MORGAN & CLAYPOOL PUBLISHERS

通用图形处理器架构

托尔·M·阿莫德特 威尔逊·伟伦·冯 蒂莫西·G·罗杰斯

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE

Margaret Martonosi, *Series Editor*

通用图形处理器架构

计算机体系结构综合讲座

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

创始名誉编辑 Mark D. Hill，

University of Wisconsin, Madison

Synthesis Lectures on Computer Architecture 发布 50 到 100 页的出版物，内容涉及设计、分析、选择和互连硬件组件以创建符合功能、性能和成本目标的计算机的科学与艺术。其范围主要涵盖顶级计算机架构会议的领域，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

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

虚拟内存的体系结构和操作系统支持 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

时空计算与时间神经网络 James E. Smith 2017

硬件和软件对虚拟化的支持 Edouard Bugnion, J
ason Nieh 和 Dan Tsafrir 2017

数据中心设计与管理：计算机架构师的视角 本杰明·C李 2016

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

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

分析分析学 Rajesh Bordawekar、Bob Blain
ey 和 Ruchir Puri 2015

可定制计算 Yu-Ting Chen, Jason Cong, Michael Gill, Glenn Reinman
和 Bingjun Xiao 2015

堆叠式架构 袁曦和赵继
深 2015

单指令多数据执行 Christopher J. Hughes 2
015

高效能计算机体系结构：最新进展 Magnus Själander、
Margaret Martonosi 和 Stefanos Kaxiras 2014

基于FPGA的计算机系统仿真 Hari Angepat, Derek Chi
ou, Eric S. Chung 和 James C. Hoe 2014

硬件预取入门 巴巴克·法尔萨菲
和 托马斯·F·温尼施 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

计算机体系结构中的优化与数学建模 Tony Nowatzki、Michael Ferris、Karthike
yan Sankaralingam、Cristian Estan、Nilay Vaish 和 David Wood 2013

计算机架构师的安全基础 Ruby B. Le
e 2013

数据中心即计算机：仓库级机器设计导论，第二版 作者：Luiz André Barroso
、Jimmy Clidaras 和 Urs Hölzle 2013

共享内存同步 Michael L. Scott
2013

针对电压变化的弹性架构设计 Vijay Janapa Red
di 和 Meeta Sharma Gupta 2013

多线程架构 马里奥·涅米罗夫斯基
和 迪恩·M·图尔森 2013

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

自动并行化：基础编译器技术概述 Samuel P. Midkiff 2012

相变存储器：从设备到系统 Moinuddin K. Qureshi、Sudha
nva 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）》 作者：Tzvetan S.
Metodi、Arvin I. Faruque 和 Frederic T. Chong 2011年

高性能数据中心网络：架构、算法和机会 Dennis Abts 和 John Kim 2011

处理器微架构：实现视角 Antonio González、Fernando L
atorre 和 Grigorios Magklis 2010

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

计算机体系结构性能评估方法 Lieven Eeckhout 2010

可重构超级计算导论 Marco Lanzagorta、Stephen
Bique 和 Robert Rosenberg 著 2009

片上网络 Natalie Enright Jerger 和 L
i-Shiuan Peh 2009

记忆系统：你无法避免，它不可忽视，也无法伪造 Bruce Jacob 2009

容错计算机体系结构 Daniel J. Sorin
2009

数据中心即计算机：仓库规模机器设计导论

路易斯·安德烈·巴罗索和乌尔斯·
赫尔茨勒 2009

计算机体系结构的功耗效率技术 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

事务内存 James R. Larus 和
Ravi Rajwar 2006

量子计算机体系结构 Tzvetan S. Metodi 和 Fr
ederic 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 ISBN：9781627056182 电子书 ISBN：97816 81733586 精装

DOI 10.2200/S00848ED1V01Y201804CAC044

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

讲座 #44
系列编辑：Margaret Martonosi，*Princeton University* 创始编辑荣誉退休：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

托尔·M·阿莫德
University of British Columbia
威尔逊·伟伦·冯
Samsung Electronics
提摩西·G·罗杰斯
Purdue University

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #44

MORGAN & CLAYPOOL PUBLISHERS

摘要

最初为支持视频游戏而开发的图形处理器单元（GPU），现在越来越多地用于通用（非图形）应用，从机器学习到加密货币的挖掘。通过将更多的硬件资源用于计算，GPU 相较于中央处理器单元（CPU）能够实现更高的性能和效率。此外，与专用领域的加速器相比，其通用的可编程性使得当代 GPU 对软件开发者更具吸引力。本书为那些对支持通用计算的 GPU 架构感兴趣的读者提供了一份入门指南，汇集了目前仅存在于各种分散来源的信息。作者领导了广泛应用于 GPU 架构学术研究的 GPGPU-Sim 模拟器的开发工作。

本书第一章描述了GPU的基本硬件结构，并简要概述了其历史。第二章总结了与本书其余部分相关的GPU编程模型。第三章探讨了GPU计算核心的架构。第四章探讨了GPU内存系统的架构。在描述现有系统架构之后，第三章和第四章概述了相关研究。第五章总结了对计算核心和内存系统均有影响的跨领域研究。

本书应为希望了解用于加速通用应用程序的图形处理单元 (GPU) 架构的人提供宝贵的资源，也为想要了解快速增长的研究领域如何改进这些 GPU 架构的人提供入门指导。

关键词

GPGPU，计算机架构

内容

Preface

本书旨在为那些希望了解图形处理单元（GPU）架构并初步了解如何改进其设计的相关研究的人提供帮助。假设读者熟悉计算机架构的概念，例如流水线和缓存，并且对与GPU架构相关的研究和/或开发感兴趣。此类工作通常侧重于不同设计之间的权衡，因此本书的写作目的是提供关于这些权衡的见解，从而帮助读者避免通过试错来学习那些经验丰富的设计师已经掌握的知识。

为了实现这一目标，本书将许多目前分散在各种不同来源（如专利、产品文档和研究论文）中的相关信息汇集成一个资源。我们希望这能帮助刚开始进行自己研究的学生或从业者缩短成为高效工作者所需的时间。

虽然本书涵盖了当前GPU设计的某些方面，但也试图“综合”已发表的研究成果。这在某种程度上是出于必要，因为供应商对具体GPU产品的微架构几乎未曾透露。在描述“基准”GPGPU架构时，本书既依赖于已发表的产品描述（期刊论文、白皮书、手册），也在某些情况下参考了专利中的描述。专利中提到的细节可能与实际产品的微架构存在显著差异。在某些情况下，微基准测试研究为研究人员澄清了一些细节，但在其他情况下，我们的基准是基于公开可用信息的“最佳猜测”。尽管如此，我们认为这将有所帮助，因为我们的重点是理解已经被研究过的或可能在未来研究中值得探索的架构权衡。

本书的多个部分集中于总结近年来关于改进GPU架构这一主题的众多研究论文。由于近年来该主题的受欢迎程度显著提高，本书无法涵盖所有内容。因此，我们不得不在涵盖的内容和遗漏的内容之间做出艰难的选择。

托尔·M·阿莫德（Tor M. Aamodt）、冯伟伦（Wilson Wai Lun Fung）和蒂莫西·G·罗杰斯（Timothy G. Rogers），2018年4月

致谢

我们要感谢我们的家人在撰写这本书期间给予的支持。此外，我们感谢我们的出版商 Michael Morgan 和编辑 Margaret Martonosi，在这本书成型过程中表现出的极大耐心。我们还要感谢 Carole-Jean Wu、Andreas Moshovos、Yash Ukidave、Aamir Raihan 和 Amruth Sandhupatla，他们为本书早期草稿提供了详细的反馈。最后，我们感谢 Mark Hill 分享了他关于撰写 Synthesis Lectures 的策略以及对本书的具体建议。

托尔·M阿莫德（Tor M. Aamodt）、冯伟伦（Wilson Wai Lun Fung）和蒂莫西·G·罗杰斯（Timothy G. Rogers），2018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领域。如今，GPU无处不在，从智能手机、笔记本电脑、数据中心，到超级计算机都可以发现它们的身影。实际上，对 Apple A8 应用处理器的分析表明，其集成 GPU 所占的芯片面积比中央处理器单元（CPU）内核还要多 [A8H]。对更逼真图形渲染的需求最初推动了 GPU 的创新 [Montrym and Moreton, 2005]。尽管图形加速仍然是其主要用途，GPU 正在越来越多地支持非图形计算。一个受到广泛关注的突出例子是利用 GPU 开发和部署机器学习系统的快速增长 [NVIDIA Corp., 2017]。因此，本书的重点是改进非图形应用性能和能源效率的相关特性。

本介绍章节简要概述了GPU。我们在第1.1节中首先考虑计算加速器这一更广泛类别的动机，以了解GPU与其他选项的比较。然后，在第1.2节中，我们快速概述了当代GPU硬件。最后，第1.4节提供了本书其余部分的路线图。

1.1 计算加速器的全景

几十年来，一代又一代的计算系统在每美元的性能上呈现出指数级增长。其根本原因是晶体管尺寸的缩小、硬件架构的改进、编译器技术的进步以及算法的改进共同作用。据一些估算，这些性能提升中有一半归因于晶体管尺寸的缩小，这使得设备运行速度更快 [Hennessy and Patterson, 2011]。然而，自2005年左右开始，晶体管的缩放未能遵循现在被称为Dennard缩放定律的经典规则 [Dennard et al., 1974]。一个重要的结果是，随着设备变得更小，时钟频率的提升速度变得更加缓慢。为了提升性能，需要寻找更高效的硬件架构。

通过利用硬件专用化，可以将能效提高多达 500× [Hameed 等人, 2010]。正如 Hameed 等人所示，达到这种能效提升的几个关键方面包括：转向矢量硬件（例如 GPU 中的硬件）通过消除指令处理的开销，可以提高约 10× 的能效。硬件专用化剩余的大部分提升则来源于尽量减少数据移动。

可以通过引入复杂操作来实现，这些操作在避免访问诸如寄存器文件之类的大型存储数组的同时，执行多种算术运算。

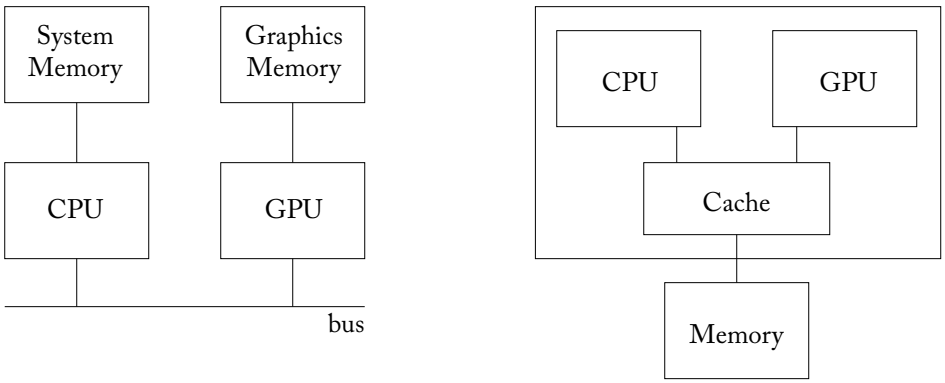
当今计算机架构师面临的一个关键挑战是找到更好的方法，在通过使用专用硬件获得效率提升与支持广泛程序所需的灵活性之间取得平衡。在没有架构的情况下，只有能够用于大量应用的算法才能高效运行。一个新兴的例子是专为支持深度神经网络而设计的硬件，例如 Google 的张量处理单元（Tensor Processing Unit）[Jouppi et al., 2017]。尽管机器学习似乎可能占据计算硬件资源的很大一部分，并且这些资源可能会迁移到专用硬件上，但我们认为，仍然需要高效支持用传统编程语言编写的软件所表达的计算。

GPU 计算在机器学习之外领域备受关注的一个原因是现代 GPU 支持图灵完备的编程模型。所谓图灵完备，我们指的是任何计算只要有足够的时间和内存都可以运行。相对于专用加速器，现代 GPU 更加灵活。对于能够充分利用 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具有吸引力。因此，现在许多顶级超算，无论是在峰值性能还是能源效率方面都使用了 GPU [top]。随着产品世代的更替，GPU 制造商已经改进了 GPU 架构和编程模型，以增加灵活性，同时提高能源效率。

1.2 GPU 硬件基础

通常，第一次接触 GPU 的人会问它们是否最终可能完全取代 CPU。这似乎不太可能。在当前系统中，GPU 并不是独立的计算设备。相反，它们与 CPU 结合在一起，要么集成在单个芯片上，要么通过将仅包含 GPU 的附加卡插入包含 CPU 的系统中实现。CPU 负责启动 GPU 上的计算并负责数据在 CPU 和 GPU 之间的传输。CPU 和 GPU 之间这种任务分工的一个原因是，计算的开始和结束通常需要访问输入/输出（I/O）设备。尽管目前正在努力开发直接在 GPU 上提供 I/O 服务的应用程序编程接口（API），但到目前为止，这些 API 都假定附近存在一个 CPU [Kim 等，2014；Silberstein 等，2013]。这些 API 的功能是通过提供便捷的接口隐藏管理 CPU 和 GPU 之间通信的复杂性，而不是完全消除对 CPU 的需求。为什么不完全消除 CPU？用于访问 I/O 设备和提供操作系统服务的软件似乎缺乏某些特性，例如大规模并行性，这些特性使得它们适合在 GPU 上运行。因此，我们从考虑 CPU 和 GPU 的交互开始分析。

一个抽象图显示了包含 CPU 和 GPU 的典型系统，如图 1.1 所示。左侧是一个典型的离散 GPU 设置，包括连接 CPU 和 GPU 的总线（例如 PCIe），适用于 NVIDIA 的 Volta GPU 等架构；右侧是一个典型的集成 CPU 和 GPU（如 AMD 的 Bristol Ridge APU 或移动 GPU）的逻辑图。请注意，包含离散 GPU 的系统具有为 CPU（通常称为系统内存）和 GPU（通常称为设备内存）分别分配的 DRAM 内存空间。这些内存使用的 DRAM 技术通常不同（CPU 使用 DDR，而 GPU 使用 GDDR）。CPU DRAM 通常优化为低延迟访问，而 GPU DRAM 则优化为高吞吐量。相比之下，集成 GPU 的系统只有一个 DRAM 内存空间，因此必然使用相同的内存技术。由于集成 CPU 和 GPU 通常出现在低功耗移动设备上，共享的 DRAM 内存通常会优化为低功耗（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 计算应用程序在 CPU 上开始运行。通常，应用程序的 CPU 部分会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 内存中分配数据结构的空间。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）提供了软件和硬件支持，能够自动将数据从 CPU 内存传输到 GPU 内存。这可以通过利用 CPU 和 GPU 上的虚拟内存支持来实现 [Gelado et al., 2010]。NVIDIA 将其称为 “ 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，不需要由程序员控制的数据从 CPU 内存到 GPU 内存的复制。然而，由于 CPU 和 GPU 使用缓存并

一些这些缓存可能是私有的，这可能会导致缓存一致性问题，这是硬件开发人员需要解决的 [Power 等人, 2013b]。

在某些时候，CPU 必须启动 GPU 上的计算。在当前系统中，这是通过运行在 CPU 上的驱动程序完成的。在 GPU 上启动计算之前，GPU 计算应用程序需要指定应该在 GPU 上运行的代码。这些代码通常被称为内核（更多细节见第 2 章）。同时，GPU 计算应用程序的 CPU 部分还需要指定应运行多少线程以及这些线程应从哪里获取输入数据。运行的内核、线程数量以及数据位置通过运行在 CPU 上的驱动程序传递给 GPU 硬件。驱动程序会翻译这些信息，并将其放置在 GPU 可访问的内存中一个 GPU 被配置为查找的位置。然后驱动程序向 GPU 发出信号，通知其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*，而 AMD 将它们称为 *compute units*。每个 GPU 核心执行一个对应于已启动运行在 GPU 上的内核的单指令多线程（SIMT）程序。每个 GPU 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存储器进行通信，并使用快速屏障操作进行同步。每个核心通常还包含一级指令和数据缓存。这些缓存充当带宽过滤器，以减少发送到内存系统较低级别的流量。当数据未命中一级缓存时，运行在核心上的大量线程被用来隐藏访问内存的延迟。

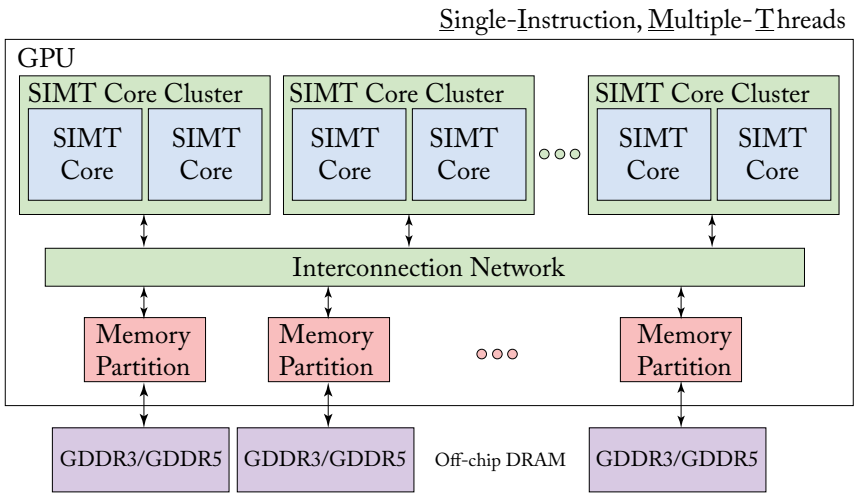


图 1.2：通用现代 GPU 架构。

为了维持高计算吞吐量，必须平衡高计算吞吐量和高内存带宽。这反过来需要在内存系统中实现并行性。

在GPU中，这种并行性通过包含多个内存通道来提供。通常，每个内存通道都与其相关联的最后一级缓存位于一个内存分区中。GPU核心和内存分区通过片上互连网络（例如交叉开关）连接。其他组织结构也是可能的。例如，直接与GPU在超级计算市场竞争的Intel Xeon Phi，将最后一级缓存分布在核心之间。

GPU 通过将更大比例的芯片面积用于算术逻辑单元（ALU），而相应减少控制逻辑的面积，在高度并行的工作负载上相较于超标量乱序 CPU 实现了每单位面积性能的提升。为了帮助理解 CPU 和 GPU 架构之间的权衡，Guz 等人 [2009] 开发了一个有洞见的分析模型，展示了性能如何随线程数量变化。为了简化模型，他们假设一个简单的缓存模型，其中线程不共享数据，且具有无限的片外内存带宽。图 1.3 复现了他们论文中的一个图表，展示了他们模型中发现的一个有趣的权衡。当一个大缓存被少量线程共享时（如多核 CPU 的情况），性能随着线程数量的增加而提高。然而，如果线程数量增加到缓存无法容纳整个工作集的程度，性能会下降。随着线程数量的进一步增加，多线程隐藏长片外延迟的能力使性能再次提高。GPU 架构在图表的右侧进行了表示。GPU 的设计目标是通过多线程来容忍频繁的缓存未命中。

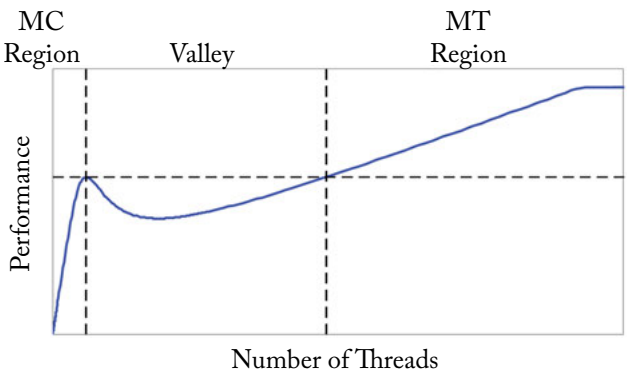


图1.3：基于分析模型的多核（MC）CPU架构与多线程（MT）架构（如GPU）性能权衡的分析显示，如果线程数量不足以覆盖片外内存访问延迟，可能会出现“性能谷”（基于Guz等人[2009]的图1）。

随着Dennard缩放结束 [Horowitz et al., 2005]，提高能效已成为计算机架构研究创新的主要驱动力。一个关键的观察是，访问大型内存结构可能消耗与计算相同甚至更多的能量。

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]。在提出新的 GPU 架构设计时，考虑能量消耗是很重要的。为此，最近的 GPGPU 架构模拟器（如 GPGPU-Sim [Bakhoda et al., 2009]）集成了能量模型 [Leng et al., 2013]。

表 1.1: 各种操作在45 nm工艺技术中的能耗（基于Han等人[2016]的表1）

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

1.3 GPU 的简史

本节简要描述了图形处理单元（GPU）的历史。计算机图形学在20世纪60年代兴起，代表性项目包括伊万·萨瑟兰的Sketchpad [Sutherland, 1963]。从最早期开始，计算机图形学就成为离线渲染动画电影以及实时渲染电子游戏发展的重要组成部分。早期的视频卡始于1981年的IBM单色显示适配器（MDA），它仅支持文本显示。后来，视频卡逐渐引入了2D加速功能，然后发展到3D加速功能。除了电子游戏，3D加速器还应用于计算机辅助设计。早期的3D图形处理器（如NVIDIA的GeForce 256）功能较为固定。NVIDIA通过引入顶点着色器 [Lindholm et al., 2001] 和像素着色器于2001年推出的GeForce 3，为GPU带来了可编程性。研究人员迅速学会了如何通过将矩阵数据映射到纹理并应用着色器，在这些早期GPU上实现线性代数运算 [Krüger and Westermann, 2003]，随后学术界致力于将通用计算映射到GPU上，使程序员无需了解图形学 [Buck et al., 2004]。这些努力激发了GPU制造商直接支持除图形处理之外的通用计算。首款支持这一功能的商业产品是NVIDIA GeForce 8系列。GeForce 8系列引入了多个创新功能，包括从着色器写入任意内存地址的能力以及通过本地存储器（scratchpad memory）来减少离芯片带宽需求，这在早期GPU中尚属空白。下一项创新是NVIDIA Fermi架构中支持读写数据缓存的功能。后续的改进包括AMD的Fusion架构，它将CPU和GPU集成到同一芯片中，以及支持动态并行化以实现更高效的计算。

从GPU本身启动线程。最近，NVIDIA的Volta引入了专为机器学习加速设计的功能，例如Tensor Cores。

1.4 书籍大纲

本书其余部分的结构如下。

在设计硬件时，重要的是要考虑它将支持的软件。因此，在第2章中，我们简要总结了编程模型、代码开发过程和编译流程。

在第 3 章中，我们探讨了支持成千上万线程执行的单个 GPU 核心的架构。我们逐步建立对支持高吞吐量和灵活编程模型所涉及权衡的日益详细的理解。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快速了解情况。

在第4章中，我们探讨了内存系统，包括GPU核心内的一级缓存和内存分区的内部组织。了解GPU的内存系统非常重要，因为运行在GPU上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 章概述了关于 GPU 计算架构的其他研究，这些研究并未完全适合第 3 章或第 4 章的内容。

CHAPTER 2

编程模型

本章的目标是提供关于如何为非图形计算编程 GPU 的足够背景，以便那些没有 GPU 经验的人可以理解后续章节的讨论。我们在这里专注于基本材料，更深入的内容可参考其他文献（例如，[Kirk 和 Wen-Mei, 2016]）。许多 GPU 计算基准套件可以用于体系结构研究。学习 GPU 的编程方法对于对 GPU 计算感兴趣的计算机架构师来说是相关的，有助于更好地理解硬件/软件接口，但如果您希望在研究中探索对硬件/软件接口进行更改，这将变得至关重要。在后一种情况下，可能不存在现有的基准，因此可能需要创建，或通过修改现有 GPU 计算应用程序的源代码来实现。例如，探索在 GPU 上引入事务内存（TM）的研究就需要这样做，因为当前的 GPU 不支持 TM（参见第 5.3 节）。

现代 GPU 使用宽 SIMD 硬件来利用 GPU 应用程序中的数据级并行性。GPU 计算 API（如 CUDA 和 OpenCL）并未直接将 SIMD 硬件暴露给程序员，而是提供了一种类似 MIMD 的编程模型，允许程序员在 GPU 上启动大量标量线程。这些标量线程中的每一个都可以遵循其独特的执行路径，并可以访问任意的内存位置。在运行时，GPU 硬件以锁步方式在 SIMD 硬件上执行标量线程组（在 AMD 术语中称为 *warps*（或 *wavefronts*，）），以利用其规律性和空间局部性。这种执行模型称为单指令多线程（SIMT）[Lindholm et al., 2008a, Nickolls and Reusch, 1993]。

本章其余部分将在此讨论的基础上展开，具体结构如下：在第2.1节中，我们探讨近期GPU编程模型所使用的概念性执行模型，并对过去十年发布的典型GPU的执行模型进行简要总结。在第2.2节中，我们探讨GPU计算应用程序的编译过程，并简要了解GPU指令集架构。

2.1 执行模型

GPU 计算应用程序在 CPU 上开始执行。对于离散 GPU，应用程序的 CPU 部分通常会分配用于 GPU 上计算的内存，然后将输入数据传输到 GPU 内存，最后在 GPU 上启动计算内核。而对于集成 GPU，仅需要最后一步。计算内核是

由（通常）数千个线程组成。每个线程执行相同的程序，但可能根据计算结果在该程序中遵循不同的控制流程。以下我们使用用 CUDA 编写的一个具体代码示例详细讨论这种流程。在接下来的部分，我们将在汇编级别考察执行模型。我们的讨论不会深入探讨 GPU 编程模型的性能方面。然而，Seo 等人 [2011] 在 OpenCL（一个类似于 CUDA 的编程模型，可以编译到多种架构）上下文中的一个有趣观察是，为某种架构（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 提供了一个 CPU 实现的 C 代码，用于执行著名的操作 *single-precision scalar value A times vector value X plus vector value Y*，称为 SAXPY。SAXPY 是著名的基本线性代数软件 (BLAS) 库的一部分 [Lawson et al., 1979]，在实现高层次的矩阵操作（如高斯消去法 [McCool et al., 2012]）时非常有用。由于其简单性和实用性，它常被用作讲授计算机体系结构的示例 [Hennessy and Patterson, 2011]。图 2.2 提供了一个对应的 CUDA 版本的 SAXPY，该版本将执行分布在 CPU 和 GPU 上。

图 2.2 中的示例展示了 CUDA 及相关编程模型（例如 OpenCL [Kaeli 等, 2015]）提供的抽象。代码以函数 `main()` 开始执行。为了使示例专注于 GPU 上计算的具体细节，我们省略了分配和初始化数组 `x` 和 `y` 的细节。接下来，调用了函数 `saxpy_serial`。该函数的输入参数包括参数 `n` 中向量 `x` 和 `y` 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。该函数对数组 `x` 和 `y` 的每个元素进行迭代。在每次迭代中，代码在第 4 行通过循环变量 `i` 读取值 `x[i]` 和 `y[i]`，将 `x[i]` 与 `a` 相乘后加上 `y[i]`，然后用结果更新 `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，我们考虑一个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。我们不会逐行分析此代码，而是首先强调与GPU执行相关的特定方面。

在 GPU 上执行的线程是由函数指定的计算 *kernel* 的一部分。在图 2.2 中显示的 CUDA 版本的 SAXPY 中，第 1 行的 CUDA 关键字 `__global__` 表明内核函数 `saxpy` 将在 GPU 上运行。在图 2.2 的示例中，我们对图 2.1 中的“ `for` ”循环进行了并行化。具体而言，图 2.1 中原始仅适用于 CPU 的 C 代码中第 4 行的“ `for` ”循环的每次迭代被翻译为一个单独的线程，该线程运行图 2.2 中第 3 – 5 行的代码。

计算内核通常由数千个线程组成，每个线程开始时运行相同的函数。在我们的示例中，CPU 在第 17 行使用 CUDA 的内核配置语法启动 GPU 上的计算。内核配置语法看起来非常像 C 中的函数调用，并附加了一些用于指定线程数量的信息。

```
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]）。

在三重尖括号之间（<<<>>>）。构成计算核的线程被组织成由 *grid* 的 *thread blocks* 组成的层次结构。在 CUDA 编程模型中，单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 硬件将线程组以同步方式一起执行。这些线程组在 NVIDIA 中被称为 *warps*，在 AMD 中被称为 *wavefronts*。NVIDIA 的 warp 由 32 个线程组成，而 AMD 的 wavefront 由 64 个线程组成。warp 被组织成一个更大的单元，由 NVIDIA 称为协作线程数组（CTA）或线程块。第 17 行表示计算核应该启动一个由 `nblocks` 个线程块组成的单个网格，其中每个线程块包含 256 个线程。CPU 代码传递给核配置语句的参数会分发到 GPU 上运行线程的每个实例。

当今许多移动设备的系统级芯片（SoC）将CPU和GPU集成到单个芯片中，正如当今笔记本电脑和台式电脑中的处理器一样。然而，传统上，GPU拥有其独立的DRAM内存，这种情况在数据中心用于机器学习的GPU中仍然持续。我们注意到，NVIDIA引入了统一内存（Unified Memory），它能够透明地将CPU内存更新到GPU内存，并将GPU内存更新到CPU内存。在启用统一内存的系统中，运行时和硬件负责代替程序员执行内存复制。鉴于对机器学习的兴趣日益增长，并且本书的目标是理解硬件，在我们的示例中，我们考虑由程序员管理的独立GPU和CPU内存的一般情况。

按照许多 NVIDIA CUDA 示例中使用的风格，我们在命名指向分配在 CPU 内存中的指针变量时使用前缀 `h_`，而在命名指向分配在 GPU 内存中的指针变量时使用前缀 `d_`。在第 13 行，CPU 调用了 CUDA 库函数 `cudaMalloc`。该函数调用 GPU 驱动程序，并请求其为程序分配 GPU 上的内存。对 `cudaMalloc` 的调用将 `d_x` 设置为指向 GPU 内存中的一个区域，该区域包含足够的空间来容纳 `n` 个 32 位浮点值。在第 15 行，CPU 调用了 CUDA 库函数 `cudaMemcpy`。该函数调用 GPU 驱动程序，并请求其将由 `h_x` 指向的 CPU 内存中数组的内容复制到由 `d_x` 指向的 GPU 内存中的数组。

让我们最后关注GPU上线程的执行。在并行编程中常用的一种策略是为每个线程分配一部分数据。为了实现这一策略，GPU上的每个线程可以在线程块网格中查找自己的身份。在CUDA中，网格、线程块和线程标识符用于实现这一机制。在CUDA中，网格和线程块具有x、y和z三个维度。在执行期间，每个线程在网格和线程块中都有固定且唯一的非负整数x、y和z坐标的组合。每个线程块在网格中具有x、y和z坐标。同样，每个线程在线程块中具有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，未指定y和z维度，因此所有线程的y和z线程块和线程坐标值都为零。在第3行，`threadIdx.x`的值标识了线程在其线程块内的x坐标以及`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`的值将评估为256，因为这是第17行中指定的值。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。正如我们将看到的，使用索引`i`，我们为每个线程分配了`x`和`y`中的一个唯一元素。

在很大程度上，编译器和硬件的结合使程序员能够忽略线程在一个warp中锁步执行的本质。编译器和硬件使得每个线程在warp中看起来像是独立执行的。在图2.2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。对于`i`小于`n`的线程，执行第5行。图2.2的第5行执行了图2.1中原始循环的一次迭代。当网格中的所有线程都完成后，计算内核在第17行后将控制权返回给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。

以下是CUDA编程模型的一些其他细节，这些细节并未在SAXPY示例中展示，但我们将在后续讨论中提到：

CTA 内的线程可以通过每个计算核心的擦写内存高效地相互通信。这种擦写内存被 NVIDIA 称为 *shared memory*。每个流式多处理器 (SM) 包含一个共享内存。共享内存中的空间在运行于该 SM 上的所有 CTA 之间分配。AMD 的图形核心下一代 (GCN) 架构 [AMD, 2012] 包括一个类似的擦写内存，AMD 称之为 *local data store* (LDS)。这些擦写内存较小，每个 SM 的容量范围为 16 – 64 KB，并以不同的内存空间形式暴露给程序员。程序员可以在源代码中使用特殊关键字（例如 CUDA 中的“`__shared__`”）将内存分配到擦写内存中。擦写内存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频繁且可预测地重复使用的数据时，应用程序使用擦写内存可以获益。与 NVIDIA 的 GPU 不同，AMD 的 GCN GPU 还包括一个 *global data store* (GDS) 擦写内存，供 GPU 上的所有核心共享。擦写内存在图形应用程序中用于在不同的图形着色器之间传递结果。例如，LDS 用于在 GCN 中的顶点着色器和像素着色器之间传递参数值 [AMD, 2012]。

在一个CTA内的线程可以使用硬件支持的屏障指令高效同步。不同CTA中的线程可以通信，但需要通过一个所有线程都可访问的全局地址空间进行。访问此全局地址空间通常比访问共享内存在时间和能量方面都更昂贵。

NVIDIA 随着 Kepler 一代 GPU 推出了 CUDA 动态并行（CDP）[NVIDIA Corporation, a]。CDP 的动机源于观察

GPU 硬件未被充分利用。在许多方面，其动机与动态波束形成（Dynamic Warp Formation, DWF）[Fung 等人，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，我们简要讨论了如何将计算内核从诸如 CUDA 和 OpenCL 等高级语言翻译为 GPU 硬件执行的汇编级别，以及当前 GPU 指令集的形式。GPU 架构的一个有趣方面是其支持指令集演化的方式，这与 CPU 架构有些不同。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可以无需任何更改便能在下一代架构上运行。因此，理论上，40 年前为 Intel 8086 编译的软件可以在今天任何一款 x86 处理器上运行。

2.2.1 NVIDIA GPU 指令集架构

鉴于有时提供GPU硬件的供应商数量众多（每个供应商都有自己的硬件设计），通过OpenGL Shading Language (OGSL) 和 Microsoft 的 High-Level Shading Language (HLSL) 实现了一定程度的指令集虚拟化，这种情况在早期GPU可编程时变得常见。当NVIDIA在2007年初引入CUDA时，他们决定遵循类似的路径，并引入了自己的用于GPU计算的高级虚拟指令集架构，称为Parallel Thread Execution ISA，或PTX [NVI, 2017]。NVIDIA在每次发布CUDA时都会完整记录该虚拟指令集架构，以至于本书的作者能够轻松开发支持PTX的GPGPU-Sim模拟器 [Bakhoda et al., 2009]。PTX在许多方面类似于标准的精简指令集计算机（RISC）指令集架构，如ARM、MIPS、SPARC或ALPHA。它还与优化编译器中使用的中间表示有相似之处。其中一个例子是使用无限制的虚拟寄存器集。图2.3展示了图2.2中SAXPY程序的PTX版本。

在 GPU 上运行 PTX 代码之前，必须将 PTX 编译为硬件支持的实际指令集架构。NVIDIA 将此级别称为 SASS，即“ Streaming ASSembler ”（流汇编器）的缩写 [Cabral, 2016]。将 PTX 转换为 SASS 的过程可以由 GPU 驱动程序或 NVIDIA CUDA Toolkit 提供的独立程序 `ptxas` 完成。NVIDIA 并未对 SASS 提供完整文档支持。尽管这使学术研究人员更难开发能捕获所有编译器优化效果的架构模拟器，但它也使 NVIDIA 不必满足客户在硬件级别上提供向后兼容性的需求，从而能够在不同代之间完全重新设计指令集架构。不可避免地，希望深入了解性能的开发人员开始创建自己的工具来反汇编 SASS。第一个这样的努力由 Wladimir Jasper van der Laan 发起，被命名为“ decuda ”[van der Laan]，于 2007 年底针对 NVIDIA 的 GeForce 8 系列（G80）发布，这距离首个 CUDA 支持硬件发布大约一年时间。

```
.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：与图 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指令集进行了足够详细的研究，以至于可以开发一个汇编器。这为在GPGPU-Sim 3.2.2 [Tor M. Aamodt et al.]中开发对SASS的支持（直至NVIDIA的GT200架构）提供了帮助。NVIDIA最终引入了一种名为`cuobjdump`的工具，并开始部分地记录SASS。NVIDIA的SASS文档[NVIDIA Corporation, c]目前（截至2018年4月）仅提供了汇编操作码名称的列表，但没有关于操作数格式或SASS指令语义的详细信息。最近，随着GPU在机器学习中的使用爆炸式增长以及对性能优化代码的需求，其他人开发了类似于decuda的工具，用于后续架构，例如NVIDIA的Fermi [Yunqing]和NVIDIA的Maxwell架构[Gray]。

图 2.4 展示了为 NVIDIA 的 Fermi 架构 [NVI, 2009] 编译并通过 NVIDIA 的 CUDA 工具包中的 `cuobjdump` (部分提取的 SAXPY 内核的 SASS 代码。图 2.4 的第一列是指令的地址，第二列是汇编代码，第三列是编码后的指令。如上所述，NVIDIA 仅部分文档化了其硬件汇编语言。比较图 2.3 和图 2.4，可以发现虚拟和硬件 ISA 层次之间的相似点和差异。总体上，有一些重要的相似点，例如两者都是 RISC（都使用加载和存储访问内存），并且都使用谓词化 [Allen et al., 1983]。更细微的差异包括：（1）PTX 版本基本上有一个无限的寄存器集合，因此每个定义通常使用一个新寄存器，类似于静态单一赋值 [Cytron et al., 1991]，而 SASS 使用的是有限的寄存器集合；（2）内核参数通过分块的常量内存传递，SASS 中的非加载/存储指令也可以访问这些内存，而在 PTX 中，参数被分配到它们自己的独立 “ 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 SASS 代码。对比图 2.5 和图 2.4，可以明显看出 NVIDIA 的 ISA 发生了显著变化，包括指令编码方面的变化。图 2.5 包含一些没有反汇编指令的行（例如，第 3 行地址为 0x0000）。这些是 NVIDIA 在 Kepler 架构中引入的特殊 “ 控制指令 ”，旨在消除使用记分板进行显式依赖性检查的需求 [NVIDIA Corporation, b]。Lai 和 Seznec [2013] 探讨了 Kepler 架构控制指令的编码。正如 Lai 和 Seznec [2013] 所指出的，这些控制指令似乎与 Tera 计算机系统中的显式依赖性前瞻类似 [Alverson et al., 1990]。Gray 描述了他们能够推断出的关于 NVIDIA Maxwell 架构控制指令编码的详细信息。据 Gray 所述，在 Maxwell 中，每三条常规指令对应一条控制指令。这在 NVIDIA 的 Pascal 架构中似乎也是如此，如图 2.5 所示。据 Gray 所述，Maxwell 的 64 位控制指令包含三个 21 位组，分别为以下三条指令编码以下信息：延迟计数、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，这在图 2.5 中也可以看到（例如，`R0.reuse` 用于第一个源）。

```
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 中计算核对应的低级 SASS 代码（使用 CUDA 8.0 为 NVIDIA Fermi 架构 sm_20 编译）。

在第 7 行的整数短乘加指令中的操作数 `.XMAD`)。这似乎表明从 Maxwell 开始，NVIDIA GPU 添加了一个“ 操作数重用缓存 ”（参见第 3.6.1 节的相关研究）。该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图形核心下一代指令集架构

与 NVIDIA 相比，当 AMD 引入 Southern Islands 架构时，他们发布了一份完整的硬件级 ISA 规范 [AMD, 2012]。Southern Islands 是 AMD Graphics Core Next (GCN) 架构的第一代。AMD 硬件 ISA 文档的可用性帮助了学术研究人员开发低层次工作的模拟器 [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，作为异构系统架构 (HSA) 的一部分。

一个关键的区别在于，AMD 的 GCN 架构和 NVIDIA GPU（包括 NVIDIA 最新的 Volta 架构 [NVIDIA Corp., 2017]）之间具有单独的标量和矢量指令。图 2.6 和图 2.7 复现了 AMD [2012] 提供的一个高层 OpenCL（类似于 CUDA）代码示例及其在 AMD South 上的等效机器指令。

```
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 中的计算核对应的低级 SASS 代码（使用 CUDA 8.0 为 NVIDIA Pascal 架构 sm_60 编译）。

恩群岛架构。在图 2.7 中，标量指令以 s_ 为前缀，向量指令以 v_ 为前缀。在 AMD GCN 架构中，每个计算单元（例如，SIMT 核心）包含一个标量单元和四个向量单元。向量指令在向量单元上执行，并为波前中的每个独立线程计算不同的 32 位值。相比之下，标量指令在标量单元上执行，并为波前中的所有线程计算一个共享的 32 位值。在图 2.7 所示的示例中，标量指令与控制流处理相关。特别是，exec 是一个特殊寄存器，用于为 SIMT 执行中的单个向量通道的执行设置谓词。使用掩码处理 GPU 上的控制流在第 3.1.1 节中有更详细的描述。GCN 架构中标量单元的另一个潜在优势是，在 SIMT 程序中，计算的某些部分经常与线程 ID 无关，因而得出相同的结果（见第 3.5 节）。

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

图 2.6：OpenCL 代码（基于 AMD [2012] 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（图形核心下一代）微代码（基于AMD [2012] 图2.2）。

AMD 的 GCN 硬件指令集手册 [AMD, 2012] 提供了许多关于 AMD GPU 硬件的有趣见解。例如，为了解决长延迟操作的数据依赖性问题，AMD 的 GCN 架构包含了 `S_WAITCNT` 指令。对于每个波前，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。每个计数器表示特定类型的未完成操作数量。编译器或程序员插入 `S_WAITCNT` 指令，使波前等待直到未完成操作的数量下降到指定的阈值以下。

CHAPTER 3

SIMT 核心：指令与寄存器数据流

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

在其传统的图形渲染角色中，GPU 访问的数据集（例如详细的纹理贴图）通常过于庞大，无法完全缓存于芯片上。为了实现高性能的可编程性（这是图形处理中所需的，因为图形模式的数量不断增加，需要降低验证成本，并使游戏开发者更容易区分他们的产品 [Lindholm et al., 2001]），必须采用能够维持大规模离芯片带宽的架构。因此，现今的 GPU 能够同时执行成千上万的线程。虽然每个线程的片上存储量较小，但缓存仍然能够有效减少大量的离芯片内存访问。例如，在图形工作负载中，相邻像素操作之间存在显著的空间局部性，这可以通过片上缓存加以利用。

图3.1展示了本章讨论的GPU流水线的微架构。该图展示了图1.2中所示的单个SIMT核心的内部组织。该流水线可以分为SIMT前端和SIMD后端。流水线由三个调度“循环”组成，它们共同在单一流水线中运行：指令获取循环、指令发射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的模块。指令发射循环包括标记为I-Buffer、Scoreboard、Issue和SIMT Stack的模块。寄存器访问调度循环包括标记为Operand Collector、ALU和Memory的模块。在本章的其余部分，我们将通过分析依赖于这些循环的架构关键方面，帮助您全面理解图中各个模块的功能。

由于全面理解该组织涉及许多细节，我们将讨论分为多个部分。我们按顺序安排这些部分，目的是逐步形成对核心微架构的详细视图。我们从整体GPU管线的高级视图开始，然后补充细节。我们将这些越来越精确的描述称为“近似”，以表明即使在我们最详细的描述中，仍然有一些细节被省略。由于当今GPU的核心组织原则是多线程，我们按此原则组织这些“近似”。

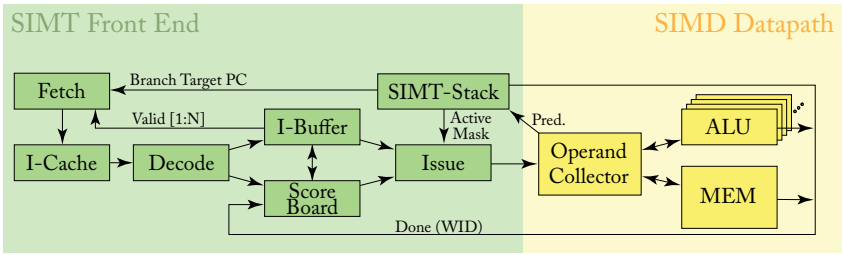


图3.1：通用GPGPU核心的微架构。

围绕上述描述的三个调度循环。我们发现，通过考虑三个逐步提高精确度的“近似循环”来组织本章是很方便的，这些循环逐步纳入这些调度循环的细节。

3.1 一回路近似

我们首先考虑具有单个调度器的GPU。这种对硬件的简化观察与仅阅读CUDA编程手册中硬件描述时可能期望的硬件行为并无太大不同。

为了提高效率，线程被组织成组，NVIDIA 称之为“ warp ”，AMD 称之为“ wave front ”。因此，调度的基本单位是 warp。在每个周期中，硬件会选择一个 warp 进行调度。在单循环近似中，warp 的程序计数器被用来访问指令存储器，以找到 warp 下一步要执行的指令。在获取指令后，指令被解码，并从寄存器文件中获取源操作数寄存器。与从寄存器文件中获取源操作数并行进行的是，SIMT 执行掩码值的确定。以下小节描述了如何确定 SIMT 执行掩码值，并将其与现代 GPU 中也采用的谓词化方法进行对比。

在执行掩码和源寄存器可用之后，执行以单指令、多数据的方式进行。只要设置了SIMT执行掩码，每个线程都会在与通道关联的功能单元上执行。与现代CPU设计类似，功能单元通常是异构的，这意味着一个给定的功能单元仅支持指令的子集。例如，NVIDIA GPU包含一个 *special function unit* (SFU)、*load/store unit*、*floating-point*、*function unit*、*integer function unit*，并且自Volta起增加了一个 *Tensor Core*。

所有功能单元名义上包含的通道数量与一个warp中的线程数量相同。然而，一些GPU采用了不同的实现方式，即一个warp或波前在多个时钟周期内执行。通过以更高的频率运行功能单元，可以在牺牲能源消耗的情况下实现更高的单位面积性能。实现功能单元更高时钟频率的一种方法是对其执行进行流水线处理或增加其流水线深度。

3.1.1 SIMT 执行屏蔽

现代 GPU 的一个关键特性是 SIMT 执行模型，从功能的角度来看（尽管不是性能的角度），它为程序员提供了一个抽象，即各个线程完全独立地执行。这种编程模型可能仅通过谓词化就能实现。然而，在当前的 GPU 中，它是通过传统谓词化与我们称为 *SIMT stack* 的谓词掩码堆栈的组合来实现的。

SIMT 堆栈有助于高效处理当所有线程可以独立执行时发生的两个关键问题。第一个问题是嵌套控制流。在嵌套控制流中，一个分支对另一个分支具有控制依赖关系。第二个问题是当一个 warp 中的所有线程都避开某个控制流路径时，完全跳过计算。对于复杂的控制流，这可以代表显著的节省。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，并且文献中已提出支持跨通道谓词测试的方案。

GPU 使用的 SIMT 堆栈可以处理嵌套控制流和跳过的计算。在专利和指令集手册中描述了几种实现方法。在这些描述中，SIMT 堆栈至少部分由专门用于此目的的特殊指令管理。相反，我们将描述一种在学术工作中提出的稍微简化的版本，该版本假设硬件负责管理 SIMT 堆栈。

为了描述SIMT堆栈，我们使用一个示例。图3.2展示了包含两个分支嵌套在do-while循环中的CUDA C代码，图3.3展示了相应的PTX汇编代码。图3.4（重现了Fung等人[Fung et al., 2007]的图5）展示了在假设GPU每个warp有四个线程的情况下，这段代码如何与SIMT堆栈交互。

图3.4a展示了与图3.2和图3.3中的代码对应的控制流图（CFG）。如CFG顶部节点内标签“ `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，该基本块对应于图3.2中第2到第6行的代码以及图3.3中第1到第6行的代码。在执行图3.3第6行的分支后，这四个线程遵循不同的（分歧的）控制流，该分支对应于图3.2第6行的“ if ”语句。具体而言，如图3.4a中标签“ `B/1110` ”所示，前三个线程落入基本块B。这三个线程分支到图3.3第7行（图3.2第7行）。如图3.4a中标签“ `F/0001` ”所示，在执行分支后，第四个线程跳转到基本块F，该基本块对应于图3.3第14行（图3.2第14行）。

类似地，当基本块 B 中执行的三个线程到达图 3.3 中第 9 行的分支时，第一个线程分歧到基本块 C，而第二个和第三个线程分歧到基本块 D。然后，所有三个线程到达基本块 E 并一起执行，如图 3.4a 中标签“ `E/1110` ”所示。在基本块 G，所有四个线程一起执行。

GPU 硬件如何在使用每个周期只能执行一条指令的 SIMD 数据路径的同时，使一个 warp 内的线程能够遵循代码中的不同路径？

```
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 PTX 汇编代码示例。

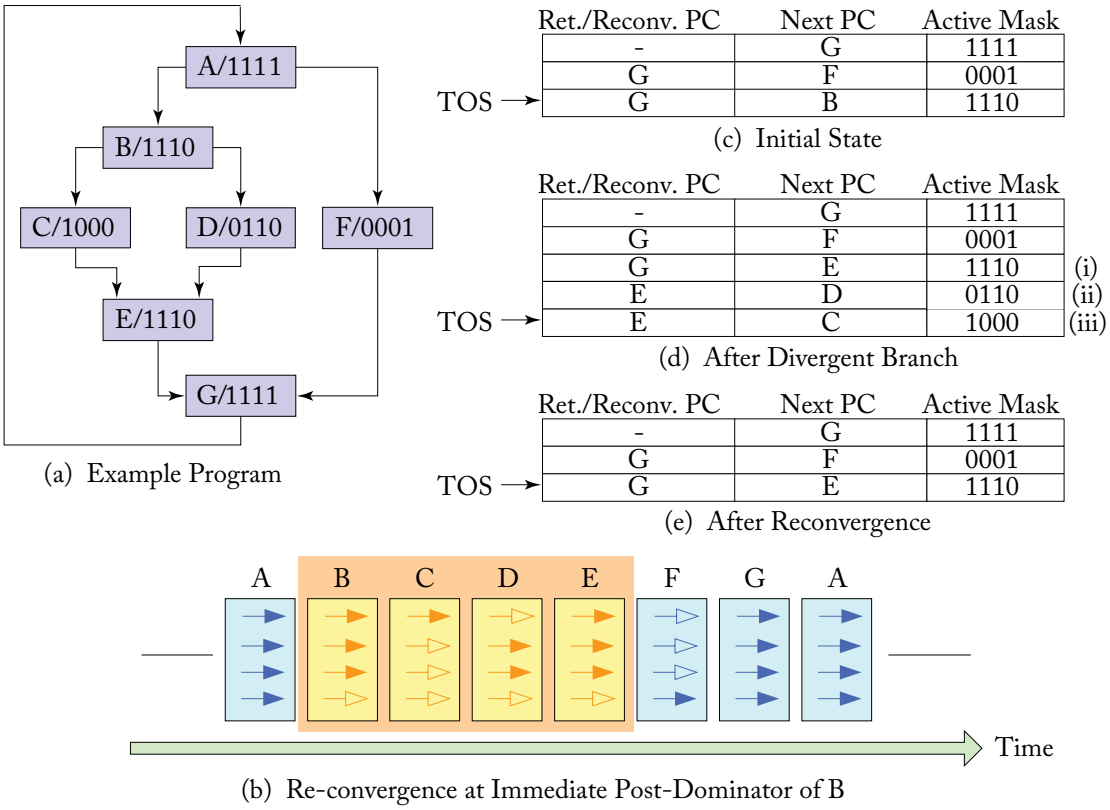


图 3.4：SIMT 堆栈操作示例（基于 Fung 等人 [2007] 的图 5）。

当前 GPU 使用的方法是串行执行同一个 warp 中沿不同路径执行的线程。如图 3.4b 所示，箭头表示线程。实心箭头表示线程正在执行对应基本块中的代码（由每个矩形顶部的字母指示）。空心箭头表示线程被屏蔽。图中时间沿底部箭头所示方向向右推进。最初，每个线程都在执行基本块 B 中的代码。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，此时线程四被屏蔽。为了保持 SIMD 执行，第四个线程将在不同的时间（在此示例中为几个周期之后）通过基本块 F 执行备用代码路径。

为了实现分歧代码路径的序列化，一种方法是使用如图 3.4c – e 所示的堆栈。该堆栈中的每个条目包含三个部分：一个重汇合程序计数器（RPC）、下一条指令的地址（Next PC）以及一个活动掩码。

图 3.4c 展示了在线程束执行图 3.3 中第 6 行的分支后，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，栈顶（TOS）新增了两个条目。线程束执行的下一条指令由栈顶（TOS）条目中的 Next PC 值决定。在图 3.4c 中，该 Next PC 值为 B，表示基本块 B 中第一条指令的地址。对应的活动掩码条目“ `1110` ”表明只有线程束中的前三个线程应执行此指令。

在纱线中的前三个线程继续从基本块 B 执行指令，直到它们到达图 3.3 中的第 9 行分支。在执行该分支后，它们分叉，如前所述。该分支分叉导致堆栈发生三个变化。首先，在执行分支之前的 TOS 条目的 Next PC 条目（图 3.4d 中标注为 (i)）被修改为该分支的 *reconvergence point*，即基本块 E 中第一条指令的地址。然后添加两个条目，分别标注为图 3.4d 中的 (ii) 和 (iii)，每个条目对应纱线在执行分支后沿的路径之一。

一个重新汇聚点是程序中的一个位置，在该位置上，原本分歧的线程可以被强制同步执行。通常，首选最接近的重新汇聚点。在给定程序执行中的最早点，能够在编译时保证曾经分歧的线程可以再次同步执行的是导致分支分歧的分支的直接后支配点。在运行时，有时可以在程序的更早位置实现重新汇聚 [Coon 和 Lindholm, 2008，Diamos 等，2011，Fung 和 Aamodt, 2011]。

一个有趣的问题是：“ 在分支分歧后，应该以什么顺序将条目添加到堆栈中？ ”为了将重汇堆栈的最大深度减少到与波束中线程数量的对数成比例，最好先将活跃线程最多的条目放入堆栈，然后再放入活跃线程较少的条目 [AMD, 2012]。在图 3.4 的第 (d) 部分中，我们遵循了这个顺序，而在第 (c) 部分中我们使用了相反的顺序。

3.1.2 SIMT 死锁与无栈 SIMT 架构

最近，NVIDIA 公布了其即将推出的 Volta GPU 架构的详细信息 [NVIDIA Corp., 2017] 。他们强调的一个变化是关于分支情况下的屏蔽行为及其与同步的交互方式。ElTantaw y 和 Aamodt [2016] 将 SIMT 的基于堆栈的实现描述为可能导致一种称为 “ SIMT 死锁 ” 的死锁情况。学术研究已经提出了用于 SIMT 执行的替代硬件 [ElTantaway et al., 2014] ，并通过一些小的改动 [ElTantawy 和 Aamodt, 2016] 可以避免 SIMT 死锁。NVIDIA 称其新的线程分支管理方法为独立线程调度（Independent Thread Scheduling）。关于独立线程调度的描述表明，他们实现的行为类似于上述学术提案所获得的行为。以下，我们首先描述 SIMT 死锁问题，然后描述一种可以避免 SIMT 死锁且一致的机制。

与 NVIDIA 对独立线程调度的描述以及最近 NVIDIA 的一项专利申请中披露的内容一致 [Diamos et al., 2015]。

图3.5的左侧部分给出了一个CUDA示例，用于说明SIMT死锁问题，中间部分显示了相应的控制流图。A行将共享变量`mutex`初始化为零，表示锁是空闲的。在B行，每个warp中的线程执行`atomicCAS`操作，该操作对包含`mutex`的内存位置执行比较交换操作。`atomicCAS`操作是一个被编译器翻译为`atom.global.cas` PTX指令的内置函数。从逻辑上讲，比较交换首先读取`mutex`的内容，然后将其与第二个输入值0进行比较。如果`mutex`的当前值是0，则比较交换操作会将`mutex`的值更新为第三个输入值1。`atomicCAS`返回的值是`mutex`的原始值。重要的是，比较交换操作对每个线程执行上述逻辑操作时是原子的。因此，同一个warp中不同线程对任何单个位置的多次访问`atomicCAS`是串行化的。由于图3.5中的所有线程访问的是同一内存位置，因此只有一个线程会看到`mutex`的值为0，其余线程将看到`mutex`的值为1。接下来，在考虑SIMT堆栈的情况下，观察B行中`while`循环在`atomicCAS`返回后的情况。不同线程会看到不同的循环条件。具体而言，一个线程希望退出循环，而其余线程希望留在循环中。退出循环的线程将到达汇聚点，因此它将在SIMT堆栈中不再处于活动状态，从而无法执行C行中的`atomicExch`操作来释放锁。而留在循环中的线程将处于SIMT堆栈的顶部并无限旋转。线程之间由此产生的循环依赖引入了一种新的死锁形式，被ElTantawy和Aamodt [2016]称为SIMT死锁，而如果线程在MIMD架构上执行，这种问题就不会存在。

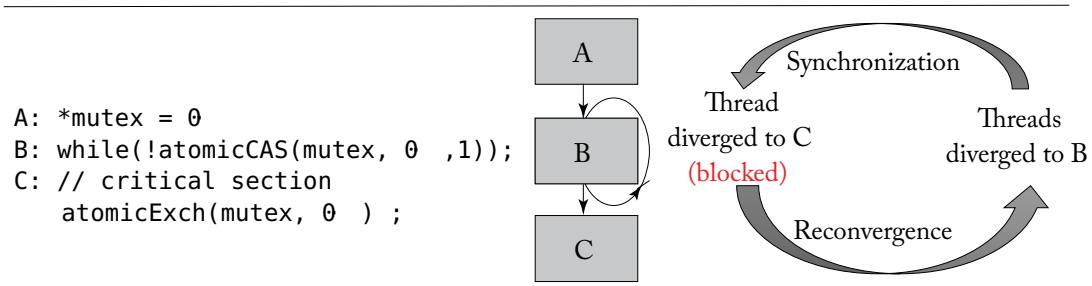


图3.5：SIMT死锁示例（基于ElTantawy和Aamodt [2016] 的图1）。

接下来，我们总结一种无堆栈的分支重新收敛机制，该机制类似于NVIDIA最近的一项美国专利申请 [Diamos 等, 2015] 中的描述。此机制与NVIDIA迄今为止对Volta重新收敛处理机制的描述一致 [Nvidia, 2017]。其关键思想是用每个warp的收敛屏障替代堆栈。图3.6展示了根据NVIDIA专利申请描述的每个warp维护的各个字段，图3.8展示了相关内容。

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Multi-Path IPDOM [ElTantaway et al., 2014] 的一种替代实现，这将在第3.4.2节中与早期的学术研究一起描述。收敛屏障机制与Fung和Aamodt [2011] 中描述的 *warp barrier* 的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，图3.8显示了类似于图3.7中CUDA代码所生成的控制流图。

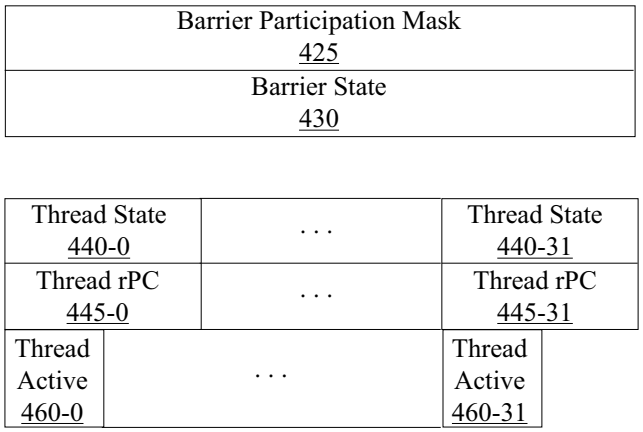


图3.6：一种最近由NVIDIA描述的基于无栈收敛屏障的分支发散处理机制（基于Diamos等人[2015]的图4B）。

```
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)）。

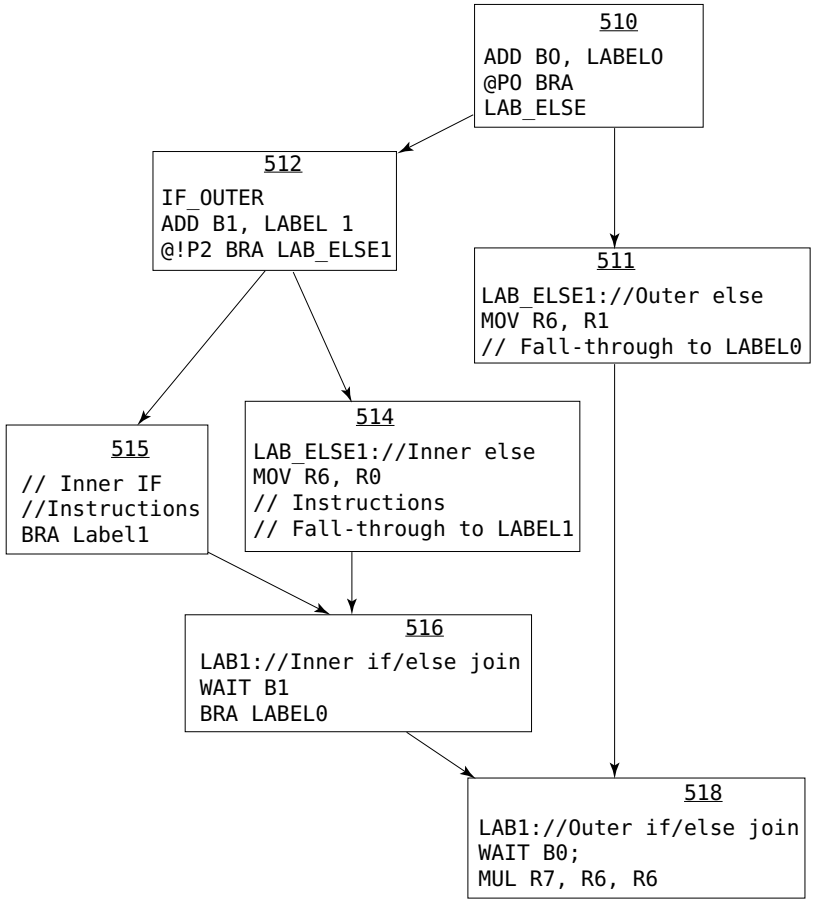


图 3.8：用于处理收敛屏障分支分歧机制的代码示例，最近由 NVIDIA 描述（基于 Diamos 等人 [2015] 的图 5B）。

接下来，我们描述图 3.6 中的字段。这些字段存储在寄存器中，并由硬件 warp 调度器使用。每个 *Barrier Participation Mask* 用于跟踪给定 warp 中的哪些线程参与了某个收敛屏障。对于一个给定的 warp，可能存在多个屏障参与掩码。在常见情况下，由某个屏障参与掩码跟踪的线程会等待彼此在程序中某个分支后的公共点处会合，从而重新收敛。为支持这一点，*Barrier State* 字段用于跟踪哪些线程已到达给定的收敛屏障。*Thread State* 字段用于跟踪 warp 中每个线程的状态，即线程是否准备好执行，是否阻塞在某个收敛屏障（如果是，则是哪一个），或者是否已让出资源。让出状态似乎可以用于在某些情况下使 warp 中的其他线程能够越过收敛屏障继续向前推进。

否则会导致SIMT死锁。线程rPC字段跟踪每个未激活线程的下一条待执行指令的地址。线程活动字段是一个位，指示波束中对应线程是否处于活动状态。

假设一个warp包含32个线程，屏障参与掩码的宽度为32位。如果某一位被设置，这意味着warp中对应的线程参与了该汇合屏障。当线程执行分支指令（例如图3.8中基本块510和512末尾的分支指令）时，它们会发生分歧。这些分支对应于图3.7中的两个“ if ”语句。屏障参与掩码由warp调度器用于在特定的汇合屏障位置停止线程，这个位置可以是分支的直接后继或其他位置。在任意时间，每个warp可能需要多个屏障参与掩码来支持嵌套的控制流结构，例如图3.7中的嵌套if语句。图3.6中的寄存器可能使用通用寄存器、专用寄存器或两者的组合来实现（专利申请未明确说明）。鉴于屏障参与掩码仅为32位宽，如果每个线程都拥有屏障参与掩码的副本（如在使用通用寄存器文件时可能会天真地这样设计），则会显得多余。然而，由于控制流可以嵌套到任意深度，一个给定的warp可能需要任意数量的屏障参与掩码，这使得对掩码的软件管理变得必要。

要初始化收敛屏障参与掩码，需要使用一种特殊的“ ADD ”指令。当执行该ADD指令时，所有处于活动状态的线程都会在该ADD指令所指示的收敛屏障中设置其位。在执行分支后，一些线程可能会发生分歧，这意味着要执行的下一条指令的地址（即程序计数器PC）将有所不同。当这种情况发生时，调度器会选择具有相同PC的一部分线程，并更新线程活动字段（Thread Active Field），以启用这些线程的warp执行。学术提案将这样一部分线程称为“ warp分裂 ”（warp split）[ElTantaway等人, 2014；ElTantawy和Aamodt, 2016；Meng等人, 2010]。与基于堆栈的SIMT实现相比，使用收敛屏障实现时，调度器可以在分歧线程组之间自由切换。这使得当warp中的一些线程获得锁而其他线程未获得时，可以实现warp线程之间的前进进度。

“ WAIT ”指令用于在一个 warp 分支到达收敛屏障时停止执行。如 NVIDIA 的专利申请中所述，WAIT 指令包含一个操作数，用于指示收敛屏障的标识。WAIT 指令的作用是将 warp 分支中的线程添加到该屏障的 Barrier State 寄存器中，并将线程的状态更改为阻塞状态。一旦屏障参与掩码中的所有线程都执行了相应的 WAIT 指令，线程调度器即可将原始 warp 分支中的所有线程切换为活跃状态，从而保持 SIMD 效率。如图 3.8 中的示例所示，包含两个收敛屏障 B1 和 B2，以及在基本块 516 和 518 中的 WAIT 指令。为了实现 warp 分支之间的切换，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），这些内容在本讨论中略去 [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展示了使用NVIDIA Volta白皮书中描述的独立线程调度的潜在时序。在图3.10中，我们可以看到语句A和B与语句X和Y交错执行，与图3.9中的行为形成对比。这种行为与上述收敛屏障机制（以及Multi-Path IPDOM [ElTantaway et al., 2014]）一致。最后，图3.11说明了无堆栈架构如何执行图3.5中的自旋锁代码以避免SIMT死锁。

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

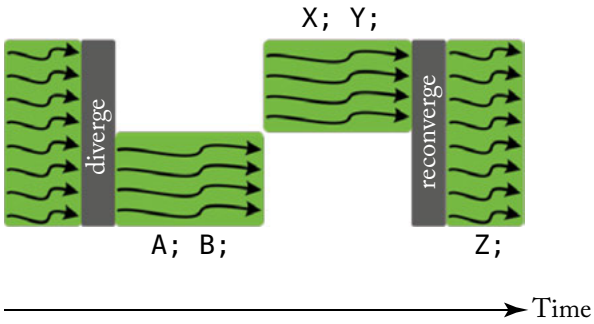


图 3.9：显示基于栈的重新汇聚行为的示例（基于 Nvidia [2017] 的图 20）。

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

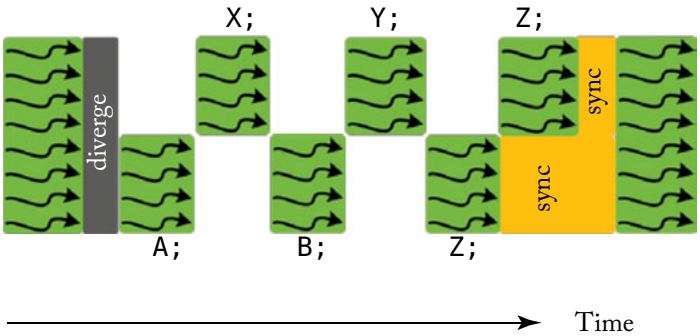


图 3.10：展示 Volta 重聚行为的示例（基于 Nvidia [2017] 的图 23）。

3.1.3 WARP 调度

每个GPU核心中包含许多warp。一个非常有趣的问题是，这些warp应该按照什么顺序进行调度。为了简化讨论，我们假设每个warp在被调度时只发出一条指令，并且该warp不具备可调度性。

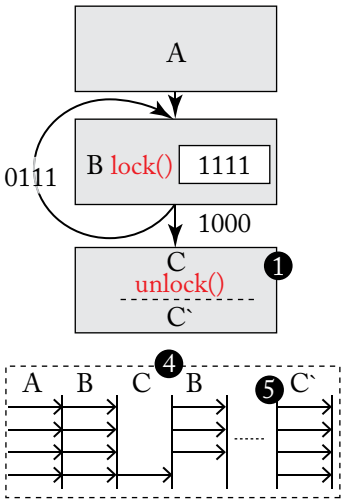


图 3.11：示例显示了类似于图 3.5 中自旋锁代码的收敛屏障的学术机制行为（基于 ElTantawy 和 Aamodt [2016] 图 6(a)）。

在第一条指令完成执行之前，不会发出另一条指令。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 ”，并且能够在某个固定延迟内响应内存请求，那么理论上可以设计核心通过细粒度多线程支持足够多的warp以隐藏这种延迟。在这种情况下，可以认为通过以“ 轮转 ”顺序调度warp，我们可以减少给定吞吐量所需的芯片面积。在轮转中，warp按照某种固定顺序，例如按线程标识符递增排序，并由调度器按此顺序选择warp。该调度顺序的一个特点是，它允许每个已发出的指令大致有相等的时间完成执行。如果核心中的warp数量乘以每个warp的发射时间超过内存延迟，则核心中的执行单元将始终保持忙碌。因此，原则上在此点之前增加warp的数量可以提高每个核心的吞吐量。

然而，这里存在一个重要的权衡：为了使不同的warp能够在每个周期发出一条指令，必须确保每个线程拥有其自己的寄存器（这样可以避免在寄存器和内存之间复制和恢复寄存器状态的需求）。因此，增加每个核心的warp数量会提高分配给寄存器文件存储的芯片面积比例，相对而言，分配给执行单元的比例会减少。在固定的芯片面积下，增加每个核心的warp数量会减少每个芯片的核心总数。

在实际中，内存的响应延迟取决于应用的局部性特性以及由离芯内存访问引发的结果争用量。

调度在考虑GPU的内存系统时会产生什么影响？这是过去几年中大量研究的主题，我们将在为我们的GPU微架构模型添加更多内存系统细节后，再次探讨这个问题。然而，简而言之，局部性属性可能会支持或抑制轮询调度：当不同线程在其执行过程中共享相似点的数据时，例如在图形像素着色器中访问纹理贴图时，让线程以相同的进度运行是有利的，因为这可以增加命中片上缓存的内存引用数量，而轮询调度正是鼓励这种行为的[Lindholm et al., 2015]。同样，当在地址空间中附近的位置在时间上接近地被访问时，访问DRAM的效率更高，这也是轮询调度所鼓励的 [Narasiman et al., 2011]。另一方面，当线程主要访问不相交的数据时（如在使用更复杂的数据结构时常见的情况），重复调度给定线程以最大化局部性可能更为有利 [Rogers et al., 2012]。

3.2 两环近似

为了帮助减少每个核心必须支持的线程束数量以隐藏长执行延迟，在早期指令尚未完成时能够从线程束中发出后续指令是有帮助的。然而，前面描述的单环微架构阻止了这一点，因为该设计中的调度逻辑仅能访问线程标识符和下一个要发出的指令的地址。具体来说，它不知道该线程束的下一条指令是否依赖于尚未完成执行的早期指令。为了提供此类依赖信息，必须首先从内存中取指令，以确定存在哪些数据和/或结构性冲突。为此，GPU实现了一个指令缓冲区，指令在缓存访问后会被放入其中。一个单独的调度器用于决定指令缓冲区中的多个指令中，哪个应该被下一步发往流水线。

指令存储器实现为一级指令缓存，由一个或多个级次级（通常是统一的）缓存支持。指令缓冲区还可以与指令缺失状态保持寄存器（MSHRs）结合使用，从而帮助隐藏指令缓存未命中延迟 [Kroft, 1981]。在缓存命中或从缓存未命中填充后，指令信息被放置到指令缓冲区中。指令缓冲区的组织形式可以有多种。一个特别直接的方法是为每个 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性。在传统的 CPU 架构中，有两种检测指令之间依赖性的传统方法：计分板和保留站。保留站用于消除名称依赖性，但引入了在面积和能量方面成本较高的关联逻辑。计分板可以设计为支持顺序执行或乱序执行。支持乱序执行的计分板，例如 CDC 6600 中使用的计分板，也相当复杂。另一方面，计分板对于一个...

单线程顺序执行的 CPU 非常简单：每个寄存器在记分牌中用单个位表示，当有指令将写入该寄存器时，该位会被设置。任何想要读取或写入寄存器的指令，如果其对应的记分牌位被设置，则会被阻塞，直到写入寄存器的指令清除该位为止。这可以防止读后写和写后写的冒险。当结合顺序指令发射时，这种简单的记分牌可以防止写后读的冒险，前提是寄存器文件的读取被限制为按顺序发生，而这通常是顺序 CPU 设计中的情况。由于这是最简单的设计，因此它将消耗最少的面积和能量，GPU 实现了顺序记分牌。然而，如下所述，当支持多个 warp 时，使用顺序记分牌会面临一些挑战。

上述简单按序记分牌设计的第一个问题是现代 GPU 中包含的大量寄存器。每个 warp 最多有 128 个寄存器，每个核心最多有 64 个 warp，因此实现记分牌每个核心需要总计 8192 位。

另一个关于上述简单顺序计分板设计的担忧是，遇到依赖关系的指令必须反复在计分板中查找其操作数，直到其所依赖的前一条指令将结果写入寄存器文件。对于单线程设计，这引入的复杂性很小。然而，在顺序发射的多线程处理器中，来自多个线程的指令可能会等待较早的指令完成。如果所有这些指令都必须探测计分板，则需要额外的读取端口。现代GPU每个核心支持多达64个warp，每个warp最多有4个操作数。如果允许所有warp每个周期都探测计分板，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计分板的warp数量，但这限制了可被考虑用于调度的warp数量。此外，如果所检查的指令都存在依赖关系，即使其他未被检查的指令恰好没有依赖关系，也可能无法发射任何指令。

这两个问题可以使用 Coon 等人 [2008] 提出的设计来解决。与每个 warp 的每个寄存器只保留一位的传统方法不同，该设计为每个 warp 包含少量条目（最近一项研究 [Lashgar 等人, 2016] 估计约为 3 或 4 个），其中每个条目是一个寄存器的标识符，该寄存器将由已发射但尚未完成执行的指令写入。常规的顺序记分板在指令发射和写回时都会被访问。而 Coon 等人的记分板在指令进入指令缓冲区和指令将结果写入寄存器文件时被访问。

当一条指令从指令缓存中取出并放入指令缓冲区时，会将该指令对应的 warp 的记分牌条目与该指令的源寄存器和目标寄存器进行比较。这会生成一个短的位向量，每个位对应该 warp 的记分牌中的一个条目（例如，3 或 4 位）。如果记分牌中的对应条目与指令的任一操作数匹配，则设置该位。然后，将该位向量与指令一起复制到指令缓冲区中。在所有位清除之前，指令调度器不会考虑该指令，这可以通过输入每个位来确定。

将向量的位输入 NOR 门。当指令缓冲区中的依赖位在指令将其结果写入寄存器文件时被清除。如果某个 warp 的所有条目都被用完，那么要么所有 warp 的取指停滞，要么该指令被丢弃并且必须重新取指。当已执行的指令准备写入寄存器文件时，它会清除在计分板中分配给它的条目，并清除存储在指令缓冲区中的来自同一 warp 的任何指令的相应依赖位。

在双循环架构中，第一个循环选择一个在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解决依赖关系的指令，并将其发送到执行单元。

3.3 三环近似

如前所述，为了隐藏长时间的内存延迟，有必要在每个核心上支持多个warp，并且为了支持逐周期在warp之间切换，需要一个大型寄存器文件，该文件包含每个正在执行的warp的独立物理寄存器。例如，在最近的NVIDIA GPU架构（如Kepler、Maxwell和Pascal架构）中，这样的寄存器文件包含256 KB。现在，SRAM内存的面积与端口数量成正比。一个寄存器文件的简单实现需要每个周期每条指令的每个操作数使用一个端口。减少寄存器文件面积的一种方法是通过使用多组单端口内存来模拟大量的端口。虽然可以通过将这些内存组暴露给指令集架构来实现这种效果，但在一些GPU设计中，似乎使用了一种称为操作数收集器的结构 [Coon et al., 2009, Lindholm et al., 2008b, Lui et al., 2008]，以更透明的方式实现这一目标。操作数收集器实际上形成了一个第三调度循环，如下所述。

为了更好地理解操作数收集器解决的问题，首先考虑图 3.12，该图显示了一种用于提供更高寄存器文件带宽的简单微架构。该图展示了 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。在实际应用中，由于寄存器文件非常大，每个逻辑组可能会进一步分解为更多的物理组（图中未显示）。逻辑组通过交叉开关连接到分段寄存器（标记为“流水线寄存器”），这些寄存器在将源操作数传递到 SIMD 执行单元之前对其进行缓冲。仲裁器控制对各个逻辑组的访问，并通过交叉开关将结果路由到合适的分段寄存器。

图 3.13 显示了每个 warp 的寄存器到逻辑 bank 的一种简单布局。在此图中，来自 warp 0 的寄存器 `r0`（`w0`）存储在 Bank 0 的第一个位置，来自 warp 0 的寄存器 `r1` 存储在 Bank 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 bank 的数量，则分配会循环。例如，warp 0 的寄存器 `r4` 存储在 Bank 0 的第二个位置。

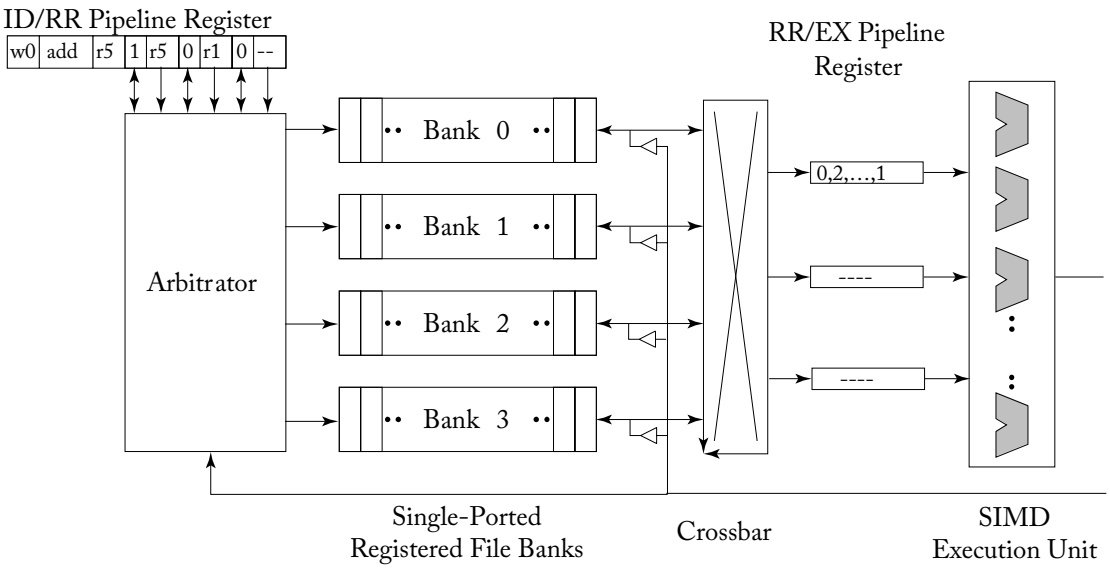


图 3.12：简单分块寄存器文件微架构。

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：简单分区寄存器布局。

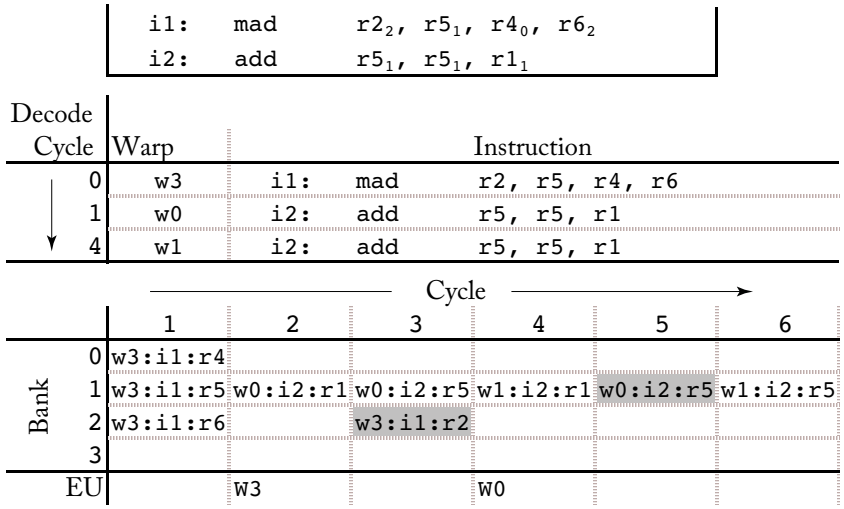


图 3.14：简单分块寄存器文件的时序。

图 3.14 展示了一个定时示例，强调了这种微架构如何导致性能下降。示例涉及顶部显示的两条指令。第一条指令 i1 是一个多重加法操作，从寄存器 r5、r4 和 r6 读取数据，这些寄存器分别分配在第 1、0 和 2 个银行中（如图中的下标所示）。第二条指令 i2 是一个加法指令，从寄存器 r5 和 r1 读取数据，它们都分配在第 1 个银行。图的中间部分展示了指令发射的顺序。在第 0 个周期，warp 3 发射指令 i1；在第 1 个周期，warp 0 发射指令 i2；在第 4 个周期，warp 1 发射指令 i2，但由于接下来的描述的银行冲突出现了延迟。图的底部部分说明了不同指令访问不同银行的定时情况。在第 1 个周期，来自 warp 3 的指令 i1 能够在第 1 个周期读取其三个源寄存器，因为它们映射到不同的逻辑银行。然而，在第 2 个周期，来自 warp 0 的指令 i2 只能读取其两个源寄存器中的一个，因为它们都映射到第 1 个银行。在第 3 个周期，这条指令的第二个源寄存器与来自 warp 3 的指令 i1 的写回操作并行读取。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，但无法读取第二个源操作数，因为它们都映射到第 1 个银行。同样地，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，因为该银行已被来自 warp 0 的指令 i2 的高优先级写回操作占用。最终，在第 6 个周期，来自 warp 1 的 i2 的第二个源操作数从寄存器文件中读取。总而言之，三条指令完成读取其源寄存器的过程需要六个周期，而在此期间，许多银行并未被访问。

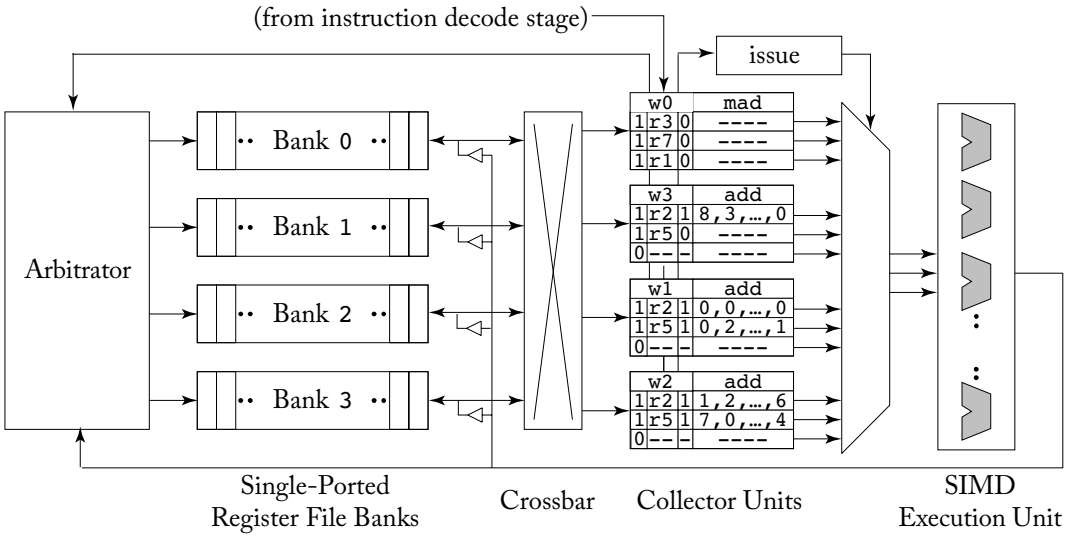


图3.15：操作数收集器微架构（基于Tor M. Aamodt等人的图6）。

3.3.1 操作数收集器

操作数收集器微架构 [Lindholm et al., 2008b] 如图 3.15 所示。主要变化是阶段寄存器被 *collector units* 替代。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。由于存在多个收集器单元，因此多条指令可以重叠读取源操作数，从而在各条指令的源操作数之间发生银行冲突的情况下有助于提高吞吐量。每个收集器单元包含执行一条指令所需的所有源操作数的缓冲空间。由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银行级并行性，以便并行访问多个寄存器文件银行。

操作数收集器通过调度在发生银行冲突时进行容忍。这引出了如何减少银行冲突数量的问题。图3.16展示了一种修改后的寄存器布局，Coon等人描述了该布局有助于减少银行冲突。其核心思想是在不同银行中为不同线程束分配等价的寄存器。例如，在图3.16中，线程束0的寄存器$_{r0}$被分配到银行0，而线程束1的寄存器$_{r0}$被分配到银行1。这并未解决单条指令的寄存器操作数之间的银行冲突。然而，它确实在减少不同线程束指令之间的银行冲突方面有所帮助。尤其是在线程束进度相对均衡的情况下（例如，由于轮转调度或两级调度 [Narasiman等，2011]，其中抓取组中的单个线程束以轮转顺序进行调度）。

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：交错的分组寄存器布局。

```
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：操作数收集器的时序。

图 3.17 显示了一个时序示例，顶部展示了一系列加法和乘加指令。在中间部分显示了指令的发射顺序。来自 warp 1 到 warp 3 的三个 `i1` 实例分别在第 0 到第 2 个周期发射。来自 warp 0 的 `i2` 指令实例在第 3 个周期发射。请注意，加法指令写入寄存器 `r1`，对于任何给定的 warp，该寄存器与源寄存器 `r5` 分配在同一个寄存器库中。然而，与图 3.13 中的寄存器布局情况不同，这里不同的 warp 访问不同的寄存器库，这有助于减少一个 warp 写回与其他 warp 读取源操作数之间的冲突。底部部分显示了操作数收集器引起的寄存器库级访问时序。在第 1 个周期，warp 1 的寄存器 `r2` 读取了寄存器库 3。在第 4 个周期，可以看到 warp 1 的寄存器 `r1` 的写回与 warp 3 的寄存器 `r5` 的读取以及 warp 0 的寄存器 `r3` 的读取并行进行。

一个微妙的问题是，如前所述的操作数收集器，因为它没有在不同指令何时准备发射之间强制任何顺序，这可能会导致读后写（WAR）冒险 [Mishkin 等人, 2016]。如果两条指令来自于同一上下文，这种情况可能会发生。

相同的warp在操作数收集器中出现，且第一条指令读取一个寄存器，该寄存器将被第二条指令写入。如果第一条指令的源操作数访问遇到重复的bank冲突，则第二条指令可能在第一条指令读取（正确的）旧值之前将新值写入寄存器。防止此类WAR（写后读）风险的一种方法是简单地要求同一warp的指令按照程序顺序从操作数收集器发送到执行单元。Mishkin等人[2016]探索了三种硬件复杂性较低的潜在解决方案，并评估了它们的性能影响。第一种方案，*release-on-commit warpboard*，每个warp最多允许一条指令在执行。不出意外，他们发现这对性能有负面影响，在某些情况下性能降低了近一半。第二种方案，*release-on-read warpboard*，每个warp在操作数收集器中一次仅允许一条指令收集操作数。这种方案在他们研究的工作负载中最多导致10%的性能下降。最后，为了在操作数收集器中实现指令级并行性，他们提出了一种使用小型布隆过滤器跟踪未完成寄存器读取的*bloomboard*机制。这种机制的影响不到（错误地）允许WAR风险情况下的几个百分点。另有一项由Gray进行的分析表明，NVIDIA的Maxwell GPU引入了一种“读取依赖屏障”，该屏障由特殊的“控制指令”管理，可以用于避免某些指令的WAR风险（见第2.2.1节）。

3.3.2 指令重放：处理结构性冲突

GPU流水线中可能存在许多潜在的结构性风险。例如，寄存器读取阶段可能会耗尽操作数收集单元。许多结构性风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。通常，由一个warp执行的单条内存指令可能需要被分解为多个独立的操作。这些独立的操作中的每一个都可能在某个周期内完全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冒险时会发生什么？在单线程顺序CPU流水线中，一个标准的解决方案是阻止较年轻的指令，直到遇到阻塞条件的指令可以进一步执行。这种方法在高度多线程的吞吐架构中可能较不理想，至少有两个原因。首先，由于寄存器文件的规模较大以及支持完整图形流水线所需的许多流水线阶段，分发阻塞信号可能会影响关键路径。对阻塞周期分发进行流水化处理需要引入额外的缓冲，从而增加面积。其次，阻止一个线程束（warp）中的一条指令可能会导致其他线程束中的指令在其后等待。如果这些指令不需要导致阻塞的指令所需的资源，吞吐量可能会受到影响。

为了避免这些问题，GPU 实现了一种指令重放机制。指令重放存在于一些 CPU 设计中，用作在依赖指令基于具有可变延迟的更早指令进行推测性调度时的恢复机制。例如，加载操作可能会命中或未命中一级缓存，但在高时钟频率的 CPU 设计中...

频率可能会通过多达四个时钟周期来流水化一级缓存访问。一些 CPU 根据加载情况推测性地唤醒指令，以提高单线程性能。相比之下，GPU 避免了推测，因为这往往会浪费能量并降低吞吐量。取而代之的是，GPU 使用指令重放来避免堵塞流水线以及由停顿导致的电路面积和/或时序开销。

为了实现指令重放，GPU可以将指令保留在指令缓冲区中，直到确定它们已完成或指令的所有部分都已执行完毕 [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 warp 内的线程通过相同的控制流路径执行，以便 GPU 可以在 SIMD 硬件上同步执行它们。鉴于线程的自主性，当 warp 的线程在数据依赖的分支处分化到不同目标时，可能会遇到 *branch divergence*。现代 GPU 包含专门的硬件来处理 warp 中的分支分化。本书第 3.1.1 节描述了基线 SIMT 堆栈，该堆栈被基线 GPU 架构使用。基线 SIMT 堆栈通过串行化不同目标的执行来处理 warp 中的分支分化。虽然基线 SIMT 堆栈能够正确处理大多数现有 GPU 应用中的分支分化，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，基础的SIMT堆栈会将每个分支目标的执行序列化。当每个目标被执行时，SIMT堆栈仅激活运行该目标的标量线程子集。这导致SIMD硬件中的一些通道处于空闲状态，从而降低了整体*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并非功能正确性的必要条件。GPU编程模型不会在warp内的标量线程之间强加任何隐式数据依赖——它们必须通过共享内存和屏障显式通信。GPU可以交错执行分歧warp的所有分支目标，以利用SIMD硬件中的空闲周期。

不足的MIMD抽象通过强制分歧的warp在编译器定义的重新收敛点重新收敛，基础SIMT堆栈隐式地在每个重新收敛点强加了一个warp范围的同步点。这适用于许多现有的GPU应用程序。然而，这种隐式同步可能与其他用户实现的同步机制（例如细粒度锁）病态地交互，导致warp死锁。编译器定义的重新收敛点也未考虑系统级结构（如异常和中断）引入的控制流分歧。

面积成本 虽然基线SIMT堆栈对每个warp的面积需求仅为32 × 64位（或低至6 × 64位），但面积会随着GPU中正在执行的warp数量而扩展。在分支发散较少的典型GPU应用中，SIMT堆栈占用的面积可以以其他方式用于提升应用吞吐量（例如，更大的缓存，更多的ALU单元等）。

无论是在工业界还是学术界，都提出了替代方案来解决上述不足。这些各种提案可以分为以下几类：warp压缩、warp内分歧路径管理、增加MIMD能力以及复杂性减少。一些提案包含了捕获多个类别特征的改进，因此被多次提及。

3.4.1 Warp 压缩

由于GPU实现了细粒度的多线程以容忍长时间的内存访问延迟，每个SIMT核心中有许多warp，总计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，它们可能会遵循相同的执行路径，并在同一组数据相关分支处遇到分支分歧。因此，分支分歧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该观察结果来提高因分支分歧而受影响的GPU应用性能的研究。该系列中的提案均涉及通过新的硬件机制将来自不同*static warps*的*compact*线程重新组合成新的*dynamic warps*，以提高这些分歧GPU应用的整体SIMD效率。这里，静态warp是指当标量线程从内核启动中生成时，由GPU硬件形成的warp。在我们的基线GPU架构中，这种安排在warp的整个执行过程中是固定的。将标量线程安排成静态warp是一种由GPU硬件强加的任意分组，这在编程模型中基本上是不可见的。

动态扭曲形成。动态扭曲形成（DWF）[Fung 等, 2007, Fung 等, 2009] 利用这一观察，通过重新排列执行相同指令的这些分散线程形成新的动态扭曲。在分支分歧处，DWF 通过将分散于多个已分歧静态扭曲中的线程压缩到更少的非分歧动态扭曲中，可以提升应用程序的整体 SIMD 效率。通过这种方式，DWF 能够在 SIMD 硬件上捕获 MIMD 硬件的大部分优势。然而，DWF 要求扭曲在短时间窗口内遇到相同的分支分歧。这种依赖时间的 DWF 特性使其对扭曲调度策略非常敏感。

Fung 和 Aamodt [2011] 的后续工作确定了 DWF 性能病理学的两个主要方面：（1）贪婪的调度策略可能会使某些线程处于饥饿状态，从而导致 SIMD

效率下降；以及 (2) DWF 中的线程重组增加了非合并内存访问和共享内存银行冲突。这些问题导致 DWF 减慢了许多现有的 GPU 应用程序。此外，依赖静态 warp 中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部分解决，该策略将计算内核有效地分为两类区域：发散区域和非发散（相干）区域。发散区域显著受益于 DWF，而相干区域虽然没有分支发散问题，但容易受到 DWF 病理问题的影响。我们发现，通过在相干区域强制 DWF 将标量线程重新排列回其静态 warp，可以显著减少 DWF 病理问题的影响。

线程块压缩。线程块压缩（TBC）[Fung 和 Aamodt, 2011] 基于这一洞察提出了一个观察，即将线程不断重新排列到新的动态 warp 中不会带来额外的收益。相反，这种重新排列或 *compaction* 仅需要在分支发生分歧之后（即分歧区域的开始）以及其重新汇合点之前（即一致性区域的开始）进行。我们注意到，现有的每个 warp 的 SIMT 栈（在第 3.1.1 节中描述）隐式地在分歧分支的重新汇合点处对分歧到不同执行路径的线程进行同步，将这些分歧的线程在执行一致性区域之前重新合并回一个静态 warp。TBC 将 SIMT 栈扩展到包含同一核心中执行的所有 warp，强制它们在分歧分支和重新汇合点进行同步和压缩，从而实现稳健的 DWF 性能收益。然而，在每个分歧分支上对核心内的所有 warp 进行同步以进行压缩可能会大大降低可用的线程级并行性（TLP）。GPU 架构依赖于丰富的 TLP 来容忍流水线和内存延迟。

TBC 通过将压缩限制在仅发生在 *thread block* 内，在 SIMD 效率和 TLP 可用性之间折中。GPU 应用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，在分歧分支处重叠压缩开销——当一个线程块中的 warp 在分歧分支处同步以进行压缩时，其他线程块中的 warp 可以保持硬件繁忙。它扩展了每个 warp 的 SIMT 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 栈来确定线程块中的 warp 何时同步并被压缩为新的 warp 集合。其结果是一个更加稳健且简单的机制，捕捉了 DWF 的大部分优势，而没有病态行为。

大规模Warp微架构。大规模Warp微架构 [Narasiman et al., 2011] 扩展了SIMT堆栈，类似于TBC，以管理一组Warp的重汇合。然而，它并未将压缩限制在分支点和重汇合点。LWM要求组内的Warp完全同步执行，以便在每条指令上对组进行压缩。这比TBC进一步减少了可用的TLP，但允许...

LWM 使用带谓词指令以及无条件跳转执行压缩。与 TBC 类似，LWM 将运行在同一核心上的 warp 分成多个组，并限制压缩仅在组内进行。它还选择了一种更复杂的记分牌微架构，该架构以线程粒度跟踪寄存器依赖性。这使得组中的一些 warp 能够比其他 warp 略微提前执行，以弥补因同步执行而导致的 TLP 损失。

紧凑性充分性预测器。Rhu 和 Erez [2012] 使用紧凑性充分性预测器（CAPRI）扩展了 TBC。该预测器识别出在每个分支上将线程压缩到少量 warp 的有效性，并且仅在预测紧凑化会带来收益的分支上同步线程。这回收了由于非收益性停顿和使用 TBC 紧凑化而丧失的 TLP。Rhu 和 Erez [2012] 还表明，一个类似于单级分支预测器的简单基于历史的预测器足以实现高精度。

组内压缩。Vaidya 等人 [2013] 提出了一种低复杂度的压缩技术，这种技术有利于宽 SIMD 执行组在较窄的硬件单元上以多个周期运行。他们的基本技术将单个执行组划分为多个与硬件宽度匹配的子组。遭遇分歧的 SIMD 执行组可以通过跳过完全空闲的子组，在较窄的硬件上更快地运行。为了创建更多完全空闲的子组，他们提出了一种混排机制，将元素压缩到更少的子组中以应对分歧。

同时扭曲交织。Brunie 等人 [2012] 提出了同时分支和扭曲交织（SBI 和 SWI）。他们扩展了 GPU 的 SIMT 前端，以支持每个周期发出两条不同的指令。他们通过将扭曲宽度扩展为原来的两倍来补偿这种增加的复杂性。SWI 从一个受分支发散影响的扭曲中与另一个发散扭曲的指令共同发出指令，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，硬件压缩通常在SIMT核心内部局部进行。由于压缩后的线程都位于同一核心上，共享相同的寄存器文件，因此通过更灵活的寄存器文件设计，可以在无需移动其架构状态的情况下执行压缩 [Fung et al., 2007]。

正如本章前面所讨论的，GPU 寄存器文件通过大型单端口 SRAM 存储器组实现，以最大化其面积效率。同一个 warp 中线程的寄存器存储在同一 SRAM 存储器组的连续区域中，因此它们可以通过一个宽端口一起访问。这种设计允许高带宽的寄存器文件访问，同时摊薄寄存器文件访问控制硬件的成本。硬件 warp 压缩会生成动态 warp，可能不遵循这种寄存器排列。Fung 等人 [2007] 提出了一个更灵活的寄存器文件设计，该设计采用具有窄端口的 SRAM 存储器组。为了保持相同的带宽，这种设计需要更多的 SRAM 存储器组。

动态微内核。Steffen 和 Zambreno [2010] 利用 *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。这种分解本身并不会提高并行性，因为每次迭代都依赖于前一次迭代的数据。相反，该启动机制通过将剩余的活动线程压缩到少量的 warp 中，改善了同一核心中不同线程之间的负载不平衡。它还与其他硬件 warp 压缩技术不同，因为这种压缩通过将线程及其架构状态迁移到每核心的暂存区（即每核心的高速缓存内存）中作为过渡区域来实现。

第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，这些提议引入了额外的内存流量，以将线程从一个SIMT核心重新定位到另一个核心。

软件中的Warp压缩

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。重新分组涉及将线程及其私有数据在内存中移动，这可能会引入显著的内存带宽开销。以下我们重点介绍几项关于软件压缩技术的工作。

条件流 [Kapasi 等，2000] 将这一概念应用于流计算。它将可能具有分支控制流的流处理器计算内核分解为多个内核。在分支点，内核根据每个数据元素的分支结果将其数据流分解为多个流。每个流由单独的内核处理，并在控制流分支结束时合并回一起。

Billeter 等人 [2009] 提出了使用并行前缀和实现 SIMD *stream compaction*。流压缩将包含不同任务的元素流重新组织为紧凑的同类任务子流。该实现利用了 GPU 片上高速缓存的访问灵活性，以实现高效能。Hoberock 等人 [2009] 提出了一种用于光线追踪的延迟着色技术，该技术利用流压缩来提高复杂场景中具有多种材质类别的像素着色的 SIMD 效率。每种材质类别都需要独特的计算。一个将所有材质类别的计算组合在一起的像素着色器在 GPU 上运行效率较低。流压缩将击中具有相似材质类别的物体的光线分组，从而使 GPU 的 SIMD 硬件能够高效地执行这些像素的着色器。

张等人 [2010] 提出了一种运行时系统，该系统能够动态地将线程重新映射到不同的warp，以提高SIMD效率以及内存访问的空间局部性。该运行时系统具有流水线系统特性，由CPU执行动态重新映射，而GPU对重新映射的数据/线程进行计算。

Khorasani 等人 [2015] 提出了 *Collective Context Collection* (CCC)，这是一种编译器技术，可以将具有潜在分支发散惩罚的 GPU 计算内核进行转换。

为了提高现有GPU上的SIMD效率，CCC专注于计算核，其中每个线程在每一步执行不规则数量的计算，例如通过不规则图进行广度优先搜索。与其为每个线程分配一个节点（或其他应用中的任务），CCC首先对计算核进行转换，使每个线程处理多个节点，节点到warp（注意：不是线程）的分配在核启动之前确定。随后，CCC转换计算核，使warp中的每个线程能够将任务的上下文卸载到存储在共享内存中的warp特定堆栈。当warp在当前任务集中经历低SIMD效率时，可以将任务卸载到堆栈，并使用这些卸载的任务填充处理后续任务集时变为空闲的线程。实际上，CCC通过将多个warp的任务分组到更少的warp集合中，并通过存储在快速片上共享内存中的warp特定堆栈将分歧任务压缩到每个warp的更少迭代中，实现了“ warp压缩 ”。

线程分配在一个Warp内的影响
在本书研究的基准GPU架构中，具有连续线程ID的线程被静态融合在一起以形成warp。关于线程到warp或warp内lane的静态分配，学术研究较少。对于大多数工作负载来说，这种默认的顺序映射效果很好，因为相邻线程倾向于访问相邻数据，从而改进了内存合并。然而，一些研究也探讨了替代方法。

SIMD通道置换。Rhu和Erez [2013b] 观察到，将线程ID按顺序映射到一个warp中连续的线程，对于本节前面描述的warp压缩技术来说并不是最优的。大多数warp压缩和形成工作的一个关键限制是，当线程被分配到一个新的warp时，它们不能被分配到不同的通道，否则它们的寄存器文件状态必须移动到向量寄存器中的不同通道。Rhu和Erez观察到，程序的结构会使某些控制流路径偏向某些SIMD通道。这种偏向性使得实现压缩更加困难，因为采用相同路径的线程往往位于相同的通道中，阻止这些线程被合并。Rhu和Erez提出了几种不同的线程映射置换方法，这些方法可以消除这些程序偏向性，并显著提高压缩率。

Warp 内循环压缩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。这意味着一个 32 线程的 warp 需要在两个核心周期内执行完毕。Vaidya 等人 [2013] 观察到，当出现分歧时，如果某条指令的一个连续 SIMD 范围内的线程被屏蔽掉，那么该指令可以仅用一个周期发射，跳过被屏蔽的线程。他们将这种技术称为周期压缩（cycle compression）。然而，如果被屏蔽的线程不是连续的，则基本技术不会带来任何性能改进。为了解决这一问题，他们提出了一种称为交换周期压缩（swizzled cycle compression）的方法，通过重新排列线程与通道之间的对应关系，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] 的工作）认为，当同一 warp 中的线程处理相同数据时，SIMT 编程模型效率较低。一些解决方案建议在流水线中包含标量单元，用于处理编译器或程序员可以先验识别为标量的工作。AMD 的 Graphics Core Next (GCN) 架构为此目的包含了标量流水线。详见第 3.5 节。

3.4.2 线程束内分岔路径管理

虽然具有直接后支配点重新汇合的SIMT堆栈可以处理具有任意控制流的分支分歧，但它在多个方面仍可以进一步改进。

1. 分支线程分散到一个分歧 warp 的不同分支目标时，可以交错执行它们的操作，从而利用 SIMD 硬件中的空闲周期。
2. 虽然分支发散的直接后支配节点是明确的汇聚点，但分支到不同目标的线程可能能够在分支发散的直接后支配节点之前汇聚。

以下小节重点介绍了几项试图在这两个方面改进SIMT堆栈的工作。

多路径并行

当一个 warp 在分支处发生分歧时，线程被分成多个组，称为 *warp-splits*。每个 warp-split 包括遵循相同分支目标的线程。在基线情况下，*single path*，SIMT 堆栈中，同一个 warp 的 warp-splits 按照逐个执行的方式进行，直到 warp-split 达到其重汇点。这种串行化方式使硬件实现相对简单，但对于功能正确性来说并不是必须的。warp 中的线程拥有独立的寄存器，并通过内存操作和同步操作（如屏障）显式地相互通信。换句话说，同一个 warp 的每个 warp-split 都可能在 *parallel* 中执行。我们称这种执行模式为 *multi-path execution mode*。

虽然不同的warp-split可能不会在同一硬件上于同一时钟周期执行（毕竟它们运行不同的指令），但它们可能像多个warp在同一数据路径上交错执行一样，交错地在同一硬件上执行。通过这种方式，多路径执行模式提高了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，多路径执行仍然提升了内存受限应用程序的整体性能，因为SIMT核心有大量空闲周期可以用有用的工作填充。

示例 3.1 显示了一个可能受益于多路径执行的简短计算内核。在此示例中，两个分支目标中的代码路径都包含从内存加载的操作。在单路径 SIMT 栈中，即使 warp 分裂因等待来自内存的数据而停滞，块 B 和块 C 也会被连续执行，直到相应的 warp 分裂到达块 D（重汇合点）。这会使整个 warp 停滞，在数据路径中引入空闲周期，如图所示。

在图 3.18 中，这些需要由其他 warp 的工作来填充。通过多路径执行，块 B 和 C 的 warp 分割可以交错执行，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
```

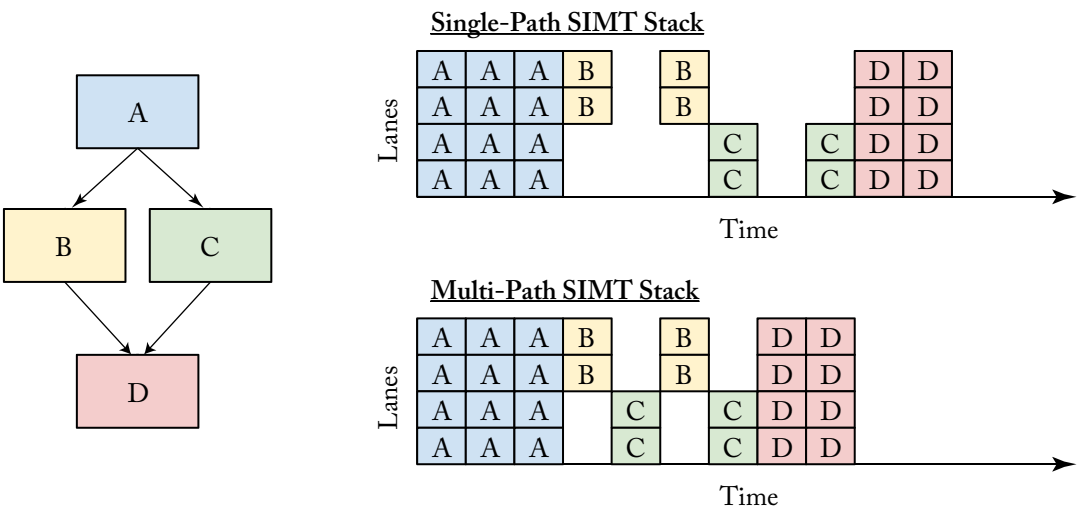


图 3.18：分支分歧处的多路径执行。

动态扭曲划分。Meng 等人 [2010] 提出 *dynamic warp subdivision* (DWS) 是首次利用多路径执行中的 TLP 提升的提案。DWS 扩展了 SIMT 堆栈，添加了一个扭曲分裂表，以将分歧的扭曲划分为并发的扭曲分裂。每个扭曲分裂分别执行分歧分支目标，可以并行执行，从而通过内存访问回收硬件闲置。扭曲分裂也会在内存分歧处创建——当一个扭曲中只有部分线程命中 L1 数据缓存时。DWS 不会等待所有线程获取数据，而是将扭曲分裂，并允许命中缓存的扭曲分裂提前执行，潜在地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），通过限制每个 warp 仅执行两个并发的 warp 分裂，解决了 DWS 的一些实现缺陷。虽然这种限制使 DPS 能够捕获大部分完整 DWS 的优势，但它带来了更简单的硬件设计。DPS 只需要通过增加一组额外的 PC 和活动掩码来扩展基线 SIMT 堆栈，用于编码额外的 warp 分裂。只有位于 warp 堆栈顶端的两个 warp 分裂可以并行执行；同一 warp 中的其他 warp 分裂会暂停，直到它们的条目到达堆栈顶端。此外，DPS 还伴随着对记分板的扩展，用于独立跟踪每个 warp 分裂的寄存器依赖性。这使得双路径执行模型能够在基线记分板的基础上实现比 DWS 更大的 TLP。

多路径执行。ElTantaway 等人 [2014] 使用多路径执行模型（MPM）消除了双路径的限制。MPM 用两个表替代了 SIMT 堆栈：一个是保持来自分歧 warp 的 warp 分裂集合的 warp-分裂表，另一个是同步具有相同重聚点的所有 warp 分裂的重聚表。

在一个分支分歧点，会在重聚表中创建一个新条目，记录分支分歧点的重聚点（即其直接后支配点）。在扭曲分裂表中会创建多个（通常为两个）条目，每个扭曲分裂对应一个条目。每个扭曲分裂条目维护该扭曲分裂的当前程序计数器（PC）、活动掩码、重聚PC（RPC），以及指向重聚表中对应条目的R索引。扭曲分裂表中的每个扭曲分裂都可以被执行，直到其PC == RPC。在这一点上，对应的重聚表条目会被更新，以反映来自该扭曲分裂的线程已经到达重聚点。当所有等待的线程都到达重聚点时，该重聚表条目会被释放，并使用重聚后的活跃线程创建一个新的扭曲分裂条目，从RPC开始执行。

MPM 还扩展了记分板，以跟踪每个线程的寄存器依赖性，而无需为每个线程完全复制记分板（这样做会因显著的面积开销而使 MPM 变得不切实际）。这是一个关键的扩展，使得 warp-splits 可以以真正独立的方式执行——如果没有该扩展，一个 warp-split 的寄存器依赖性可能会被误认为是同一 warp 的另一个 warp-split 的依赖性。

MPM 进一步通过机会性早期重收敛扩展，提升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，与第3.4.1节讨论的warp压缩技术是正交的。例如，TBC中的块范围SIMT堆栈可以通过扩展DWS来提升可用的TLP。

更好的收敛性

后支配者（PDOM）基于堆栈的重聚机制 [Fung 等人，2007；Fung 等人，2009] 使用统一算法识别的重聚点，而不是通过将源代码中的控制流习语翻译为指令 [AMD，2009；Coon 和

Lindholm, 2008，Levinthal 和 Porter, 1984]。被选择为重新汇合点的分支的直接后支配点是程序中分支线程 *guaranteed* 重新汇合的最早位置。在某些情况下，线程可以在 *earlier point* 重新汇合，如果硬件能够利用这一点，将提高 SIMD 效率。我们认为这一观察促使了最近 NVIDIA GPU 中包含 `break` 指令 [Coon 和 Lindholm, 2008]。

示例 3.2 中的代码（来自 [Fung 和 Aamodt, 2011]）展示了这种较早的重新汇聚。它生成了图 3.19 中的控制流图，其中边标注了单个标量线程沿该路径的概率。块 `F` 是 `A` 和 `C` 的直接后支配块，因为 `F` 是 *all* 起始于 `A`（或 `C`）的路径首次汇合的位置。在基线机制中，当一个 warp 在 `A` 处分岔时，重新汇聚点被设置为 `F`。然而，从 `C` 到 `D` 的路径很少被采用，因此在 *most* 的情况下，线程可以更早地在 `E` 处重新汇聚。

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

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

可能收敛点。Fung 和 Aamodt [2011] 提出扩展 SIMT 栈，添加 *likely convergence points*。此扩展为每个 SIMT 栈条目新增两个字段：一个用于记录可能收敛点的程序计数器 (LPC)，另一个 (LPos) 是指针，用于记录在分支具有与直接后支配点不同的可能收敛点时创建的特殊可能收敛条目的栈位置。每个分支的可能收敛点可以通过控制流分析或剖析信息（可能在运行时收集）来识别。Fung 和 Aamodt [2011] 的提案将可能收敛点限制为最近的包围向后执行分支，以捕获循环中“`break`”语句的影响 [Coon 和 Lindholm, 2008]。

当一个线程束（warp）在具有可能收敛点的分支处发生分歧时，会在SIMT堆栈上推入三个条目。第一个条目是为该分支的可能收敛点创建的LPC条目。另两个条目分别对应分支的执行路径（taken）和未执行路径（fall through），与基线机制中的创建方式相同。这两个其他条目的LPC字段被填充为分支的可能收敛点，LPos字段被填充为LPC条目的堆栈位置。LPC条目的RPC被设置为直接后支配点（即明确的）。

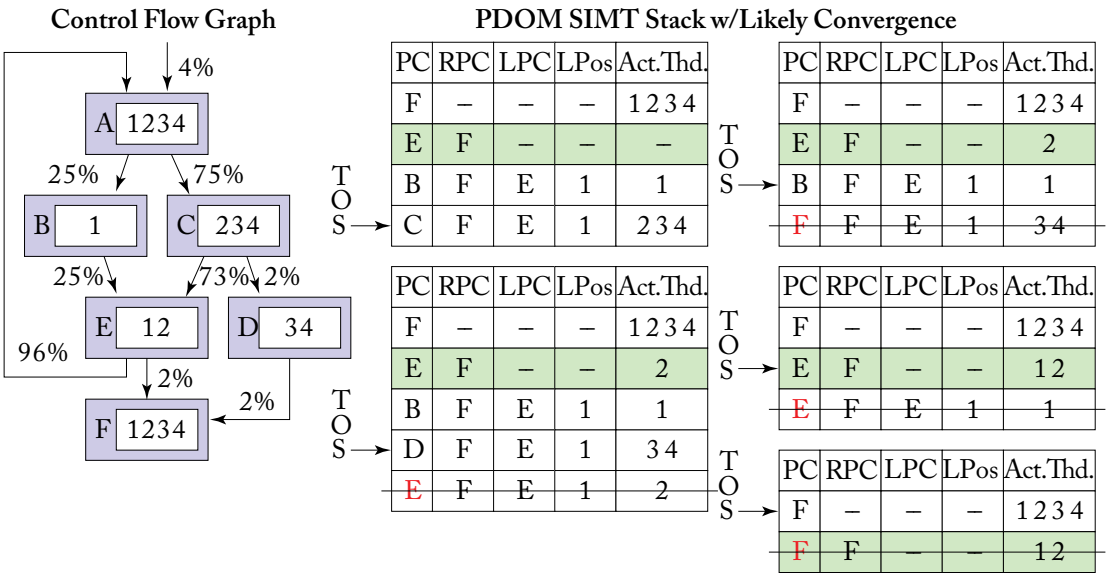


图 3.19：较早的重新收敛点 *before* 是直接后支配点。可能的收敛点在 E 捕获了这一较早的重新收敛。

汇合点，分支的发散部分，使得该入口中的线程将重新汇合到明确的汇合点。

当一个warp以SIMT栈中的顶部条目执行时，它会将其PC与RPC字段（与基线SIMT栈相同）以及LPC字段进行比较。如果PC == LPC，则弹出SIMT栈，并将弹出条目中的线程合并到LPC条目中。否则，如果PC == RPC，则仅弹出SIMT栈——RPC条目已在其活动掩码中记录了这些线程。当LPC条目到达SIMT栈顶部时，它会像任何其他SIMT栈条目一样被执行，或者如果其活动掩码为空，则直接弹出。

线程边界。Diamos 等人 [2011] 完全放弃了 SIMT 堆栈，转而提出通过 *thread frontiers* 在分歧后重新汇合线程。支持线程边界的编译器根据内核的拓扑顺序对基本块进行排序。通过这种方式，在较高 PC 处执行指令的线程永远不会跳转到较低 PC 处的指令。循环通过将循环出口放置在循环体的末尾来处理。在这种排序的代码布局中，分歧的线程束最终会通过优先处理较低 PC 的线程（使它们赶上）重新汇合。

与使用直接后支配点重新收敛的SIMT堆栈相比，通过线程边界进行重新收敛能够为具有非结构化控制流的应用程序提供更高的SIMD效率。多表达式条件语句的求值语义和异常的使用都可能生成具有非结构化控制流的代码。扩展了可能性的SIMT堆栈

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

机会主义的早期重收敛。ElTantaway 等人 [2014] 提出了 *opportunistic early reconvergence* (OREC)，在不需要额外编译器分析的情况下，提升了具有非结构化控制流的 GPU 应用的 SIMD 效率。OREC 基于同一篇论文中介绍的多路径（MP）SIMT 栈（参见第 3.4.2 节）。MP SIMT 栈使用一个单独的 warp 分裂表，记录当前可供执行的 warp 分裂集合。在分支发生分歧时，会创建新的 warp 分裂，其中包含分支目标程序计数器（PC）和分歧分支的重收敛 PC。通过 OREC，硬件不仅将这些新的 warp 分裂插入 warp 分裂表，还会在 warp 分裂表中搜索具有相同起始 PC 和重收敛 PC 的现有 warp 分裂。如果存在这样的 warp 分裂，硬件会在重收敛表中创建一个早期重收敛点，用于在原始重收敛 PC 之前将两个 warp 分裂收敛。早期重收敛点会在特定 PC 同步两个 warp 分裂，使其即使在现有的 warp 分裂已经沿分歧路径推进时也可以合并。在 ElTantaway 等人 [2014] 的研究中，早期重收敛点是现有 warp 分裂的下一个 PC。

3.4.3 添加 MIMD 功能

以下提案通过引入有限的 MIMD 功能，改善了 GPU 对分歧控制流的兼容性。所有这些提案都提供了两种操作模式：

- 一种SIMD模式，其中前端发出一条指令以在一个warp中的所有线程上执行；或

- 一种MIMD模式，其中前端为分支warp中的每个线程发出不同的指令。

当一个warp没有发生分歧时，它以SIMD模式执行，以捕获warp中线程表现出的控制流局部性，其能源效率与传统SIMD架构相当。当warp发生分歧时，它切换到MIMD模式。在这种模式下，warp的运行效率较低，但性能损失低于传统SIMD架构的损失。

向量线程架构。向量线程（VT）架构 [Krashinsky 等，2004] 结合了 SIMD 和 MIMD 架构的特点，旨在利用两种方法的优点。VT 架构具有一组与公共 L1 指令缓存连接的通道。在 SIMD 模式下，所有通道直接从 L1 指令缓存接收指令以进行同步执行，但每个通道可以切换到 MIMD 模型，独立运行其自身的 $v*$。

根据其 L0 缓存的指令执行操作。Lee 等人 [2011] 最近对传统 SIMT 架构（如 GPU）进行了比较，结果表明，VT 架构在处理常规并行应用程序时具有可比的效率，而在处理非规则并行应用程序时表现得更加高效。

时序SIMT。时序SIMT [Keckler等，2011，Krashinsky，2011] 允许每个通道以MIMD方式执行，类似于VT架构。然而，它不是在所有通道上同步运行一个warp，而是通过一个通道对warp的执行进行时间复用，每个通道运行一组单独的warp。时序SIMT通过为整个warp只获取一次指令，实现了SIMD硬件的效率。这在时间上均摊了控制流开销，而传统的SIMD架构在空间上通过多个通道均摊了相同的开销。

可变Warp大小架构。可变Warp大小（VWS）架构 [Rogers等人，2015] 包含多个（例如，8个）切片，每个切片包含一个取指和解码单元，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。与通过窄数据路径对大Warp进行时间复用不同，VWS中的每个切片由窄（4宽）Warp组成。这些窄Warp随后被分组为更大的执行实体，称为*gangs*。每个gang包含来自每个切片的一个Warp。

在没有分支分歧的应用程序中，一个 gang 中的 warp 以锁步方式执行，从共享的取指单元和共享的 L1 指令缓存获取指令。遇到分支分歧（或内存分歧）时，gang 会分裂为多个 gang。新的 gang 可能会进一步分裂，直到每个 warp 都在自己的 gang 中为止。在这一点上，这些单 warp gang 将通过 slice 的取指单元和私有的 L0 指令缓存各自独立地执行。这些分裂的 gang 会通过硬件比较各个 gang 的 PC 值，适时地重新合并为原始 gang。如果所有 PC 值都匹配，原始 gang 会被重新创建。Rogers 等人 [2015] 还提出在第一个分歧分支的直接后支配点插入一个 gang 级同步屏障。

本书还评估了每个切片中 L0 指令缓存容量对性能的影响，以及与共享 L1 指令缓存带宽的关系。在非联动模式下，切片中的 L0 缓存可能会同时从 L1 缓存请求指令，从而造成带宽瓶颈。他们的评估显示，即使对于分支性较大的应用程序，256 字节的 L0 缓存也可以过滤掉大多数对共享 L1 缓存的请求。因此，L1 缓存仅凭基线 SIMT 架构 2× 带宽就可以覆盖大部分带宽不足的问题。

同时分支交织。Brunie 等人 [2012] 在线程块压缩技术发表后，提出了同时分支和线程束交织（SBI 和 SWI）。他们扩展了 GPU SIMT 前端，以支持每个周期发出两条不同的指令。SBI 在遇到分支分歧时从同一线程束共同发出指令。同时执行分歧分支的两个目标显著消除了其性能损失。

基线SIMT堆栈对每个warp的面积需求仅为32 × 64位（或者在使用AMD GCN的优化时低至6 × 64位）。虽然与SIMT核心中的寄存器文件相比，这个面积很小，但该面积会随着GPU中并行执行的warp数量以及每个warp的线程数量直接扩展。此外，在分支发散较少的典型GPU应用中，SIMT堆栈占用的面积本可以用于以其他方式提升应用吞吐量。一些提案用替代机制取代了SIMT堆栈，这些机制在warp没有遇到任何分支发散时，可以将资源用于其他用途。

在标量寄存器文件中的SIMT栈。AMD GCN [AMD, 2012] 具有一个由一个warp中的所有线程共享的标量寄存器文件。其寄存器可以用作谓词寄存器，以控制warp中每个线程的活动。当编译器在计算内核中检测到潜在的分支分歧时，它使用该标量寄存器文件在软件中模拟一个SIMT栈。GCN架构具有专门的指令来加速SIMT栈的模拟。

一种优化方法是优先执行活动线程数较少的目标，从而最小化支持最坏情况分歧所需的标量寄存器数量。这使得最坏情况分歧可以通过 $\log_2(\#threads\ per\ warp)$ 个标量寄存器来支持，远少于基线 SIMT 堆栈所需的条目数。此外，当计算内核没有潜在的分歧分支时，编译器可以将为 SIMT 堆栈保留的标量寄存器用于其他标量计算。

线程前沿。正如第 3.4.2 节所述，Diamos 等人 [2011] 用 *thread frontiers* 替换了 SIMT 堆栈。通过线程前沿，每个线程在寄存器文件中维护自己的程序计数器（PC），并且代码按拓扑排序，使分支的重新收敛点始终具有更高的 PC。当一个 warp 分叉时，它总是优先处理所有线程中 PC 最低的线程。这组线程被称为 warp 的线程前沿。优先执行前沿中的线程会隐式地迫使程序中更前方的所有线程在分支的重新收敛点等待合并。

由于仅当计算内核包含潜在的分支分歧时才需要每线程的PC，因此编译器只需要在这些计算内核中分配一个PC寄存器。在其他计算内核中，额外的寄存器存储有助于提高warp占用率，从而增加每个SIMT核心可以维持的warp数量，以更好地容忍内存延迟。

无栈SIMT。Asanovic等人 [2013] 提议通过一个 `syncwarp` 指令扩展时间SIMT架构。在该提议中，当执行计算核的 *convergent regions* 时，warp 内的线程以锁步方式运行，编译器保证 warp 永远不会分岔。在分岔分支处，warp 中的每个线程利用时间SIMT架构中的MIMD能力，遵循其私有PC的控制流路径。

编译器在分支发散的重汇点放置了一条syncwarp指令。这迫使分歧的warp中的所有线程在进入计算内核的另一个收敛区域之前，在重汇点同步。

虽然该机制无法捕获嵌套分支分歧可能的重新汇合，但它仍然是一种更廉价的替代方案，可以为很少表现出分支分歧的GPU应用提供与基线SIMT堆栈相当的性能。本文提出了一种结合收敛和变异分析的方法，使编译器能够确定任意计算内核中符合 *scalarization* 和/或 *affine transfor- mation* 条件的操作。在无堆栈SIMT的上下文中，相同的分析使编译器能够确定任意计算内核中的收敛和分歧区域。

1. 编译器首先假设所有基本块为 *thread-invariant*。
2. 它将所有依赖线程 ID 的指令、原子指令以及对易失性内存的内存指令标记为 *thread-variant*。
3. 然后，它会迭代地将所有依赖于线程变量指令的指令也标记为线程变量指令。
4. 所有在线程变体分支指令上的基本块中的指令 *control dependent* 也都是线程变体。实质上，线程变体分支的直接后支配者之后的指令可能仍然是线程不变的，只要它们未因其他条件被标记为线程变体。

此分析使编译器能够检测到在每个 warp 中所有线程都一致执行的分支。由于这些分支不会导致 warp 分歧，编译器无需插入代码来检测这些分支的动态分歧，也无需在其直接后主导点插入 `syncwarp` 指令以强制重新汇合。

谓词化。在将完整的SIMT堆栈集成到架构之前，具有可编程着色器的GPU已经通过 *predications* 支持着色器程序中的有限控制流结构，就像传统的向量处理器一样。谓词化在现代GPU中仍然是一种低开销的方式，用于处理简单的if分支，从而避免推入和弹出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，以指定其谓词化寄存器。谓词化寄存器本质上是专用于控制流的标量寄存器。

Lee 等人 [2014b] 提出了一个 *thread-aware prediction algorithm*，它将谓词化的应用扩展到任意控制流，其性能与 NVIDIA 的 SIMT 堆栈相当。线程感知谓词化算法通过在每个分支处添加谓词化节点，扩展了标准控制流依赖图 (CDG)。每个基本块所需的预测可以基于它们的控制流依赖关系进行计算，并在不破坏功能行为的前提下进一步严格优化。论文随后描述了两个

基于此线程感知的CDG的优化，以及其先前工作中对收敛性和方差的分析 [Asanovic 等，2013]。

- 当编译器可以通过收敛分析保证整个 warp 中分支可以被统一采用时，应用 *Static branch-uniformity optimization*。在这种情况下，编译器可以用统一的分支指令替换谓词生成。 • 在其他情况下，应用 *Runtime branch-uniformity optimization*。编译器发出仅在谓词为空（即所有线程禁用）时才采用的共识分支（`cbranch.ifnone`）。这允许 warp 跳过带有空谓词的代码，这是 SIMT 堆栈提供的关键优势。此方法与用于向量处理器的先前方法（如 BOSCC）不同，它依赖于结构分析来确定此优化的候选对象。

尽管谓词化和SIMT堆栈在本质上以相似的能量和面积成本提供了相同的功能，Lee等人 [2014b] 强调了这两种方法之间的以下权衡。

- 由于不同分支目标由不同的谓词寄存器保护，编译器可以调度来自不同分支目标的指令，将不同分支目标的执行交错进行，以利用线程级并行性 (TLP)，而这通常需要更高级的硬件分支分歧管理来实现。- 谓词化往往会增加寄存器压力，从而降低线程束占用率，并带来整体性能损失。这是因为保守的寄存器分配无法在分支的两侧复用寄存器。它无法可靠地证明来自不同分支目标的指令在寄存器中的操作通道集是互斥的。两种优化方法插入的统一和一致的分支指令缓解了这个问题。- 谓词化可能以多种方式影响动态指令计数。在某些情况下，检查统一分支的开销会显著增加动态指令计数。或者，不执行检查意味着某些路径以空谓词掩码执行。在其他情况下，它移除了维持 SIMT 堆栈所需的 push/pop 指令。

最终，论文提出了新的指令以减少谓词化的开销。

- 对于函数调用和间接分支，他们提出了一种新的 `find_unique` 指令，通过循环顺序执行每个分支目标/函数。 - `cbranch.ifany` (除了现有的共识分支指令 `cbranch.ifnone` 和 `cbranch.ifall`) 外，还将有助于减少由动态统一分支检测引入的指令数量开销。

3.5 标量化和仿射执行的研究方向

正如第 2 章所述，GPU 计算 API（如 CUDA 和 OpenCL）具有类似 MIMD 的编程模型，使程序员可以将大量标量线程部署到 GPU 上。尽管每个标量线程可以遵循其独特的执行路径并可能访问任意内存位置，但在常见情况下，它们通常遵循一小组执行路径并执行类似的操作。GPU 线程之间的汇聚控制流在大多数（如果不是全部）现代 GPU 上通过 SIMT 执行模型加以利用，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观察到执行相同计算内核的线程之间的 *value structure* [Kim 等, 2013]。这两种值结构类型，*uniform* 和 *affine*，在示例 3.3 中的计算内核中进行了说明。

统一变量 在计算内核中，所有线程的值都相同且不变的变量。在算法3.3中，变量`a`以及字面量`THRESHOLD`和`Y_MAX_VALUE`在计算内核中的所有线程之间具有统一的值。统一变量可以存储在单个标量寄存器中，并被计算内核中的所有线程重复使用。

仿射变量 一个变量，其值对于计算核中的每个线程是线程 ID 的线性函数。在算法 3.3 中，变量 `y[idx]` 的内存地址可以表示为线程 ID `threadIdx.x` 的 *affine* 变换：

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

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

关于如何在GPU中对 *detect* 和 *exploit* 进行统一或仿射变量的研究提案有很多。本节的其余部分从这两个方面总结了这些提案。

3.5.1 检测均匀或仿射变量

在 GPU 计算内核中检测统一变量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。这是可行的，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某种类型的方法。

算法 3.3 计算内核中标量和仿射操作的示例（摘自 [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;
}
```

在整个计算内核中保持恒定，并提供用于线程 ID 的特殊变量。编译器可以执行控制依赖性分析，以检测仅依赖于常量和线程 ID 的变量，并将其标记为统一/仿射。仅作用于统一/仿射变量的操作随后成为 *scalarization* 的候选。

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

Asanovic 等人 [2013] 引入了一种结合收敛和变体分析的方法，使编译器能够确定任意计算内核中适用于 *scalariza-*、*tion* 和/或 *affine transformation* 的操作。计算内核的收敛区域内的指令可以转换为标量/仿射指令。在从计算内核的发散区域到收敛区域的任何过渡中，编译器会插入 `syncwarp` 指令，以处理两个区域之间由控制流引起的寄存器依赖性。Asanovic 等人 [2013] 采用了这种分析方法，为 Temporal-SIMT 架构 [Keckler 等人, 2011, Krashinsky, 2011] 生成标量操作。

解耦仿射计算（DAC）[Wang 和 Lin, 2017] 依赖于类似的编译器分析，以提取标量和仿射候选者，并将其解耦到一个单独的 warp 中。Wang 和 Lin [2017] 在此过程中增加了一个分歧仿射分析，目的是提取从计算内核开始就已是仿射的指令链。这些仿射指令链从主内核中解耦，进入一个仿射内核，并通过硬件队列向主内核提供数据。

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

1. 这允许标量化和仿射执行与原始GPU指令集架构一起应用。这节省了与硬件共同开发专用标量化编译器的工作。

2. 硬件检测在计算内核执行期间发生。因此，它能够检测到动态出现的均匀/仿射变量，而这些变量被静态分析所忽略。

基于标签的检测。Collange 等人 [2010] 引入了一种基于标签的检测系统。在该系统中，每个 GPU 寄存器都被扩展为一个标签，指示寄存器中是否包含统一、仿射或通用向量值。在计算内核启动时，包含线程 ID 的寄存器的标签被设置为仿射状态。从常量或共享内存中单一位置广播值的指令会将目标寄存器的标签设置为统一状态。在内核执行期间，根据表 3.1 中的简单规则，寄存器的状态通过算术指令从源操作数传播到目标操作数。尽管这种基于标签的检测硬件开销很小，但它往往是保守的——例如，它保守地将统一和仿射变量之间的乘法输出视为向量变量。

表 3.1：来自 Collange 等人 [2010] 的关于指令中统一和仿射状态传播规则的示例。对于每个操作，第一行和第一列显示输入操作数的状态，其余条目显示每种输入操作数状态排列下输出操作数的状态（U = 统一，A = 仿射，= 向量）。

+	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 等人, 2013] 将 Collange 等人 [2010] 提出的基于标签的检测机制进行了扩展，为分支提供了更好的支持。仿射分支，或者比较仿射操作数的分支，如果其中一个操作数是统一的，则通过标量数据路径解决。Kim 等人 [2013] 还提出了一种 *lazy expansion* 方案，其中仿射寄存器在分支分歧或谓词指令之后被懒惰地扩展为完整的向量寄存器。这种扩展是必要的，以允许分歧 warp 中的一个线程子集更新目标寄存器中的槽位，而其他槽位保持不变——这保持了 SIMT 执行语义。相比于一种更简单、急切的扩展方案（在第一个分支分歧后扩展每个仿射寄存器），懒惰扩展方案消除了许多不必要的扩展。

写回时的比较。Gilani 等人 [2013] 引入了一种更激进的机制，通过在每次向量指令写回时比较一个线程束中所有线程的寄存器值来检测统一变量。在检测到统一变量时，检测逻辑会将写回重定向到标量寄存器文件，并更新内部表以记录寄存器的状态。随后对该寄存器的使用将被重定向到标量寄存器文件。所有操作数均来自标量寄存器文件的指令将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。他们没有采用简单的统一检测器，而是在寄存器写回阶段增加了一个寄存器值压缩器，该压缩器使用 Pekhimenko 等人 [2012] 提出的算法将输入的值向量转换为 $< base, delta, immediate >$ (BDI) 元组。

Wong 等人 [2016] 提出了 *Warp Approximation*，这是一个利用 warp 内近似计算的框架，该框架还具有在寄存器回写时的检测功能。检测器计算向寄存器文件回写的向量中所有值中与两个给定值共享 *d*-MSB 的最小 *d-similarity*。具有高于阈值 d-相似度的寄存器被标记为 *similar*，随后用于确定后续依赖指令中近似执行的适用性。

与 Lee 等人 [2015] 的提案类似，G-Scalar [Liu 等人, 2017] 在寄存器回写阶段引入了一个寄存器值压缩器，但该压缩器采用了更简单的算法，仅提取所有通道中所有值所共用的字节。如果所有字节都是共用的，则该寄存器包含一个统一变量。任何仅对统一变量操作的指令都可以被标量化。

G-Scalar还扩展了寄存器值压缩器，以检测在分支分歧下适合标量执行的操作。所有先前的提案在线程束分歧时都会回退到矢量执行。Liu等人[2017]观察到，在许多分支分歧下的指令中，活动通道的操作数值是统一的。使用这些部分统一寄存器的指令实际上适合标量执行。他们随后扩展了寄存器值压缩器，通过一种特殊逻辑仅检查活动通道的值。这大大增加了各种GPU计算工作负载中的标量指令数量。需要注意的是，在分歧下，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存在的值结构。

压缩寄存器存储

统一变量和仿射变量的紧凑表示使它们能够以更少的位数存储在寄存器文件中。节省下来的存储空间可以用于支持更多的飞行线程，从而在使用相同的寄存器文件资源的情况下，提高GPU对内存延迟的容忍度。

标量寄存器文件。许多提案/设计在 GPU 功能中利用统一或仿射变量，为标量/仿射值提供专用的寄存器文件。

- AMD GCN 架构具有一个标量寄存器文件，该寄存器文件可被标量和矢量流水线访问。 - FG-SIMT 架构 [Kim 等人, 2013] 将统一/仿射值存储在一个单独的仿射 SIMT 寄存器文件 (ASRF) 中。ASRF 记录每个寄存器的状态（仿射/统一/矢量）。

寄存器，使控制逻辑能够检测可直接在控制处理器上执行的操作。

- 来自 Gilani 等人 [2013] 的动态统一检测提议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] 将基值、增量、立即数（BDI）压缩应用于写回寄存器文件的寄存器。被压缩的寄存器在作为源操作数读回时会解压为正常向量。在这种方案中，每个压缩的寄存器仍然占用与未压缩寄存器相同的存储槽，但仅占用寄存器组的一个子集，因此读取寄存器的压缩表示形式消耗的能量更少。

Warp Approximate 架构 [Wong 等, 2016] 通过仅访问通过相似性检测选择的代表线程对应的通道，减少了寄存器读写的能量消耗。

同样地，G-Scalar [Liu 等, 2017] 采用压缩寄存器，这些寄存器仅占用为未压缩寄存器分配的一部分存储单元，从而降低寄存器读取的能耗。

专用仿射扭曲。解耦仿射计算（DAC）[Wang and Lin, 2017] 将所有由编译器提取的仿射变量缓存在专用仿射扭曲的寄存器中。该仿射扭曲与其他非仿射扭曲一样共享相同的矢量寄存器文件存储，但仿射扭曲使用每个寄存器条目的单独通道存储基数和不同非仿射扭曲的增量。

标量化操作

除了高效存储外，使用统一或仿射变量的操作可以是 *scalarized*。与通过SIMD数据路径在一个warp中的所有线程上重复相同操作不同，标量操作可以在单一标量数据路径上执行一次，从而在此过程中消耗更少的能量。一般来说，如果其输入操作数仅由统一或仿射变量组成，则算术操作可以被标量化。

专用标量流水线。AMD 的 GCN 架构具有一个专用的标量流水线，用于执行由编译器生成的标量指令。FG-SIMT 架构 [Kim et al., 2013] 配备了一个控制处理器，该处理器能够直接执行动态检测的仿射操作，而无需调用 SIMD 数据路径。

在这两种实现中，标量流水线还负责处理SIMD流水线的控制流和谓词操作。这种解耦意味着许多系统相关功能（例如，与主处理器的通信）也可以卸载到标量流水线，从而免除SIMD数据路径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] 都在其中一个动态检测的标量指令上执行。

SIMD 数据路径中的通道。当这种情况发生时，其他通道会进行时钟门控以减少动态功耗。

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

聚合为仿射warp。解耦仿射计算（DAC）[Wang and Lin, 2017] 将多个warp的仿射操作聚合为每个SIMT核心中的单个仿射warp。这个仿射warp与其他warp一样在SIMD数据路径上执行，但每条指令的执行会同时作用于多个warp的仿射表示。

内存访问加速

当使用统一变量或仿射变量表示内存操作（加载/存储）的地址时，内存操作触及的内存位置是高度可预测的——每个连续位置之间的间隔为已知的步长。这允许进行各种优化。例如，具有已知步长的内存位置的合并要比任意随机位置的合并简单得多。仿射变量还可以用来通过单条指令表示批量传输，而不是通过加载/存储指令的循环实现。

FG-SIMT架构 [Kim et al., 2013] 在控制过程中具有一个特殊的地址生成单元，用于将具有仿射地址的内存访问扩展为实际地址。由于仿射地址在线程之间具有固定的步幅，将这些仿射内存访问合并到缓存行中可以通过更简单的硬件实现。

解耦仿射计算（DAC）[Wang 和 Lin, 2017] 也具有类似的优化，用于利用仿射内存访问中的固定步幅。此外，它使用仿射warp领先于其他非仿射warp执行，为这些warp预取数据。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 使用大量的硬件线程（warp），在数量远少于线程数（但仍然庞大）的 ALU 上多路复用它们的执行，以容忍流水线和内存访问延迟。为了实现 warp 之间的快速高效切换，GPU 使用硬件 warp 调度器，并将所有硬件线程的寄存器存储在片上寄存器文件中。在许多 GPU 架构中，由于 GPU 使用的宽 SIMD 数据路径，以及为容忍数百个周期的内存访问延迟所需的大量 warp，这些寄存器文件的容量通常相当大，有时甚至超过最后一级缓存的容量。例如，

NVIDIA 的 Fermi GPU 可以维持超过 20,000 个正在执行的线程，并具有 2 MB 的总寄存器容量。

为了最小化寄存器文件存储所占用的面积，GPU上的寄存器文件通常通过低端口计数的SRAM存储单元实现。SRAM存储单元被并行访问，以提供所需的操作数带宽，从而在宽SIMD流水线的峰值吞吐量下维持指令运行。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，以最小化存储单元冲突的惩罚。

访问这些大型寄存器文件在每次访问时都会消耗大量的动态能量，其大的尺寸也会导致较高的静态功耗。在 NVIDIA GTX280 GPU 上，寄存器文件几乎消耗了 GPU 总功耗的 10%。这为创新 GPU 寄存器文件架构以降低其能耗提供了明确的动机。因此，近年来在该主题上发表了大量的研究论文。本节的其余部分总结了若干旨在实现这一目标的研究提案。

3.6.1 分级寄存器文件

Gebhart 等人 [2011b] 观察到，在一组实际的图形和计算工作负载中，一条指令生成的值中多达 70% 仅被读取一次，而只有 10% 被读取超过两次。为了捕捉大多数寄存器值的这种短生命周期，他们提出在 GPU 上扩展一个 *register file cache* (RFC) 的主寄存器文件。这形成了寄存器文件的层次结构，并显著降低了对主寄存器文件的访问频率。

在这项工作中，RFC 为每条指令的目标操作数分配一个新条目，使用 FIFO 替换策略。未命中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 驱逐的每个值都会写回到主寄存器文件。然而，由于许多值从未再次被读取，Gebhart 等人 [2011b] 将仅硬件实现的 RFC 扩展为结合编译时生成的静态活性信息。在指令编码中增加了一个额外的位，用于指示消耗寄存器值的最后一条指令。已被最后一次读取的寄存器在 RFC 中被标记为死亡。在驱逐时，它不会被写回主寄存器文件。

为了进一步减少RFC的大小，Gebhart等人 [2011b] 将其与两级warp调度器结合。该两级warp调度器将执行限制在一个包含*active*个warp的池中，该池仅由每个SIMT核心中warp的一个小子集组成。该工作考虑了一个由4 – 8个warp组成的活动warp池，而每个SIMT核心总共有32个warp。RFC仅保存来自活动warp的值，因此体积更小。在发生长延迟操作（如全局内存加载或纹理获取）时，warp会从活动池中移除。当这种情况发生时，该warp的RFC条目会被清除，从而为由二级调度器激活的其他warp腾出空间。

编译时管理的寄存器文件层次结构。Gebhart 等人 [2011a] 进一步扩展了这种寄存器文件层次结构，引入了一个“最后结果文件”（Last Result File，LRF），其作用是缓冲每个活跃 warp 的最后一条指令产生的寄存器值。该工作还将硬件管理的 RFC 替换为编译时管理的操作数寄存器文件（Operand Register File，ORF）。ORF 中值的进出由编译器显式管理，从而消除了 RFC 所需的标签查找操作。编译器还能够更全面地了解大多数 GPU 工作负载中的寄存器使用模式，从而做出更优的决策。该研究还扩展了两级 warp 调度器，使编译器能够指示何时可以将一个 warp 从活跃池中切换出去。为了协调 ORF 的内容与 warp 的活跃状态，在 warp 被切换出去之前，需要将 ORF 中所有的活动数据移动回主寄存器文件。

3.6.2 瞌睡状态寄存器文件

Abdel-Majeed 和 Annavaram [2013] 提出了一个三模寄存器文件设计，能够减少大型 GPU 寄存器文件的泄漏功耗。三模寄存器文件中的每个条目可以在 ON、OFF 和 Drowsy 模式之间切换。ON 模式是正常的操作模式；OFF 模式不保留寄存器的值；Drowsy 模式保留寄存器的值，但在访问之前需要将条目唤醒到 ON 模式。在该研究中，所有未分配的寄存器都处于 OFF 模式，所有已分配的寄存器在每次访问后立即进入 Drowsy 状态。该策略利用了 GPU 上对同一寄存器连续访问之间的长延迟（由于 GPU 的细粒度多线程特性），使寄存器文件中的寄存器大部分时间处于 Drowsy 模式。GPU 中的长流水线也意味着将寄存器从 Drowsy 状态唤醒所需的额外延迟不会引入显著的性能损失。

3.6.3 寄存器文件虚拟化

Tarjan 和 Skadron [2011] 观察到，在等待内存操作时，GPU 线程中的活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。他们提出通过使用寄存器重命名来虚拟化物理寄存器，从而将物理寄存器文件的大小减少最多 50%，或将并发执行线程的数量翻倍。在所提出的机制中，线程在开始执行时没有分配任何寄存器，物理寄存器在指令解码时被分配给目标寄存器。Tarjan 和 Skadron 进一步建议，通过使用编译器分析来确定寄存器的最后一次读取，可以增强物理寄存器的释放。他们提出了“最后读取注释”，并建议为每个操作数添加“一个位以指示是否为最后一次读取”，并指出这可能需要在指令编码中增加额外的位。

Jeon 等人 [2015] 通过将寄存器溢出到内存来量化减少 GPU 寄存器文件大小的影响。他们发现，通过采用溢出将寄存器文件大小减少 50%，执行时间平均增加了 73%。他们回顾了在使用乱序执行的 CPU 上采用寄存器重命名时，提前回收物理寄存器的较早提案。

执行。为了解决增加“最终读取注释”所需的额外位数问题，提出通过添加“元数据指令”来高效编码物理寄存器何时可以被回收，并通过寄存器生命周期活跃性分析生成这些指令。他们的重要观察是，在确定何处可以安全地回收物理寄存器时，必须考虑分支分歧（由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，通过不对生命周期较长的寄存器进行重命名，这一开销可以减少到1 KB。为了利用这一机会，他们提议仅对逻辑寄存器号大于编译器确定的阈值的寄存器进行重命名。Jeon等人还进一步提出，通过重命名实现寄存器文件子阵列的电源门控。他们评估了支持通过寄存器重命名实现寄存器文件虚拟化的详细提案的有效性，显示将寄存器文件的大小减少50%且性能无损失确实是可以实现的。

3.6.4 分区寄存器文件

Abdel-Majeed 等人 [2017] 引入了 *Pilot Register File*，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 的访问能耗和漏电功率都要低得多。作为交换，访问 NTV SRAM 的延迟要慢得多，通常需要多个周期（而常规 SRAM 通常只需一个周期）。在这项工作中，SRF 的规模显著大于 FRF。每个 warp 在 FRF 中有 4 个条目。关键在于利用 FRF 服务大部分访问，以弥补 SRF 的慢速。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 还增强了低功耗模式。这使得非活动 warp 的 FRF 可以切换到低功耗模式，从而使 FRF 在无需显式调度 warp 进入和退出活动池的情况下，获得两级调度器的优势。

该工作与分层寄存器文件不同，不同分区持有一组独占的寄存器，并且分区在整个 warp 生命周期中保持不变。Abdel-Majeed 等人 [2017] 并未使用编译器来确定放置在 FRF 中的寄存器集合，而是在每次内核启动时采用试点 CTA 对最常使用的寄存器进行分析。这组高频使用的寄存器被记录在一个查找表中，该表可供内核启动后的每个 warp 访问。

3.6.5 无回归

Kloosterman 等人 [2017] 引入了 *RegLess*，旨在消除寄存器文件并用操作数暂存缓冲区替代。论文观察到，在相对较短的时间跨度内，访问的寄存器数量仅占寄存器文件总容量的一小部分。例如，在 100 个周期内，他们评估的许多应用程序访问的寄存器数量少于

在使用 GTO 或两级 warp 调度器时，占用 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。区域之间的边界是根据限制活跃寄存器的数量来选择的。利用区域注释，容量管理器（CM）确定哪些 warp 有资格进行调度。当一个 warp 开始执行一个新区域的指令时，该区域使用的寄存器会从分配在全局内存中的后备存储区域加载到操作数暂存单元（OSU），这些寄存器可能会缓存到 L1 数据缓存中。OSU 本质上是一个由八个存储体组成的缓存，能够提供足够的带宽以支持每周期两条指令的处理。为了避免在 OSU 中访问数据时发生停顿，CM 在发出区域中的第一条指令之前预加载寄存器。为管理预加载过程，CM 为每个 warp 维护一个状态机，用于指示 OSU 中是否存在下一区域所需的寄存器。为了减少 OSU 与内存层次结构之间产生的内存流量，RegLess 采用了寄存器压缩技术，这些技术利用了仿射值（参见第 3.5 节）。

Kloosterman 等人对其提案进行了详细评估，包括 Verilog 合成以及由 RegLess 引入的硬件单元的寄生电容和电阻值提取。评估结果显示，512 项 OSU 在性能上略优于 2048 KB 的寄存器文件，同时只占用其 25% 的空间，并将整体 GPU 能耗降低了 11%。

CHAPTER 4

记忆系统

本章探讨了GPU的内存系统。GPU计算内核通过加载和存储指令与内存系统交互。传统的图形应用程序与多个内存空间交互，例如纹理、常量和渲染表面。虽然这些内存空间可以通过CUDA等GPGPU编程API访问，但本章将重点讨论GPGPU编程中使用的内存空间，尤其是实现这些内存空间所需的微架构支持。

CPU 通常包括两个独立的内存空间：寄存器文件和内存。现代 GPU 在逻辑上进一步将内存细分为局部内存空间和全局内存空间。局部内存空间是每个线程私有的，通常用于寄存器溢出，而全局内存用于多个线程共享的数据结构。此外，现代 GPU 通常实现了一种程序员管理的暂存内存（scratchpad memory），该内存可以在协作线程数组中被执行的线程共享访问。引入共享地址空间的一个动机是，在许多应用中，程序员知道在计算的特定步骤中需要访问哪些数据。通过将所有这些数据一次性加载到共享内存中，他们可以在长延迟的片外内存访问时重叠进行计算，从而避免在处理这些数据时对内存的长延迟访问。更重要的是，在给定时间内，GPU 和片外内存之间可以传输的字节数（DRAM 带宽）相对于同一时间内可以执行的指令数量而言是较小的。此外，将数据从片外内存传输到 GPU 所消耗的能量比从片上内存访问数据消耗的能量高出几个数量级。因此，从片上内存访问数据能够提供更高的性能并节省能量。

我们将关于内存系统的讨论分为两部分，分别对应内存划分为驻留在GPU核心内的部分和连接到芯片外DRAM芯片的内存分区的部分。

4.1 一级内存结构

本节描述了 GPU 上的一级缓存结构，重点是统一的 L1 数据缓存和用于“ 共享内存 ”的暂存区，以及它们如何与核心流水线交互。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 计算应用中用途有限，但提供了一些关于 GPU 与 CPU 区别的见解和直觉。最近的一项专利描述了如何将纹理缓存和 L1 数据缓存统一起来（例如，在 NVIDIA 的实现中）。

Maxwell 和 Pascal GPUs）[Heinrich 等，2017]。关于此设计的讨论，我们将推迟到首先考虑纹理缓存的组织方式之后。GPU 中一级存储结构的一个有趣方面是它们在遇到冲突时如何与核心流水线交互。如第 3 章所述，可以通过重新执行指令来处理流水线冲突。在本章中，我们将扩展之前关于重新执行的讨论，重点关注内存系统中的冲突。

4.1.1 记事本存储器与 L1 数据缓存

在 CUDA 编程模型中，“ 共享内存 ”指的是一种相对较小的内存空间，预计具有低延迟，但可以被同一个 CTA 内的所有线程访问。在其他架构中，这种内存空间有时被称为暂存内存（scratchpad memory）[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。实际上，早期的 NVIDIA 专利将 CUDA “ 共享内存 ”称为全局寄存器文件（Global Register File）[Acocella and Goudy, 2010]。在 OpenCL 中，这种内存空间被称为 “ 局部内存 ”。从程序员的角度来看，使用共享内存时需要考虑的一个关键方面（除了其容量有限之外）是潜在的 *bank conflicts*。共享内存实现为静态随机存取存储器（SRAM），一些专利[Minkin et al., 2012]中描述其实现方式为每个通道（lane）对应一个存储库（bank），每个存储库具有一个读端口和一个写端口。每个线程可以访问所有存储库。如果在某一周期中有多个线程访问同一个存储库且这些线程希望访问该存储库中的不同位置，就会发生 *bank conflict*。在详细讨论共享内存的实现方式之前，我们先来看看 L1 数据缓存。

L1数据缓存在缓存中维护全局内存地址空间的一个子集。在某些架构中，L1缓存仅包含未被内核修改的位置，这有助于避免由于GPU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不同线程在给定warp中访问的内存位置彼此之间的关系。如果warp中的所有线程访问的内存位置都位于单个L1数据缓存块中，并且该块不在缓存中，则只需向更低级缓存发送一个请求。这种访问被称为 “ 合并访问 ”。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请求。这种访问被称为非合并访问。程序员试图避免银行冲突和非合并访问，但为了简化编程，硬件允许两者的存在。

图4.1展示了类似于Minkin等人[2012]所描述的GPU缓存组织结构。图中所示的设计实现了统一的共享内存和L1数据缓存，这一特性在NVIDIA的Fermi架构中首次引入，也存在于Kepler架构中。图的中心是一个SRAM数据阵列，可通过配置[Minkin等人, 2013]部分用于共享内存的直接映射访问，部分作为集合关联缓存。该设计通过使用重放机制在处理银行冲突和L1数据缓存未命中时支持与指令流水线的无停顿接口。为了帮助解释其操作方式，

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

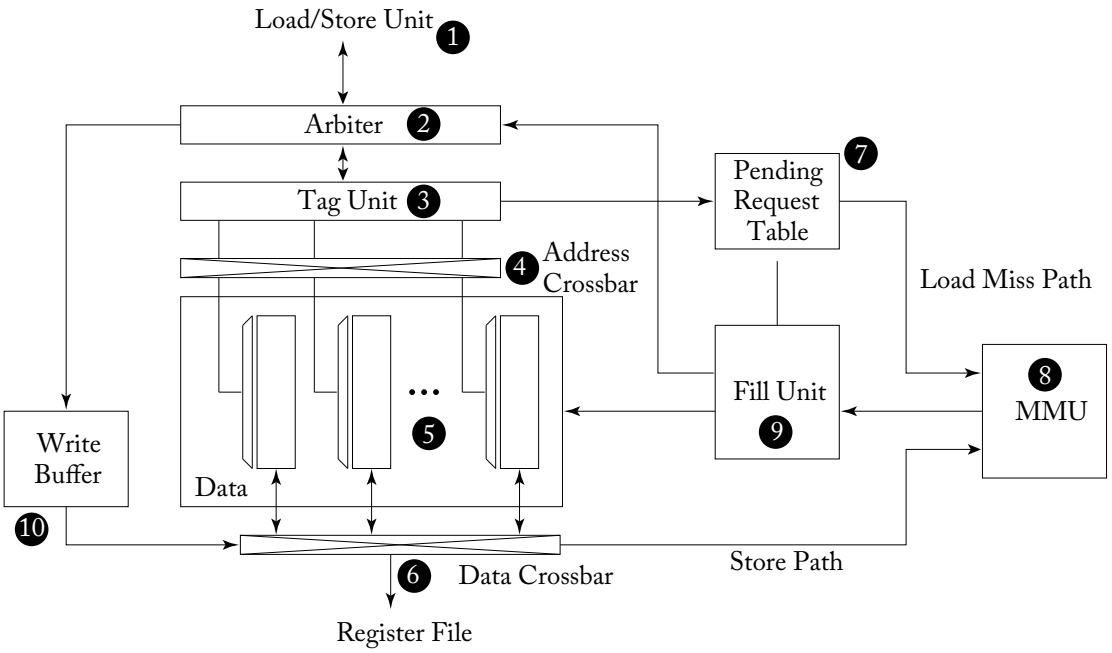


图 4.1：统一的 L1 数据缓存和共享内存 [Minkin 等, 2012]。

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银行冲突。如果请求的地址会引发一个或多个银行冲突，仲裁器会将请求分为两部分。第一部分包含warp中不引发银行冲突的线程地址。仲裁器接受这一部分的原始请求，进行缓存的进一步处理。第二部分包含与第一部分地址发生银行冲突的那些地址。这一部分原始请求将返回到指令流水线，并且必须重新执行。此后续执行被称为“重放”。在存储原始共享内存请求的重放部分的位置上存在权衡。虽然通过从指令缓冲区重放内存访问指令可以节省面积，但这会消耗访问大型寄存器文件的能量。一种更节能的替代方案可能是为重放内存访问指令提供有限的缓冲。

加载/存储单元，并在指令缓冲区的可用空间开始耗尽时避免调度内存访问操作。在考虑重放请求会发生什么之前，让我们先考虑已接受的内存请求部分是如何处理的。

共享内存请求的被接受部分绕过了标签单元3中的标签查找，因为共享内存是直接映射的。在接受共享内存加载请求时，仲裁器会调度一个写回事件到指令流水线中的寄存器文件中，因为在没有银行冲突的情况下，直接映射内存查找的延迟是恒定的。标签单元确定每个线程的请求映射到哪个银行，以便控制地址交叉开关4，该交叉开关将地址分配到数据阵列中的各个银行。数据阵列中的每个银行5宽度为32位，并具有自己的解码器，允许每个银行中的不同行独立访问。数据通过数据交叉开关6返回到相应线程的通道，并存储在寄存器文件中。仅与活跃线程对应的通道会向寄存器文件写入值。

假设共享内存查找具有单周期延迟，重新播放的共享内存请求部分可以在前一个被接受部分之后的一个周期访问 L1 缓存仲裁器。如果该重新播放部分遇到银行冲突，它将进一步被划分为一个被接受部分和一个重新播放部分。

缓存读取操作

接下来，让我们考虑如何处理对全局内存空间的加载。由于只有全局内存空间的一个子集被缓存在 L1 缓存中，因此标签单元需要检查数据是否存在于缓存中。尽管数据数组被高度分块以便于单个 warp 灵活访问共享内存，但对全局内存的访问每个周期仅限于单个缓存块。这种限制有助于相对于缓存数据量减少标签存储开销，同时也是标准 DRAM 芯片标准接口的结果。在 Fermi 和 Kepler 架构中，L1 缓存块大小为 128 字节，而在 Maxwell 和 Pascal 架构中进一步被划分为四个 32 字节的扇区 [Liptay, 1968] [NVIDIA Corp.]。32 字节扇区大小对应于从现代图形 DRAM 芯片（例如 GDDR5）中单次访问时可以读取的最小数据大小。每个 128 字节的缓存块由 32 个银行中同一行的 32 位条目组成。

存储/加载单元1计算内存地址，并应用合并规则将一个warp的内存访问分解为单个合并访问，然后将其传递到仲裁器2。若资源不足，仲裁器可能会拒绝请求。例如，如果访问映射到的缓存组中的所有路都处于繁忙状态，或者待处理请求表7（如下所述）中没有空闲条目。假设有足够的资源来处理未命中情况，仲裁器请求指令流水线在与缓存命中相对应的未来固定周期内安排写回到寄存器文件。与此同时，仲裁器还请求标签单元3检查该访问是否实际导致缓存命中或未命中。如果发生缓存命中，将在所有存储银行中访问数据阵列5的相应行。

数据被返回到指令流水线中的寄存器文件中。与共享内存访问的情况一样，仅更新与活动线程对应的寄存器通道。

当访问标签单元时，如果确定某个请求触发了缓存未命中，仲裁器会通知加载/存储单元必须重放请求，并且同时将请求信息发送到未决请求表（PRT）7。未决请求表提供的功能与传统的未命中状态保持寄存器（Miss-Status Holding Registers, MSHR）[Kroft, 1981] 在 CPU 缓存存储系统中支持的功能类似。NVIDIA 专利 [Minkin et al., 2012, Nyland et al., 2011] 中描述了至少两种版本的未决请求表。与图 4.1 中显示的 L1 缓存架构相关的版本看起来与传统的 MSHR 有些相似。传统数据缓存的 MSHR 包含缓存未命中块的地址，以及块偏移量和在块填充到缓存时需要写入的相关寄存器信息。通过记录多个块偏移量和寄存器，支持对同一块的多次未命中。图 4.1 中的 PRT 支持对同一块的两个请求进行合并，并记录通知指令流水线重放哪个延迟内存访问所需的信息。

L1数据缓存如图4.1所示，是虚拟索引和虚拟标记的。这可能与现代CPU微架构形成对比令人感到意外，因为现代CPU大多采用虚拟索引/物理标记的L1数据缓存。CPU使用这种组织方式来避免在上下文切换时清空L1数据缓存的开销[Hennessy和Patterson, 2011]。尽管GPU在每个warp发出时每个周期都有效地执行上下文切换，但这些warp是同一应用程序的一部分。即使GPU每次只能运行单一操作系统应用程序，基于页的虚拟内存仍然是有利的，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，内存请求会被转发到内存管理单元(MMU)进行虚拟地址到物理地址的翻译，然后通过交叉互连转发到相应的内存分区单元。如4.3节将进一步说明，内存分区单元包含一个L2缓存组以及一个内存访问调度器。除了关于访问哪个物理内存地址以及读取多少字节的信息外，内存请求还包含一个“ subid ”，用于在内存请求返回核心时查找PRT中关于该请求的信息。

一旦加载的内存请求响应返回到核心，它会通过MMU传递给填充单元9。填充单元随后使用内存请求中的subid字段在PRT中查找有关请求的信息。这包括可以由填充单元通过仲裁器2传递给加载/存储单元的信息，以重新调度加载，并通过在数据阵列5中放置数据后锁定缓存中的行来保证其命中缓存。

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，对全局内存的存储指令（写入）可以通过多种方式处理。具体的内存

空间写入决定了写操作是被视为直写 (write through) 还是回写 (write back)。在许多 GPGPU 应用中，对全局内存的访问通常被认为具有非常差的时间局部性，因为常见的内核编写方式是线程在退出之前将数据写入一个大的数组。对于这样的访问，采用“直写且无写分配”策略 [Hennessy 和 Patterson, 2011] 可能是合理的。相比之下，将寄存器溢出写入堆栈的本地内存写操作可能会表现出较好的时间局部性，随后加载操作也能够证明“回写且写分配”策略 [Hennessy 和 Patterson, 2011] 的合理性。

要写入共享内存或全局内存的数据首先被放置在写数据缓冲区 (WDB) 10 中。对于未合并的访问或[redacted]些线程被屏蔽的情况，仅写入缓存块的一部分。如果该块存在于缓存中，数据可以通过数据交叉开关 6 写入数据阵列。如果数据不在缓存中，则必须先从 L2 缓存或 DRAM 内存读取该块。[redacted]完全填充缓存块的合并写操作可能会绕过缓存，但前提是它们会使缓存中任何过时数据的标记失效。

请注意，图 4.1 中描述的缓存组织不支持缓存一致性。例如，假设在线程块 SM 1 上执行的线程读取了内存位置 A，并将该值存储在 SM 1 的 L1 数据缓存中，然后在线程块 SM 2 上执行的另一个线程写入了内存位置 A。如果随后 SM 1 上的任何线程在该值从 SM 1 的 L1 数据缓存中被驱逐之前再次读取内存位置 A，它将获得旧值而不是新值。为避免此问题，从 Kepler 开始，NVIDIA GPU 仅允许用于寄存器溢出和堆栈数据或只读全局内存数据的本地内存访问放置在 L1 数据缓存中。最近的研究探讨了如何在 GPU 上启用一致的 L1 数据缓存 [Ren and Lis, 2017, Singh et al., 2013] 以及明确定义 GPU 内存一致性模型的需求 [Alglave et al., 2015]。

4.1.2 L1 纹理缓存

最近，NVIDIA 的 GPU 架构将 L1 数据缓存和纹理缓存结合在一起以节省面积。为了更好地理解这种缓存的工作原理，首先需要对独立纹理缓存的设计有一定了解。这里讨论的细节旨在为开发高吞吐量处理器的微架构提供更多直观的认识。本文的大部分讨论基于 Igehy 等人（1998）的论文，该论文旨在弥补关于工业纹理缓存设计如何容忍缓存未命中时长时间的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。由于本书的重点不在图形领域，我们仅对激励纹理缓存包含的纹理操作进行了简要总结。

在3D图形中，使场景看起来尽可能逼真是很重要的。为了在实现实时渲染所需的高帧率的同时实现这种逼真效果，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，将一幅图像（称为纹理）应用到3D模型的表面上，以使该表面看起来更逼真。例如，可以使用纹理为场景中的桌子赋予天然木材的外观。

在纹理映射中，渲染管线首先确定纹理图像内一个或多个采样点的坐标。这些采样点被称为纹素（texels）。然后使用这些坐标找到包含这些纹素的内存位置的地址。由于屏幕上相邻的像素映射到相邻的纹素，并且通常会平均附近纹素的值，因此纹理内存访问中存在显著的局部性，这种局部性可以通过缓存加以利用 [Hakura and Gupta, 1997]。

图 4.2 展示了 Igehy 等人 [1998] 所描述的 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，标记阵列 2 和数据阵列 5 之间通过一个 FIFO 缓冲器 3 分隔开。设置此 FIFO 的目的是为了隐藏可能需要从 DRAM 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很频繁，并且工作集的大小相对较小。为了保持标记和数据阵列的尺寸较小，标记阵列实际上比数据阵列运行得更快。标记阵列的内容反映了数据阵列未来的状态，这个未来的时间大致等于一次未命中请求到内存并返回的往返时间。虽然在容量有限和未命中处理资源有限的情况下，相较于常规 CPU 设计提高了吞吐量，但缓存命中和未命中都经历了大致相同的延迟。

具体来说，如图 4.2 所示，纹理缓存的操作如下。加载/存储单元 1 将计算出的纹素地址发送到标签阵列 2 进行查找。如果访问命中，则指向数据阵列中数据位置的指针会与完成纹理操作所需的其他信息一起放入片段 FIFO 3 的尾部条目中。当条目到达片段 FIFO 的头部时，控制器 4 使用指针从数据阵列 5 查找纹素数据，并将其返回给纹理过滤单元 6 。尽管未详细显示，但对于如双线性和三线性过滤（mipmap 过滤）等操作，每个片段（即屏幕像素）实际上会进行四次或八次纹素查找。纹理过滤单元将纹素组合生成一个单一的颜色值，然后通过寄存器文件将其返回到指令流水线。

在标记查找期间发生缓存未命中时，标记数组通过未命中请求FIFO 8发送内存请求。未命中请求FIFO向内存系统的下一级发送请求9。通过使用内存访问调度技术 [Eckert, 2005, 2015]，可以提高GPU内存系统中DRAM带宽的利用率，这些技术可能会无序地服务内存请求以减少行切换惩罚。为了确保数据数组 5 的内容反映标记数组 2 的时间延迟状态，必须按顺序从内存系统返回数据。这是通过使用重新排序缓冲区10实现的。

4.1.3 统一纹理和数据缓存

在NVIDIA和AMD最近的GPU架构中，数据和纹理值的缓存是通过统一的L1缓存结构实现的。为了以最直接的方式实现这一点，仅缓存可以保证为只读的数据值。对于遵循这一限制的数据，纹理缓存硬件几乎无需修改即可使用，只需对寻址逻辑进行更改。这种设计在最近的一项专利中有所描述 [Heinrich et al.,

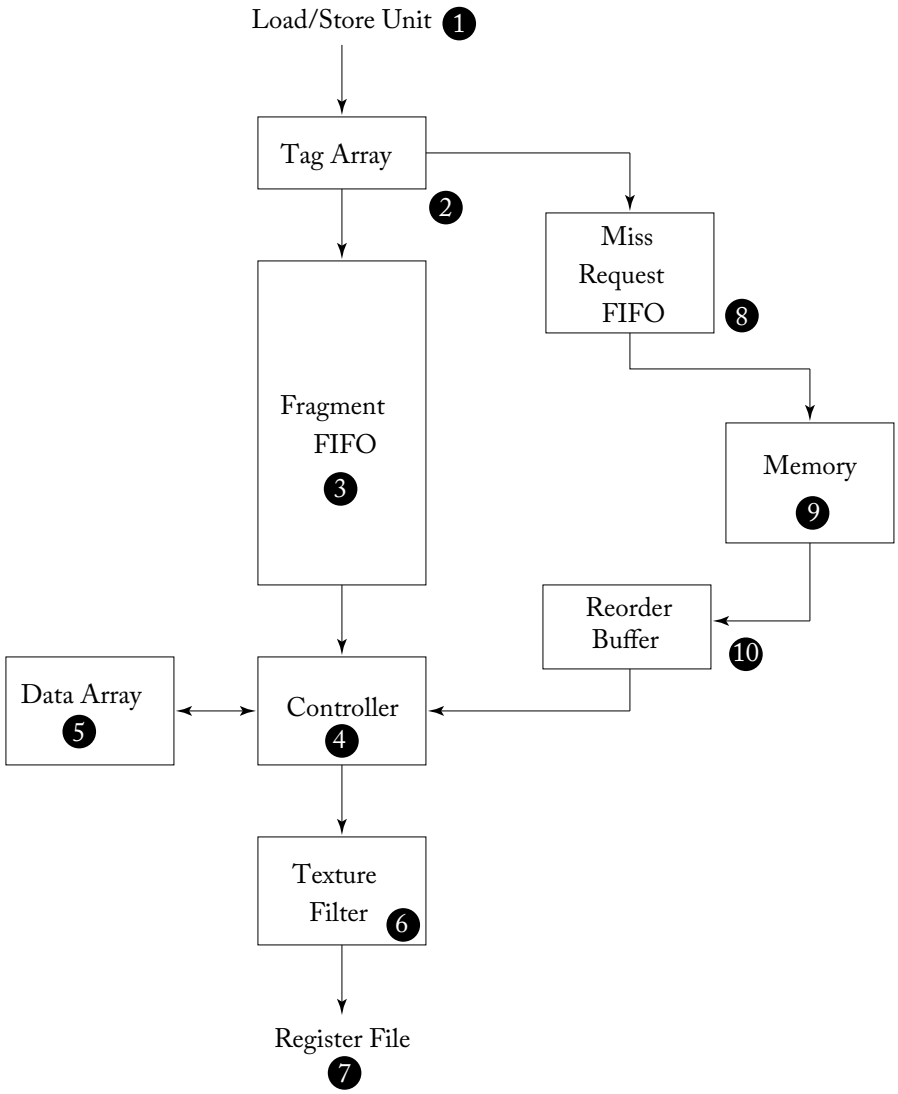


图 4.2：L1 纹理缓存（部分基于 [Igehy et al., 1998] 的图 2）。

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（第4.3节描述）并行连接多个DRAM芯片。内存流量通过地址交错分布在内存分区单元之间。NVIDIA的一项专利描述了用于在多达6个内存分区之间平衡流量的地址交错方案，其粒度为256字节或1,024字节 [Edmondson 和 Van Dyke, 2011]。

SIMT核心通过片上互连网络连接到内存分区单元。最近NVIDIA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，我们描述了与几项最近的 NVIDIA 专利相关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 的 Fermi GPU 架构发布大约一年前提交的。如图 4.3 所示，每个内存分区单元包含部分二级（L2）缓存，以及一个或多个内存访问调度器（也称为“ 帧缓冲器 ”或 FB）和光栅操作（ROP）单元。L2 缓存包含图形和计算数据。内存访问调度器会重新排序内存读写操作，以减少访问 DRAM 的开销。ROP 单元主要用于图形操作，例如 Alpha 混合，并支持图形表面的压缩。ROP 单元还支持 CUDA 编程模型中提供的原子操作。这三个单元紧密耦合，以下将对其进行详细描述。

4.3.1 L2 缓存

L2 缓存设计包括多种优化，以提高 GPU 每单位面积的总体吞吐量。每个内存分区内的 L2 缓存部分由两个切片组成 [Edmondson 等, 2013]。每个切片包含独立的标签数组和数据数组，并按顺序处理传入的请求 [Roberts 等, 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具有四个 32 字节的扇区。缓存行分配用于存储指令或加载指令。为优化在写缺失情况下完全覆盖每个扇区的合并写入的常见情况，不会首先从内存读取任何数据。这与标准计算机体系结构教科书中常见的 CPU 缓存描述方式有很大不同。我们检查的专利中未描述如何处理未合并写入（未完全覆盖一个扇区），但两种解决方案是存储字节级有效位和完全绕过 L2 缓存。为了减少内存访问调度器的面积，正在写入内存的数据会在 L2 的缓存行中缓冲，同时等待调度的写入操作。

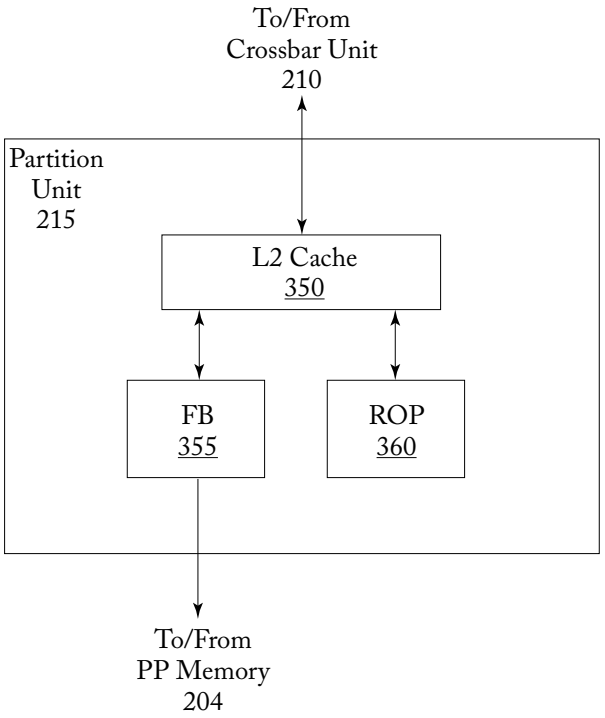


图 4.3：内存分区单元概述（基于 Edmondson 等 [2013] 的图 3B）。

4.3.2 原子操作

正如Glasco等人 [2012] 所描述的那样，ROP单元包括用于执行原子和归约操作的功能单元。访问相同内存位置的一系列原子操作可以被流水化，因为ROP单元包括一个本地ROP缓存。原子操作可用于实现运行在不同线程块中的线程之间的同步。

4.3.3 内存访问调度器

为了存储大量数据，GPU 使用诸如 GDDR5 gdd 等特殊的动态随机存取存储器（DRAM）。DRAM 将单个位存储在小电容中。例如，为了从这些电容中读取值，首先将称为页面的一行位读入一个称为行缓冲区的小型存储结构中。为了完成此操作，将连接各个存储电容和行缓冲区的位线（它们自身也具有电容）首先预充电至介于 0 和供电电压之间的中间电压。当存储单元在激活操作期间通过接入晶体管连接到位线时，由于电荷从存储单元流入或流出位线，位线的电压会略微上升或下降。

放大器随后放大这一微小变化，直到读取到一个干净的逻辑 0 或 1。将值读取到行缓冲区的过程中会刷新存储在电容中的值。预充电和激活操作引入了延迟，在此期间无法对 DRAM 阵列进行数据读写。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，在访问数据时切换行的延迟通常也无法完全隐藏。这导致了内存访问调度器的使用 [Rixner et al., 2000, Zuravleff 和 Robinson, 1997]，这些调度器重新排序 DRAM 内存访问请求，从而减少数据在行缓冲区和 DRAM 单元之间移动的次数。

为了启用对DRAM的访问，GPU中的每个内存分区可能包含多个内存访问调度器 [Keil and Edmondson, 2012]，用于将其包含的L2缓存部分连接到片外DRAM。实现这一目标的最简单方法是让L2缓存的每个切片拥有自己的内存访问调度器。每个内存访问调度器包含用于对L2缓存发送的读取请求和写入请求（“脏数据通知”）进行排序的独立逻辑 [Keil et al., 2012]。为了将对同一DRAM银行行的读取请求分组在一起，使用了两个独立的表。第一个表称为读取请求排序器，它是一个按内存地址访问的集合关联结构，将所有对给定银行同一行的读取请求映射到单个指针。该指针用于在第二个表中查找单个读取请求列表，该表称为读取请求存储。

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

袁等人 [2009] 探讨了针对运行 CUDA 编写的 GPU 计算应用程序的 GPU 内存访问调度器设计。他们观察到由单个流多处理器（SM）生成的请求具有行缓冲区局部性。如果某个内存分区的一系列内存请求在序列中接近出现并访问同一 DRAM 行中的同一 DRAM 银行，则称该序列具有行缓冲区局部性。然而，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 的请求混杂在一起，这些请求也发送到相同的内存分区。结果是，进入内存分区的请求序列的行缓冲区局部性降低。袁等人 [2009] 提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实现这一目标，这些策略优先处理包含来自同一 SM 的内存请求或具有相似行-银行地址的内存请求的数据包。

Bakhoda 等人 [2010, 2013] 探讨了用于 GPU 的片上互连网络的设计。这种互连将流处理器（streaming multiprocessors）与内存分区连接起来。他们认为，随着 SM 数量的增加，采用可扩展拓扑（如网格）将变得必要。他们研究了片上网络设计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，发现其具有多对少对多的模式。他们提出了一种更受限制的可扩展拓扑结构，由“ 半路由器 ”组成，通过利用这种流量模式减少路由器的面积成本。

4.4.2 缓存效果

Bakhoda 等人 [2009] 研究了在使用 GPGPU-Sim 模拟器模拟的支持 CUDA 的 GPU 上，为全局内存访问添加 L1 和/或 L2 缓存的影响，并表明虽然某些应用程序受益，但其他应用程序则没有。

后续由Jia等人[2012]开展的研究通过在NVIDIA Fermi GPU硬件上启用或禁用缓存，描述了缓存的有效性，并发现了类似的结果。观察到通过L1缓存将数据读取到共享内存（scratchpad shared memory）的应用程序并未从启用L1缓存中获益。即使排除这些应用程序，Jia等人[2012]仍观察到，仅缓存命中率不足以预测缓存是否会提升性能。他们发现，相反，需要考虑缓存对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缓存未进行分段，因此启用缓存可能在未命中时导致更大的128字节片外内存访问。对于内存带宽受限的应用程序，这种额外的片外内存流量可能导致性能下降。Jia等人[2012]引入了三种局部性的分类法：线程束内（within-warp）、线程块内（within-block）和跨指令（cross-instruction）。线程束内局部性发生在单个线程束内的不同线程从单次加载执行的内存读取访问相同的缓存块时。线程块内局部性发生在来自同一线程块的不同线程束中的线程从单次加载执行的内存读取访问相同的缓存块时。跨指令局部性发生在来自同一线程块的线程执行的不同加载指令的内存读取访问相同的缓存块时。Jia等人[2012]引入了一种基于此分类法的编译时算法，以帮助推断对单个加载指令启用缓存是否有利。

4.4.3 内存请求优先级与缓存旁路

根据上述特征研究 [Jia et al., 2012] 以及 Rogers 等人 [2012] 的工作（该工作表明 warp 调度可以提高缓存效率，详见第 5.1.2 节），Jia 等人 [2014] 提出了针对 GPU 的内存请求优先级和缓存绕过技术。相对于线程数量而言，关联度较低的缓存可能会遭受显著的冲突未命中 [Chen and Aamodt, 2009]。Jia 等人 [2014] 指出，许多用 CUDA 编写的 GPGPU 应用程序中包含的数组索引会导致单个 warp 的单个内存请求在使用标准模缓存组索引函数时 [Hennessy and Patterson, 2011, Patterson and Hennessy, 2013] 映射到同一缓存组，从而导致未命中。Jia 等人 [2014] 将此称为 warp 内部竞争。假设在缓存中分配空间时，

检测到未命中[1]，并且有限数量的未命中状态保持寄存器[2]的线程束内争用可能导致内存管道阻塞。[3] 为了解决线程束内争用问题，Jia 等人 [2014] 提出了在发生未命中且由于关联性阻塞无法分配缓存块时绕过 L1 数据缓存的方法。关联性阻塞发生在缓存集合中的所有块都被保留用于为未完成的缓存未命中提供数据存储空间的情况下。

贾等人 [2014] 还研究了他们称之为跨线程束争用的现象。这种形式的缓存争用发生在一个线程束将另一个线程束引入的数据驱逐时。为了解决这种争用问题，贾等人 [2014] 建议采用一种他们称之为 “ 内存请求优先化缓冲区 ”（MRPB）的结构。与 CCWS [Rogers et al., 2012] 类似，MRPB 通过修改对缓存的访问顺序以提高局部性，从而减少容量未命中。然而，不同于 CCWS 间接地通过线程调度实现这一点，MRPB 试图通过在线程调度后更改单个内存访问的顺序来提高局部性。

MRPB 在第一级数据缓存之前实现了内存请求重新排序。MRPB 的输入是指令发射流水线阶段在执行内存请求合并之后生成的内存请求。MRPB 的输出将内存请求传递到第一级缓存。在内部，MRPB 包含多个并行的先进先出 (FIFO) 队列。缓存请求通过一个 “ 签名 ” 分配到这些 FIFO 队列中。在评估的几种选项中，他们发现最有效的签名是使用 “ warp ID ”（一个在 0 到流式多处理器上可运行的最大 warp 数之间的数字）。MRPB 采用了一种 “ 排空策略 ” 来确定从哪个 FIFO 中选择内存请求以用于下一次访问缓存。在探索的几种选项中，最佳版本是一个简单的固定优先级方案，其中每个队列被分配一个静态优先级，包含请求的优先级最高的队列将首先被服务。

详细评估显示，使用MRPB结合绕过和重排序机制相较于64路16 KB实现了4%的几何平均加速。Jia等人[2014]还与CCWS进行了比较，显示了更大的改进。我们顺便提到，Rogers等人[2012]的评估采用了一个基线架构，该架构使用了更复杂的集合索引哈希函数[4]，以减少关联性停滞的影响。此外，Nugteren等人[2014]的后续工作试图逆向工程NVIDIA Fermi架构中实际使用的集合索引哈希函数的细节，并发现它使用了异或运算（这也有助于减少此类冲突）。

与 Rogers 等人 [2013] 类似，Jia 等人 [2014] 表明，他们以程序员透明的方式提高性能的方法可以缩小使用缓存的简单代码与使用片上共享内存的高度优化代码之间的差距。

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

Arunkumar 等人 [2016] 探讨了在静态指令中存在的内存分歧程度的基础上绕过和改变缓存行大小的效果。他们利用观察到的重用距离模式和内存分歧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循环的缓存绕过方法，该方法尝试在运行时逐条指令预测重用行为。缓存行的重用行为被监控。如果由特定程序计数器加载的缓存行未经历足够的重用，则绕过该指令的访问。

4.4.4 利用跨Warp异构性

Ausavarungnirun 等人 [2015] 提出了在 GPU 的共享 L2 缓存和内存控制器上的一系列改进，以缓解不规则 GPU 应用中的内存延迟分歧。这些技术统称为内存分歧校正（Memory Divergence Correction, MeDiC），利用了以下观察：同一内核中，不同 warp 的内存延迟分歧程度存在异质性。根据它们与共享 L2 缓存的交互方式，每个内核中的 warp 可以被分类为全部/大部分命中、全部/大部分未命中或平衡类型。作者证明，对于非全部命中的 warp，性能提升的空间很小，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。他们还证明，L2 缓存的排队延迟可能对性能产生非微不足道的影响，而通过对所有非全部命中的 warp 的所有请求（即使可能命中）绕过 L2 缓存，可以减轻这种影响。这种方法通过减少排队延迟，降低了全部命中 warp 的访问延迟。除了自适应绕过技术，他们还提出了对缓存替换策略和内存控制器调度器的修改，以尝试最小化检测到的全部命中 warp 的延迟。他们进一步证明，即使对于全部命中的 warp，由于 L2 缓存银行之间的排队延迟差异可能会导致额外且潜在可避免的排队延迟，因为 L2 缓存银行之间的排队延迟存在不平衡。

作者提出的微架构机制包括四个组件：（1）一个warp类型检测模块——将GPU中的warp分类为五种可能的类型之一：全未命中、主要未命中、均衡、主要命中或全命中；（2）一个基于warp类型的旁路逻辑模块，决定请求是否应绕过L2缓存；（3）一个基于warp类型的插入策略，确定在LRU堆栈中插入的位置；以及（4）一个基于warp类型的内存调度器，决定如何将L2未命中/旁路发送到DRAM。

检测机制通过按区间采样每个warp的命中率（总命中次数/访问次数）来运行。基于该命中率，warp会被划分为上述五种分类中的一种。确定这些分类边界的具体命中率会针对每个工作负载进行动态调整。在分类区间内，为了响应每个warp的L2特性中的阶段变化，任何请求都不会绕过缓存。

绕过机制位于 L2 缓存之前，接收带有生成它们的 warp 类型标记的内存请求。该机制试图消除来自全未命中 warp 的访问，并将主要未命中 warp 转换为全未命中 warp 。该模块简单地将所有标记为来自全未命中和主要未命中 warp 的请求直接发送到内存调度器。

MeDiC 的缓存管理策略通过更改从 DRAM 返回的请求在 L2 的 LRU 栈中的位置来运行。由主要未命中 warp 请求的缓存行被插入到 LRU 位置，而所有其他请求则插入到传统的 MRU 位置。

最后，MeDic 修改了基线内存请求调度器，使其包含两个内存访问队列：一个用于全命中和大部分命中的 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的 warp 的低优先级队列。内存调度器仅简单地优先处理高优先级队列中的所有请求，而不是低优先级队列中的任何请求。

4.4.5 协调缓存绕过

谢等人 [2015] 探讨了通过选择性启用缓存绕过来提高缓存命中率的潜力。他们使用性能分析来确定 GPGPU 应用中每个静态加载指令的局部性是良好、较差还是中等，并相应地标记这些指令。被标记为具有良好局部性的加载操作被允许使用 L1 数据缓存；被标记为局部性较差的加载操作则始终绕过缓存；而被标记为中等局部性的加载指令使用一种自适应机制，该机制的工作方式如下：自适应机制以线程块为粒度运行。对于给定的线程块，所有具有中等局部性的加载操作都会被统一处理——要么使用 L1 缓存，要么绕过缓存。这种行为是在线程块启动时确定的，依据的是一个在线调整的阈值。该阈值通过使用一个性能指标来动态调整，该指标综合考虑了 L1 缓存命中率和流水线资源冲突。他们的评估显示，与静态 warp 限制相比，这种方法显著提高了缓存命中率。

4.4.6 自适应缓存管理

Chen 等人 [2014b] 提出了协调缓存绕过和 warp 节流的方法，该方法利用 warp 节流和缓存绕过的优势来提高对缓存高度敏感的应用程序的性能。该机制在运行时检测缓存争用和内存资源争用，并相应地协调节流和绕过策略。该机制通过现有的 CPU 缓存绕过技术（保护距离）实现缓存绕过，该技术防止缓存行在一定数量的访问中被驱逐。缓存行插入缓存时，会被分配一个保护距离，计数器会跟踪缓存行剩余的保护距离。一旦剩余的保护距离达到 0，该行不再受保护，可以被驱逐。当一个新的内存请求尝试将新行插入到没有未保护行的集合时，该内存请求会绕过缓存。

保护距离是全局设置的，其最优值在不同的工作负载之间有所不同。在这项工作中，Chen 等人 [2014b] 对静态保护距离进行了全面分析，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

李等人 [2015] 观察到，warp节流可以优化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）显著未被充分利用。他们提出了一种通过为warp分配令牌来决定哪些warp可以将数据行分配到L1缓存中的机制。额外的“ 非污染warp ”未被分配令牌，因此尽管它们可以执行，但不被允许从L1缓存中驱逐数据。这导致了一个优化空间，其中可以调度的warp数量 (W) 和拥有令牌的warp数量 (T) 都可以设置为小于可以执行的warp最大数量。他们表明，静态选择W和T的最佳值可以在静态warp限制的CCWS基础上实现17%的性能提升。

基于这一观察，Li 等人 [2015] 探讨了两种机制来学习 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思想。在这种方法中，称为 dynPCALMTLP，一个采样周期运行内核，将 W 设置为最大数量的线程块，然后在不同的 SIMT 核心上变化 T。随后选择能够实现最大性能的 T 值。这种方法实现了与 CCWS 可比的性能，同时显著减少了面积开销。第二种方法称为 dynPCALCCWS，最初使用 CCWS 设置 W，然后使用 dynPCALMTLP 确定 T。然后，它监控共享结构的资源使用情况，动态增加或减少 W。这种方法相比 CCWS 带来了 11% 的性能提升。

4.4.8 虚拟内存页面放置

Agarwal 等人 [2015] 考虑了在异构系统中支持跨多种物理内存类型的缓存一致性所带来的影响，包括容量优化和带宽优化的内存。由于针对带宽优化的 DRAM 在成本和能耗上比针对容量优化的 DRAM 更昂贵，未来的系统可能会同时包含这两种类型。Agarwal 等人 [2015] 指出，目前操作系统的页面放置策略（例如 Linux 中的策略）并未考虑内存带宽的不均匀性。他们研究了一种未来系统，在这种系统中，GPU 可以以低延迟访问低带宽/高容量的 CPU 内存——延迟代价为 100 个核心周期。他们的实验使用了一个经过修改的 GPGPU-Sim 3.2.2 版本，该版本配置了额外的 MSHR 资源以模拟更新的 GPU。

通过这种设置，他们首先发现，对于受内存带宽限制的应用程序，通过同时使用 CPU 和 GPU 内存来增加总内存带宽，可以显著提高性能。他们发现，对于内存延迟限制较小的 GPGPU 应用程序，这种情况并不成立。在假设页面被均匀访问且带宽优化内存的容量不是限制条件的情况下，他们表明，分配

将页面按可用内存带宽比例分配到内存区域是最优的。在假设带宽受限内存的容量不是问题的情况下，他们发现一种简单的策略，即以与内存带宽成比例的概率随机将页面分配到带宽优化或容量优化的内存，在实际的GPGPU程序中效果良好。然而，当带宽优化内存的容量不足以满足应用需求时，他们发现有必要细化页面放置策略，以考虑访问频率。

为了优化页面放置，他们提出了一个系统，该系统通过使用经过修改的 NVIDIA 开发工具 nvcc 和 ptxas 实现的分析阶段，以及对现有 CUDA API 的扩展以包含页面放置提示来实现。使用基于分析的页面放置提示可以获得接近于 oracle 页面放置算法约 90% 的效益。他们将页面迁移策略留待未来研究。

4.4.9 数据放置

陈等人 [2014a] 提出了PORPLE，这是一种便携的数据放置策略，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。他们利用了这样一个观察结果：在GPU上可用的各种类型的内存中，选择将数据放置在何处对于程序员来说是困难的，并且通常无法在不同的GPU架构之间移植。PORPLE的目标是可扩展、输入自适应，并且普遍适用于规则和非规则的数据访问。他们的方法依赖于三个解决方案。

第一个解决方案是一种内存规范语言，有助于扩展性和可移植性。内存规范语言根据访问这些空间时的序列化条件，描述了GPU上的各种形式的内存。例如，对相邻全局数据的访问是合并的，因此是并发访问的，但对同一共享内存银行的访问必须进行序列化。

第二种解决方案是一个名为 PORPLE-C 的源到源编译器，它将原始 GPU 程序转换为与放置无关的版本。编译器在对内存的访问周围插入保护，选择与预测的最佳数据放置位置相对应的访问。

最后，为了预测哪种数据放置方式最优，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。该函数在 CPU 上运行一段短时间，并在启动内核之前帮助确定最佳的基于 GPU 的数据放置。在本研究的范围内，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。

轻量级模型用于在PORPLE中进行放置预测，其生成的估算基于内存的串行化条件来预测生成的事务数量。对于具有缓存层次结构的内存，它使用缓存的重用距离估算。

命中率。当多个数组共享一个缓存时，每个数组被分配到的缓存量的估算是基于缓存的线性划分，该划分依据数组的大小进行。

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 性能增长的减缓。他们提出通过在多芯片模块上由较小的 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、考虑局部性和首次访问页面分配的 CTA 调度，可以实现单个大型（且不可实现的）单片 GPU 性能的 90%。根据他们的分析，这比在相同工艺技术下使用最大可实现的单片 GPU 所能实现的性能高出 45%。

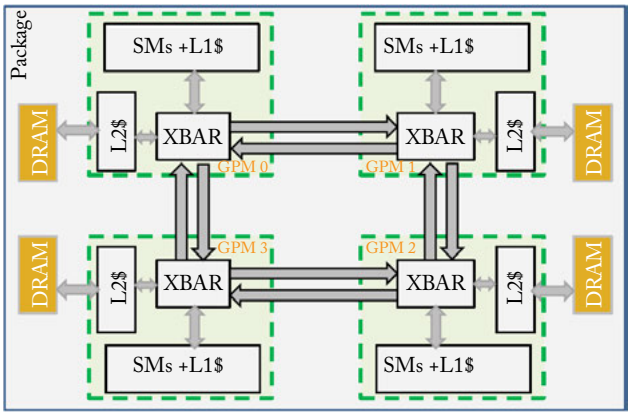


图4.4：一个多芯片模块GPU（基于Arunkumar等人[2017]的图3）。

CHAPTER 5

交叉研究GPU计算架构

本章详细介绍了GPGPU架构中几个未能很好地融入前几章的研究方向，这些章节主要集中在GPU架构的特定部分。第5.1节探讨了GPU中线程调度的相关研究。第5.2节研究了替代的编程方法论，第5.4节分析了异构CPU/GPU计算的相关工作。

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。无论程序如何指定（例如，使用OpenCL、CUDA、OpenACC等），没有广泛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上的线程主要通过三种方式组织和调度。

线程分配到Warp 由于GPU使用SIMD单元执行由MIMD编程模型定义的线程，这些线程必须融合在一起以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的线程被静态融合在一起以形成warp。第3.4.1节总结了关于warp内部替代线程排列的研究提案，以实现更好的warp压缩。

线程块到核心的动态分配 与 CPU 中线程可以逐个分配到硬件线程不同，在 GPU 中，工作以批量形式分配给 GPU 核心。该工作单元由以线程块形式存在的多个 warp 组成。在我们的基准 GPU 中，线程块以轮询方式分配给核心。核心的资源（如 warp 插槽、寄存器文件和共享内存空间）按线程块粒度进行订阅。由于与每个线程块相关的大量状态，现代 GPU 不会中断其执行。线程块中的线程运行至完成后，其资源才能分配给另一个线程块。

逐周期调度决策 在线程块被分配到 GPU 核心后，一组细粒度硬件调度器在每个周期决定选择哪一组 warps。

获取指令，其中包括发出指令以执行的步骤，以及何时为每个已发出的指令读取/写入操作数。

调度多个内核 线程块级别和逐周期调度决策可以在单个内核内部以及在同一 GPU 上并发运行的不同内核之间进行。传统的内核调度限制一次仅允许一个内核在 GPU 上活动。然而，NVIDIA 的 Streams 和 HyperQ 调度机制的引入使并发内核的运行成为可能。这种情况在某些方面类似于 CPU 上的多道程序设计。

5.1.1 线程块分配到核心的研究

当一个内核启动时，每个内核启动中的线程被分组为线程块。一个面向整个GPU的线程块调度机制根据资源可用性将每个线程块分配给一个SIMT核心。每个核心都有固定数量的临时存储器（在CUDA中称为共享内存，在OpenCL中称为本地内存）、寄存器数量、用于保存warp的槽位和用于保存线程块的槽位。在内核启动时，这些参数对于每个线程块都是已知的。最明显的线程块调度算法是以轮转方式将线程块分配给核心，以最大化参与核心的数量。线程块会被连续调度，直到每个核心中至少有一种资源耗尽。需要注意的是，一个内核可能由超过GPU能够同时运行的线程块组成。因此，一个内核中的一些线程块可能在其他线程块执行时根本没有在GPU上运行。一些研究技术研究了线程块调度领域中的权衡。

线程块级别的限流。Kayiran 等人 [2013] 提出通过限制分配给每个核心的线程块数量，以减少线程过度订阅引起的内存系统争用。他们开发了一种算法，用于监控核心空闲周期和内存延迟周期。该算法首先为每个核心分配其最大线程块数量的一半，然后监控空闲周期和内存延迟周期。如果某个核心主要在等待内存，则不会为其分配更多线程块，并且可能暂停现有线程块以阻止它们发出指令。该技术实现了一种粗粒度的并行限流机制，限制了内存系统的干扰并提高了整体应用性能，即使同时活动的 CTA 数量减少了。

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，能够动态监控资源争用情况，并调整线程数量、核心频率和内存频率，以改善能耗和性能。该系统基于四个参数做出决策：(1) SM 中活跃的 warp 数量；(2) 等待从内存获取数据的 warp 数量；(3) 准备执行算术指令的 warp 数量；(4) 准备执行内存指令的 warp 数量。基于这些参数，系统首先决定在一个 SM 上保持活跃的 warp 数量，然后根据该值和其他三个计数器的值（这些计数器作为内存的代理）进行后续决策。

争用、计算强度和内存强度），它决定如何最好地调整核心和内存系统的频率。

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约能源，从而将能耗降至最低，同时减轻其对性能的影响。在性能增强模式下，均衡器通过提升瓶颈资源以提高性能，同时以节能高效的方式进行。

他们通过研究与改变内存频率、计算频率以及并发运行线程数相关的性能和能量权衡，表征了来自 Rodinia 和 Parboil 的一组工作负载，将其分类为计算密集型、内存密集型、缓存敏感型或未饱和型。如果目标是最小化能耗（而不牺牲性能），那么计算密集型内核应以较低的内存频率运行，而内存内核应以较低的 SIMT 核心频率运行。这有助于减少系统中在基准速率下未被充分利用时不必要消耗的能量。

均衡器以间隔为基础对频率和并发性进行决策。该技术为每个SIMT核心添加了监控硬件，该硬件基于前面列出的四个计数器做出局部决策。它在每个SIMT核心局部决定当前周期的三个输出参数（CTA数量、内存频率和计算频率）应为多少。它将该SM应使用的CTA数量告知全局工作分配引擎，如果SIMT核心需要更多工作，则发布新的块。如果SM应运行较少的CTA，则会暂停核心上的某些CTA。在决定运行的CTA数量后，每个SIMT核心将内存/计算电压目标提交给全局频率管理器，后者根据多数功能设置全芯片频率。

本地决策是通过观察等待执行内存指令的warp数量以及等待执行ALU指令的warp数量来做出的。如果尝试等待内存的warp数量大于CTA中的warp数量，那么运行在该SIMT核心上的CTA数量将减少，从而可能提升对缓存敏感的工作负载的性能。如果准备发出内存（或ALU）指令的warp数量超过CTA中的warp数量，则该SIMT核心被认为是内存（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，但如果超过一半的活动warp正在等待并且等待内存的warp数量不超过两个，则该SIMT核心仍然可以被认为是ALU或内存受限。在这种情况下，核心上的活动CTA数量将增加一个，并根据等待计算的warp数量是否多于等待内存的warp数量来确定该SIMT核心是计算受限还是内存受限。

一旦SIMT核心做出了本地决策，内存和核心的频率将根据Equalizer运行的操作模式按±15%进行调整。

循环调度的早期特性Lakshminarayana 和 Kim [2010] 在早期不带硬件管理缓存的 GPU 环境中探索了许多 warp 调度策略，并显示，对于执行对称（平衡）动态指令计数的应用程序，基于公平性的 warp 和 DRAM 访问调度策略可以提高性能。这种策略在他们研究中使用的常规 GPU 工作负载上表现良好，因为不同 warp 之间的常规内存请求在核心内被合并，并且更好地利用了 DRAM 行缓冲区局部性。该论文还描述了其他几种 warp 调度策略，包括 ICOUNT，该策略最早由 Tullsen 等人 [1996] 为同时多线程 CPU 提出。ICOUNT 的设计目标是通过优先处理进展最快的 warp（或线程）来提高系统吞吐量。Lakshminarayana 和 Kim [2010] 表明，在他们的早期无缓存 GPU 上，针对早期的常规工作负载，仅优先处理少数几个 warp 通常无法提高性能。

两级调度。Gebhart 等人 [2011c] 引入了使用两级调度器以提高能效。他们的两级调度器将核心中的 warp 分为两个池：一个是用于下一周期调度的活动 warp 池，另一个是非活动 warp 池。当 warp 遇到编译器识别出的全局或纹理内存依赖时，会从活动池中移出，并以轮转方式从非活动池重新加入活动池。每个周期从较小的 warp 池中选择可以减少 warp 选择逻辑的规模和能耗。

Narasiman 等人 [2011] 提出的两级调度器通过允许线程组在不同时间到达相同的长延迟操作来提高性能。这有助于确保在提取组内维持缓存和行缓冲区的局部性。系统随后可以通过在提取组之间切换来隐藏长延迟操作。相比之下，缓存感知 Warp 调度（见下文）通过根据系统丧失的 Warp 内部局部性量自适应地限制系统能够维持的多线程数量来提高性能。

面向缓存的Warp调度。Rogers等人[2012]将GPU内核中的内存局部性分类为*intra-warp*（即warp加载随后引用其自身数据），或者*inter-warp*（即warp与其他warp共享数据）。他们证明，在缓存敏感型工作负载中，warp内部的局部性是最常见的局部性形式。基于这一观察，他们提出了一种面向缓存的波前调度（CCWS）机制，通过基于内存系统反馈控制活跃调度的warp数量，以利用这种局部性。

在更少的线程束之间主动调度，使得每个单独的线程束能够消耗更多的缓存空间，并减少 L1 数据缓存争用。特别是，当具有局部性的工作负载对缓存造成冲击时，会发生节流现象。为检测这种冲击，CCWS 引入了一种基于 L1 数据缓存替换受害者标签的局部性丢失检测机制。

图5.1 绘制了CCWS的高层微架构。在每次从缓存逐出的过程中，受害者的标签被写入私有线程（warp）受害者标签数组。每个线程都有其自己的受害者标签数组，因为CCWS仅关注检测线程内的局部性。在每次随后的缓存未命中时，会探查未命中线程的受害者标签数组。如果在受害者标签中找到该标签，则说明丢失了一些线程内的局部性。CCWS假设，如果该线程对L1数据缓存拥有更多的独占访问权，可能会命中该行，因此可能通过限流获得潜在的收益。

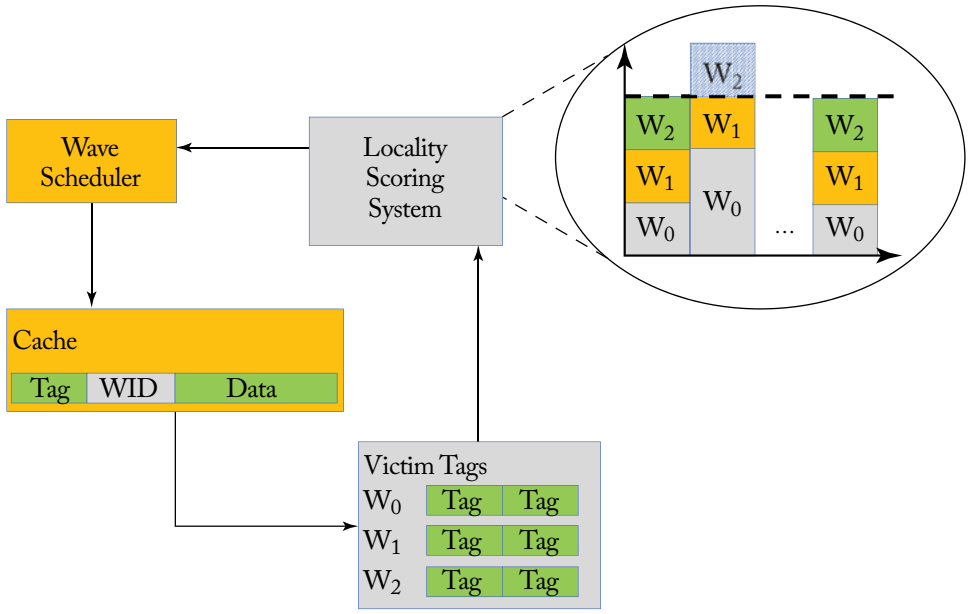


图5.1：面向缓存的波前调度微架构。

为了反映这种局部性的丧失，会向调度系统发送一个信号。问题调度器使用一个局部性评分系统来估算系统中每个warp丧失的局部性程度，这相当于估算每个warp需要多少额外的缓存容量。在局部性评分系统中，所有warp都会被分配一个初始评分，假设所有warp需要相同的缓存容量且没有发生限制（图5.1中的堆叠柱状图）。随着时间的推移，当检测到局部性丧失时，单个warp的评分会增加。在图5.1的示例中，warp 0经历了局部性的丧失，其评分有所增加。这一评分的增加使得warp 3超过了阈值，从而阻止其发出L1数据缓存请求，有效地限制了在核心上主动调度的warp数量。随着时间的推移，如果没有发生局部性丧失，则warp 0的评分会降低，直到warp 2能够低于阈值并再次能够发出内存请求。

CCWS 通过将各种调度机制与缓存替换策略进行比较，进一步展示了调度决策对缓存命中率的敏感性。论文表明，warp 调度器可用的决策空间远大于替换策略所能做出的相对受限的决策空间。论文进一步展示了，即使使用 Belady 最优缓存替换策略，CCWS 方案通过使用 LRU 替换策略仍能比之前的调度机制提高缓存命中率。

Rogers 等人 [2013] 提出了 Divergence-Aware Warp Scheduling (DAWS)，它通过对每个 warp 的缓存占用量进行更准确的估计扩展了 CCWS。DAWS 利用 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 创建了一个按 warp 划分的缓存占用估计值。基于每个 warp 预测的循环占用量，DAWS 会预先限制循环中 warp 的数量。DAWS 还根据每个 warp 体验到的控制流分歧程度来调整其缓存占用估计。已经退出循环的 warp 中的线程不再对占用估计作贡献。DAWS 进一步探索了 GPU 的可编程性方面，展示了通过一个更智能的 warp 调度器，一个没有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于同一基准的 GPU 优化版本的性能表现。

预取感知的Warp调度。Jog等人[2013b]在GPU上研究了一种预取感知的Warp调度器。他们基于两级调度机制构建调度器，但从非连续的Warp中形成了Fetch组。该策略增加了DRAM中的银行级并行性，因为预取器不会对同一个DRAM银行发出连续访问请求。他们进一步扩展了这一思想，根据Warp组分配来操控预取器。通过为其他组中的Warp预取数据，他们可以提高行缓冲区的局部性，并在预取请求和数据需求之间提供间隔。

CTA感知调度。Jog等人 [2013a] 提出了一个CTA感知的warp调度器，该调度器基于两级调度器，通过选择性组合CTA来形成fetch组。他们利用了多种基于CTA的特性来提高性能。他们采用了一种节流优先级技术，通过限制核心中活动warp的数量来实现，类似于其他节流调度器。结合节流技术，他们利用了不同核心中CTA之间的页面局部性。在仅基于局部性感知的CTA调度器下，连续的CTA通常会同时访问相同的DRAM bank，从而降低bank级并行性。他们将其与预取机制相结合，还提高了DRAM行的局部性。

调度对分支发散缓解技术的影响。Meng 等人 [2010] 提出了动态束划分 (Dynamic Warp Subdivision, DWS)，该技术在一些通道命中缓存而其他通道未命中时对束进行划分。此方案允许命中缓存的单独标量线程继续执行，即使它们的束内某些同伴未命中。DWS 通过允许命中线程继续运行来提高性能。

提前线程以更早地启动它们的缺失请求，并为落后的线程创建了一种预取效果。DWS 尝试通过提高数据加载到缓存中的速率来改善线程束内部的局部性。

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

冯和阿莫德特 [2011] 还提出了三种线程块优先机制，以补充其线程块压缩 (TBC) 技术。这些优先机制试图将同一CTA内的线程一起调度。他们的方法类似于Narasiman 等人 [2011] 提出的两级调度的相关工作，不同之处在于线程块是一起调度的，而不是取指组。

第3.4节包含了DWS、DWF和TBC的更详细总结。

调度和缓存重新执行。Sethia 等人 [2015] 提出了 Mascar，其旨在更好地在内存密集型工作负载中重叠计算与内存访问。Mascar 包括两个相互交织的机制。

- 一种内存感知的warp调度器（MAS），在核心中的MSHR和L1未命中队列条目超出容量时，优先执行单个warp。这种优先级策略即使在工作负载不包含数据局部性时也有助于提高性能，因为它可以让在顺序核心上执行的warp更快地到达其计算操作，从而实现优先warp的计算与其他warp的内存访问的重叠。

- 一种缓存访问重执行 (CAR) 机制，通过在缓存中的数据访问被阻塞时允许 L1 数据缓存未命中时的命中，从而避免 L1 数据缓存冲突。当具有缓存数据的 warp 因低局部性访问阻塞内存流水线而无法发出时，该机制可以起作用。

MAS有两种操作模式：等优先级模式（EP）和内存访问优先级模式（MAP）。系统根据L1 MSHR和内存未命中队列的填充情况在EP和MAP模式之间切换。一旦这些结构几乎满载，系统将切换到MAP模式。MAS包含两个队列，一个用于内存warp（尝试发出内存指令的warp），另一个用于计算warp（尝试发出其他类型指令的warp）。在每个队列内，warp按照“贪心-最早”顺序调度。对内存相关指令的跟踪通过扩展记分板来完成，记分板会指示基于加载操作何时填充输出寄存器。当观察到工作负载平衡且内存系统未过载时，调度器以EP模式运行。在EP模式下，调度机制优先处理内存warp。由于内存系统未过载，预测提前启动内存访问会提升性能。在MAP模式下，调度器优先处理计算warp，以更好地将可用计算与受限的内存系统重叠。在此模式下，仅允许一个内存warp，即“拥有者warp”，发出内存指令，直到其到达依赖于待处理内存请求的操作为止。

除了调度机制，Sethia 等人 [2015] 表明，内存密集型内核的峰值 IPC 比计算密集型内核的峰值 IPC 要低得多。他们说明，在内存密集型应用中，由于过多的内存访问导致的内存回压，大量的周期花费在 SIMT 核心的加载存储单元（LSU）停滞状态中。当 LSU 停滞时，有相当一部分时间是准备就绪的 warp 的数据已经在 L1 数据缓存中，但由于 LSU 被其他 warp 的内存请求阻塞，这些 warp 无法发出指令。缓存访问重新执行（CAR）机制试图通过在 LSU 管道旁边提供一个缓冲区来改善这种行为，该缓冲区存储停滞的内存指令并允许其他指令进入 LSU。只有在 LSU 未停滞且没有新的请求需要发出时，才会处理重新执行队列中的请求，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，直到队列中释放出空间。

当重新执行队列与内存感知调度器结合使用时，需要特别注意，因为重新执行队列中的请求可能来自于非优先级所有者线程束的线程束。在 MAP 模式下，重新执行队列中来自非所有者线程束的请求在发送到 L1 时，如果未命中 L1，将会进一步延迟。具体来说，当非所有者线程束的请求未命中 L1 时，请求不会被转发到 L2 缓存，而是重新插入到重新执行队列的尾部。

5.1.3 关于调度多个内核的研究

支持 GPU 上的抢占。Park 等人 [2015] 解决了在 GPU 上支持抢占式多任务处理的挑战。它采用了一种更宽松的幂等性定义，以使线程块内的计算刷新成为可能。更宽松的幂等性定义涉及从线程执行开始检测执行是否具有幂等性。他们的提案 Chimera 动态地在三种实现每个线程块上下文切换的方法中进行选择：

- 完整的上下文保存/存储；

- 等待线程块完成；

- 如果由于幂等性，可以安全地从头重新启动线程块，则直接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被停止，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。他们通过使用真实的 GPU 硬件演示了在线程自旋等待锁时会产生显著的开销。他们指出，简单地让未能获取锁的线程所属的 warp 后退执行，可能会阻碍或减慢同一 warp 中已持有锁的其他线程的进程。他们提出了一种硬件结构，用于动态识别涉及自旋锁的循环，这在使用基于堆栈的重新收敛机制时变得更具挑战性 [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以及一个单独的谓词寄存器更新历史记录，以准确检测自旋锁循环。为减少争用并提高性能，他们建议在 warp 执行自旋锁循环的后向分支时，降低被识别为执行自旋循环的 warp 的优先级，此时 warp 中持有锁的线程已经释放了这些锁。他们发现，与 Lee 和 Wu [2014] 相比，这种方法可以将性能提高 1.5× 倍，能耗降低 1.6× 倍。

5.

细粒度工作队列。Kim 和 Batten [2014] 提出了在 GPU 的每个 SIMT 核心中添加一个细粒度硬件工作列表的方案。他们利用了这样一个观察结果：不规则的 GPGPU 程序通常在使用数据驱动的方法并以软件实现时表现最佳，这种方法中工作是动态生成的，并在线程之间进行平衡，而不是采用拓扑方法，即启动固定数量的线程——其中许多线程并没有执行有用的工作。数据驱动方法有可能提高工作效率和负载均衡，但如果没有广泛的软件优化，其性能可能较差。本文提出了一种片上硬件工作列表，支持核心内以及核心之间的负载均衡。他们使用线程等待机制，并以间隔为基础重新平衡由线程生成的任务。他们在 lonestar GPU 基准测试套件中对不规则应用的各种实现（包括拓扑和数据驱动工作分配）评估了他们的硬件机制。

核心硬件工作列表解决了数据驱动软件工作列表的两个主要问题：（1）线程在推送生成的工作时导致内存系统争用；（2）由于基于线程 ID 的静态分区工作导致的负载平衡差。未依赖静态分区的软件实现在推送和拉取时都会遭遇内存争用。静态分区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，减少了争用。它通过在线程变为空闲之前动态重新分配生成的工作来改善负载平衡。作者为硬件队列的推送和拉取添加了特殊的 ISA 指令。核心中的每条通道都分配有一个小型的单端口 SRAM，用作存储给定通道使用和生成的工作 ID 的存储器。

本文提出了一种基于区间和需求驱动（仅在推/拉请求时重新分配）的工作重新分配方法，并深入评估了前者。基于区间的方法通过简单的阈值或更复杂的排序机制重新分配工作。阈值方法将工作量超过阈值的通道分类为贪婪型（工作量过多），而将工作量少于阈值的通道分类为匮乏型（工作量不足）。排序过程随后从贪婪通道重新分配工作到匮乏通道。基于排序的技术更为复杂，但由于所有匮乏通道也可以向其他匮乏通道捐赠工作，因此实现了更好的负载均衡。他们的技术还包括一种全局排序机制，可用于在核心之间分配工作。此外，该架构支持对硬件工作列表进行虚拟化，使其能够扩展到生成动态工作量超过硬件结构容量的工作负载。

基于嵌套并行模式的编程。Lee 等人 [2014a] 提出了嵌套并行模式在 GPU 上的局部性感知映射，利用了嵌套并行计算到 GPU 线程的映射没有普遍最优解的观察结果。具有嵌套并行性的算法（例如 map/reduce 操作）可以根据 GPU 程序的编写方式，在不同层次上将其并行性暴露给 GPU。作者利用了嵌套并行映射在 GPU 上的三种泛化：

- 一个一维映射，它并行化了顺序程序的外层循环；
- 一个线程块/线程映射，将顺序程序外层循环的每次迭代分配给一个线程块，并在线程块内对内部模式进行并行化；并且
- 基于warp的映射，将外层循环的每次迭代分配给一个warp，并在warp中并行化内部模式。

本文提出了一个自动编译框架，该框架基于局部性和嵌套模式中暴露的并行度生成预测性能分数，从而选择最适合一组常见嵌套并行模式的映射。这些模式包括诸如 map、reduce、foreach、filter 等集合操作。框架尝试将线程映射到集合中每个元素的操作上。该框架通过首先将应用程序中的每个嵌套级别分配到一个维度（例如 x, y, z 等）来处理模式的嵌套。一个双层嵌套模式（例如包含 reduce 的 map）有两个维度。然后，映射决定了 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分割的概念进一步控制内核中的并行度。在二维内核中（即两个嵌套级别的模式），如果每个维度被分配为 span(1)，那么在内核中启动的每个线程仅负责操作集合中的一个元素。此映射暴露了最大的并行度。相比之下，span(all) 表示每个线程操作集合中的所有元素。跨度可以是 (1) 和 (all) 之间的任何数值。Span(all) 被用于

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

由于 span(all) 会严重限制暴露的并行性并导致 GPU 未被充分利用，该框架还提供了 *split* 的概念。split(2) 表示每个线程在给定维度上操作一半的元素（可以将其视为 span(all)/2）。当使用 split 时，框架会启动第二个内核（称为组合器内核）来聚合跨分割的结果，从而生成与使用 span(all) 分区的内核相同的结果。

要选择每个维度中的块大小以及每个维度的分割/跨度，框架使用基于硬约束和软约束的评分算法。该算法遍历整个搜索空间的所有可能维度、块大小和跨度。搜索空间随着循环嵌套级别呈指数增长。然而，该指数的基数小于100，而典型的内核包含少于3个级别。因此，该空间可以在几秒钟内完全搜索。搜索会剪除违反硬约束的配置——例如导致执行错误的配置，如块中的最大线程数过高。它对软约束分配加权分数，例如确保将顺序内存访问的模式分配到x维以提高内存合并效率。

该框架还执行了两种常见的 GPU 优化：预先分配内存而不是在嵌套内核中动态分配全局内存，以及在确定将数据预取到共享内存对嵌套模式有利时利用共享内存。结果表明，自动生成的代码在性能上与专家调优的代码具有竞争力。

动态并行性。Wang 和 Yalamanchili [2014] 对 Kepler GPU 硬件上使用 CUDA 动态并行性的开销进行了特性分析，发现这些开销可能相当大。具体而言，他们确定了几个限制他们所研究工作负载效率的关键问题。首先，应用程序使用了非常大量的设备发起的内核。其次，每个内核通常只有 40 个线程（略多于一个 warp）。第三，尽管每个动态内核中执行的代码类似，但启动配置不同，导致内核配置信息需要大量存储开销。最后，为了实现并发，设备发起的内核被放置在不同的流中，以利用 Kepler 支持的 32 个并行硬件队列（Hyper-Q）。他们发现，这些因素的结合导致了非常低的利用率。

王等人 [2016a] 随后提出了动态线程块启动（DTBL），该方法修改了CUDA编程模型，使设备启动的内核能够共享硬件队列资源，从而实现更大的并行性和更好的GPU硬件利用率。他们提议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。这通过维护一个聚合线程块的链表来实现，该链表由修改后的硬件在启动内核时使用。他们

通过修改 GPGPU-Sim 评估 DTBL，发现 DTBL 相较于 CDP 性能提升了 1.4×，相比不使用 CDP 的高度优化 CUDA 版本性能提升了 1.2×。

王等人[2016b]随后探讨了动态启动的线程块被调度到哪个SM的影响。他们发现，通过鼓励子线程块调度到与父SM相同的SM，同时考虑SM之间的工作负载分布，与简单的轮转分配机制相比，他们能够将性能提高27%。

5.3 支持事务内存

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

这些提案的动机是TM编程模型具有潜力，可以缓解GPU应用中管理线程间不规则、细粒度通信的挑战，这些应用具有大量的不规则并行性。在现代GPU上，应用程序开发者可以通过屏障来粗化线程之间的同步，或者尝试使用许多现代GPU上可用的单字原子操作来实现这些通信的细粒度锁定。前一种方法可能涉及对底层算法的重大更改，而后一种方法则涉及细粒度锁定开发工作中的不确定性，这对以市场驱动的软件开发来说过于冒险（有一些例外）。在GPU上启用TM简化了同步，并提供了一种强大的编程模型，促进细粒度通信和并行工作负载的强扩展性。TM的这种承诺希望鼓励软件开发者探索这些不规则应用的GPU加速。

在 GPU 上支持 TM 的独特挑战。GPU 的高度多线程特性为 TM 系统设计引入了一系列新挑战。与在多核处理器上运行的 TM 研究主要关注具有相对较大访问范围的几十个并发事务不同，GPU 上的 TM 系统旨在扩展到成千上万个小型并发事务。这反映了 GPU 的高度多线程特性，成千上万个线程协同工作，每个线程执行一项小任务，共同实现一个目标。这些小型事务以字级粒度进行跟踪，从而比缓存块提供更精细的冲突检测分辨率。此外，GPU 中每个核心的私有缓存由数百个 GPU 线程共享。这大大降低了利用缓存一致性协议来检测冲突的优势，这一技术被大多数针对传统具有大规模 CPU 核心的 CMP 设计的硬件事务内存所采用。

5.3.1 KILO TM

Kilo TM [Fung 等, 2011] 是第一个针对 GPU 架构的已发表硬件 TM 提案。

Kilo TM采用基于价值的冲突检测方法 [Dalessandro et al., 2010, Olszewski et al., 2007]，以消除冲突检测对全局元数据的需求。每个事务

仅读取全局内存中的现有数据进行验证——以确定是否与其他已提交事务存在冲突。这种形式的验证利用了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，并在最细粒度上检测冲突。

然而，基于值的冲突检测的本地实现需要事务以串行方式提交。为了提高提交并行性，Kilo TM 结合了现有 TM 系统的思想 [Cha fi et al., 2007, Spear et al., 2008]，并扩展了这些思想，提出了创新的解决方案。特别是，Fung 等人 [2011] 引入了 *recency bloom filter*，一种新颖的数据结构，它利用时间和顺序的概念来压缩大量的小项集。Kilo TM 使用该结构来压缩所有提交事务的写集合。每个提交的事务查询近期布隆过滤器，以获取一个近似的冲突事务集合——该冲突集合中的某些事务是误报。Kilo TM 使用这一近似信息来调度数百个非冲突事务进行并行验证和提交。近期布隆过滤器的这种近似特性使其保持小型化，仅几个 kB，因此可以驻留在芯片上以实现快速访问。利用近期布隆过滤器提高事务提交并行性是 Kilo TM 的一个核心部分。

分支分歧与事务内存。事务内存编程模型引入了一种新的分支分歧类型。当一个warp完成事务时，其所有活跃线程都会尝试提交。一些线程可能会中止并需要重新执行其事务，而其他线程可能通过验证并提交其事务。由于这种结果可能在整个warp中并不一致，warp可能在验证后发生分歧。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分歧。该扩展与Kilo TM的其他设计方面无关，但它是支持GPU上的TM所必需的组成部分。

图 5.2 展示了如何扩展 SIMT 堆栈以处理由于事务中止导致的控制流分歧。当一个 warp 进入事务（在 B 行，`tx_begin`），它会在 SIMT 堆栈上压入两个特殊条目 1 。第一条目类型为 R，存储重新启动事务的信息。它的活动掩码初始为空，其 PC 字段指向 `tx_begin` 之后的指令。第二条目类型为 T，用于跟踪当前的事务尝试。在 `tx_commit`（行 F），任何未通过验证的线程会在 R 条目中设置其掩码位。当 warp 完成提交过程（即其活动线程已经提交或中止）时，T 条目会被弹出 2 。此时会使用 R 条目的活动掩码和 PC 压入一个新的 T 条目，以重新启动已中止的线程。然后，R 条目的活动掩码被清空 3 。如果 R 条目的活动掩码为空，则 T 和 R 条目都会被弹出，恢复原始的 N 条目 5 。其 PC 随后被修改为指向 `tx_commit` 之后的指令，warp 恢复正常执行。事务中的 warp 分支分歧处理方式与非事务性分歧相同 4 。

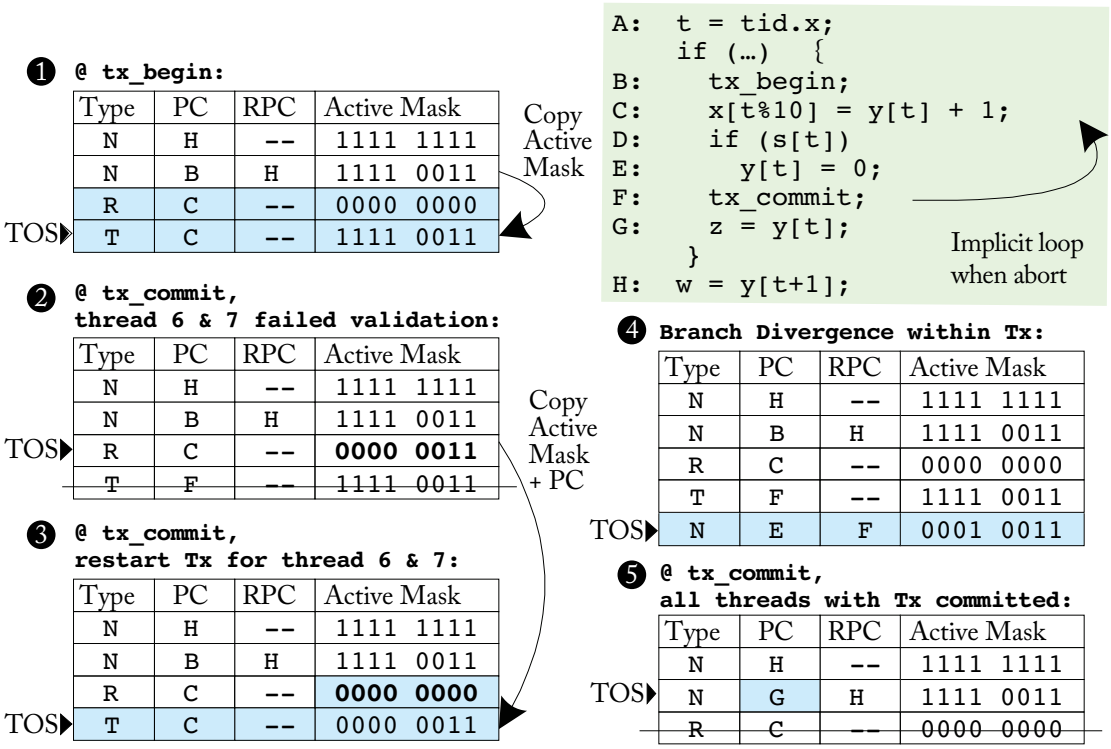


图 5.2：SIMT 堆栈扩展以处理由于事务中止（验证失败）导致的分歧。线程 6 和 7 未通过验证并被重新启动。堆栈条目类型：正常 (N)、事务重试 (R)、事务顶部 (T)。对于每种情况，新增的条目或修改的字段以阴影标示。

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] 提出了两项独特的改进，提升了 Kilo TM 的性能和能效：线程束级事务管理（WarpTM）和时间冲突检测（TCD）。

基于Warp的事务管理利用了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。具体来说，WarpTM摊薄了Kilo TM的控制开销，并提升了GPU内存子系统的利用率。如果能够高效地解决warp内的冲突，这些优化才有可能实现，因此，一个低开销的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] 提出了一种两阶段的并行warp内冲突解决方法，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能够合并标量内存访问...

将同一warp中多个事务的验证和提交的内存访问合并为更宽的访问。这种优化称为 *validation and commit coalescing*，是使Kilo TM能够充分利用优化用于向量宽访问的GPU宽内存系统的关键。

时间冲突检测是一种低开销机制，使用一组全局同步的片上计时器检测只读事务的冲突。一旦初始化，每个片上计时器将在其微架构模块中本地运行，并且不会与其他计时器通信。这种无需通信的隐式同步使得 TCD（时间冲突检测）区别于现有的软件事务内存系统中使用的基于时间戳的冲突检测方法 [Dalessandro et al., 2010, Spear et al., 2006, Xu et al., 2014]。TCD 使用从这些计时器捕获的时间戳来推断事务内存读取相对于其他事务更新的顺序。Kilo TM 集成了 TCD，用于检测无冲突的只读事务，这些事务可以直接提交，而无需基于值的冲突检测。通过这样做，它显著减少了这些事务的内存带宽开销，这些事务在使用事务进行数据结构遍历的 GPU-TM 应用中可能频繁发生。

5.4 异构系统

异构系统中的并发管理。Kayiran 等人 [2014] 提出了一个限制并发的方案，通过限制 GPU 的多线程以减少多程序化 CPU/GPU 系统中的内存和网络争用。在异构系统中，GPU 的干扰可能导致同时执行的 CPU 应用程序性能显著下降。他们提出的线程级并行性（TLP）限制方案通过观察共享 CPU/GPU 内存控制器和互联网络中的拥塞指标，来估计每个 GPU 核心上应主动调度的 GPU warp 数量。他们提出了两种方案，一种专注于仅提升 CPU 性能，另一种则通过平衡受限多线程对 GPU 性能的影响与 CPU 干扰，寻求优化整体系统吞吐量（包括 CPU 和 GPU）。作者评估了在 GPU 核心与 CPU 核心比为 2:1 的平铺异构架构中，warp 调度对性能的影响，并解释了这种比率是由于基于相同工艺技术的 NVIDIA GPU SM 的面积大约是现代无序执行 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，以最大化资源利用率。通过这一方案，作者观察到限制 GPU 的 TLP 对 GPU 性能可能产生正面或负面的影响，但绝不会对 CPU 性能产生负面影响。

为了提升CPU性能，作者提出了一种以CPU为中心的并发管理技术，该技术监控全局内存控制器中的停滞情况。该技术分别统计由于内存控制器输入队列已满而导致的内存请求停滞数量，以及由于从内存控制器（MC）到核心的回复网络已满而导致的内存请求停滞数量。这些指标在每个内存控制器本地进行监控，并且被聚合。

集中在一个中心化单元中，该单元将信息发送到 GPU 核心。启发式驱动方案为这些值设置了高阈值和低阈值。如果两个请求停滞计数的总和较低（基于阈值），则 GPU 上主动调度的 warp 数量会增加。如果两个计数的总和较高，则活跃 warp 的数量会减少，希望通过减少 GPU 内存流量来提高 CPU 性能。

一种更为平衡的技术尝试最大化整体系统吞吐量，通过增强以CPU为中心的方法来考虑由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（其工作中为1,024个周期）内无法发出指令的周期数。当前多线程限制级别下GPU停顿的移动平均值存储在每个GPU核心上，用于确定是否应增加或减少多线程级别。平衡技术以两个阶段调节GPU的线程级并行（TLP）。在第一阶段，其操作与以CPU为中心的解决方案相同，此时不考虑GPU停顿，GPU的TLP仅根据内存争用进行限制。在第二阶段（当GPU的TLP节流开始导致GPU性能下降，因为GPU对延迟的容忍度已降低时），如果系统预测继续节流GPU并发性会损害GPU性能，则停止节流。这一预测通过查阅目标多线程级别的GPU停顿移动平均值实现，该值来自该级别先前执行的记录。如果目标多线程级别与当前多线程级别的观察到的GPU停顿值之间的差异超过阈值k，则不会减少TLP级别。这个k值可以由用户设置，是指定GPU性能优先级的代理参数。

异构系统一致性。Power 等人 [2013a] 提出了一种硬件机制，用于高效支持集成系统中 CPU 和 GPU 之间的缓存一致性。他们发现，由 GPU 生成的内存流量增加导致目录带宽成为一个重要瓶颈。他们采用了粗粒度区域一致性 [Cantin 等人，2005] 来减少传统基于缓存块一致性目录导致的过多目录流量。一旦获得了粗粒度区域的权限，大多数请求将无需访问目录，并且一致性流量可以卸载到非一致性直接访问总线，而不是低带宽一致性互连网络。

异构 TLP 感知的 CPU-GPU 架构缓存管理。Lee 和 Kim [2012] 评估了在异构环境中，在 CPU 核心和 GPU 核心之间管理共享的最后一级缓存（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，因为线程级并行性可以隐藏内存延迟。为了确定 GPU 应用程序是否对缓存敏感，他们开发了一种按核心性能采样技术，其中一些核心绕过共享 LLC，而另一些核心插入到最近使用位置。基于这些核心的相对性能，他们可以为其余的 GPU 核心设置绕过策略：如果性能有所提升，则插入 LLC；如果性能不敏感，则绕过。

其次，他们观察到，以往以CPU为中心的缓存管理更倾向于访问频率较高的核心。研究表明，GPU核心在最后一级缓存（LLC）上的流量是CPU核心的五到十倍。这导致缓存容量偏向于GPU，从而降低了CPU应用程序的性能。他们提议扩展基于效用的缓存分区方法 [Qureshi and Patt, 2006]，以考虑LLC访问的相对比例。当GPU核心对缓存敏感时，为了弥补CPU和GPU在访问规模和延迟敏感性方面的差异，CPU核心的缓存分配会超出基于效用的缓存分区所提供的范围。

参考文献

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

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

Top500.org 2

托尔·M·阿莫德 (Tor M. Aamodt)、威尔逊·W·L·冯 (Wilson W. L. Fung)、英德普里特·辛格 (Inderpreet Singh)、艾哈迈德·埃尔·沙菲 (Ahmed El-Shafiey)、吉米·夸 (Jimmy Kwa)、泰勒·赫瑟林顿 (Tayler Hetherington)、阿尤布·古布兰 (Ayub Gubran)、安德鲁·博克托 (Andrew Boktor)、蒂姆·罗杰斯 (Tim Rogers)、阿里·巴霍达 (Ali Bakhoda) 和哈迪·乔伊巴尔 (Hadi Jooybar)。*GPGPU-Sim 3.x Manual*, 16, 38

M. Abdel-Majeed 和 M. Annavaram. Warped register file：一种适用于 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. Pilot 寄存器文件：用于 GPU 的节能分区寄存器文件。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2017 年 2 月。DOI: 10.1109/h-pca.2017.47. 65

多米尼克·阿科切拉和马克·R·古迪。美国专利#7,750,915：在共享内存资源的多个存储库中并发访问数据元素（受让人：NVIDIA公司），2010年7月。68

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

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

约翰·R·艾伦、肯·肯尼迪、卡莉·波特菲尔德和乔·沃伦。控制依赖向数据依赖的转换。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》。发表于 *Proc. of the ACM International Conference on Supercomputing (ICS)*，第 1 – 6 页，1990 年。DOI: 10.1145/77726.255132. 16

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

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

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

阿基尔·阿伦库马尔、叶夫根尼·博洛廷、本杰明·乔、乌格利耶萨·米利奇、埃曼·埃布拉希米、奥雷斯特·维拉、阿梅尔·贾利尔、卡罗尔-让·吴和大卫·内兰斯。MCM-GPU：用于持续性能扩展的多芯片模块GPU。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第320 – 332页，2017年。DOI：10.1145/3079856.3080231. 84

克里斯特·阿萨诺维奇、斯蒂芬·W凯克勒、尹石燮、罗尼·克拉辛斯基和维诺德·格罗弗。《数据并行架构的收敛与标量化》。发表于 *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*，2013。DOI: 10.1109/cgo.2013.6494995。54, 56, 58

拉查塔·奥萨瓦龙尼伦、索加塔·戈斯、奥努尔·凯兰、加布里埃尔·H·洛、奇塔·R·达斯、马哈穆特·T·坎德米尔和奥努尔·穆图鲁。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)* 中利用线程束间的异构性提高 GPGPU 性能，2015 年。DOI: 10.1109/pact.2015.38. 80

阿里·巴科达、麦Owens J. D.、冯Wilson W. L.，黄亨利，以及阿莫德Tor M.。使用详细的GPU模拟器分析CUDA工作负载。在 *Proc. of the IEEE Symposium of Performance and Analysis of Systems and Software, (ISPASS'09)*，第163-174页，2009年。DOI: 10.1109/ispass.2009.4919648。6, 14, 78

阿里·巴科达、约翰·金和托尔·M·阿莫特。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)* 中针对多核加速器的高吞吐量片上网络，第 421 – 432 页，2010 年。DOI: 10.1109/micro.2010.50. 77

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

马库斯·比列特、奥拉·奥尔松和乌尔夫·阿萨松。《宽SIMD多核架构上的高效流压缩》。发表于 *Proc. of the ACM Conference on High Performance Graphics*，第159 – 166页，2009年。DOI: 10.1145/1572769.1572795。

Nicolas Brunie，Sylvain Collange 和 Gregory Diamos。分支与 warp 同时交织以维持 GPU 性能。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 49 – 60 页，2012 年。DOI: 10.1109/isca.2012.6237005。44, 53

伊恩·巴克、蒂姆·弗利、丹尼尔·霍恩、杰里米·苏格曼、凯文·法塔哈利安、迈克·休斯顿和帕特·汉拉汉。《Brook for GPUs：基于图形硬件的流计算》。发表于 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，第777-786页，2004年。DOI: 10.1145/1186562.1015800。

布莱恩·卡布拉尔。“ SASS ”是什么的缩写？ https://stackoverflow.com/questions/9798258/what-is-sass-short-for, 2016年11月。 14

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

埃德温·卡特穆尔。《用于计算机显示曲面细分的算法》。*Technical Report*，DTIC 文档，1974。72

Hassan Chafi，Jared Casper，Brian D. Carlstrom，Austen Mcdonald，Chi Cao，Minh Woongki Baek，Christos Kozyrakis 和 Kunle Olukotun。《一种可扩展的非阻塞事务内存方法》。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 97 – 108 页，2007 年。DOI: 10.1109/hpca.2007.346189。97

陈国阳，吴波，李东，沈熹鹏。PORPLE：一个用于 GPU 上可移植数据放置的可扩展优化器。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014a。DOI: 10.1109/micro.2014.20。83

陈西峨和Tor M. Aamodt. 一阶细粒度多线程吞吐量模型. 见 *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

陈旭豪、李文昌、Christopher I. Rodrigues、吕杰、王志英和胡文美。面向能效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。在 GPGPU 计算中动态检测均匀和仿射向量。发表于 *Proc. of the European Conference on Parallel Processing (Euro-Par)*，2010 年。DOI: 10.1007/978-3-642-14122-5_8。59

布雷特·W·库恩 (Brett W. Coon) 和约翰·埃里克·林德霍尔姆 (John Erik Lindholm)。美国专利 #7,353,369：《用于管理SIMD架构中分歧线程的系统和方法》（受让人：NVIDIA公司），2008年4月。26, 49, 50

Brett W. Coon，Peter C. Mills，Stuart F. Oberman 和 Ming Y. Siu。美国专利 #7,434,032：使用具有独立内存区域的计分板跟踪多线程处理中的寄存器使用情况并存储顺序寄存器大小指示器（受让方 NVIDIA Corp.），2008年10月。34

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

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

卢克·达莱桑德罗、迈克尔·F斯皮尔和迈克尔·L斯科特。《NOrec：通过取消所有权记录简化STM》。发表于 *Proc. of the ACM Symposium on Principles and Practices of Parallel Programming (PPoPP)*，第67 – 78页，2010年。DOI: 10.1145/1693453.1693464。96, 99

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

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

格雷戈里·弗雷德里克·迪亚莫斯，理查德·克雷格·约翰逊，维诺德·格罗弗，奥利维尔·吉鲁，杰克·H·乔奎特，迈克尔·阿兰·费特曼，阿杰伊·S·提鲁马拉，彼得·尼尔森，以及罗尼·迈尔·克拉辛斯基。使用收敛屏障执行分岔线程，2015年7月13日。27, 28, 29, 30

罗杰·埃克特。美国专利#7,376,803：用于DRAM系统的页面流分类器（受让人：NVIDIA公司），2008年5月。73

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

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

约翰·H·埃德蒙森等，美国专利 #8,464,001：带有帧缓冲区管理的脏数据提取和高优先级清理机制的缓存及相关方法，2013年6月，第75页，第76页

Ahmed ElTantaway，Jessica Wenjie Ma，Mike O’Connor，Tor M. Aamodt. 一种用于高效GPU控制流的可扩展多路径微架构。发表于 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，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

亚历山大·L明肯等人，美国专利 #7,649,538：具有高级过滤功能的可重构高性能纹理管线（受让人：NVIDIA公司），2010年1月。72

冯威尔逊（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)*, 第 25 – 36 页, 2011. DOI: 10.1109/hpca.2011.5749714. 26, 28, 42, 43, 50, 91

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

Wilson W. L. Fung，Ivan Sham，George Yuan，和 Tor M. Aamodt。动态线程束形成与调度用于高效 GPU 控制流。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 407 – 420 页，2007 年。DOI: 10.1109/micro.2007.4408272。14, 23, 25, 42, 44, 49, 91

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

冯威尔逊等人. 动态warp形成：在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. 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。发表于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2011 年 12 月。DOI: 10.1145/2155620.2155675. 63

马克·盖布哈特、丹尼尔·R约翰逊、大卫·塔尔让、斯蒂芬·W凯克勒、威廉·J达利、埃里克·林德霍尔姆和凯文·斯卡德隆。能效机制用于吞吐量处理器中的线程上下文管理。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，2011b。DOI: 10.1145/2000064.2000093。 马克·盖布哈特、丹尼尔·R约翰逊、大卫·塔尔让、斯蒂芬·W凯克勒、威廉·J达利、埃里克·林德霍尔姆和凯文·斯卡德隆。能效机制用于吞吐量处理器中的线程上下文管理。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 235 – 246 页，2011c。DOI: 10.1145/2000064.2000093。 伊萨克·格拉多、约翰·E斯通、贾维尔·卡贝萨斯、桑杰·帕特尔、纳乔·纳瓦罗和黄文美。用于异构并行系统的非对称分布式共享内存模型。发表于 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*，第 347 – 358 页，2010。DOI: 10.1145/1736020.1736059。 赛义德·佐海布·吉拉尼、金南成和迈克尔·J舒尔特。面向计算密集型 GPGPU 应用的能效计算。发表于 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2013。DOI: 10.1109/hpca.2013.6522330。 大卫·B格拉斯科等。美国专利#8,135,926：与外部算术逻辑单元块结合的基于缓存的原子操作控制，2012 年 3 月。 大卫·B格拉斯科等。美国专利#8,539,130：有效的数据包传输的虚拟通道，2013 年 9 月。 斯科特·格雷。适用于 NVIDIA Maxwell 架构的汇编器。`https://github.com/NervanaSyst ems/maxas` 16, 40 兹维卡·古兹、叶夫根尼·博洛廷、伊迪特·凯达尔、阿维诺阿姆·科洛德尼、阿维·门德尔森和乌里·C魏泽尔。多核与多线程机器：远离“谷底”。*IEEE Computer Architecture Letters*，8(1):25 – 28，2009。DOI: 10.1109/l-ca.2009.4。 齐亚德·S哈库拉和阿努普·古普塔。纹理映射缓存架构的设计与分析。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 108 – 120 页，1997。DOI: 10.1145/264107.264152。 雷汉·哈米德、瓦加哈特·卡迪尔、梅根·瓦克斯、奥米德·阿齐兹、亚历克斯·索洛马特尼科夫、本杰明·C李、斯蒂芬·理查森、克里斯托斯·科齐拉基斯和马克·霍洛维茨。理解通用芯片中效率低下的根源。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 37 – 47 页，2010。DOI: 10.1145/1815961.1815968。 韩松、刘星宇、毛惠子、蒲靖、阿尔达文·佩德拉姆、马克·A霍洛维茨和威廉·J达利。EIE：基于压缩深度神经网络的高效推理引擎。发表于

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 11

蒂姆·哈里斯、詹姆斯·拉鲁斯和拉维·拉杰瓦尔。《*Transactional Memory*》，第2版。摩根与克莱普尔，2010年。DOI: 10.1201/b11417-16。96

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

约翰·亨尼斯（John Hennessy）和大卫·帕特森（David Patterson）。*Computer Architecture—A Quantitative Approach*，第5版。摩根考夫曼出版社（Morgan Kaufmann），2011年。1, 10, 71, 72, 78

Maurice Herlihy 和 J. Eliot B. Moss。《事务内存：对无锁数据结构的架构支持》。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architec- ture (ISCA)*，第 289－300 页，1993 年。DOI: 10.1109/isca.1993.698569。96

贾里德·霍伯罗克、维克多·卢、贾云涛和约翰·C·哈特。《延迟着色的流压缩》。发表于 *Proc. of the ACM Conference on High Performance Graphics*，第173-180页，2009年。DOI：10.1145/1572769.1572797。45

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

马克·霍洛维茨（Mark Horowitz），埃拉德·阿隆（Elad Alon），迪内什·帕蒂尔（Dinesh Patil），塞缪尔·纳夫齐格尔（Samuel Naffziger），拉杰什·库马尔（Rajesh Kumar），以及凯瑞·伯恩斯坦（Kerry Bernstein）。《CMOS的缩放、功耗及未来》。发表于 *IEEE International Electron Devices Meeting*，2005年。DOI: 10.1109/iedm.2005.1609253

霍曼·伊格希（Homan Igehy）、马修·埃尔德里奇（Matthew Eldridge）和凯科阿·普劳德富特（Kekoa Proudfoot）。在纹理缓存架构中进行预取。见 *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics hardware*，第133页及后续，1998年。DOI：10.1145/285305.285321。72, 73, 74

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

贾伟, K. A. Shaw 和 M. Martonosi. MRPB：用于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 Com- puter Architecture (HPCA)*, 2014. DOI: 10.1109/hpca.2014.6835938. 78, 79

贾文浩、凯利·A·肖、玛格丽特·马托诺西。在 *Proc. of the ACM International Conference on Supercom- puting (ICS)* 中，表征和改进 GPU 中需求获取缓存的使用，第 15-24 页，2012 年。DOI: 10.1145/2304576.2304582. 78

阿德维特·乔格（Adwait Jog），奥努尔·凯伊兰（Onur Kayiran），纳奇阿帕·奇丹巴拉姆·纳奇阿帕（Nachiappan Chidambaram Nachiappan），阿西特·K·米什拉（Asit K. Mishra），马哈穆特·T·坎德米尔（Mahmut T. Kandemir），奥努尔·穆特鲁（Onur Mutlu），拉维尚卡尔·艾耶尔（Ravishankar Iyer），奇塔·R·达斯（Chita R. Das）。OWL：面向提高GPGPU性能的协作线程阵列感知调度技术。在 *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015a。DOI: 10.1145/2451116.2451158。90

阿德瓦伊特·乔格（Adwait Jog）、奥努尔·卡伊兰（Onur Kayiran）、阿西特·K·米什拉（Asit K. Mishra）、马赫穆特·T·坎德米尔（Mahmut T. Kandemir）、奥努尔·穆图鲁（Onur Mutlu）、拉维尚卡尔·艾耶尔（Ravishankar Iyer）和奇塔·R·达斯（Chita R. Das）。面向GPGPU的协调调度与预取。在 *Proc. of the ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015b。DOI: 10.1145/2749469.2750418。90

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers 等人。在数据中心中对张量处理单元的性能分析。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017。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。10

乌贾尔·J·卡帕西等人。数据并行架构的高效条件操作。发表于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 159 – 170 页，2000 年。DOI: 10.1109/micro.2000.898067。45

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

Onur Kayiran、Adwait Jog、Mahmut T. Kandemir 和 Chita R. Das。《不多也不少：为 GPGPU 优化线程级并行性》。发表于 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2013。86

S. W. Keckler，W. J. Dally，B. Khailany，M. Garland 和 D. Glasco。GPU 与并行计算的未来。*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 #8,195,858：管理共享 L2 总线上的冲突，2012 年 6 月。77

Shane Keil 等，美国专利 #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，第77页

法尔扎德·霍拉萨尼（Farzad Khorasani）、拉吉夫·古普塔（Rajiv Gupta）和拉克斯米·N·布亚扬（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 warp 执行。发表于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2015。DOI: 10.1145/2830772.2830796。45

金俊勇和C. Batten. 使用细粒度硬件工作列表在GPGPU上加速不规则算法。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MI- CRO)*，2014。DOI : 10.1109/micro.2014.24。

金基姆、克里斯托弗·托恩、什里沙·斯里纳特、德里克·洛克哈特和克里斯托弗·巴滕。微架构机制在SIMT架构中利用值结构。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2013。DOI: 10.1145/2508148.2485934。57, 58, 59, 60, 61, 62

金尚满、许成九、张新雅、胡艺歌、阿米尔·沃特、埃米特·威彻尔和马克·西尔伯斯坦。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

David B. Kirk 和 Hwu Wen-Mei. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016. DOI: 10.1016/c2011-0-04129-7. 9

约翰·克鲁斯特曼 (John Kloosterman)、乔纳森·博蒙特 (Jonathan Beaumont)、D. Anoushe Jamshidi、乔纳森·贝利 (Jonathan Bailey)、特雷弗·穆奇 (Trevor Mudge) 和斯科特·马尔克 (Scott Mahlke)。Regless: GPU 的即时操作数分阶段处理。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，第 151 – 164 页，2017 年。DOI: 10.1145/3123939.3124554。65

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

罗尼·M·克拉申斯基。美国专利申请 #20130042090 A1：时间SIMT执行优化，2011年8月。53, 58

David Kroft。《无锁存指令提取/预取缓存组织》。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 81 – 87 页，1981 年。DOI: 10.1145/285930.285979。33, 71

延斯·克吕格 (Jens Krüger) 和鲁迪格·韦斯特曼 (Rüdiger Westermann)。用于数值算法 GPU 实现的线性代数算子。发表于 *ACM Transactions on Graphics (TOG)*, V. 22, 第 908 – 916 页, 2003 年。DOI: 10.1145/882262.882363. 6

赖俊杰和安德烈·塞兹内克。对Fermi和Kepler GPU上SGEMM性能上限的分析与优化。发表于*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

纳格什·B·拉克希米纳拉亚纳（Nagesh B. Lakshminarayana）和金慧顺（Hyesoon Kim）。指令获取和内存调度对GPU性能的影响。发表于 *Workshop on Language, Compiler, and Architecture Support for GPGPU*, 2010. 88

艾哈迈德·拉什加尔、埃巴德·萨利希和阿米拉里·巴尼亚萨迪。关于逆向工程GPGPU的一个案例研究：突出的内存处理资源。*ACM SIGARCH Computer Architecture News*，43(4):15 – 21，2016。DOI: 10.1145/2927964.2927968。C. L. Lawson、R. J. Hanson、D. R. Kincaid 和 F. T. Krogh。用于Fortran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。

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

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

李允燮，Rimas Avizienis，Alex Bishara，Richard Xia，Derek Lockhart，Christopher Batten，Krste Asanović。在数据并行加速器中探索可编程性与效率之间的权衡。见 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第129 – 140页，2011年。DOI：10.1145/2000064.2000080。

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

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。GPUWattch：实现 GPGPU 的能量优化。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，页面 487 – 498，2013。DOI：10.1145/2508148.2485964。6 Adam Levinthal 和 Thomas Porter。Chap—一种 SIMD 图形处理器。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，页面 77 – 82，1984。DOI：10.1145/800031.808581。50 Dong Li、Minsoo Rhu、Daniel R. Johnson、Mike O'Connor、Mattan Erez、Doug Burger、Donald S. Fussell 和 Stephen W. Redder。基于优先级的缓存分配在吞吐处理器中的应用。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2015。DOI：10.1109/hpca.2015.7056024。82 E. Lindholm、J. Nickolls、S. Oberman 和 J. Montrym。NVIDIA Tesla：一个统一的图形和计算架构。*Micro, IEEE*，28(2):39 – 55，2008 年 3 – 4 月。DOI：10.1109/mm.2008.31。9 Erik Lindholm、Mark J. Kilgard 和 Henry Moreton。一个用户可编程的顶点引擎。在 *Proc. of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*，页面 149 – 158，2001。DOI：10.1145/383259.383274。6, 21 John Erik Lindholm、Ming Y. Siu、Simon S. Moy、Samuel Liu 和 John R. Nickolls。美国专利 #7,339,592：使用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。美国专利 #9,189,242：基于信用的流多处理器 Warp 调度（受让人 NVIDIA Corp.），2015 年 11 月。33, 41 John S. Liptay。System/360 型号 85 的结构特性，II：缓存。*IBM Systems Journal*，7(1):15 – 21，1968。DOI：10.1147/sj.71.0015。70 Z. Liu、S. Gilani、M. Annavaram 和 N. S. Kim。G-Scalar：针对功耗高效 GPU 的具有成本效益的通用标量执行架构。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，2017。DOI：10.1109/hpca.2017.51。60, 61, 62 Samuel Lui、John Erik Lindholm、Ming Y. Siu、Brett W. Coon 和 Stuart F. Oberman。美国专利申请 11/555,649

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

孟佳源、David Tarjan 和 Kevin Skadron。动态 warp 子划分用于集成分支和内存分歧容忍。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第 235 – 246 页，2010 年。DOI: 10.1145/1815961.1815992。30, 48, 90

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

亚历山大·L·明金等，《美国专利#8,266,383：使用延迟/重放机制处理缓存未命中》（受让人：NVIDIA公司），2012年9月，第68、69、71页

亚历山大·L·明金等人，美国专利 #8,595,425：《用于多个客户端的可配置缓存》（受让方：NVIDIA公司），2013年11月。68

Michael Mishkin、Nam Sung Kim 和 Mikko Lipasti。GPGPUSIM 中的读后写冲突预防。在 *Workshop on Deplicating, Deconstructing, and Debunking (WDDD)*，2016 年 6 月。39, 40

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

维努·纳拉西曼（Veynu Narasiman）、迈克尔·谢巴诺夫（Michael Shebanow）、李昌柱（Chang Joo Lee）、鲁斯塔姆·米夫塔胡丁诺夫（Rustam Miftakhutdinov）、奥努尔·穆图鲁（Onur Mutlu）和耶鲁·N.帕特（Yale N. Patt）。通过大规模线程块和两级线程块调度提升GPU性能。发表于
Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)，第 308 – 31 7，2011 年。DOI: 10.1145/2155620.2155656。33, 38, 43, 88, 91

约翰·R·尼克尔斯（John R. Nickolls）和约亨·罗伊施（Jochen Reusch）。MP-1 和 MP-2 中自主 SIMD 灵活性。在 *Proc. of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*，第 98 – 99 页，1 993 年。DOI: 10.1145/165231.165244。9

塞德里克·努格特伦（Cedric Nugteren）、格特-扬·范登布拉克（Gert-Jan Van den Braak）、亨克·科波拉尔（Henk Corporaal）和亨利·巴尔（Henri Bal）。基于重用距离理论的详细GPU缓存模型。发表于 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第37-48页，2014年。DOI: 10.1109/hpca.2014.6835955. 79

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

英伟达。*NVIDIA tesla V100 GPU architecture*。2017年。27, 31

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

NVIDIA公司. 内部Volta：世界上最先进的数据中心GPU. https://devblogs.nvidia.com/parallelforall/inside-volta/, 2017年5月. 1, 17, 26 NVIDIA公司. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110*, a. 13 NVIDIA公司. *NVIDIA GeForce GTX 680*, b. 16 NVIDIA公司. *CUDA Binary Utilities*, c. 16 *Parallel Thread Execution ISA (Version 6.1)*. NVIDIA公司, CUDA Toolkit 9.1版, 2017年11月. 14 Lars Nyland等. 美国专利 #8,086,806: 用于合并并行线程内存访问的系统和方法 (受让人: NVIDIA公司), 2011年12月. 71 Marek Olszewski, Jeremy Cutler, 和 J. Gregory Steffan. JudoSTM: 一种用于软件事务内存的动态二进制重写方法. 见 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 第365 – 375页, 2007. DOI: 10.1109/pact.2007.4336226. 96 Jason Jong Kyu Park, Yongjun Park, 和 Scott Mahlke. Chimera: 在共享GPU上的协作抢占多任务处理. 见 *Proc. of the ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015. DOI: 10.1145/2694344.2694346. 92 David A. Patterson 和 John L. Hennessy. *Computer Organization and Design: The Hardware/-Software Interface*. 2013. 78 Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, 和 Todd C. Mowry. 基础-增量-即时压缩：片上缓存的实用数据压缩方法. 见 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2012. 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. 见 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2013年12月a. 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. 见 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 第457 – 467页, 2013年b. DOI: 10.1145/2540708.2540747. 4

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

任晓薇和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)*，第625-636页，2017年。DOI: 10.1109/hpca.2017.40. 72

闵秀·鲁（Minsoo Rhu）和马坦·埃雷兹（Mattan Erez）。CAPRI：用于处理GPGPU架构中控制分歧的压缩适当性预测。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*，第61-71页，2012年。DOI：10.1109/isca.2012.6237006。44

闵洙·鲁（Minsoo Rhu）和马坦·埃雷兹（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。DOI: 10.1109/hpca.2013.6522353. 49

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

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

蒂莫西·G·罗杰斯、迈克·奥康纳和托尔·M·阿莫德特。意识到分歧的warp调度。在 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2013。DOI：10.1145/2540708.2540718。79, 90

蒂莫西·G·罗杰斯、丹尼尔·R·约翰逊、迈克·奥康纳和斯蒂芬·W·凯克勒。可变线程束大小架构。发表于 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015。DOI: 10.1145/2749469.2750410。53

徐尚敏，赵江元，李在镇。OpenCL 中 NAS 并行基准测试的性能特征。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: 通过减少内存停顿加速 GPU warp。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*，第 174 – 185 页，2015。DOI: 10.1109/hpca.2015.7056031。91, 92

安基特·塞西亚和斯科特·马尔克。Equalizer：GPU资源的动态调整以实现高效执行。发表于 *Proc. of the ACM/IEEE International Symposium on Microarchitecture (MICRO)*，2014年。DOI: 10.1109/micro.2014.16。86

Ryan Shrout。AMD ATI Radeon HD 2900 XT 评测：R600 到来。*PC Perspective*，2007年5月。75

Mark Silberstein, Bryan Ford, Idit Keidar 和 Emmett Witchel. GPUfs：将文件系统与 GPU 集成。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。用于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：使用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 Steffen 和 Joseph Zambreno。通过动态微内核的架构支持改进全局渲染算法的 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。针对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。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：可实现的同时多线程处理器的指令提取和发射。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 1996. DOI: 10.1145/232973.232993. 88 Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, 和 David Kaeli. Multi2Sim: 一个用于CPU-GPU计算的仿真框架。在 *Proc. of the ACM/IEEE International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 页码 335 – 344, 2012. DOI: 10.1145/2370816.2370865. 17 Aniruddha S. Vaidya, Anahita Shayesteh, Dong Hyuk Woo, Roy Saharoy, 和 Mani Azimi. 通过线程组内压缩优化SIMD分歧。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 368 – 379, 2013. DOI: 10.1145/2485922.2485954. 44, 46 Wladimir J. van der Lann. Decuda. `http://wiki.github.com/laanwj/decuda/` Jin Wang 和 Sudhakar Yalamanchili. 非结构化GPU应用中动态并行的表征与分析。在 *Proc. of the IEEE International Symposium on Workload Characterization (IISWC)*, 页码 51 – 60, 2014. DOI: 10.1109/iiswc.2014.6983039. 95 Jin Wang, Norm Rubin, Albert Sidelnik, 和 Sudhakar Yalamanchili. 动态线程块启动：支持GPU上非规则应用的一种轻量级执行机制。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 页码 528 – 540, 2016a. DOI: 10.1145/2749469.2750393. 95 Jin Wang, Norm Rubin, Albert Sidelnik, 和 Sudhakar Yalamanchili. Laperm: 针对GPU动态并行性的局部性感知调度器。在 *Proc. of the ACM/IEEE International Sympo- sium on Computer Architecture (ISCA)*, 2016b. DOI: 10.1109/isca.2016.57. 96 Kai Wang 和 Calvin Lin. 用于SIMT GPU的解耦仿射计算。在 *Proc. of the ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2017. DOI: 10.1145/3079856.3080205. 58, 61, 62 D. Wong, N. S. Kim, 和 M. Annavaram. 使用线程组内操作数值相似性近似线程组。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016. DOI: 10.1109/hpca.2016.7446063. 60, 61 X. Xie, Y. Liang, Y. Wang, G. Sun, 和 T. Wang. 协调静态和动态GPU缓存绕过。在 *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2015. DOI: 10.1109/hpca.2015.7056023. 81

徐云龙，王锐，Nilanjan Goswami，李涛，高兰，钱德培。适用于GPU架构的软件事务内存。在 *Proc. of the ACM/IEEE International Symposium on Code Generation and Optimization (CGO)*，第1:1 – 1:10页，2014年。DOI: 10.1145/2581122.2544139。

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

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

侯云青。NVIDIA FERMI 汇编器。`https://github.com/hyqneuron/asfermi` 16 Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, 和 Xipeng Shen。运行时线程数据重新映射通过消除线程分歧动态优化 GPU 应用程序。在 *Proc. of the ACM International Conference on Supercomputing (ICS)*, 第 115 – 126 页, 2010 年。DOI: 10.1145/1810085.1810104。 45 William K. Zuravleff 和 Timothy Robinson。美国专利 # 5,630,096：同步 DRAM 控制器，通过允许内存请求和命令乱序发出以最大化吞吐量，1997 年 5 月 13 日。 77

作者简介

托尔·M·阿莫德特

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 年以来一直担任该系教职。他当前的研究重点是通用 GPU 的架构和能效计算，最近包括机器学习加速器。他与其研究小组的学生共同开发了广泛使用的 GPGPU-Sim 模拟器。他的三篇论文被 *IEEE Micro Magazine* 评为“ Top Picks ”，第四篇论文被评为“ Top Picks ”荣誉提名。他的一篇论文还被 *Communications of the ACM.* 选为“ Research Highlight ”。他被列入 MICRO 名人堂。他曾于 2012 – 2015 年担任 *IEEE Computer Architecture Letters* 的副编辑，2012 – 2016 年担任 *International Journal of High Performance Computing Applications* 的副编辑，曾担任 ISPASS 2013 的程序主席、ISPASS 2014 的大会主席，并参与了众多程序委员会的工作。他于 2012 – 2013 年在斯坦福大学计算机科学系担任访问副教授。他于 2010 年获得 NVIDIA 学术合作奖，2016 – 2019 年获得 NSERC Discovery Accelerator 奖，并于 2016 年获得 Google Faculty Research 奖。

托尔在多伦多大学获得了工程科学学士学位（BASc）、硕士学位（MASc）和博士学位（Ph.D.）。他的大部分博士研究工作是在他作为英特尔微架构研究实验室实习生期间完成的。随后，他在 NVIDIA 工作，负责 GeForce 8 系列 GPU 的内存系统架构（“ 帧缓冲器 ”）——这是第一款支持 CUDA 的 NVIDIA GPU。

Tor已在不列颠哥伦比亚省注册为专业工程师。

威尔逊·伟伦·冯

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，他致力于下一代 GPU IP 的开发。他对计算机架构的理论和实践方面都很感兴趣。冯伟伦曾获得 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。他是广泛使用的 GPGPU-Sim 模拟器的主要贡献者之一。他的两篇论文被 *IEEE Micro Magazine.* 评为计算机架构领域的“ Top Pick ”。冯伟伦在不列颠哥伦比亚大学完成了计算机工程学士学位 (BASc)、硕士学位 (MASc) 和博士学位 (Ph.D.)。在攻读博士学位期间，他曾在 NVIDIA 实习。

提摩西·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。他感兴趣的是探索能够提高程序员生产力和能源效率的计算机系统和架构。蒂莫西曾获得 NVIDIA 研究生奖学金和 NSERC 亚历山大·格雷厄姆·贝尔加拿大研究生奖学金。他的研究工作被 *IEEE Micro Magazine* 评为“计算机架构领域的顶尖成果”，并被 *Communications of the ACM.* 选为“研究亮点”。在攻读博士学位期间，蒂莫西曾在 NVIDIA Research 和 AMD Research 实习。在攻读研究生之前，蒂莫西曾在 Electronic Arts 担任软件工程师，并获得麦吉尔大学电气工程学士学位。