2 缓存发送的读取请求和写入请求（“脏数据通知”）进行排序 [Keil et al.，2012]。为了将读取分组到 DRAM 组中的同一行，使用了两个单独的表。第一个表称为读取请求分类器，是一个通过内存地址访问的组关联结构，它将给定组中同一行的所有读取请求映射到单个指针。该指针用于在第二个表（称为读取请求存储库）中查找单个读取请求的列表。

4.4 GPU内存系统的研究方向

4.4.1 内存访问调度和互连网络设计

Yuan 等人 [2009] 探索了运行用 CUDA 编写的 GPU 计算应用程序的 GPU 的内存访问调度程序设计。他们观察到单个流式多处理器 (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一系列内存请求在序列中相邻的请求访问同一 DRAM 组中的同一 DRAM 行，则称该序列具有行缓冲区局部性。但是，当来自一个 SM 的内存请求发送到内存分区时，它们会与来自其他 SM 的请求混合在一起，这些请求向同一内存分区发送请求。结果是进入内存分区的请求序列的行缓冲区局部性较低。Yuan 等人 [2009] 建议通过修改互连网络以保持行缓冲区局部性来降低内存访问调度的复杂性。他们通过引入仲裁策略来实现这一点，这些策略优先授予包含来自同一 SM 或具有相似行缓冲区地址的内存请求的数据包。

Bakhoda 等人 [2010, 2013] 探索了 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，将有必要采用可扩展拓扑，例如网格。他们探索了片上网络设计如何影响系统吞吐量，并发现

许多 CUDA 应用程序的吞吐量对互连延迟相对不敏感。他们分析了互连流量，发现它具有多对少对多的模式。他们提出了一种由“ 半路由器 ”组成的更受限制的可扩展拓扑，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] 研究了使用他们的 GPGPU-Sim 模拟器模拟的对支持 CUDA 的 GPU 添加 L1 和/或 L2 缓存以进行全局内存访问的影响，并表明虽然一些应用程序受益，但其他应用程序却没有。

Jia 等人 [2012] 随后的研究通过启用或禁用 NVIDIA Fermi GPU 硬件上的缓存来描述缓存的有效性，并发现了类似的结果。据观察，通过 L1 缓存将数据读入暂存器共享内存的应用程序不会从启用 L1 缓存中受益。即使排除此类应用程序，Jia 等人 [2012] 也观察到单靠缓存命中率不足以预测缓存是否会提高性能。他们发现，有必要考虑缓存对 L2 缓存（例如，内存分区）请求流量的影响。在他们研究的 Fermi GPU 上，L1 缓存没有分区，因此启用缓存会在未命中时引发更大的 128 字节片外内存访问。在内存带宽有限的应用程序中，这种额外的片外内存流量可能会导致性能下降。Jia 等人[2012] 引入了三种局部性的分类法：warp 内、块内和跨指令。当由单个 warp 内的不同线程执行的单个加载的内存读取访问访问同一缓存块时，就会发生 warp 内局部性。当由同一线程块的不同 warp 中的线程执行的单个加载的内存读取访问访问同一缓存块时，就会发生块内局部性。当由同一线程块中的线程执行的不同加载指令的内存读取访问访问同一缓存块时，就会发生跨指令局部性。Jia 等人 [2012] 引入了一种使用此分类法的编译时算法，以帮助推断何时启用缓存对各个加载指令有帮助。

4.4.3 内存请求优先级和缓存绕过

继上述特性研究 [Jia et al., 2012] 和 Rogers et al. [2012] 的研究（证明了 warp 调度可以提高缓存效率（见第 5.1.2 节））之后，Jia et al. [2014] 提出了用于 GPU 的内存请求优先级和缓存绕过技术。相对于线程数而言关联性较低的缓存可能会出现严重的冲突未命中 [Chen and Aamodt, 2009]。Jia et al. [2014] 指出，用 CUDA 编写的几个 GPGPU 应用程序包含数组索引，当使用标准模缓存集索引函数时，这会导致来自单个 warp 的单个内存请求导致映射到同一缓存集的未命中 [Hennessy and Patterson, 2011, Patterson and Hennessy, 2013]。Jia et al. [2014] 将此称为内部 Warp 争用。假设在

检测到未命中[1]，且有限数量的未命中状态保持寄存器[2]，warp 内争用可能导致内存流水线停顿。[3] 为了解决 warp 内争用问题，Jia 等人 [2014] 建议在发生未命中且由于关联性停顿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都被保留以提供空间用于由未解决的缓存未命中提供的数据时，就会发生关联性停顿。

Jia 等人 [2014] 还研究了他们所谓的跨 Warp 争用问题。当一个 Warp 逐出另一个 Warp 带来的数据时，就会发生这种形式的缓存争用。为了解决这种形式的争用问题，Jia 等人 [2014] 建议采用一种他们称之为“ 内存请求优先级缓冲区 ”（MRPB）的结构。与 CCWS [Rogers et al., 2012] 一样，MRPB 通过修改访问缓存的顺序来增加局部性，从而减少容量未命中。然而，与通过线程调度间接实现这一点的 CCWS 不同，MRPB 试图通过在线程调度后更改各个内存访问的顺序来增加局部性。

MRPB 在第一级数据缓存之前实现内存请求重新排序。MRPB 的输入是在执行内存请求合并之后在指令发出管道阶段生成的内存请求。MRPB 的输出将内存请求输入到第一级缓存中。在内部，MRPB 包含几个并行的先进先出 (FIFO) 队列。缓存请求使用“ 签名 ”分发到这些 FIFO。在评估的几个选项中，他们发现最有效的签名是使用“ warp ID ”（一个介于 0 到流式多处理器上可以运行的最大 warp 数之间的数字）。MRPB 采用“ 排出策略 ”来确定从哪个 FIFO 中选择内存请求以用于下一次访问缓存。在探索的几个选项中，最好的版本是一个简单的固定优先级方案，其中每个队列都分配有一个静态优先级，并且包含请求的最高优先级队列首先得到服务。

详细评估表明，使用 MRPB 的绕过和重新排序组合机制在 64 路 16 KB 上实现了 4% 的几何平均加速。Jia 等人 [2014] 还与 CCWS 进行了一些比较，显示出更大的改进。我们顺便指出，Rogers 等人 [2012] 的评估采用了基线架构和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，随后，Nugteren 等人 [2014] 对 NVIDIA Fermi 架构中使用的实际集合索引哈希函数的细节进行了逆向工程，发现它使用了 XOR-ing（这也倾向于减少此类冲突）。

与 Rogers 等人 [2013] 类似，Jia 等人 [2014] 表明，他们采用的对程序员透明的性能提升方法可以缩小使用缓存的简单代码与使用暂存器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] 探索了基于静态指令中内存发散程度的绕过和改变缓存行大小的影响。他们使用观察到的重用距离模式和内存发散程度来预测绕过和最佳缓存行大小。

Lee 和 Wu [2016] 提出了一种基于控制环路的缓存绕过方法，该方法试图在运行时逐条预测重用行为。缓存行的重用行为受到监控。如果特定程序计数器加载的缓存行没有经历足够的重用，则绕过该指令的访问。

4.4.4 利用经线间的异质性

Ausavarungnirun 等人 [2015] 提出了一系列改进 GPU 共享 L2 和内存控制器的方法，以缓解不规则 GPU 应用中的内存延迟差异。这些技术统称为内存差异校正 (MeDiC)，利用了以下观察结果：同一内核中各个 Warp 的内存延迟差异水平存在差异。根据它们与共享 L2 缓存的交互方式，内核中的每个 Warp 可分为全部/大部分命中、全部/大部分未命中或平衡。作者表明，拥有并非全部命中的 Warp 几乎没有好处，因为大部分命中的 Warp 必须等待最慢的访问返回后才能继续。他们还表明，L2 缓存的排队延迟会对性能产生不小的影响，可以通过对所有非全部命中的 Warp 的所有请求（即使是可能命中的请求）绕过 L2 缓存来减轻这种影响。通过减少排队延迟，这可以降低全命中 Warp 的访问延迟。除了自适应绕过技术之外，他们还建议修改缓存替换策略和内存控制器调度程序，以尽量减少检测到的全命中 Warp 的延迟。他们还证明，即使对于全命中 Warp，L2 缓存库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 库之间的排队延迟不平衡。

作者提出的微架构机制由四个部分组成：（1）一个扭曲类型检测块 - 将 GPU 中的扭曲分类为五种潜在类型之一：全部未命中、大部分未命中、平衡、大部分命中或全部命中；（2）一个扭曲类型感知旁路逻辑块，用于决定请求是否应绕过 L2 缓存；（3）一个扭曲类型感知插入策略，用于确定 L2 中的插入在 LRU 堆栈中的什么位置；（4）一个扭曲类型感知内存调度程序，用于排序如何将 L2 未命中/旁路发送到 DRAM。

检测机制通过按间隔对每个 Warp 的命中率（总命中数/访问数）进行采样来运行。根据此比率，Warp 会采用上面列出的五种分类之一。确定这些分类边界的准确命中率会针对每个工作负载进行动态调整。在分类间隔期间，没有请求会绕过缓存来对每个 Warp 的 L2 特性中的相位变化做出反应。

旁路机制位于 L2 缓存的前面，接收标记为生成它们的 warp 类型的内存请求。此机制尝试消除来自全未命中 warp 的访问，并将大部分未命中 warp 转换为全未命中 warp。该块只是将所有标记为来自全未命中和大部分未命中 warp 的请求直接发送到内存调度程序。

MeDiC 的缓存管理策略通过改变从 DRAM 返回的请求在 L2 的 LRU 堆栈中的位置来运行。大部分未命中的 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程序，使其包含两个内存访问队列：一个用于全命中和大部分命中的 Warp 的高优先级队列，以及一个用于平衡、大部分未命中和全未命中 Warp 的低优先级队列。内存调度程序只是将高优先级队列中的所有请求优先于低优先级队列中的任何请求。

4.4.5 协调缓存绕过

Xie 等人 [2015] 探索了选择性启用缓存旁路以提高缓存命中率的潜力。他们使用分析来确定 GPGPU 应用程序中的每个静态加载指令是否具有良好局部性、较差局部性或中等局部性。他们相应地标记指令。标记为具有良好局部性的加载操作可以使用 L1 数据缓存。标记为具有较差局部性的加载操作始终被绕过。标记为中等局部性的加载指令采用自适应机制，其工作方式如下。自适应机制以线程块粒度运行。对于给定的线程块，所有执行的中等局部性加载都得到统一处理。它们要么使用 L1，要么旁路。行为是在启动线程块时根据阈值确定的，该阈值使用考虑了 L1 缓存命中和管道资源冲突的性能指标在线调整。他们的评估表明，这种方法比静态扭曲限制更能提高缓存命中率。

4.4.6 自适应缓存管理

Chen 等人 [2014b] 提出了协调的缓存绕过和 warp 节流，利用 warp 节流和缓存绕过的优点来提高高度缓存敏感应用程序的性能。所提出的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 缓存绕过保护距离技术实现缓存绕过，这可防止缓存行因多次访问而被逐出。在插入缓存后，该行会被分配一个保护距离，并且计数器会跟踪行的剩余保护距离。一旦剩余保护距离达到 0，该行将不再受保护并可被逐出。当新的内存请求尝试将新行插入到没有未受保护行的集合中时，内存请求会绕过缓存。

保护距离是全局设置的，最佳值因工作负载而异。在本文中，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

Li 等人 [2015] 观察到，warp 节流优化了 L1 缓存命中率，但可能会导致其他资源（如片外带宽和 L2 缓存）的利用率大大降低。他们提出了一种向 warp 分配令牌的机制，以确定哪些 warp 可以将行分配到 L1 缓存中。额外的“ 无污染 warp ”没有被赋予令牌，因此虽然它们可以执行，但不允许从 L1 中逐出数据。这导致了一个优化空间，其中可调度的 warp 数量 (W) 和具有令牌的数量 (T) 都可以设置为小于可执行的最大 warp 数量。他们表明，静态选择 W 和 T 的最优值可以使性能比使用静态 warp 限制的 CCWS 提高 17%。

基于这一观察，Li 等人 [2015] 探索了两种机制来学习 W 和 T 的最佳值。第一种方法基于在提高缓存命中率的同时保持高线程级并行性的理念。在这种方法中，称为 dynPCALMTLP，采样周期运行一个内核，其中 W 设置为最大 warp 数，然后在不同的 SIMT 核心之间改变 T。然后选择实现最大性能的 T 值。这可以实现与 CCWS 相当的性能，但面积开销明显更少。第二种方法称为 dynPCALCCWS，最初使用 CCWS 设置 W，然后使用 dynPCALMTLP 确定 T。然后它监视共享结构的资源使用情况以动态增加或减少 W。与 CCWS 相比，这可以使性能提高 11%。

4.4.8 虚拟内存页面布局

Agarwal 等人 [2015] 考虑在包括容量优化和带宽优化内存的异构系统中支持跨多种物理内存类型的缓存一致性的影响。由于针对带宽优化的 DRAM 比针对容量优化的 DRAM 成本更高、能耗更高，因此未来的系统可能会同时包含这两种内存。Agarwal 等人 [2015] 观察到当前的操作系统页面放置策略（例如在 Linux 中部署的策略）并未考虑内存带宽的不均匀性。他们研究了一种未来的系统，其中 GPU 可以以低延迟访问低带宽/高容量 CPU 内存——损失 100 个核心周期。他们的实验使用配置了额外 MSHR 资源的修改版 GPGPU-Sim 3.2.2 来模拟更新的 GPU。

通过这种设置，他们首先发现，对于内存带宽受限的应用程序，通过使用 CPU 和 GPU 内存来增加总内存带宽，有很大的机会获得性能提升。他们发现内存延迟较少的 GPGPU 应用程序并非如此。在假设页面访问均匀且带宽优化内存的内存容量不受限制的情况下，他们表明分配

按照区域可用内存带宽的比例将页面分配到内存区域是最佳的。假设带宽受限内存的容量不是问题，他们发现一个简单的策略，即按照内存带宽的比例将页面随机分配到带宽或容量优化的内存中，这种策略在实际的 GPGPU 程序中是可行的。然而，当带宽优化的内存容量不足以满足应用程序需求时，他们发现有必要优化页面布局以考虑访问频率。

为了优化页面布局，他们提出了一个系统，该系统涉及使用 NVIDIA 开发工具 nvcc 和 ptxas 的修改版本以及现有 CUDA API 的扩展来实现的分析过程，以包含页面布局提示。使用配置文件引导的页面布局提示可获得 Oracle 页面布局算法的约 90% 的好处。他们将页面迁移策略留待将来研究。

4.4.9 数据放置

Chen 等人 [2014a] 提出了一种可移植数据放置策略 PORPLE，它由规范语言、源到源编译器和自适应运行时数据放置器组成。他们利用了以下观察：由于 GPU 上有各种类型的内存，程序员很难决定将哪些数据放置在何处，而且通常无法从一个 GPU 架构移植到另一个 GPU 架构。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。

第一个解决方案是内存规范语言，以帮助实现可扩展性和可移植性。内存规范语言根据对这些空间的访问进行序列化的条件来描述 GPU 上各种形式的内存。例如，对相邻全局数据的访问是合并的，因此可以同时访问，但对同一共享内存组的访问必须进行序列化。

第二种解决方案是名为 PORPLE-C 的源到源编译器，它将原始 GPU 程序转换为与位置无关的版本。编译器在内存访问周围插入保护，选择与预测的最佳数据位置相对应的访问。

最后，为了预测哪种数据放置方式最为理想，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跟踪运行时访问模式并尝试进行预测。此函数在 CPU 上运行一小段时间，有助于在启动内核之前确定最佳的基于 GPU 的数据放置方式。在本工作范围内，系统仅处理数组的放置，因为它们是 GPU 内核中最常用的数据结构。

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事务数量的估计值。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。当多个阵列共享一个缓存时，对每个阵列分配多少缓存的估计是基于基于阵列大小的缓存的线性分区。

4.4.10 多芯片模块 GPU

Arunkumar 等人 [2017] 指出，摩尔定律的放缓将导致 GPU 性能提升放缓。他们建议通过在多芯片模块上用较小的 GPU 模块构建大型 GPU 来扩展性能扩展（见图 4.4）。他们证明，通过结合远程数据的本地缓存、考虑局部性的模块 CTA 调度和首次接触页面分配，可以实现单个大型（且不可实现）单片 GPU 的 10% 性能。根据他们的分析，这比使用相同工艺技术的最大可实现单片 GPU 的性能高出 45%。

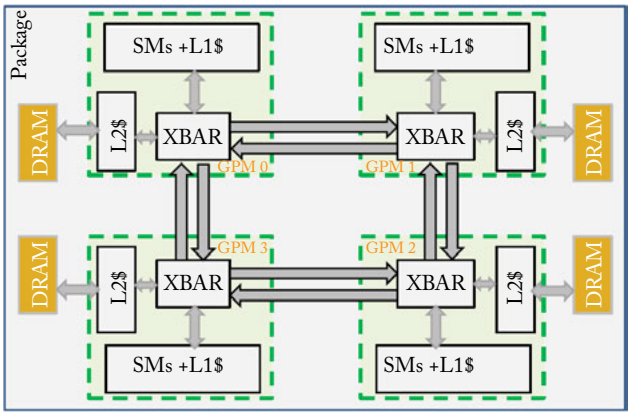


图 4.4：多芯片模块 GPU（基于 Arunkumar 等人 [2017] 的图 3）。

CHAPTER 5

GPU 计算架构的交叉研究

本章详细介绍了 GPGPU 架构中的几个研究方向，这些方向与之前专注于 GPU 架构特定部分的章节不太契合。第 5.1 节探讨了如何在 GPU 中调度线程。第 5.2 节介绍了替代编程方法，第 5.4 节研究了异构 CPU/GPU 计算方面的工作。

5.1 线程调度

当代 GPU 与 CPU 的根本区别在于它们依赖于大规模并行性。无论程序是如何指定的（例如，使用 OpenCL、CUDA、OpenACC 等），没有大量软件定义并行性的工作负载都不适合 GPU 加速。GPU 采用多种机制来聚合和调度所有这些线程。GPU 上的线程组织和调度主要有三种方式。

将线程分配到 Warp 由于 GPU 使用 SIMD 单元来执行由 MIMD 编程模型定义的线程，因此必须将线程融合在一起，以 Warp 的形式进行锁步执行。在本书研究的基线 GPU 架构中，具有连续线程 ID 的线程会静态融合在一起以形成 Warp。第 3.4.1 节总结了有关 Warp 内替代线程排列的研究提案，以实现更好的 Warp 压缩。

将线程块动态分配给核心 与 CPU 不同，在 CPU 中，线程可以一次分配给一个硬件线程，而在 GPU 中，工作是批量分配给 GPU 核心的。此工作单元由多个线程块形式的 warp 组成。在我们的基准 GPU 中，线程块按循环顺序分配给核心。核心的资源（如 warp 槽、寄存器文件和共享内存空间）以线程块粒度订阅。由于每个线程块都与大量状态相关，当代 GPU 不会抢占它们的执行。线程块中的线程运行完毕后，才能将其资源分配给另一个线程块。

逐周期调度决策将线程块分配给 GPU 核心后，一组细粒度的硬件调度程序将在每个周期决定要使用哪组 Warp

获取指令，它会扭曲以发出需要执行的指令，以及何时读取/写入每个发出的指令的操作数。

调度多个内核 线程块级和逐周期调度决策既可以在内核中进行，也可以在同一 GPU 上同时运行的不同内核之间进行。传统内核调度限制一次只能在 GPU 上激活一个内核。但是，NVIDIA 的 Streams 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 上的多道编程。

5.1.1 线程块与核心的分配研究

启动内核时，每个内核启动中的线程被分组为线程块。GPU 范围的线程块调度机制根据资源可用性将每个线程块分配给 SIMT 内核之一。每个内核都有固定数量的暂存器内存（在 CUDA 中称为共享内存，在 OpenCL 中称为本地内存）、寄存器数量、用于 warp 的插槽和用于线程块的插槽。在内核启动时，每个线程块的所有这些参数都是已知的。最明显的线程块调度算法是以循环方式将线程块分配给内核，以最大化所涉及的内核数量。线程块被连续调度，直到每个内核中的至少一个资源耗尽。请注意，内核可能由比 GPU 上一次可以运行的线程块更多的线程块组成。因此，内核中的一些线程块甚至可能没有在 GPU 上运行，而其他线程块则在执行。有几种研究技术研究了线程块调度空间中的权衡。

线程块级别的节流。Kayiran 等人 [2013] 建议限制分配给每个内核的线程块数量，以减少线程超额订阅导致的内存系统争用。他们开发了一种监控内核空闲周期和内存延迟周期的算法。该算法首先为每个内核分配其最大线程块的一半。然后监控空闲和内存延迟周期。如果内核主要在等待内存，则不会再分配线程块，并且现有线程块可能会暂停以阻止它们发出指令。该技术实现了一种粗粒度并行节流机制，即使同时活动的 CTA 较少，也可以限制内存系统干扰并提高整体应用程序性能。

动态调整 GPU 资源。Sethia 和 Mahlke [2014] 提出了 Equalizer，这是一种硬件运行时系统，可动态监控资源争用情况并调整线程数、核心频率和内存频率，以改善能耗和性能。该系统根据四个参数做出决策：(1) SM 中的活动 Warp 数；(2) 等待内存数据的 Warp 数；(3) 准备执行算术指令的 Warp 数；(4) 准备执行内存指令的 Warp 数。利用这些参数，它首先决定 SM 上要保持活动的 Warp 数，然后根据此值和其他三个计数器（充当内存的代理）的值

通过权衡竞争、计算强度和内存强度，它决定了如何最好地扩展核心和内存系统的频率。

均衡器有两种运行模式：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源，以最大限度地降低能耗，同时减轻其对性能的影响。在性能增强模式下，均衡器以节能的方式增强瓶颈资源，从而提高性能。

他们通过检查与更改内存频率、计算频率和同时运行的线程数相关的性能和能源权衡，将 Rodinia 和 Parboil 的一组工作负载描述为计算密集型、内存密集型、缓存敏感型或不饱和型。如果目标是最小化能源（而不牺牲性能），那么计算密集型内核应该以较低的内存频率运行，内存内核应该以较低的 SIMT 核心频率运行。这有助于减少系统中不必要消耗的能源，这些能源在基准速率下没有得到充分利用。

均衡器会根据间隔做出有关频率和并发性的决策。该技术为每个 SIMT 核心添加监控硬件，该硬件会根据前面列出的四个计数器做出本地决策。它在每个 SIMT 核心中本地决定此时期的三个输出参数（CTA 数量、内存频率和计算频率）应该是什么。它会通知全局工作分配引擎此 SM 应使用的 CTA 数量，如果 SIMT 核心需要更多工作，则会发出新块。如果 SM 应该使用更少的 CTA 运行，它会暂停核心上的某些 CTA。在决定要运行的 CTA 数量后，每个 SIMT 核心都会向全局频率管理器提交内存/计算电压目标，该管理器根据多数函数设置芯片范围的频率。

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 数量来做出本地决策。如果尝试等待内存的 Warp 数量大于 CTA 中的 Warp 数量，则在此 SIMT 核心上运行的 CTA 数量会减少，从而可能有助于提高缓存敏感型工作负载的性能。如果准备发出内存（或 ALU）的 Warp 数量大于 CTA 中的 Warp 数量，则 SIMT 核心被视为内存（或计算）密集型。如果等待内存（或计算）的 Warp 数量少于 CTA 中的 Warp 数量，则如果超过一半的活动 Warp 正在等待并且等待内存的 Warp 不超过两个，则 SIMT 核心仍可被视为 ALU 或内存受限。如果是这种情况，则核心上活动 CTA 的数量将增加一，并且根据是否存在更多计算等待 warp 或更多内存等待 warp 来确定 SIMT 核心是计算限制还是内存限制。

一旦 SIMT 核心做出了本地决策，内存和核心的频率就会根据均衡器所运行的操作模式按 ±15% 进行缩放。

逐周期调度的早期特征。Lakshminarayana 和 Kim [2010] 在没有硬件管理缓存的早期 GPU 背景下探索了多种 warp 调度策略，并表明对于每个 warp 执行对称（平衡）动态指令数的应用程序，基于公平性的 warp 和 DRAM 访问调度策略可以提高性能。该策略在他们研究中使用的常规 GPU 工作负载上效果很好，因为 warp 之间的常规内存请求都在核心内合并，并且更好地利用了 DRAM 行缓冲区局部性。本文还描述了其他几种 warp 调度策略，包括 ICOUNT，它是由 Tullsen 等人 [1996] 首次为同时多线程 CPU 提出的。ICOUNT 旨在通过优先考虑速度最快的 warp（或线程）来提高系统吞吐量。 Lakshminarayana 和 Kim [2010] 表明，在早期的无缓存 GPU 中，在早期的常规工作负载下仅优先考虑少数 Warp 通常不会提高性能。

两级调度。Gebhart 等人 [2011c] 引入了两级调度程序来提高能源效率。他们的两级调度程序将核心中的 Warp 分为两个池：一个是活动 Warp 池，其中的 Warp 将在下一个周期进行调度；另一个是非活动 Warp 池，其中的 Warp 不会在下一个周期进行调度。每当 Warp 遇到编译器识别的全局或纹理内存依赖关系时，它就会从活动池中转移出来，并以循环方式从非活动池中转移回活动池。每个周期从较小的 Warp 池中进行选择可减少 Warp 选择逻辑的大小和能耗。

Narasiman 等人 [2011] 提出的两级调度程序着重于通过允许线程组在不同时间达到相同的长延迟操作来提高性能。这有助于确保保持提取组内的缓存和行缓冲区局部性。然后，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，缓存意识 Warp 调度（见下文）着重于通过根据丢失的 Warp 内局部性程度自适应地限制系统可以维持的多线程数量来提高性能。

缓存意识 Warp 调度。Rogers 等人 [2012] 将 GPU 内核中存在的内存局部性分类为 *intra-warp*，其中 Warp 加载会引用其自己的数据，或 *inter-warp*，其中 Warp 与其他 Warp 共享数据。他们证明，Warp 内局部性是缓存敏感型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 Warp 数量来利用这种局部性。

在较少的 Warp 之间进行主动调度可使每个 Warp 占用更多的缓存空间，并减少 L1 数据缓存争用。具体而言，当具有局部性的工作负载使缓存发生抖动时，就会发生节流。为了检测这种抖动，CCWS 引入了一种基于 L1 数据缓存中的替换受害者标签的丢失局部性检测机制。

图 5.1 绘制了 CCWS 的高级微架构。每次从缓存中驱逐时，受害者的标签都会写入 warp 私有受害者标签数组。每个 warp 都有自己的受害者标签数组，因为 CCWS 只关心检测 warp 内部局部性。每次后续缓存未命中时，都会探测丢失的 warp 的受害者标签数组。如果在受害者标签中找到标签，则部分 warp 内部局部性已丢失。CCWS 假设，如果 warp 对 L1 数据缓存拥有更多的独占访问权限，则该 warp 可能能够命中此行，因此可以从节流中受益。

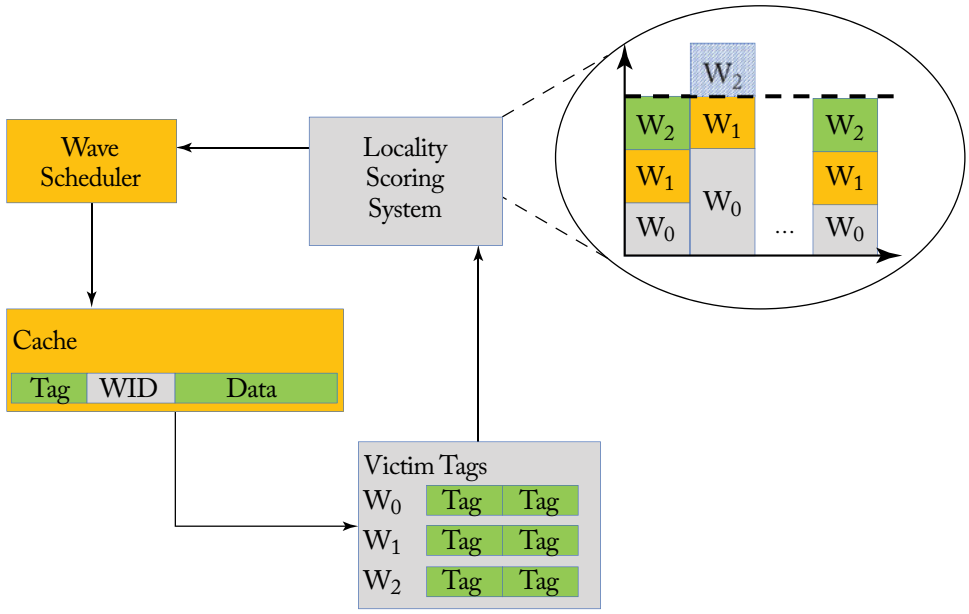


图 5.1：缓存意识的波前调度微架构。

为了反映局部性的损失，会向调度系统发送一个信号。发布调度程序使用局部性评分系统来估计系统中的每个 Warp 丢失了多少局部性，这是每个 Warp 需要多少额外缓存容量的近似值。局部性评分系统中的所有 Warp 都被分配了一个初始分数，假设所有 Warp 都需要相同的缓存容量，并且不会发生限制（图 5.1 中的堆叠条）。随着时间的推移和检测到丢失的局部性，各个 Warp 的分数会增加。在图 5.1 的示例中，Warp 0 经历了局部性的损失，其分数增加了。其分数的增加使 Warp 3 超过了阈值，这将阻止它发出 L1 数据缓存请求，从而有效地限制了核心上主动调度的 Warp 数量。随着时间的推移，如果没有丢失局部性，那么 warp 0 的分数就会减少，直到 warp 2 能够降至阈值以下并能够再次发出内存请求。

CCWS 通过将各种调度机制与缓存替换策略进行比较，进一步证明了缓存命中率对调度决策的敏感性。本文证明了 Warp 调度程序可用的决策空间比替换策略做出的相对受限的决策空间大得多。本文进一步证明了使用 LRU 替换策略的 CCWS 方案可以比以前的调度机制更好地提高缓存命中率，即使它们使用 Belady 最优缓存替换策略。

Rogers 等人 [2013] 提出了发散感知 Warp Scheduling (DAWS)，它扩展了 CCWS，可以更准确地估算每个 Warp 的缓存占用空间。DAWS 利用了 GPU 工作负载中大多数 Warp 内局部性发生在循环中的事实。DAWS 为循环中的 Warp 创建每个 Warp 的缓存占用空间估算。DAWS 根据每个 Warp 预测的循环占用空间，预先限制循环中的 Warp 数量。DAWS 还会根据每个 Warp 经历的控制流发散程度调整其缓存占用空间估算。离开循环的 Warp 中的线程不再对占用空间估算产生影响。DAWS 接着进一步探索 GPU 的可编程性方面，证明使用更智能的 Warp 调度程序，未对内存传输进行优化（例如，使用共享内存而不是缓存）的基准测试可以非常接近 GPU 优化版本的同一基准测试。

预取感知 Warp 调度。Jog 等人 [2013b] 探索了 GPU 上的预取感知 Warp 调度程序。他们基于两级调度机制设计了调度程序，但从非连续 Warp 中形成提取组。此策略增加了 DRAM 中的库级并行量，因为预取器不会查询一个 DRAM 库以进行连续访问。他们进一步扩展了这个想法，以根据 Warp 组分配来操纵预取器。通过预取其他组中 Warp 的数据，他们可以改善行缓冲区局部性，并在预取请求和数据需求之间提供间隔。

CTA 感知调度。Jog 等人 [2013a] 提出了一种 CTA 感知的 Warp 调度程序，该调度程序也基于两级调度程序，根据选择性组合的 CTA 形成提取组。他们利用了多种基于 CTA 的属性来提高性能。他们采用了一种节流优先级技术，该技术可以限制核心中的活动 Warp 数量，类似于其他节流调度程序。结合节流，他们利用不同核心上的 CTA 之间的 CTA 间页面局部性。在仅感知局部性的 CTA 调度程序下，连续的 CTA 通常会同时访问同一个 DRAM 库，从而降低库级并行性。他们将其与预取机制相结合，以改善 DRAM 行局部性。

调度对分支发散缓解技术的影响。Meng 等人 [2010] 引入了动态 Warp 细分 (DWS)，当某些通道命中缓存而某些通道未命中时，它会拆分 Warp。此方案允许命中缓存的单个标量线程即使其某些 Warp 同伴未命中也能取得进展。DWS 通过允许运行来提高性能

提前线程启动未命中，并为落后线程创建预取效果。DWS 尝试通过提高数据加载到缓存中的速率来改善 warp 内部局部性。

Fung 等人 [2007] 探索了 Warp 调度策略对其动态 Warp 形成 (DWF) 技术效果的影响。当同一 Warp 中的标量线程在分支指令上采用不同路径时，DWF 尝试通过动态创建新 Warp 来缓解控制流发散。他们提出了五种调度程序并评估了它们对 DWF 的影响

。

Fung 和 Aamodt [2011] 还提出了三种线程块优先级机制来补充他们的线程块压缩 (TBC) 技术。优先级机制尝试将同一 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，只是线程块是一起调度的，而不是按提取组进行调度。

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] 提出了 Mascar，旨在更好地将内存密集型工作负载中的计算与内存访问重叠。Mascar 由两个相互交织的机制组成。

- 内存感知 Warp 调度程序 (MAS)，当核心中的 MSHR 和 L1 未命中队列条目超额订阅时，它会优先执行单个 Warp。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，它也能使按顺序核心执行的 Warp 更快地完成计算操作，从而使优先 Warp 的计算与其他 Warp 的内存访问重叠。

- 缓存访问重新执行 (CAR) 机制。当由于低局部性访问阻碍了内存管道，导致缓存中带有数据的 warp 无法发出时，该机制可通过启用 L1 数据缓存命中 - 未命中来避免 L1 数据缓存抖动。

MAS 有两种操作模式：同等优先级 (EP) 和内存访问优先级 (MAP) 模式。系统根据 L1 MSHR 和内存未命中队列的满载程度在 EP 和 MP 之间切换。一旦这些结构几乎已满，系统就会切换到 MP 模式。MAS 包含两个队列，一个用于内存 Warp（试图发出内存指令的 Warp），另一个用于计算 Warp（试图发出其他类型指令的 Warp）。在每个队列中，Warp 都按照先贪婪后最旧的顺序进行调度。通过增强记分板来指示输出寄存器何时根据负载填满，可以跟踪内存相关指令。当观察到工作负载平衡且内存系统未超额认购时，调度程序将在 EP 模式下运行。在 EP 模式下，调度机制首先优先考虑内存 Warp。由于内存系统未超额认购，因此可以预测尽早启动内存访问将提高性能。在 MAP 模式下运行时，调度程序会优先考虑计算 Warp，以便更好地将可用计算与瓶颈内存系统重叠。只有一个内存 Warp，即 “ 所有者 Warp ”，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。

除了调度机制之外，Sethia 等人 [2015] 还表明，内存密集型内核的峰值 IPC 执行率远低于计算密集型内核。他们指出，在内存密集型应用中，由于过多的内存访问导致内存背压，SIMT 核心的加载存储单元会停顿，从而占用大量周期。当 LSU 停顿时，有相当一部分时间，就绪 warp 的数据位于 L1 数据缓存中，但 warp 无法发出，因为 LSU 备份了其他 warp 的内存请求。缓存访问重新执行 (CAR) 机制试图通过在 LSU 管道侧提供缓冲区来纠正此行为，该缓冲区存储停顿的内存指令并允许其他指令进入 LSU。仅当 LSU 没有停滞并且没有新的请求要发出时，才会处理来自重新执行队列的请求，除非重新执行队列已满，在这种情况下，重新执行队列中的访问将被优先处理，直到队列中的空间释放为止。

当重新执行队列与内存感知调度程序结合使用时，需要特别小心，因为重新执行队列中的请求可能来自优先级所有者 warp 以外的 warp。在 MAP 模式下操作时，从重新执行队列发送到 L1 的非所有者 warp 的请求如果未在 L1 中命中，则会进一步延迟。具体而言，当来自非所有者 warp 的请求在 L1 中未命中时，该请求不会中继到 L2 缓存，而是重新插入到重新执行队列的尾部。

5.1.3 多核调度研究

支持 GPU 上的抢占。Park 等人 [2015] 解决了支持 GPU 上的抢占式多任务处理的挑战。它采用了更宽松的幂等性定义，以支持线程块内的计算刷新。更宽松的幂等性定义涉及从线程执行开始时检测执行是否是幂等的。他们的提案 Chimera 在三种方法中动态选择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，则只需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，一种算法会估算当前正在运行的线程块子集，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，warp 调度的影响。他们使用真实的 GPU 硬件证明了当线程旋转等待锁时会产生大量开销。他们指出，如果单纯地退出包含未能获取锁的线程的 warp 的执行，可能会阻止或减慢同一 warp 中已持有锁的其他线程的进度。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，而使用基于堆栈的重新收敛使这一过程更具挑战性 [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记录和单独的谓词寄存器更新历史记录来准确检测在锁上旋转的循环。为了减少争用并提高性能，他们建议在执行自旋循环的反向分支时降低被识别为执行自旋循环的线程束的优先级，因为在线程束中持有锁的任何线程都释放了这些锁之后，线程束才会执行自旋循环。他们发现，与 Lee 和 Wu [2014] 相比，这分别提高了性能并降低了 1.5 个 v0 和 1.6 个 v1 的能耗。

5.2 表达并行性的替代方法

细粒度工作队列。Kim 和 Batten [2014] 建议在 GPU 中的每个 SIMT 核心中添加一个细粒度硬件工作列表。他们利用了以下观察结果：不规则的 GPGPU 程序通常在使用数据驱动方法在软件中实现时表现最佳，其中工作是动态生成并在线程之间平衡的，而不是使用拓扑方法，其中启动固定数量的线程 — 其中许多线程不做有用的工作。数据驱动方法有可能提高工作效率和负载平衡，但如果没有广泛的软件优化，可能会出现性能不佳的情况。本文提出了一种片上硬件工作列表，支持核心内和核心之间的负载平衡。他们使用线程等待机制并按间隔重新平衡线程生成的任务。他们在 lonestar GPU 基准测试套件中利用拓扑和数据驱动的工作分布的各种不规则应用程序实现上评估了他们的硬件机制。

内核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) 线程推送生成的工作时内存系统中的争用和 (2) 基于线程 ID 静态划分工作导致的负载平衡不佳。不依赖静态划分的软件实现在推送和拉取时都会遭遇内存争用。静态划分工作解决了拉取争用问题。硬件工作列表分布在多个结构中，从而减少了争用。它通过在线程空闲之前动态地将生成的工作重新分配给线程来改善负载平衡。作者向 ISA 添加了特殊指令，用于从硬件队列推送和拉取。内核中的每个通道都分配有一个小型单端口 SRAM，用作给定通道使用和生成的工作 ID 的存储。

本文提出了一种基于间隔和需求驱动（仅根据推送/拉取请求进行重新分配）的工作重新分配方法，并对前者进行了深入评估。基于间隔的方法根据简单的阈值或更复杂的排序重新分配工作。阈值方法将工作量超过阈值的通道归类为贪婪（工作量过多），将工作量少于阈值的通道归类为贫困（工作量不足）。然后，排序过程将工作从贪婪的银行重新分配给贫困的银行。基于排序的技术更复杂，但可以实现更好的负载平衡，因为所有贫困的银行也可以将工作捐赠给其他贫困的银行。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，该架构还支持虚拟化硬件工作列表，使其可扩展到生成比硬件结构中可用容量更多的动态工作的工作负载。

基于嵌套并行模式的编程。Lee 等人 [2014a] 提出了一种 GPU 上嵌套并行模式的局部感知映射，该映射利用了以下观察结果：嵌套并行计算到 GPU 线程没有普遍最优的映射。具有嵌套并行性的算法（例如 map/reduce 操作）可以在不同级别将其并行性暴露给 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种概括：

- 一维映射，将顺序程序的外循环并行化；
- 线程块/线程映射，将顺序程序外循环的每次迭代分配给一个线程块，并将内层模式在线程块中并行化；
- 基于 warp 的映射，将外循环的每次迭代分配给一个 warp，并在整个 warp 中并行化内层模式。

本研究提出了一个自动编译框架，该框架根据嵌套模式中公开的局部性和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。这些模式由集合操作组成，例如 map、reduce、foreach、filter 等。该框架尝试将线程映射到集合中每个元素上的操作。该框架通过首先将应用程序中的每个嵌套级别分配给一个维度（x、y、z 等）来处理模式的嵌套。双重嵌套模式（即包含 Reduce 的 Map）有两个维度。然后，映射确定 CUDA 线程块中给定维度的线程数。在设置线程块的维度和大小之后，框架通过使用线程跨越和拆分的概念为每个线程分配多个元素，进一步控制内核中的并行度。在二维内核（即两层模式嵌套）中，如果每个维度都分配有 span(1)，则内核中启动的每个线程只负责操作集合中的一个元素。此映射可实现最大程度的并行性。相反，span(all) 表示每个线程都操作集合中的所有元素。span 可以是 (1) 和 (all) 之间的任何数字。Span(all) 用于

在两种特殊情况下：当维度的大小直到内核启动后才知道（例如，当内部模式中操作的元素数量是动态确定的）以及当模式需要同步时（例如，reduce 操作）。

由于 span(all) 会严重限制暴露的并行性并导致 GPU 利用率不足，因此框架还提供了 *split* 的概念。split(2) 表示每个线程对给定维度中的一半元素进行操作（将其视为 span(all)/2）。使用 split 时，框架会启动第二个内核（称为组合器内核）来聚合拆分的结果，产生的结果与使用 span(all) 对内核进行分区时的结果相同。

为了选择每个维度的块大小和每个维度的拆分/跨度，该框架使用基于硬约束和软约束的评分算法。该算法会扫描整个搜索空间的所有可能维度、块大小和跨度。搜索空间随循环嵌套的级别呈指数增长。但是，指数的底数小于 100，典型的内核包含的级别少于 3。因此，只需几秒钟即可完全搜索该空间。搜索会修剪违反硬约束的配置 - 即导致执行不正确的配置，例如块中的最大线程数过高。它为软约束分配加权分数，例如确保将模式顺序内存访问分配给 x 维度以改善内存合并。

该框架还执行两项常见的 GPU 优化：在嵌套内核中预分配内存而不是动态分配全局内存，以及在确定将数据预取到共享内存对嵌套模式有益时使用共享内存。结果表明，自动生成的代码具有与专家调整的代码相媲美的性能。

动态并行。Wang 和 Yalamanchili [2014] 描述了在 Kepler GPU 硬件上使用 CUDA 动态并行的开销，并发现这些开销可能非常大。具体而言，他们确定了限制他们所研究的工作负载效率的几个关键问题。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（略多于一个 warp）。第三，虽然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，第四，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在一起导致利用率非常低。

Wang 等人 [2016a] 随后提出了动态线程块启动 (DTBL)，该模型修改了 CUDA 编程模型，使设备启动的内核能够共享硬件队列资源，从而实现更高的并行性和更好地利用 GPU 硬件。他们的提案的关键是使动态启动的内核能够与运行相同代码的现有内核聚合在一起。这通过维护修改后的硬件在启动内核时使用的聚合线程块的链接列表来支持。他们

通过修改 GPGPU-Sim 来评估 DTBL，发现 DTBL 的性能相对于 CDP 提高了 1.4×，相对于不使用 CDP 的高度调整的 CUDA 版本提高了 1.2×。

Wang 等人 [2016b] 随后探索了动态启动线程块被调度到哪个 SM 的影响。他们发现，通过鼓励将子线程块调度到与父 SM 相同的 SM 上，同时考虑跨 SM 的工作负载分配，他们能够将性能提高 27%，而使用简单的循环分配机制则不行。

5.3 对事务内存的支持

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

这些提议的动机是 TM 编程模型的潜力，它可以减轻管理具有充足不规则并行性的 GPU 应用程序中线程之间不规则、细粒度通信的挑战。在现代 GPU 上，应用程序开发人员可以通过屏障来粗化线程之间的同步，或者他们可以尝试使用许多现代 GPU 上提供的单字原子操作来实现这些通信的细粒度锁定。前一种方法可能涉及对底层算法的重大更改，而后一种方法涉及细粒度锁定的开发工作的不确定性，对于实际的、市场驱动的软件开发来说风险太大（除了几个例外）。在 GPU 上启用 TM 简化了同步，并提供了一个强大的编程模型，该模型促进了细粒度通信和并行工作负载的强大扩展。TM 的这一承诺希望鼓励软件开发人员探索这些不规则应用程序的 GPU 加速。

在 GPU 上支持 TM 的独特挑战。GPU 的高度多线程特性为 TM 系统设计带来了一系列新挑战。GPU 上的 TM 系统旨在扩展到数万个小型并发事务，而不是运行数十个占用空间相对较大的并发事务（这是最近多核处理器 TM 研究的重点）。这反映了 GPU 的高度多线程特性，数万个线程协同工作，每个线程执行一项小任务以实现共同目标。这些小事务以字级粒度进行跟踪，从而实现比缓存块更精细的冲突检测分辨率。此外，GPU 中的每个每核私有缓存都由数百个 GPU 线程共享。这大大降低了利用缓存一致性协议检测冲突的好处，这种技术在大多数为具有大 CPU 核心的传统 CMP 设计的硬件事务内存中都采用。

5.3.1 千克

Kilo TM [Fung et al.，2011] 是第一个发布的针对 GPU 架构的硬件 TM 提案。

Kilo TM 采用基于价值的冲突检测 [Dalessandro et al., 2010, Olszewski et al., 2007]，从而消除了冲突检测对全局元数据的需求。每笔交易

只需读取全局内存中的现有数据进行验证，以确定它是否与另一个已提交的事务发生冲突。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任何直接交互，并以最精细的粒度检测冲突。

但是，基于价值的冲突检测的本机实现要求事务串行提交。为了提高提交并行性，Kilo TM 吸收了现有 TM 系统 [Chafi 等，2007；Spear 等，2008] 中的思想，并通过创新解决方案对其进行了扩展。特别是，Fung 等人 [2011] 引入了 *recency bloom filter*，这是一种新颖的数据结构，它使用时间和顺序的概念来压缩大量小项目集。Kilo TM 使用这种结构来压缩所有提交事务的写入集。每个提交事务都会向新近布隆过滤器查询一组近似的冲突事务——该冲突集中的一些事务是误报。Kilo TM 使用这些近似信息来安排数百个非冲突事务进行验证并并行提交。新近布隆过滤器的这种近似性质使其能够保持较小，大约为几 kB，因此它可以驻留在片上以便快速访问。使用新近布隆过滤器来提高事务提交并行性是 Kilo TM 的一个组成部分。

分支发散和事务内存。事务内存编程模型引入了一种新型的分支发散。当一个 warp 完成事务时，其每个活动线程都将尝试提交。一些线程可能会中止并需要重新执行其事务，而其他线程可能会通过验证并提交其事务。由于这个结果在整个 warp 中可能不一致，因此 warp 可能会在验证后发散。Fung 等人 [2011] 提出了一种对 SIMT 硬件的简单扩展，以处理由事务中止引入的这种特定类型的分支发散。此扩展与 Kilo TM 的其他设计方面无关，但它是支持 GPU 上的 TM 的必要部分。

图 5.2 显示了如何扩展 SIMT 堆栈以处理由于事务中止而导致的控制流分歧。当 warp 进入事务时（在行 B，`tx_begin`），它会将两个特殊条目推送到 SIMT 堆栈 1 。类型 R 的第一个条目存储重新启动事务的信息。其活动掩码最初为空，其 PC 字段指向 `tx_begin` 之后的指令。类型 T 的第二个条目跟踪当前事务尝试。在 `tx_commit` (行 F)，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）2 时，会弹出 T 条目。然后，将使用来自 R 条目的活动掩码和 PC 将新的 T 条目推送到堆栈上，以重新启动已中止的线程。然后，清除 R 条目中的活动掩码 3。如果 R 条目中的活动掩码为空，则 T 和 R 条目都会弹出，显示原始的 N 条目 5 ，然后修改其 PC 以指向紧接在 `tx_commit` 之后的指令，并且 warp 恢复正常执行。事务内 warp 的分支发散的处理方式与非事务发散 4 相同。

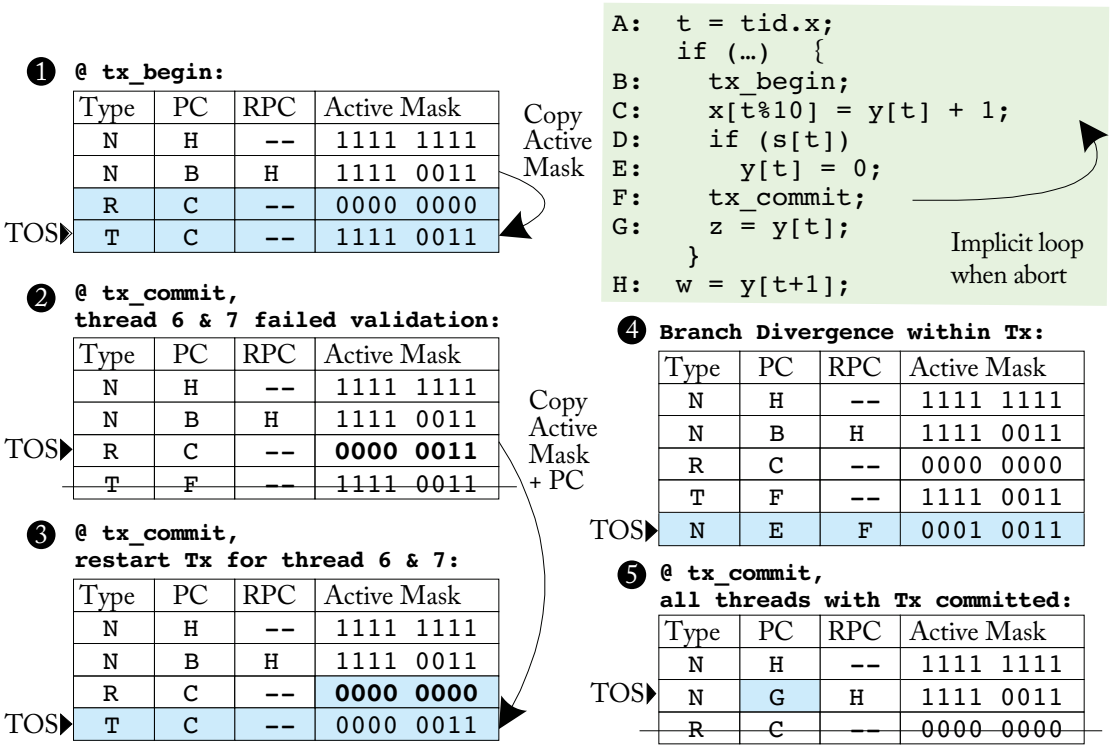


图 5.2：SIMT 堆栈扩展用于处理由于事务中止（验证失败）而导致的分歧。线程 6 和 7 验证失败并重新启动。堆栈条目类型：正常 (N)、事务重试 (R)、事务顶部 (T)。对于每个场景，添加的条目或修改的字段都用阴影表示。

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] 提出了两项不同的增强技术来提高 Kilo TM 的性能和能源效率：warp 级事务管理 (WarpTM) 和时间冲突检测 (TCD)。

Warp 级事务管理利用 GPU 编程模型中的线程层次结构（即 Warp 内线程之间的空间局部性）来提高 Kilo TM 的效率。具体而言，WarpTM 可以摊销 Kilo TM 的控制开销，并提高 GPU 内存子系统的效用。这些优化只有在可以有效解决 Warp 内的冲突时才有可能实现，因此低开销的 Warp 内冲突解决机制对于保持 WarpTM 的优势至关重要。为此，Fung 和 Aamodt [2013] 提出了一种两阶段并行的 Warp 内冲突解决机制，可以高效地并行解决 Warp 内的冲突。在解决所有 Warp 内冲突后，Kilo TM 可以合并标量内存

验证和提交同一 warp 中的多个事务的复杂度访问被简化为更宽的访问。这种优化称为 *validation and commit coalescing*，是 Kilo TM 充分利用宽 GPU 内存系统的关键，该系统针对矢量范围访问进行了优化。

时间冲突检测是一种低开销机制，它使用一组全局同步的片上计时器来检测只读事务的冲突。初始化后，每个片上计时器都在其微架构模块中本地运行，不与其他计时器通信。这种无需通信的隐式同步将 TCD 与各种软件 TM 系统中使用的现有基于时间戳的冲突检测区分开来 [Dalessandro 等，2010；Spear 等，2006；Xu 等，2014]。TCD 使用从这些计时器捕获的时间戳来推断事务相对于其他事务更新的内存读取顺序。Kilo TM 结合 TCD 来检测无冲突的只读事务，这些事务无需基于值的冲突检测即可直接提交。这样做可以显著减少这些事务的内存带宽开销，这对于使用事务进行数据结构遍历的 GPU-TM 应用程序来说可能经常发生。

5.4 异构系统

异构系统中的并发管理。Kayiran 等人 [2014] 提出了一种并发限制方案来限制 GPU 多线程，从而减少多程序 CPU/GPU 系统中的内存和网络争用。在异构系统中，来自 GPU 的干扰可能会导致并发执行的 CPU 应用程序的性能显著下降。他们提出的线程级并行 (TLP) 限制方案观察共享 CPU/GPU 内存控制器和互连网络中的拥塞指标，以估计应在每个 GPU 核心上主动调度的 GPU 扭曲数量。他们提出了两种方案，一种只注重提高 CPU 性能，另一种试图通过平衡受约束的多线程和 CPU 干扰导致的 GPU 性能下降来优化整体系统吞吐量（CPU 和 GPU）。作者评估了 Warp 调度对平铺异构架构的性能影响，该架构的 GPU 核心与 CPU 核心比率为 2:1，这是因为 NVIDIA GPU SM 的面积大约是使用相同工艺技术的现代无序 Intel 芯片的一半。基线配置完全共享 CPU 和 GPU 之间的网络带宽和内存控制器，以最大限度地提高资源利用率。使用这种方案，作者观察到限制 GPU TLP 会对 GPU 性能产生积极和消极影响，但绝不会损害 CPU 性能。

为了提高 CPU 性能，作者引入了一种以 CPU 为中心的并发管理技术，用于监控全局内存控制器中的停顿情况。该技术分别计算由于内存控制器输入队列已满而停顿的内存请求数量，以及由于从 MC 到内核的回复网络已满而停顿的内存请求数量。这些指标在每个内存控制器上进行本地监控，并由 ag-

这些信息被聚集在一个将信息发送到 GPU 核心的集中单元中。启发式驱动方案为这些值设置了高阈值和低阈值。如果两个请求停顿计数的总和较低（基于阈值），则增加在 GPU 上主动调度的 Warp 数量。如果两个计数的总和较高，则减少活动 Warp 数量，以期通过减少 GPU 内存流量来提高 CPU 性能。

一种更平衡的技术试图最大化整个系统的吞吐量，它增强了以 CPU 为中心的方法，以解决 Warp 节流对 GPU 性能的影响。这种平衡技术监控 GPU 在并发重新平衡间隔内无法发出的周期数（其工作中为 1,024 个周期）。当前多线程限制级别的 GPU 停滞的移动平均值存储在每个 GPU 核心上，并用于确定是否应增加或减少多线程级别。平衡技术分两个阶段调节 GPU 的 TLP。在第一阶段，其操作与以 CPU 为中心的解决方案相同，其中不考虑 GPU 停滞，并且仅根据内存争用限制 GPU TLP。在第二阶段（一旦 GPU TLP 节流开始导致 GPU 性能下降，因为 GPU 对延迟的容忍度已经降低，就会开始），如果系统预测这样做会损害 GPU 性能，则停止限制 GPU 并发。此预测是通过查找目标多线程级别上 GPU 停顿的移动平均数来进行的，该平均数是从此级别的早期执行中记录下来的。如果在目标多线程级别和当前多线程级别上观察到的 GPU 停顿之间的差异超过阈值 k 值，则 TLP 级别不会降低。此 k 值可由用户设置，是指定 GPU 性能优先级的代理。

异构系统一致性。Power 等人 [2013a] 提出了一种硬件机制，以有效地支持集成系统上 CPU 和 GPU 之间的缓存一致性。他们发现，随着 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 等人，2005] 来减少传统基于缓存块的一致性目录中造成的过多目录流量。一旦获得了粗粒度区域的权限，大多数请求就不必访问目录，并且可以将一致性流量卸载到非一致性直接访问总线，而不是带宽较低的一致性互连网络。

面向 CPU-GPU 架构的异构 TLP 感知缓存管理。Lee 和 Kim [2012] 评估了在异构环境中管理 CPU 核心和 GPU 核心之间共享的末级缓存 (LLC) 的效果。他们表明，虽然缓存命中率是 CPU 工作负载的关键性能指标，但许多 GPU 工作负载对缓存命中率不敏感，因为内存延迟可能被线程级并行性所隐藏。为了确定 GPU 应用程序是否对缓存敏感，他们开发了一种每核性能采样技术，其中一些核心绕过共享 LLC，一些核心插入最近使用的位置。根据这些核心的相对性能，他们可以为其余 GPU 核心设置绕过策略，如果性能得到改善，则插入 LLC，如果性能不敏感，则绕过。

其次，他们观察到，以前以 CPU 为中心的缓存管理更倾向于访问频率更高的内核。结果表明，GPU 内核在 LLC 上产生的流量是后者的五到十倍。这增加了 GPU 的缓存容量偏向性，从而降低了 CPU 应用程序的性能。他们建议扩展以前提出的基于实用程序的缓存分区 [Qureshi and Patt，2006] 工作，以考虑 LLC 访问的相对比率。当 GPU 内核对缓存敏感时，CPU 内核的访问缓存路数分配会超过基于实用程序的缓存分区所提供的分配，以考虑 CPU 和 GPU 之间访问量级和延迟敏感性的差异。

参考书目

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

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

Top500.org 2

Tor M. Aamodt、Wilson W. L. Fung、Inderpreet Singh、Ahmed El-Shafiey、Jimmy Kwa 、Tayler Hetherington、Ayub Gubran、Andrew Boktor、Tim Rogers、Ali Bakhoda 和 Had i Jooybar。 *GPGPU-Sim 3.x Manual*。 16, 38

M. Abdel-Majeed 和 M. Annavaram。扭曲寄存器文件：GPGPU 的节能寄存器文件。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。试点寄存器文件 ：用于 GPU 的节能分区寄存器文件。在 *Proc. of the IEEE International Sympo- sium on High-Performance Computer Architecture (HPCA)*，2017 年 2 月。DOI：10.1109/h- p ca.2017.47。65

Dominic Acocella 和 Mark R. Goudy。美国专利号 7,750,915：共享内存资源中跨多个存储体存储的数据元素的并发访问（受让人：NVIDIA Corp.），2010 年 7 月。68

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

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

John R. Allen、Ken Kennedy、Carrie Porterfield 和 Joe Warren。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

Robert Alverson、David Callahan、Daniel Cummings、Brian Koblenz、Allan Porterfield 和 Burton Smith。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，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：GPU 的基于指令和内存分歧的缓存管理。在 *Proc. of the IEEE International Symposium on Workload Char- acterization (IISWC)*，2016 年。DOI：10.1109/iiswc.2016.7581276。 79

Akhil Arunkumar、Evgeny Bolotin、Benjamin Cho、Ugljesa Milic、Eiman Ebrahimi、Oreste Villa、Aamer Jaleel、Carole-Jean Wu 和 David Nellans。MCM-GPU：用于持续性能可扩展性的多芯片模块 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和标量化。在 *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*，2013 年。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。利用扭曲间异构性来提高 GPGPU 性能。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2015 年。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)* 中，第 163 – 174 页，2009 年。DOI：10.1109/ispass.2009.4919648。6、14、78

Ali Bakhoda、John Kim 和 Tor M. Aamodt。多核加速器吞吐量有效的片上网络。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 421 – 432 页，2010 年。DOI：10.1109/micro.2010.50。77

Ali Bakhoda、John Kim 和 Tor M. Aamodt。设计用于吞吐量加速器的片上网络。*ACM Transactions on Architecture and Code Optimization (TACO)*，10(3):21，2013 年。DOI：10.1145/2509420.2512429。77

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

Nicolas Brunie、Sylvain Collange 和 Gregory Diamos。同时进行分支和扭曲交织，以保持 GPU 性能。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

Ian Buck、Tim Foley、Daniel Horn、Jeremy Sugerman、Kayvon Fatahalian、Mike Houston 和 Pat Hanrahan。Brook for GPUs：图形硬件上的流计算。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，第 777 – 786 页，2004 年。DOI：10.1145/1186562.1015800。6

Brian Cabral。“ SASS ”的缩写是什么？ `https://stackoverflow.com/questions/9798258/what-is-sass-short-for`，2016 年 11 月 14 日。

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

Edwin Catmull。一种用于计算机显示曲面的细分算法。*Technical Report*，DTIC 文档，1974 年。72

Hassan Chafi、Jared Casper、Brian D. Carlstrom、Austen Mcdonald、Chi Cao、Minh Woongki Baek、Christos Kozyrakis 和 Kunle Olukotun。一种可扩展、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

Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. PORPLE: An extensible optimizer for portable data placement on GPU. In *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)*，第 329 – 340 页，2009 年。DOI：10.1109/hpca.2009.4798270。78

Xuhao Chen、Li-Wen Chang、Christopher I. Rodrigues、Jie Lv、Zhiying Wang 和 Wen-Mei Hwu。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 年。DOI：10.1007/978-3-642-14122-5_8。59

Brett W. Coon 和 John Erik Lindholm。美国专利 #7,353,369：SIMD 架构中管理发散线程的系统和方法（受让人 NVIDIA Corp.），2008 年 4 月。26、49、50

Brett W. Coon、Peter C. Mills、Stuart F. Oberman 和 Ming Y. Siu。美国专利号 7,434,032：使用具有独立内存区域和存储连续寄存器大小指示器的记分板跟踪多线程处理期间的寄存器使用情况（受让人 NVIDIA Corp.），2008 年 10 月。34

Brett W. Coon、John Erik Lindholm、Gary Tarolli、Svetoslav D. Tzvetkov、John R. Nickolls 和 Ming Y. Siu。美国专利号 7,634,621：寄存器文件分配（受让人 NVIDIA Corp.），2009 年 12 月。35

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

Luke Dalessandro、Michael F. Spear 和 Michael L. Scott。NOrec：通过废除所有权记录简化 STM。在 *Proc. of the ACM Symposium on Principles and Practices of Parallel Programming (PPoPP)*，第 67 – 78 页，2010 年。DOI：10.1145/1693453.1693464

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

Gregory Diamos、Benjamin Ashbaugh、Subramaniam Maiyuran、Andrew Kerr、吴海城和 Sudhakar Yalamanchili。 SIMD 在线程边界重新收敛。载于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 477 – 488 页，2011 年。DOI：10.1145/2155620.2155676。 26、51、54

Gregory Frederick Diamos、Richard Craig Johnson、Vinod Grover、Olivier Giroux、Jack H. Choquette、Michael Alan Fetterman、Ajay S. Tirumala、Peter Nelson 和 Ronny Meir Krashinsky。使用收敛屏障执行发散线程，2015 年 7 月 13 日。27、28、29、30

Roger Eckert。美国专利号 7,376,803：DRAM 系统的页面流分类器（受让人：NVIDIA Corp.），2008 年 5 月。73

Roger Eckert。美国专利号 9,195,618：用于调度内存请求的方法和系统（受让人：NVIDIA Corp.），2015 年 11 月。73

John H. Edmondson 和 James M. Van Dyke。美国专利号 7872657：使用分区步幅的内存寻址方案，2011 年 1 月。75

John H. Edmondson 等人，美国专利 #8,464,001：带有帧缓冲区管理脏数据提取和高优先级清理机制的缓存和相关方法，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，用于实现高效的 GPU 控制流。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)* 中，2014 年。DOI：10.1109/h- pca.2014.6835936。26、28、30、31、49、52

Ahmed ElTantawy 和 Tor M. Aamodt。SIMT 架构上的 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。用于细粒度同步的 Warp 调度。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 375 – 388 页，2018 年。DOI：10.1109/hpca.2018.00040。93

Alexander L. Minken 等人，美国专利号 7,649,538：具有高级过滤功能的可重构高性能纹理管道（受让人：NVIDIA Corp.），2010 年 1 月。72

Wilson W. L. Fung。*GPU Computing Architecture for Irregular Parallelism*。博士论文，不列颠哥伦比亚大学，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)* 中，第 25 – 36 页，2011 年。DOI：10.1109/hpca.2011.5749714。26、28、42、43、50、91

Wilson W. L. Fung 和 Tor M. Aamodt。通过时空优化实现节能的 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。动态扭曲形成和调度以实现高效的 GPU 控制流。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)* 中，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。在 *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 等人。动态扭曲形成：SIMD 图形硬件上的高效 MIMD 控制流。*ACM Transactions on Architecture and Code Optimization (TACO)*，6(2):7:1 – 7:37，2009 年。DOI：10.1145/1543753.1543756。42，49

M. Gebhart、S. W. Keckler 和 W. J. Dally。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。用于管理吞吐量处理器中线程上下文的节能机制。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2011b 年。DOI：10.1145/2000064.2000093。63 Mark Gebhart、Daniel R. Johnson、David Tarjan、Stephen W. Keckler、William J. Dally、Erik Lindholm 和 Kevin Skadron。用于管理吞吐量处理器中线程上下文的节能机制。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 235-246 页，2011c 年。DOI：10.1145/2000064.2000093。 88 Isaac Gelado、John E. Stone、Javier Cabezas、Sanjay Patel、Nacho Navarro 和 Wen-mei W. Hwu。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)*，第 347 – 358 页，2010 年。DOI：10.1145/1736020.1736059。3 Syed Zohaib Gilani、Nam Sung Kim 和 Michael J. Schulte。计算密集型 GPGPU 应用程序的节能计算。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2013 年。DOI：10.1109/hpca.2013.6522330。59、61 David B. Glasco 等人。美国专利 #8,135,926：与外部 Alu 块结合的基于缓存的原子操作控制，2012 年 3 月。76 David B. Glasco 等人。美国专利 #8,539,130：用于有效数据包传输的虚拟通道，2013 年 9 月。75 Scott Gray。NVIDIA Maxwell 架构的汇编程序。`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：远离低谷。*IEEE Computer Architecture Letters*，8(1):25 – 28，2009 年。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和分析。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 108-120 页，1997 年。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。了解通用芯片的低效率来源。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 37-47 页，2010 年。DOI：10.1145/1815961.1815968。1 Song Han、Xingyu Liu、Huizi Mao、Jing Pu、Ardavan Pedram、Mark A. Horowitz 和 William J. Dally。 EIE：压缩深度神经网络上的高效推理引擎。

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*，第二版。Morgan & Claypool，2010 年。DOI：10.1201/b11417-16。96

Steven J. Heinrich 等人。美国专利号 9,595,075：纹理硬件中的加载/存储操作（受让人：NVIDIA Corp.），2017 年 3 月。68、73

John Hennessy 和 David Patterson。*Computer Architecture—A Quantitative Approach*，第 5 版。Morgan Kaufmann，2011 年。1、10、71、72、78

Maurice Herlihy 和 J. Eliot B. Moss。事务内存：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

Jared Hoberock、Victor Lu、Yuntao Jia 和 John C. Hart。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。高效处理器架构和单元处理器。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 258 – 262 页，2005 年。DOI：10.1109/hpca.2005.26。68

Mark Horowitz、Elad Alon、Dinesh Patil、Samuel Naffiziger、Rajesh Kumar 和 Kerry Bernstein。CMOS 的扩展、功率和未来。在 *IEEE International Electron Devices Meeting*，2005 年。DOI：10.1109/iedm.2005.1609253。5

Homan Igehy、Matthew Eldridge 和 Kekoa Proudfoot。纹理缓存架构中的预取。在 *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*，第 133-ff 页，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。在 *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。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2014 年。DOI：10.1109/hpca.2014.6835938。78，79

Wenhao Jia、Kelly A Shaw 和 Margaret Martonosi。描述和改进 GPU 中按需获取缓存的使用。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：协作线程阵列感知调度技术，用于提高 GPGPU 性能。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、Ravisha nkar Iyer 和 Chita R. Das。协调 GPGPU 的调度和预取。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2013b。 DOI ：10.1145/2508148.2485951。 90

Norman P. Jouppi、Cliffi Young、Nishant Patil、David Patterson、Gaurav Agrawal、Rami nder Bajwa、Sarah Bates、Suresh Bhatia、Nan Boden、Al Borchers 等人。张量处理单元的数据中心内性能分析。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2017 年。DOI：10.1145/3079856.3080246。2

David R. Kaeli、Perhaad Mistry、Dana Schaa 和 Dong Ping Zhang。*Heterogeneous Computing with OpenCL 2.0*。Morgan Kaufmann，2015 年。10

Ujval J. Kapasi 等人。数据并行架构的高效条件操作。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 159 – 170 页，200 0 年。DOI：10.1109/micro.2000.898067。45

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

Onur Kayiran、Adwait Jog、Mahmut T. Kandemir 和 Chita R. Das。不多不少：优化 GPG PU 的线程级并行性。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2013 年。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

Shane Keil 和 John H. Edmondson。美国专利号 8,195,858：管理共享 L2 总线上的冲突，2012 年 6 月。77

Shane Keil 等人，美国专利号 8,307,165：对 Dram 的请求进行排序以实现高页面局部性，2012 年 11 月。77

Farzad Khorasani、Rajiv Gupta 和 Laxmi N. Bhuyan。在存在分歧的情况下通过协作上下文收集实现高效的扭曲执行。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2015 年。DOI：10.1145/2830772.2830796。45

J. Y. Kim 和 C. Batten。使用细粒度硬件工作列表加速 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。微架构机制利用 SIMT 架构中的值结构。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 年。DOI：10.1145/2963098。2

大卫·柯克 (David B. Kirk) 和胡文梅 (W. Hwu Wen-Mei)。*Programming Massively Parallel Processors: A Hands-onApproach*。摩根考夫曼，2016。DOI：10.1016/c2011-0-04129-7。9

John Kloosterman、Jonathan Beaumont、D. Anoushe Jamshidi、Jonathan Bailey、Trevor Mudge 和 Scott Mahlke。Regless：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。矢量线程架构。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

David Kroft。无锁定指令提取/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

Jens Krüger 和 Rüdiger Westermann。用于 GPU 实现数值算法的线性代数运算符。在 *ACM Transactions on Graphics (TOG)*，卷 22，第 908-916 页，2003 年。DOI：10.1145/882262.882363。6

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

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

Ahmad Lashgar、Ebad Salehi 和 Amirali Baniasadi。GPG-PU 逆向工程案例研究：出色的内存处理资源。*ACM SIGARCH Computer Architecture News*，43(4):15 – 21，2016 年。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。GPU 上嵌套并行模式的局部感知映射。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)* 中，2014a。DOI：10.1109/micro.2014.23。94

J. Lee 和 H. Kim。TAP：一种用于 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)* 中，2012 年。DOI：10.1109/hpca.2012.6168947。100 S. Y. Lee 和 C. J. Wu。Ctrl-C：基于指令感知控制循环的 GPU 自适应缓存旁路。在 *Proc. of the IEEE International Conference on Computer Design (ICCD)*，第 133 – 140 页，2016 年。DOI：10.1109/iccd.2016.7753271。80 Sangpil Lee、Keunsoo Kim、Gunjae Koo、Hyeran Jeon、Won Woo Ro 和 Murali Annavaram。扭曲压缩：通过寄存器压缩实现节能 GPU。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2015 年。DOI：10.1145/2749469.2750417。59、60、61 Shin-Ying Lee 和 Carole-Jean Wu。CAWS：GPGPU 工作负载的关键感知扭曲调度。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*，2014 年。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 等人。揭穿 GPU 与 CPU 100X 神话：对 CPU 和 GPU 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 年。DOI：10.1145/1815961.1816021。2

Yunsup Lee、Rimas Avizienis、Alex Bishara、Richard Xia、Derek Lockhart、Christopher Bat-ten 和 Krste Asanovi ć。探索数据并行加速器中可编程性和效率之间的权衡。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 Asanovi ć。探索 SPMD 散度管理在数据上的设计空间

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。GPUWatchch：在 GPG-PU 中实现能量优化。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 487-498 页，2013 年。DOI：10.1145/2508148.2485964。6 Adam Levinthal 和 Thomas Porter。章节 — 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 年。DOI：10.1145/800031.808581。 50 Dong Li、Minsoo Rhu、Daniel R. Johnson、Mike O’Connor、Mattan Erez、Doug Burger、Don- ald S. Fussell 和 Stephen W. Redder。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 年。DOI：10.1109/hpca.2015.7056024。82 E. Lindholm、J. Nickolls、S. Oberman 和 J. Montrym。NVIDIA Tesla：统一的图形和计算架构。*Micro, IEEE*，28(2):39 – 55，2008 年 3 月至 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 年。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：使用低端口数内存模拟多端口内存（受让人 NVIDIA Corp.），2008 年 3 月。35、38 Erik Lindholm 等人。美国专利 #9,189,242：基于信用的流式多处理器 Warp 调度（受让人 NVIDIA Corp.），2015 年 11 月。33、41 John S. Liptay。系统/360 模型 85 的结构方面，II：缓存。 *IBM Systems Journal*，7(1):15 – 21，1968 年。DOI：10.1147/sj.71.0015。70 Z. Liu、S. Gilani、M. Annavaram 和 N. S. Kim。G-Scalar：适用于节能 GPU 的经济高效的通用标量执行架构。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2017 年。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（受让人 NVIDIA Corp.），2008 年 5 月。35

Michael D. McCool、Arch D. Robison 和 James Reinders。*Structured Parallel Programming: Patterns for Efficient Computation*。Elsevier，2012 年。10

Jiayuan Meng、David Tarjan 和 Kevin Skadron。动态扭曲细分以实现集成分支和内存发散容差。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

Alexander L. Minken 和 Oren Rubinstein。美国专利号 6,629,188：纹理缓存预取数据的电路和方法（受让人：NVIDIA Corp.），2003 年 9 月。72

Alexander L. Minkin 等人，美国专利号 8,266,383：使用延迟/重放机制的缓存未命中处理（受让人：NVIDIA Corp.），2012 年 9 月。68、69、71

Alexander L. Minkin 等人，美国专利号 8,595,425：可配置多客户端缓存（受让人：NVIDIA Corp.），2013 年 11 月。68

Michael Mishkin、Nam Sung Kim 和 Mikko Lipasti。GPGPUSIM 中的“读后写”风险预防。在 *Workshop on Deplicating, Deconstructing, and Debunking (WDDD)* 中，2016 年 6 月。39、40

John Montrym 和 Henry Moreton。 GeForce 6800。*IEEE Micro*，25(2):41 – 51，2005 年。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 年。DOI：10.1145/165231.165244。9

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

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

Nvidia。*NVIDIA tesla V100 GPU architecture*。2017 年。27、31

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

NVIDIA Corp. 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 Corporation。*NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110*，a。13 NVIDIA Corporation。*NVIDIA GeForce GTX 680*，b。16 NVIDIA Corporation。*CUDA Binary Utilities*，c。16 *Parallel Thread Execution ISA (Version 6.1)*。NVIDIA Corporation，CUDA Toolkit 9.1 ed.，2017 年 11 月。14 Lars Nyland 等人。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 Corp.），2011 年 12 月。71 Marek Olszewski、Jeremy Cutler 和 J. Gregory Steffan。JudoSTM：一种动态二进制重写软件事务内存的方法。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：共享 GPU 上多任务的协作抢占。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 年。78 Gennady Pekhimenko、Vivek Seshadri、Onur Mutlu、Phillip B. Gibbons、Michael A. Kozuch 和 Todd C. Mowry。基本增量即时压缩：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。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。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，2013b。DOI：10.1145/2540708.2540747。4

M. K. Qureshi 和 Y. N. Patt。基于实用程序的缓存分区：一种低开销、高性能的运行时机制，用于对共享缓存进行分区。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 423 – 432 页，2006 年。DOI：10.1109/micro.2006.49。101

任晓伟和 Mieszko Lis。通过相对论缓存一致性实现 GPU 中的高效顺序一致性。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)* 中，第 625 – 636 页，2017 年。DOI：10.1109/hpca.2017.40。 72

Minsoo Rhu 和 Mattan Erez。CAPRI：预测 GPGPU 架构中处理控制发散的压缩充分性。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。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 591 – 602 页，2013a 年。DOI：10.1109/hpca.2013.6522352。49

Minsoo Rhu 和 Mattan Erez。使用 SIMD 通道置换最大化 GPGPU 中的 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)*，第 128-138 页，2000 年。DOI：10.1109/isca.2000.854384。77 James Roberts 等人。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

Timothy G. Rogers、Mike O’Connor 和 Tor M. Aamodt。缓存意识波前调度。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2012 年。DOI：10.1109/micro.2012.16。33、78、79、88

Timothy G. Rogers、Mike O’Connor 和 Tor M. Aamodt。发散感知扭曲调度。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2013 年。DOI：10.1145/2540708.2540718。79，90

Timothy G. Rogers、Daniel R. Johnson、Mike O’Connor 和 Stephen W. Keckler。可变扭曲尺寸架构。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2015 年。DOI：10.1145/2749469.2750410。53

Sangmin Seo、Gangwon Jo 和 Jaejin Lee。OpenCL 中 NAS 并行基准的性能表征。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：通过减少内存中断来加速 GPU 扭曲。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 174 – 185 页，2015 年。DOI：10.1109/hpca.2015.7056031。91

Ankit Sethia 和 Scott Mahlke。均衡器：动态调整 GPU 资源以实现高效执行。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)* 中，2014 年。DOI：10.1109/micro.2014.16。86

Ryan Shrout。AMD ATI radeon HD 2900 XT 评论：R600 上市。*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。在 *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 架构的缓存一致性。在 *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和验证策略。在 *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：具有单个原子指令的可扩展事务。在 *Proc. of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*，第 275 – 284 页，2008 年。DOI：10.1145/1378533.1378583。97 Michael Steffin 和 Joseph Zambreno。通过对动态微内核的架构支持提高全局渲染算法的 SIMT 效率。在 *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。多线程处理器的按需寄存器分配和释放，2011 年 6 月 30 日。美国专利申请 12/649,238。64 Sean J. Treichler 等人。美国专利 #9,098,383：支持多种流量类型的整合交叉开关，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：可实现同步多线程处理器上的指令获取和发出。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)* 中，1996 年。DOI：10.1145/232973.232993。88 Rafael Ubal、Byunghyun Jang、Perhaad Mistry、Dana Schaa 和 David Kaeli。Multi2Sim：用于 CPU-GPU 计算的模拟框架。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)* 中，第 335-344 页，2012 年。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 压缩实现 SIMD 发散优化。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 368 – 379 页，2013 年。DOI：10.1145/2485922.2485954。44、46 Wladimir J. van der Laan。Decuda。`http://wiki.github.com/laanwj/decuda/` 14 Jin Wang 和 Sudhakar Yalamanchili。非结构化 GPU 应用中动态并行性的特征和分析。在 *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*，第 51 – 60 页，2014 年。DOI：10.1109/iiswc.2014.6983039。 95 Jin Wang、Norm Rubin、Albert Sidelnik 和 Sudhakar Yalamanchili。动态线程块启动：一种轻量级执行机制，用于支持 GPU 上的不规则应用程序。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 上动态并行的局部感知调度程序。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 年。DOI：10.1145/3079856.3080205。58、61、62 D. Wong、N. S. Kim 和 M. Annavaram。使用内部扭曲操作数值相似性近似扭曲。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2016 年。DOI：10.1109/hpca.2016.7446063。60、61 X. Xie、Y. Liang、Y. Wang、G. Sun 和 T. Wang。协调 GPU 的静态和动态缓存旁路。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2015 年。DOI：10.1109/hpca.2015.7056023。81

Yunlong Xu, Rui Wang, Nilanjan Goswami, Tao Li, Lan Gao, and Depei Qian. Soft- ware tra nsactional memory for GPU architectures. In *Proc. of the ACM/IEEE Interna- tional Symposium on Code Generation and Optimization (CGO)*, pages 1:1 – 1:10, 2014. DOI: 1 0.1145/2581122.2544139. 99

Y. Yang、P. Xiang、M. Mantor、N. Rubin、L. Hsu、Q. Dong 和 H. Zhou。SIMT 架构中灵活标量单元的案例。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。多核加速器架构的复杂度有效内存访问调度。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 34-44 页，2009 年。DOI：10.1145/1669112.1669119。77

侯云清。NVIDIA FERMI 汇编程序。`https://github.com/hyqneuron/asfermi` 16 Eddy Z. Zhang、Yunlian Jiang、Ziyu Guo 和 Xipeng Shen。在运行中简化 GPU 应用程序：通过运行时线程数据重新映射消除线程发散。在 *Proc. of the ACM International Conference on Supercomputing (ICS)*，第 115-126 页，2010 年。D OI：10.1145/1810085.1810104。45 William K. Zuravleffi 和 Timothy Robinson。美国专利 #5,630,096：同步 DRAM 控制器，通过允许无序发出内存请求和命令来最大化吞吐量，1997 年 5 月 13 日。77

作者简介

TOR M. AAMODT

Tor M. Aamodt 是不列颠哥伦比亚大学电气与计算机工程系的教授，自 2006 年起担任该系教员。他目前的研究重点是通用 GPU 的架构和节能计算，最近又研究了机器学习加速器。他与研究小组的学生一起开发了广泛使用的 GPGPU-Sim 模拟器。他的三篇论文被 *IEEE Micro Magazine* 选为“ 最佳论文 ”，第四篇被选为“ 最佳论文 ”荣誉奖。他的一篇论文还被 *Communications of the ACM.* 选为“ 研究亮点 ”。他已入选 MICRO 名人堂。他曾于 2012 年至 2015 年担任 *IEEE Computer Architecture Letters* 副主编，并于 2012 年至 2016 年担任 *International Journal of High Performance Computing Applications* 副主编，还曾担任 ISPASS 2013 计划主席、ISPASS 2014 总主席，并曾担任多个计划委员会成员。他于 2012 年至 2013 年担任斯坦福大学计算机科学系客座副教授。他曾于 2010 年获得 NVIDIA 学术合作伙伴奖，2016 年至 2019 年获得 NSERC 发现加速器奖，并于 2016 年获得 Google 教师研究奖。

Tor 在多伦多大学获得了理学学士（工程科学）、理学硕士和博士学位。他的大部分博士研究工作都是在英特尔微架构研究实验室实习期间完成的。随后，他在 NVIDIA 工作，负责 GeForce 8 系列 GPU 的内存系统架构（“ 帧缓冲区 ”）——这是第一款支持 CUDA 的 NVIDIA GPU。

Tor 是一名不列颠哥伦比亚省注册的专业工程师。

冯伟伦

Wilson Wai Lun Fung 是三星电子奥斯汀研发中心 (SARC) 高级计算实验室 (ACL) 的架构师，他为下一代 GPU IP 的开发做出了贡献。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 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 *IEEE Micro Magazine.* 选为计算机架构“ 首选 ”Wilson 在英属哥伦比亚大学获得了学士（计算机工程专业）、硕士和博士学位。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 (Timothy G. Rogers) 是普渡大学电气与计算机工程系的助理教授，他的研究重点是大规模多线程处理器设计。他对探索能够提高程序员工作效率和能源效率的计算机系统和架构很感兴趣。蒂莫西是 NVIDIA 研究生奖学金和 NSERC Alexander Graham Bell 加拿大研究生奖学金的获得者。他的作品被 *IEEE Micro Magazine* 选为计算机架构类别的“首选”，并被 *Communications of the ACM.* 选为“研究亮点”。在攻读博士学位期间，蒂莫西在 NVIDIA 研究中心和 AMD 研究中心实习。在进入研究生院之前，蒂莫西曾在 Electronic Arts 担任软件工程师，并在麦吉尔大学获得电气工程学士学位。