第 3 章

GPGPU 控制核心架构

抽象为本，GPGPU 编程模型从较高的层次抽象了 GPGPU 的计算模型、线程模型和存储模型，这有利于编程人员采用传统串行思想进行并行程序的设计；架构为魂，GPGPU 架构和微体系结构的设计是抽象的根本，与编程模型息息相关。

本章将在 SIMT 计算模型基础上，介绍 GPGPU 控制核心架构和微体系结构的设计。本章的介绍以桌面 GPGPU 为实例，但不拘泥于特定工业产品的设计，试图以更广泛和深入的视角探索在 SIMT 架构下如何进行高效的 GPGPU 控制核心架构设计，有序地组织起大规模线程的并行执行，以揭示 GPGPU 架构进行高性能通用计算的机理。

3.1 GPGPU 架构概述

3.1.1 CPU-GPGPU 异构计算系统

遵循经典的冯·诺依曼架构，GPGPU 大规模线程并行的方式，与传统的 CPU 一起构成了当前普遍存在于桌面计算机和工作站的异构计算平台。虽然两者的并行度都在增加，但 GPGPU 大规模并行计算的方式是串行 CPU 的重要补充。两者采用分工合作的模式，为当前众多应用程序提供了卓越的处理性能。

一个由 CPU 和 GPGPU 构成的异构计算平台，可以在较为宏观的层面上对其计算、存储和互连等主要特征加以描述。典型的 CPU-GPGPU 异构计算平台如图 3-1 所示，GPGPU 通过 PCI-E[①] 接口连接到 CPU 上。CPU 作为控制主体统筹整个系统的运行。PCI-E 充当 CPU 和 GPGPU 的交流通道，CPU 通过 PCI-E 与 GPGPU 进行通信，将程序中的内核函数加载到 GPGPU 中的计算单元阵列和内部的计算单元上执行。为了驱动内核函数的计算，所有需要的代码、配置和运行数据都需要从硬盘加载到主机端存储器中，然后由一系列运行和驱动 API 将数据传送到 GPGPU 的设备端存储器中。一旦所有的配置、代码及数据都准备完善之后，GPGPU 则启动内核函数的运算，通过大算力完成计算。在计算结

① PCI-E(Peripheral Component Interconnect Express)是一种高速串行计算机扩展总线标准。

果输出之后，CPU 再将结果由设备端存储器传送回主机端存储器，等待下一次调用。

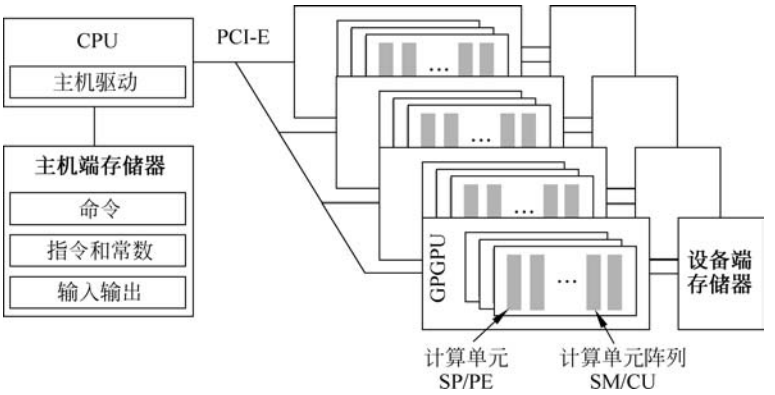


图 3-1 典型的 CPU-GPGPU 异构计算平台

与图形图像处理中利用 OpenGL 和 Direct3D 提供的 API 操作将 GPU 作为图形协处理器的方式类似，在通用处理中，CUDA 和 OpenCL 也提供了 API 操作向 GPGPU 发送命令、程序和数据，将 GPGPU 视为计算协处理器来使用，实现管控。通过这种方式，CPU 与 GPGPU 串并相协，优势共补，构建起一个强大的异构计算平台。

当然，CPU+GPGPU 的异构计算架构也不仅仅拘泥于上述形式。一种变种的异构计算平台架构就是统一存储结构系统。这种系统往往仅配备主机端存储器而省去设备端存储器，而 CPU 和 GPGPU 两者共用主机端存储器。这种系统的一个实例是 AMD 的异构系统架构(Heterogeneous System Architecture，HSA)。它采用硬件支持的统一寻址，使得 CPU 和 GPGPU 能够直接访问主机端存储器，无须在主机端存储器和设备端存储器之间进行显式的数据复制。借助 CPU 与 GPGPU 之间的内部总线作为传输通道，通过动态分配系统的物理存储器资源保证了两者的一致性，提高了两者之间数据通信的效率。但由于 GPGPU 专用的设备端存储器(如 GDDR)往往具有更高的带宽，共用主机端存储器(如 DDR)构建的这种系统容易受到存储带宽的限制，也可能由于存储器的争用导致访问延时的增加。

另外一种高性能变种是使用多个 GPGPU 并行工作。这种形式需要借助特定的互连结构和协议，将多个 GPGPU 有效地组织起来。这种系统的一个典型实例是 NVIDIA 的 DGX 系统。它通过 NVIDIA 开发的一种总线及通信协议 NVLink，采用点对点结构、串列传输等技术，实现多 GPGPU 之间的高速互连。为了解决 GPGPU 通信编程的问题，NVIDIA 还提供了 NCCL(NVIDIA Collective Communications Library)等支持，采用多种通信原语在 PCI-E、NVLink 及 InfiniBand 等多种互连上实现多 GPGPU 和 CPU 之间的高速通信。

3.1.2 GPGPU 架构

虽然不同厂商、不同架构、不同型号的 GPGPU 产品有所差异，但 GPGPU 核心的整体架构存在一定的共性特征。图 3-2 显示了典型的 GPGPU 架构及可编程多处理器的组成，

其核心部分包含了众多可编程多处理器，NVIDIA 称之为流多处理器（Streaming Multiprocesosr，SM），AMD 称之为计算单元（Compute Unit，CU）。每个可编程多处理器又包含了多个流处理器（Streaming Processor，SP），NVIDIA 称之为 CUDA 核心，AMD 称之为 PE（Processing Element），支持整型、浮点、特殊函数、矩阵运算等多种不同类型的计算。

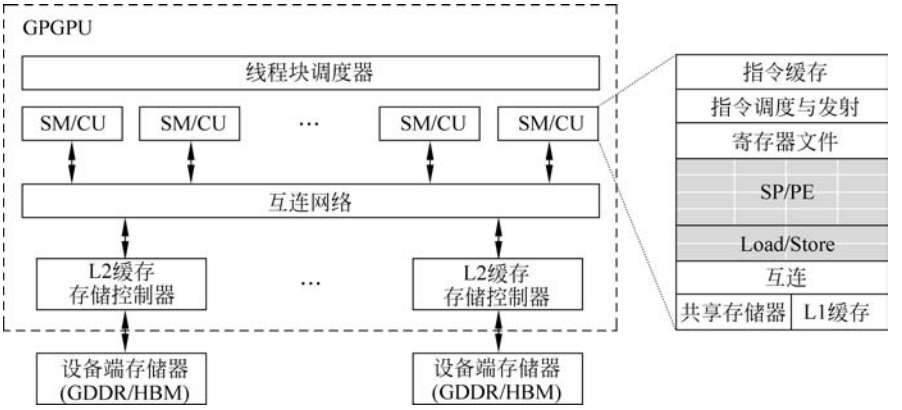


图 3-2　典型的 GPGPU 架构及可编程多处理器的组成

可编程多处理器构成了 GPGPU 核心架构的主体。它们从主机接口的命令队列接收 CPU 发送来的任务，并通过一个全局调度器分派到各个可编程多处理器上执行。可编程多处理器通过片上的互连结构与多个存储分区相连实现更高并行度的高带宽访存操作。每个存储分区包含了第二级缓存（L2 cache）和对应的 DRAM 分区。通过调整可编程多处理器和存储分区的数量，GPGPU 的规模可大可小，并通过编程框架实现对这些灵活多变架构的统一编程。

在这样的架构下，用 CUDA 或 OpenCL 编写的通用计算程序主要在可编程多处理器和它内部的流处理器中完成。由于 GPGPU 的主体结构由数量可扩展的可编程多处理器构成，每个可编程多处理器又包含了多个流处理器，所以可编程多处理器可以在很大规模上并行执行细粒度的线程操作。可编程多处理器的重复性和独立性也简化了硬件设计，同时与线程块的编程模型抽象相互对应，使得线程块可以非常直接地映射到可编程多处理器上执行。

如图 3-2 所示，可编程多处理器的一个特点就是包含了大量的流处理器。流处理器由指令驱动，以流水化的方式执行指令，提高指令级并行度。每个流处理器都有自己的寄存器，如果单个线程使用的寄存器少，则可以运行更多的线程，反之则运行较少的线程。编译器会优化寄存器分配，以便在线程并行度和寄存器溢出之间寻找更高效的平衡。每个流处理器都配备一定数量的算术逻辑单元，如整型和浮点单元，使得可编程多处理器形成了更为强大的运算能力。可编程多处理器中还包含特殊功能单元（Special Function Unit，SFU），执行特殊功能函数及超越函数。可编程多处理器通过访存接口执行外部存储器的加载、存

储访问指令。这些指令可以和计算指令同时执行。另外 NVIDIA 从 Volta 架构的 GPGPU 开始，在可编程多处理器中还增加了专用的功能单元，如张量核心（Tensor Core）等，支持灵活多样的高吞吐率矩阵运算。

可以看到，GPGPU 架构所采用的可编程多处理器和流处理器的二级层次化组织结构与 CUDA 和 OpenCL 编程模型的二级线程结构具有直接的对应关系。GPGPU 所采用的 SIMT 架构体现为硬件多线程，每个线程运行自己的指令流。同时，传统的图形流水线中对顶点、几何和像素渲染的处理也可以在可编程多处理器和流处理器中完成，视为统一的可编程图形渲染架构。另外的输入装配、建立和光栅化等图形处理的固定功能模块则被插入 GPGPU 架构当中，成为可编程图形渲染结构，与可编程多处理器一起实现图形专用功能的处理，达到了架构的统一。

这种统一的 GPGPU 的架构有如下的优点。

（1）有利于掩盖存储器加载和纹理预取的延时。硬件多线程提供了数以千计的并行独立线程，这些线程可以在一个多处理器内部充分利用数据局部性共享数据，同时利用其他线程的计算掩盖存储访问延时。由于典型的 GPGPU 只有小的流缓存而不像 CPU 那样具有大的工作集缓存，因此一个存储器和纹理读取请求通常需要经历全局存储器的访问延迟加上互连和缓冲延迟，可能高达数百个时钟周期。在一个线程等待数据和纹理加载时，硬件可以执行其他线程。尽管对于单个线程来说存储器访问延迟还是很长，但整体访存延时被掩盖，计算吞吐率得以提升。

（2）支持细粒度并行图形渲染编程模型和并行计算编程模型。一个图形顶点或像素渲染是一个处理单个顶点或像素的单一线程程序。类似地，一个 CUDA/OpenCL 程序也是一个单一线程计算的类 C/C++程序。图形和计算程序通过调用众多的并行线程以渲染复杂图形或解决复杂计算问题。在图形渲染程序或通用计算程序中，硬件多线程可以动态地轮换各自的线程，采用硬件管理成百上千的并发线程，简化了调度开销。

（3）将物理处理器虚拟化成线程和线程块以提供透明的可扩展性，简化并行编程模型。为支持独立的顶点、像素程序或 CUDA/OpenCL 的类 C/C++程序，每个线程都有自己的私有寄存器、存储器、程序计数器和线程执行状态，从而执行独立的代码路径。编程人员可以假想为一个线程编写一个串行程序，而必要时在线程块的并发线程之间进行同步栅栏。轻量级的线程创建、调度和同步有效地支持了 SIMT 计算模型。

面对数以万计的线程，硬件资源仍然有限，因此硬件仍然会对海量的线程进行分批次的处理。GPGPU 中往往采用线程束（NVIDIA 称为 warp，AMD 称为 wavefront）的方式创建、管理、调度和执行一个批次的多个线程。当前，一种典型的配置是一个 warp 包含 32 个线程，一个 wavefront 包括 64 个线程。当这些线程具有相同的指令路径时，GPGPU 就可以获得最高的效率和性能。在线程束粒度基础上，SIMT 计算与标量指令的执行方式类似，只不过有多个线程束交织在一起，整体上实现了所有线程随时间向前推进的效果。

3.1.3　扩展讨论：架构特点和局限性

1. 架构特点

GPGPU是由GPU发展而来的，所以GPGPU是在图形处理硬件的基础上，以可编程多处理器阵列为基础来构建的并行结构，以支持如CUDA和OpenCL等编程模型所需要的大规模并行线程。GPGPU在可编程多处理器阵列中统一了图形处理中顶点、几何、像素渲染处理和通用并行计算的需求，并在其中紧密集成了原有图形处理中的固定功能处理单元，如纹理滤波、光栅建立、光栅操作和高清视频处理等。

与多核CPU相比，GPGPU的架构具有本质的不同。GPGPU提供的线程数量是CPU的2～3个数量级，例如在NVIDIA最新的Ampere架构中线程数达到221 184。硬件中数量众多的可编程流多处理器和流处理器很好地适应了这种特点。

基于计算的重复和控制的相对单一性，GPGPU所采用的SIMT计算模型借助数据流之间的独立性简化了线程间的数据交互。这种数据并行的编程模型不但可以简化GPGPU的架构，有效地提高了用于计算的晶体管比例，还使得GPGPU的并行度可以持续提升。

GPGPU架构有着良好的扩展性和延续性。用户往往只是期望游戏、图形、图像和通用计算功能能够运行，而且要足够快，对它到底有多大并行规模并不关心。因此，可以根据不同的性能、市场和价格需求，通过调整可编程多处理器和存储分区的数量、缩放阵列的规模，快速迭代出合适的GPGPU设计。GPGPU的编程模型和架构设计可以以透明扩展的方式支持不同规模的产品。

GPGPU采用了大量计算逻辑部件来实现算力的提升。虽然GPGPU还是使用传统的硬件，但其背后将各种部件重新整合，使其能保证大算力的同时保留了良好的可编程能力，从而满足了如图形渲染、机器学习、大数据挖掘和数字货币等诸多新兴任务的需求，在一定程度上延续了摩尔定律的发展和冯·诺依曼架构的生命力。这就是架构设计的魅力。

2. 架构局限性

以CUDA和OpenCL为代表的GPGPU编程模型提供了高度灵活的可编程能力。但为了提高GPGPU硬件的执行效率并减少设计开销，经典的GPGPU编程模型也做出了一些改变。

(1) 为了能使GPGPU程序可以在任意数量的可编程多处理器上运行，同一个线程网格中的线程块之间不允许存在依赖而能够独立执行。由于线程块独立且能以任意的顺序执行，多个线程块之间的同步和通信往往需要更高开销的操作才能完成，例如通过全局存储器通信，或利用原子操作进行协同，抑或利用新的线程网格来处理。线程块内的同步则可以利用同步栅栏等在线程块中的所有线程上实行。不过，随着GPGPU编程模型的不断发展和通用性的不断增强，线程块的独立性也出现了一些变化，正如2.4.2节所介绍的协作组(cooperative groups)就允许重新选择线程构成协作组以实现多种粒度的协同操作。

(2) 递归程序早期也并不被允许。在大规模并行的很多情况下，递归操作并没有太大

的用处，而且可能会消耗大量的存储器空间。通常使用递归编写的程序，如快速排序，都可以变换成并行结构来实现。不过为了支持更为通用的编程，NVIDIA 在计算能力 2.0 的 GPGPU 架构中也开始支持有限制的递归程序。

(3) 典型的 CPU-GPGPU 异构计算还是需要各自拥有独立的存储空间，因此需要在主机端存储器和设备存储器之间复制数据和结果。这虽然会带来额外的开销，但可以通过执行足够大的计算密集型问题来分摊。当然，这个问题不仅仅是编程模型和架构设计的问题，也和存储器件本身的特性密切相关。

(4) 在早期的 GPGPU 中，线程块和线程只能通过 CPU 创建，而不能在内核函数执行过程中创建，这种方式有利于简化运行时管理和减小硬件多线程的开销。不过一些新的 GPGPU 架构也开始支持这一特性。例如，NVIDIA 从计算能力 3.0 的 Kepler 架构及 CUDA 5.0 中引入了对动态内核函数的支持，可以在内核函数中启动新的内核函数。

3.2 GPGPU 指令流水线

流水线技术是利用指令级并行，提高处理器 IPC[①] 的重要技术之一。它在标量处理器中已经得到了广泛应用。不同功能的电路单元组成一条指令处理流水线，利用各个单元同时处理不同指令的不同阶段，可使得多条指令同时在处理器内核中运行，从而提高各单元的利用率和指令的平均执行速度。在大多数 GPGPU 架构中，虽然指令的执行粒度变为包含多个线程的线程束，但为了提高指令级并行，仍然会采用流水线的方式提高线程束指令的并行度。与单指令流水线相比，可以想象成水管变得更粗。当线程束中所有的线程具有相同的指令路径时，指令流水的方式与标量流水线类似。但当线程束中线程发生分支，不同线程执行不同的代码路径时，GPGPU 则采用了专门的技术来解决这一问题，例如 3.3 节中将介绍的 SIMT 堆栈技术。

图 3-3 显示了一种典型的 GPGPU 架构流水线设计[②]。可以看到，每个线程束按照流水方式执行指令的读取(fetch)、解码(decode)、发射(issue)、执行(execute)及写回(writeback)过程。这一过程与标量流水线非常类似，但不同之处在于从取指令开始，GPGPU 的流水线以线程束为粒度执行，各个线程束相互独立。同时 GPGPU 的指令调度器原则上可以在任何已经就绪的线程束中挑选一个并采用锁步(lockstep)的方式执行。锁步执行使得所有的执行单元都执行同一条指令，从而简化控制逻辑，把硬件更多地留给执行单元。GPGPU 的流水线不必像动态流水线那样利用高复杂度和高开销的控制执行逻辑来提高指令并行性。

---

① Instruction Per Cycle，每周期指令数。它的值越高，说明指令级并行度越高。

② 该流水线结构参考了 GPGPU-Sim 的流水线设计。GPGPU-Sim 是加拿大 UBC 大学研究团队根据 NVIDIA 的 Fermi 架构 GPGPU 设计的一款周期级架构模拟器，广泛应用于 GPGPU 体系结构设计研究。

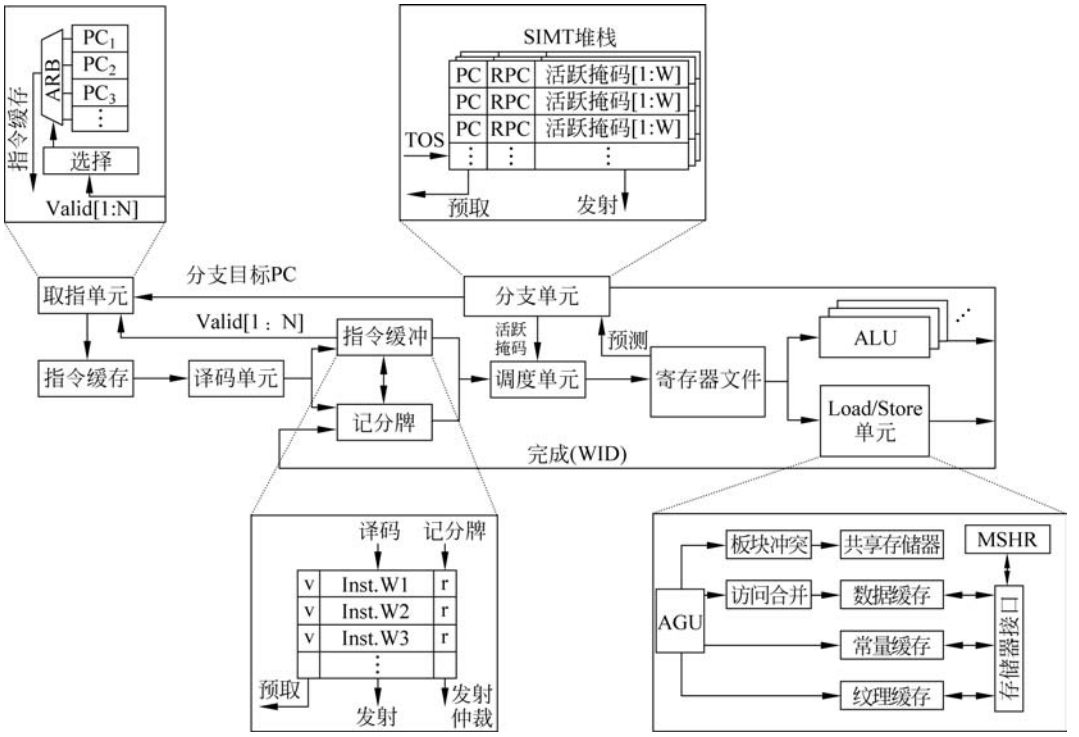


图 3-3　一种典型的 GPGPU 架构流水线设计

3.2.1　前段：取指与译码

流水线始于取指。GPGPU 的指令流水线前段主要涉及取指单元(fetch)、指令缓存(I-cache)、译码单元和指令缓冲(I-buffer)等部件。

1. 取指单元

取指单元是根据程序计数器(Program Counter,PC)的值,从指令缓存中取出要执行指令的硬件单元。取出来的指令经过译码后会保存在指令缓冲中,等待指令后续的调度、发射和执行。

在标量流水线中,一般只需要一个 PC 来记录下一条指令的地址。但由于 GPGPU 中同时存在多个线程束且每个线程束执行的进度可能并不一致,取指单元中就需要保留多个 PC 值,用于记录每个线程束各自的执行进度和需要读取的下一条指令位置。这个数目应该与可编程多处理器中允许的最大线程束数量相同。众多线程束进而通过调度单元选出一个线程束来执行。

2. 指令缓存

指令缓存接收到取指单元的 PC,读取缓存中的指令并发送给译码单元进行解码。指令高速缓存可以减少直接从设备端存储器中读取指令的次数。

本质上，指令缓存也是缓存，可以采用传统的组相联结构及 FIFO 或 LRU 等替换策略来进行设计。取指单元对指令缓存的访问也可能会发生不同的情况：如果命中，指令会被传送至译码单元；如果缺失，会向下一层存储请求缺失的块，等到缺失块回填指令缓存后，访问缺失的线程束指令会再次访问指令缓存。对 GPGPU 来说，不管命中还是缺失，调度器都会处理下一个待调度线程束的取指请求。还有一种可能的情况是指令缓存的资源不足，此时则无法响应取指单元的请求，只能停顿直到指令缓存可以来处理。

3. 译码单元

译码单元对指令缓存中取出的指令进行解码，并且将解码后的指令放入指令缓冲中对应的空余位置上。

根据 SASS 指令集的定义和二进制编码规则，译码单元会判断指令的功能、指令所需的源寄存器、目的寄存器和相应类型的执行单元或存储单元等信息，进而给出控制信号，控制整个线程束流水线的运行。

4. 指令缓冲

指令缓冲用于暂存解码后的指令，等待发射。考虑到每个可编程多处理器中会有许多线程束在执行，指令缓冲可以采用静态划分的方式来为每个线程束提供专门的指令条目，保留已解码待发射的指令。这样，每个线程束就可以直接索引到相应的位置，避免每次从指令缓冲中查找指令所带来较高的延时和功耗开销。

每个指令条目一般包含一条解码后的指令和两个标记位，即一个有效位(valid)和一个就绪位(ready)。有效位表示该条指令是有效的已解码未发射指令，而就绪位表示该指令已经就绪可以发射。就绪的指令往往需要通过诸如记分牌的相关性检查等一系列条件，并且需要有空闲的硬件资源才能得以发射。一旦某指令发射完成，就会重置对应的标记位等待进一步填充新指令。在初始时，这些标记位也会被清除以表明指令缓冲空闲。

指令缓冲中的有效位还会反馈给取指单元，表明指令缓冲中是否有空余的指定条目用于取指新的线程束指令。如果有空余条目，应尽快利用取指单元从指令缓存中获得该线程束的后续指令；如果没有空余条目，则需要等待指令缓冲中该线程束的指令被发射出去后，条目被清空才能进行指令读取。

3.2.2 中段：调度与发射

指令的调度与发射作为指令流水的中段，连接了前段取指和后段执行部分，对流水线的执行效率有着重要的影响。

1. 调度单元

调度单元通过线程束调度器(warp scheduler)选择指令缓冲中某个线程束的就绪指令发射执行。发射会从寄存器文件中读取源寄存器传送给执行单元。调度器则很大程度上决定了流水线的执行效率。

为了确保指令可以执行，调度单元需要通过各种检查以确保指令就绪并且有空闲执行单元才能发射。这些检查包括没有线程在等待同步栅栏及没有数据相关导致的竞争和冒

险等。

不同指令在不同类型的流水线上执行。例如，运算类型指令在算术逻辑部件(Arithmetic Logic Unit，ALU)中执行；访存类型指令会在存储访问单元(Load/Store 单元)中执行。当遇到条件分支类指令时，需要合理地处置指令缓冲中的指令。例如，在跳转发生时清空指令缓冲中该线程束的指令条目，同时该线程束的 PC 也需要调整，并根据分支单元如 SIMT 堆栈来管理线程分支下的流水线执行。

2. 记分牌

记分牌单元(scoreboard)主要是检查指令之间可能存在的相关性依赖，如写后写(Write-After-Write，WAW)和写后读(Read-After-Write，RAW)，以确保流水化的指令仍然可以正确执行。

经典的记分牌算法会监测每个目标寄存器的写回状态确保该寄存器写回完成前不会被读取或写入，避免后续指令的读操作或写操作引发 RAW 冒险或 WAW 冒险。记分牌算法通过标记目标寄存器的写回状态为“未写回”，确保后续读取该寄存器的指令或再次写入该寄存器的指令不会被发射出来。直到前序指令对该目的寄存器的写回操作完成，该目的寄存器才会被允许读取或写入新的数据。

3. 分支单元和 SIMT 堆栈

对于指令中存在条件分支的情况，例如 if…else…语句，它们会破坏 SIMT 的执行方式。条件分支会根据线程束内每个线程运行时得到的判断结果，对各个线程的执行进行单独控制，这就需要借助分支单元，主要是活跃掩码(active mask)和 SIMT 堆栈进行管理，解决一个线程束内线程执行不同指令的问题。

GPGPU 架构一般都会采用串行化不同线程执行的方式来处理分支的情况。例如，可以先执行 if 分支(true 路径)再执行 else 分支(false 路径)。活跃掩码用来指示哪个线程应该执行，哪个线程不应该执行，普遍采用 n 比特的独热(one-hot)编码形式(n 值与线程束内线程的数量一致)，其中每一位对应了一个线程的条件判断结果。如果该线程需要执行该指令，则对应位为 1，否则为 0。活跃掩码会传送给发射单元，用于指示该发射周期的线程束中哪些线程需要执行，从而实现分支线程的独立控制和不同分支的串行化执行。

线程分支会严重影响 SIMT 的执行效率，导致大量执行单元没有被有效利用。研究人员对此提出了不同的技术来减轻这种影响。

4. 寄存器文件和操作数收集

指令执行之前会访问寄存器文件(register file)获取源操作数。指令执行完成后还需要写回寄存器文件完成目标寄存器的更新。

寄存器文件作为每个可编程多处理器中离执行单元最近的存储层次，需要为该可编程多处理器上所有线程束的线程提供寄存器数值。为了掩盖如存储器访问等长延时操作，GPGPU 会在多个线程束之间进行调度，这也就要求寄存器文件需要有足够大的容量能够同时为多个线程束保留寄存器数据，因此其设计与传统 CPU 有显著不同。例如，GPGPU 的寄存器文件与其他存储层次会呈现“倒三角”结构。出于电路性能、面积和功耗的考虑，寄

存器文件会分板块设计，且每个板块只有少量访问端口（如单端口）的设计方式。对不同板块的数据同时读取可以在同周期完成，但是不同请求如果在同一板块，就会出现板块冲突而影响流水线性能。板块冲突也有不同的处理方式。NVIDIA 的 GPGPU 借助操作数收集器（operand collector）结构和寄存器板块交织映射等方式减轻板块冲突的可能性。

3.2.3 后段：执行与写回

作为指令执行的后段，计算单元是对指令执行具体操作的实现，存储访问单元则完成数据加载及存储操作。计算单元主要包括整型、浮点和特殊功能单元在内的多种功能单元。NVIDIA 的 GPGPU 从 Volta 架构起还引入了张量核心单元（tensor core）来支持大规模矩阵计算。

1. 计算单元

GPGPU 需要为每个可编程多处理器配备许多相同的流处理器单元来完成一个线程束中多个线程的计算需求，同时还配备了多种不同类型的计算单元，用来支持不同的指令类型，如整型、浮点、特殊函数、矩阵运算等。不同类型的指令从寄存器文件中获得源操作数，并将各自的结果写回到寄存器文件中。

作为基本的算术需求，GPGPU 中提供了较为完整的算术逻辑类指令，支持通用处理程序的执行。在 NVIDIA 的 GPGPU 架构中，流处理器单元体现为 CUDA 核心，它提供了整型运算能力和单精度浮点运算能力。不同的架构会配备不同数量的双精度浮点硬件单元，以不同的方式对双精度浮点操作进行支持，以满足高性能科学计算的需求。

某些指令需要在特殊功能单元（Special Function Unit，SFU）上执行，这些指令包括倒数、倒数平方根和一些超越函数。这些单元也以 SIMT 方式执行。但由于这些特殊功能单元往往对硬件的消耗很高，所以一般数量不会很多，而是采用分时复用的方式。例如，在 NVIDIA 的 GPGPU 架构中，一个 SFU 可能会被 4 个 SP 共享，吞吐率就降为原来的 1/4。另外，这些单元的另一个特点是它们并不一定严格遵循 IEEE 754 标准中对单精度浮点的精确性要求，这是因为对于许多 GPGPU 应用来说，更高的计算吞吐率往往是更重要的。如果应用对精确性有更高的要求，可以利用 CUDA 数学库中精确的函数来实现，这往往需要软件的介入。

近年来，为了支持深度神经网络的计算加速，NVIDIA 的 Volta、Turing 和 Ampere 架构开始增加了张量核心单元，主要为低精度的矩阵乘法提供更高的算力支持。关于张量计算单元的详细介绍，详见第 5 章的内容。

2. 存储访问单元

存储访问单元负责通用处理程序中 load 和 store 等指令的处理。由于配备了具有字节寻址能力的 load 和 store 等指令，GPGPU 可以执行通用处理程序。

如 2.3.1 节所介绍的，GPGPU 一般会包含多种类型的片上存储空间，如共享存储器、L1 数据缓存、常量缓存和纹理缓存等。存储访问单元实现了对这些存储空间的统一管理，进而实现对全局存储器的访问。同时针对 GPGPU 的大规模 SIMT 架构特点，存储访问单

元还配备了地址生成单元(Address Generation Unit,AGU)、冲突处理(bank conflict)、地址合并、MSHR(Miss Status Handling Registers)等单元来提高存储器访问的带宽并减小开销。当需要访问共享存储器中的数据时,冲突处理单元会处理可能存在的板块冲突,并允许在多周期完成数据的读取。对于全局存储器和局部存储器中的数据,load/store 指令会将同一线程束中多个线程产生的请求合并成一个或多个存储块的请求。面对 GPGPU 巨大的线程数量,存储访问单元通过合并单元将零散的请求合并成大块的请求,利用 MSHR 单元支持众多未完成的请求,有效地掩盖了对外部存储器的访问延时,提升了访问的效率。纹理存储器具有特殊的存储模式,需要经由特定的纹理单元进行访问。

由于不同的存储空间在 GPGPU 程序中会起到不同的作用,存储访问单元对各种存储空间实施差异化的管理。具体请参见第 4 章的内容。

3.2.4 扩展讨论:线程束指令流水线

1. 与其他流水线的比较

1) 与标量处理器流水线的比较

从 GPGPU 的流水线可以看出,它与标量处理器的流水线是非常相似的。通过将线程束指令划分为几个阶段,GPGPU 可以实现指令级并行。不同之处在于,从取指令开始,GPGPU 的流水线就以线程束为粒度,多个线程独立执行。GPGPU 采用了更为简单的锁步执行方式,所有执行单元都执行同一个操作,因此能够从已经就绪的线程束中选择一个进行执行。由于每个可编程多处理器内部都会有大量的线程和线程束等待执行,原则上 GPGPU 具有很大的调度空间来掩盖缓存缺失带来的访存操作等长延时操作。这使得 GPGPU 可以简化高速缓存的设计,不必像动态调度流水线一样利用高复杂度和高硬件开销的乱序执行方式寻找可以执行的指令来填充流水线,以及掩盖长延时操作带来的流水线停顿。GPGPU 的这种执行和调度方式在保证了指令并行性的同时,可以简化控制逻辑,使得 GPGPU 可以将硬件资源更多地留给计算等功能操作单元。

当然,并不是每时每刻线程束中的所有线程都能够完美地打包在一起执行,必然会有线程执行不同分支路径的情况,因此条件分支是 GPGPU 性能的重要影响因素之一,可编程多处理器必须要能够对这种情况进行有效的管理。同时,如何管理数量众多的线程束,并选择一个合理的线程束来执行,也是 GPGPU 调度器要解决的新问题。另外,线程产生数据访问请求时也可能会因为庞大的线程数量而相对分散,从而对访存性能来说也是非常不利的影响,GPGPU 需要更高效的访问策略对访问请求进行组织。

2) 与向量处理器流水线的比较

向量处理器和以 SIMT 为核心的 GPGPU 处理器起初都是为了支持数据级并行程序而设计的,但它们选取了不同的技术路径。数量更多的执行单元、灵活性更高的动态分支管理、更为复杂的存储架构、更强的存储访问能力及特有的线程和线程束调度机制是 GPGPU 流水线与向量处理器最显著的区别。

向量处理器采用数据流水的方式来一次性处理所有的向量元素,所以每次载入和存储

指令都需要进行大块的数据传输，往往存在较大的一次性启动延时代价。有的向量处理器配备了集中/分散(gather/scatter)及地址跳跃(striding)等地址访问能力来应对复杂的地址访问模式，但 GPGPU 基于单个线程独立的地址计算能力则更为灵活。同时，GPGPU 利用线程的切换来掩藏长延时的访存，等于在数据并行的维度上增加了延时掩藏的能力。

另外，在条件分支指令的处理上，两种架构都采用了活跃掩码的方式。区别在于，向量处理器可能会利用软件来管理活跃掩码的保存、求补和恢复等操作，而 GPGPU 普遍采用硬件的管理方式。这种方式往往更加灵活，也利于取得更好的性能。

这些机制和硬件单元使得 GPGPU 更为灵活，也具有更良好的可编程性来应对数据级并行(Data-Level Parallelism，DLP)之外的可并行任务。当然，并不是说向量处理器不能支持这些机制，新的融合体系结构设计可能会在两种架构之间找到更好的平衡点。更多关于向量处理器的内容可以参见文献[3]中的介绍。

3) 与 SIMD 流水线的比较

2.2.3 节中已经讨论了 SIMD 和 SIMT 在编程模型上的差异。从硬件层面上，SIMD 流水线保持了与标量流水线的高度相似性，可以认为仅仅是增加了 SIMD 扩展指令及在硬件上增加了独立并行的执行通路。而 GPGPU 流水线除了扩展执行单元的数量，还设计了完整的体系结构支持更为灵活的 SIMT 计算模型和不同的存储访问机制。GPGPU 的 SIMT 编程模型还可以通过不同的编程手法实现类似 MIMD 的并行计算模型。这种灵活性显然是 SIMD 流水线无法比拟的。

2. 线程束的宽度选择

在 GPGPU 的编程模型中，线程网格和线程块的大小都是编程人员可以根据应用需求进行调节的，而唯有线程束的大小是与硬件绑定且固定的。NVIDIA 的 GPGPU 将线程束(称为 warp)的宽度即线程(thread)的个数设置为 32，而 AMD 的 GPGPU 将线程束(称为 wavefront)的宽度即工作项(work-item)的个数设置为 64。为什么两者会选择不同的数值，为什么是这两个数值，或者说线程束的宽度究竟设置为多少才合适？人们对这些问题也进行了多种分析和研究。

一方面，对于使用相同线程数量执行的应用来说，如果线程束的宽度增加，那么执行应用所需的线程束数量就会变少，这可能会影响到线程束的并行度或 GPGPU 的调度能力，进而影响性能。一旦发生线程分支，不同的线程会执行不同的代码。越大的线程束遭遇分支的可能性会越高，导致性能损失的可能性也会越大。另一方面，由于每个线程束都需要独立地取指、访问 L1 高速缓存等资源，因此从直观上来看，越大的线程束在前端取指的次数也减少，访问 L1 高速缓存的次数也会越少。同理，越小的线程束在前端取指的次数也越多，访问 L1 缓存的次数也会变多，这都可能会带来性能上的差异。因此，虽然不能确切地推断为什么两种 GPGPU 会选择不同的数值，但很大可能是架构和应用方面多个因素折中的结果。

针对这一问题，研究人员在文献[4]中将不同线程束宽度对不同类型应用的性能影响进行了量化的研究。该研究针对 165 个真实应用的内核函数，将它们分成了三类：随着线程

束宽度下降而性能上升的发散型应用(divergent applications)、随着线程束宽度下降而性能基本不变的不敏感型应用(insensitive applications)及随着线程束宽度下降而性能下降的收敛型应用(convergent applications)。不同线程束宽度对应用的性能呈现差异化的影响结果如图 3-4 所示。

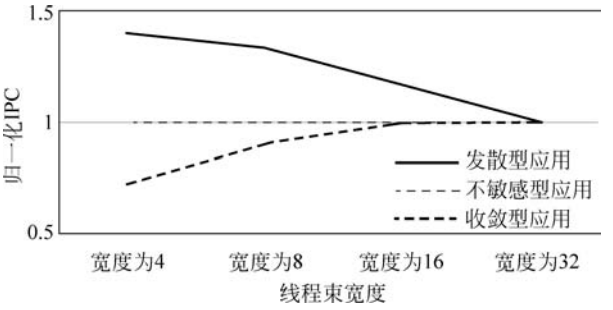


图 3-4　不同线程束宽度对应用的性能呈现差异化的影响

从图 3-4 中可以看到,不同应用的性能对线程束宽度的变化反应不一,这个结果可以这样来理解:当线程束宽度下降时,L1 高速缓存的访问次数会增大,这与直觉相符。对于一些应用,当用宽度较小的线程束来代替较大的线程束时,原本按照较大线程束合并的访问会因存储器等资源的限制被分散到多个周期,呈现存储合并能力的退化。一般出现这种情况会使得整体性能下降。但对于发散型应用,这种性能的下降又会被下面两个因素弥补。

(1) 应用的控制流存在分支,且较少的线程会参与存储访问。

(2) 虽然出现了存储合并退化的现象,但是 L1 高速缓存的命中次数提升较为明显,同时较小的线程束宽度会提升 SIMT 通道的利用率,从而弥补性能。

在收敛型应用中,这种情况对于性能的影响是负面的,说明上述两个原因可能不能弥补存储合并退化带来的性能损失。根据该文献的统计,这些应用在线程束宽度下降时,存储访问总数和 L1 高速缓存未命中数均增加,有些应用还增加了 MSHR 的合并数量,这意味着访问 L1 高速缓存的次数增加了。

线程束的宽度除了对于性能存在影响,对于前端的压力也显著增大。当线程束宽度下降时,由于每个小线程束需要读取指令,从而相比于大线程束需要读取更多的指令。这对于收敛型应用和不敏感型应用的影响比较明显,因为根据该文献的统计,取指请求的数量对于线程束宽度下降有近乎线性的提升。而对于发散型应用,虽然取指请求也增加了,但增加了更多独立的控制路径,也会变得更加复杂多样。这对于提升发散型应用的性能可能至关重要。

从上面的分析可以看到,不同应用的内核函数对线程束宽度的相关性也并不一致,线程束的宽度与诸多架构因素也有着复杂的关系,因此线程束的宽度很大程度上也是折中的结果。从另一个角度来看,静态的线程束宽度设定并不能适合所有的内核函数和架构,那么是否可以动态调整线程束的宽度以适应更多的应用和架构,这也是值得进一步研究的问题。

3.3 线程分支

从整个流水线的角度，GPGPU 遵循了 SIMT 计算模型，按照线程束的组织进行指令的取指、译码和执行。这种方式使得编程人员可以按照串行化的思维完成大部分的代码，也允许每个线程独立地执行不同的工作。在执行阶段，如果遭遇了 if…else…等条件分支语句，不同线程需要执行的代码路径可能会不一致，就会出现线程分支或分叉。

代码 3-1 给出了一个包含嵌套分支的内核函数 CUDA 代码（左）和所对应的 PTX 代码（右）。假设线程束中有 4 个线程。起初，4 个线程执行基本块 A 中的代码，这时没有发生线程分支。但是当指令块 A 到达执行末尾时需要执行第 6 行的 if…else…语句，对应 PTX 第 6 行的分支指令 bar。假设有 3 个线程在执行时判断条件成立会去执行块 B 中的代码，1 个线程不成立而去执行块 F 中的代码，此时就发生了线程分支。同理，执行完指令块 B 代码后也发生了线程分支，一部分线程会去执行 C，而另一部分线程会去执行 D。图 3-5 展示了这段 CUDA 代码和 PTX 代码所提取出的分支流图。其中，每个框表示了需要执行的指令块及哪个线程将执行这个指令块，如 A/1111 表示 4 个线程都会执行指令块 A，C/1000 表示只有第 1 个线程会执行指令块 C。每个框之间的连线意味着相继执行的指令块。

代码 3-1　包含嵌套分支的内核函数示例

```
1   do {
2       t1 = tid * N;          // A
3       t2 = t1 + i;
4       t3 = data1[t2];
5       t4 = 0;
6       if(t3 != t4){
7           t5 = data2[t2];    // B
8           if(t5 != t4) {
9               x += 1;        // C
10          }else{
11              y += 2;        // D
12          }
13      } else {
14          z += 3;            // F
15      }
16      i++;                   // G
17  } while(i < N);
```

```
1   A:  mul.lo.u32      t1, tid, N;
2       add.u32         t2, t1, i;
3       ld.global.u32   t3, [t2];
4       mov.u32         t4, 0;
5       setp.eq.u32     p1, t3, t4;
6       @p1 bra         F;
7   B:  ld.global.u32   t5, [t2];
8       setp.eq.u32     p2, t5, t4;
9       @p2 bra         D;
10  C:  add.u32         x, x, 1;
11      bra             E;
12  D:  add.u32         y, y, 2;
13  E:  bra             G;
14  F:  add.u32         z, z, 3;
15  G:  add.u32         i, i, 1;
16      setp.le.u32     p3, i, N;
17      @p3 bra         A;
```

为了支持上述条件分支的执行，GPGPU 采取的方法也很直观，就是分别执行分支的不同路径，即按照 A/1111→B/1110→C/1000→D/0110→E/1110→F/0001→G/1111 的顺序分别执行其中给定的一个或几个线程，最终执行完所有线程。为了实现这种执行方式，GPGPU 往往会利用谓词寄存器和硬件 SIMT 堆栈相结合的方式对发生了条件分支的指令流进行管理。本节将介绍这一原理及它是如何解决分支问题的。为了提高执行的效率，还将针对线程分支的效率问题展开深入的讨论。

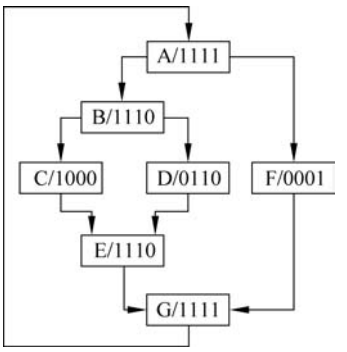


图 3-5　嵌套分支内核函数示例的分支流图

3.3.1　谓词寄存器

在理解 GPGPU 如何处理线程条件分支之前，先介绍谓词(predicate)寄存器的概念。谓词寄存器是为每个执行通道配备的 1 比特寄存器，用来控制每个通道是否开启或关闭。通常，谓词寄存器设置为 1 时，对应的执行通道将被打开，该通道的线程将得以执行并存储结果；谓词寄存器设置为 0 的通道将被关闭，该通道不会执行指令的任何操作。谓词寄存器广泛应用于向量处理器、SIMD 和 SIMT 等架构中用来处理条件分支。

GPGPU 架构普遍采用显式的谓词寄存器来支持线程分支，每个线程都配备有若干谓词寄存器。例如，在代码 3-1 的 PTX 代码中，第 5、8、16 行的 setp 指令就是根据运行时的实际结果来设置 p1、p2、p3 三个谓词寄存器。而在后续的代码中，如第 6、9、17 行的 bra 指令，可以在 p 或!p(p 取反)的指示下根据各自的谓词寄存器控制每个线程是否需要执行。

在这段嵌套分支的 PTX 代码中，第 1 行～第 4 行是指令块 A 的计算部分，每个线程通过自己的线程号 tid 计算出各自的 t3 和 t4，准备比较。

第 5 行是一个比较操作，对应 CUDA 代码中第 6 行的比较。每个线程执行 setp 指令，将 t3 和 t4 的值进行比较。如果 t3 和 t4 相等，则该线程的谓词寄存器 p1 设为 1。注意这是每个线程独立的操作，所以不同线程的 p1 值可能会不同。根据图 3-5 中的假设，只有第 4 个线程的谓词寄存器 p1 被设置为 1。

第 6 行，标记有@p1 的指令表示每个线程在执行该指令前，需要先检测谓词寄存器 p1 中的值。如果为 1 则执行 bra，跳转至 F 块中，否则不跳转继续执行第 7 行 B 块的指令。由于只有第 4 个线程的谓词寄存器 p1 为 1，所以只有该线程将跳转到 F，发生线程分支。

第 8 行，每个线程执行 setp 指令，将 t4 和 t5 的值进行对比，对应 CUDA 代码中第 8 行的比较。如果 t4 和 t5 相等，则将该线程的谓词寄存器 p2 设为 1。根据图 3-5 可知，线程 2、3 将设置谓词寄存器 p2 为 1。

第 9 行，标记@p2 的指令执行前会检查谓词寄存器 p2 的值，并根据检查结果选择执行 D 块或继续执行 C 块。这里线程 2、3 将执行 D 块。

第 11 行，bra 指令使得执行完 C 块的线程将无条件跳转至 E 块，而之前跳转至 D 块的

线程也会顺序执行到E块，这样执行了C块和D块的前3个线程会在执行E块时发生线程重聚(reconverge)。

第13行，bra指令会使E块重聚的线程无条件跳转至G块，与执行F块的线程重聚。

第16行，所有的线程都需要执行，setp指令会对比i和N的值，若i小于N则设置谓词寄存器p3为1。

第17行，标记@p3的指令会检查p3的值，判断所有线程是否需要跳转回A块，继续执行循环操作。

可以看到，当线程束内部的不同线程出现分叉时，带有谓词标记的指令会根据谓词寄存器中的0或1值产生不同的执行路径，从而能够使得不同线程独立地开启和关闭，此时多个个线程的执行也就不再整齐划一。

另外，对于条件分支的执行效率问题，如果是if…then…else这种对称分支结构且两个分支路径的长度相等，那么SIMT的执行效率降低为50%。同理，对于双重嵌套分支结构，如果路径长度相等，那么SIMT的执行效率就为25%。这意味着大多数SIMT单元在执行嵌套分支时是空闲的，执行效率大幅降低。因此，线程分支是GPGPU性能损失的一个重要因素。

3.3.2 SIMT堆栈

当代码发生分支时，谓词寄存器决定了每个线程是否应该被独立地开启或关闭。从整体来看，GPGPU的线程调度器会对线程束的多个线程进行管理，保证具有相同路径的线程能够聚集在一起执行，从而尽可能地维持SIMT的执行效率。为此，GPGPU采用了一种称为SIMT堆栈(SIMT stack)的结构。它可以根据每个线程的谓词寄存器形成线程束的活跃掩码(active mask)信息，帮助调度器来确定哪些线程应该开启或关闭，从而实现分支线程的管理。

正如图3-5中看到的那样，代码块后面的编码代表了线程束的活跃掩码信息。起初所有线程都会执行A/1111。当遭遇了第6行的bra指令会产生分叉，线程不再整齐划一，形成了B/1110和F/0001两条互斥的路径，直到G/1111处再恢复到整齐划一的状态。这里的A称为线程分叉点(devergent point)，G称为分叉线程的重聚点(reconvergent point)。如果存在嵌套分支的代码，会使得已分叉的线程进一步分叉，如B/1110遭遇了第9行的bra再次分叉，形成了C/1000和D/0110的路径，直到E/1110处再重聚恢复B/1110的状态。

随着周期的推进和不同线程束代码的调度和执行，活跃掩码也需要随之不断地更新。从上面的例子可以看到，识别线程的分叉点和重聚点是管理活跃掩码的关键。一种思路是，当线程发生分叉时，记录下重聚点的位置和当前的活跃掩码，然后进入分叉，根据分支判断的结果执行其中一些线程(如true路径上的线程)，直到一条分支路径执行完成后切换到余下的线程(如false路径上的线程)执行。当所有路径的线程都执行完毕后，分叉的线程就可以在重聚点处恢复之前的活跃掩码，继续执行下面的指令。

SIMT 堆栈实现了对活跃掩码的管理。SIMT 堆栈本质上仍是一个栈，栈内条目的进出以压栈和出栈的方式进行，栈顶指针(top-of-stack，TOS)始终指向栈最顶端的条目。每个条目包含以下三个字段。

(1) 分支重聚点的 PC(Reconvergence PC，RPC)，PC 值独一无二的特性刚好可以用来识别重聚点的位置。RPC 的值由最早的重聚点指令 PC 确定，因此称为直接后继重聚点(Immediate Post-DOMinate reconvergence point，IPDOM)。在图 3-5 的例子中，代码块 B 执行完毕后，三个线程经由两条分支路径 C 和 D 在 E 处重聚，我们就称 E(确切来说，是代码块 E 的第一条指令)为一个 IPDOM。同样，E 和 F 的重聚点为 G 的第一条指令。

(2) 下一条需要被执行指令的 PC(Next PC，NPC)，为该分支内需要执行的指令 PC。

(3) 线程活跃掩码(Active Mask)，代表了这条指令的活跃掩码。

这里借助图 3-5 的例子来详细解释 SIMT 堆栈对活动掩码的管理方式。随着时钟周期的推进，线程的执行过程如图 3-6(a)所示。实体箭头代表对应的线程被唤醒，空心箭头代表对应的线程未被唤醒，每个代码块内线程分支情况保持一致。SIMT 堆栈通过选择不同的分支路径执行完所有的指令，如图 3-6(b)～图 3-6(d)的过程，最终所有线程都会恢复到共同执行的状态。初始时如图 3-6(b)所示，所有线程(活跃掩码为 1111)执行指令块 A 时，NPC 为指令块 G 的第一条指令 PC，即后面所有线程的重聚点。当到达了 A 的最后一条指令(PTX 代码第 6 行)时，由于指令块 A 产生了分支，RPC 应更新为当前指令块的 NPC，即 G 的第一条指令 PC。此后线程分为两个互补的执行路径，前三个线程将执行指令块 B(活跃掩码为 1110)，而最后一个线程将执行指令块 F(活跃掩码为 0001)。SIMT 堆栈会将指令块 B 和 F 及它们的活跃掩码压入栈中，并记录 B 和 F 的 RPC 为 G。

当前线程束需要执行的指令将从 TOS 条目的 NPC 获得。在本例中，会弹出指令块 B 的第一条指令(第 7 行)，其活跃掩码字段给出 1110 来控制内部线程的执行，同时 B 的 NPC 为 E 压栈，如图 3-6(c)中步骤(i)所示。当到达指令块 B 的结尾(第 9 行)时，这三个线程再次遭遇条件分支，硬件会采取类似的操作来更新 SIMT 堆栈：首先将 RPC 更新为当前指令块 B 的 NPC，即 E 的第一条指令。然后 B 的两个分支路径，即 C 和 D 及它们的活跃掩码会被压入栈中，同时标记其 RPC 为 E 的第一条指令，如图 3-6(c)中步骤(ii)和(iii)所示。

当前线程束会从 TOS 条目中选取接下来要执行的指令(块)，本例为指令块 C 且活跃掩码为 1000。当这个唯一的活跃线程到达指令块 C 的最后一条指令(第 11 行)时，其目标跳转 PC 与 RPC 相同，为指令块 E，所以 SIMT 堆栈会将 C 弹栈。接下来，当前线程束会再次从 TOS 条目选取指令块 D 且活跃掩码为 0110。当 D 执行完成后，其 NPC 与 RPC 相同，为指令块 E，所以 SIMT 堆栈会将 D 弹栈。此时，SIMT 堆栈更新为图 3-6(d)的状态。上述过程就是 SIMT 堆栈对活跃掩码的管理过程，保证了分支代码的正确性，还可以很好地应对嵌套分支的情况。

为了实现 SIMT 堆栈中如压栈、出栈的操作，一种方法是引入压栈、求反和恢复等专门指令针对 SIMT 堆栈进行操作，并通过编译器在 PTX 代码合适的位置插入这些指令，实现对活跃掩码的管理。GPGPU 则普遍采用硬件 SIMT 堆栈的方式提高线程分支的执行效

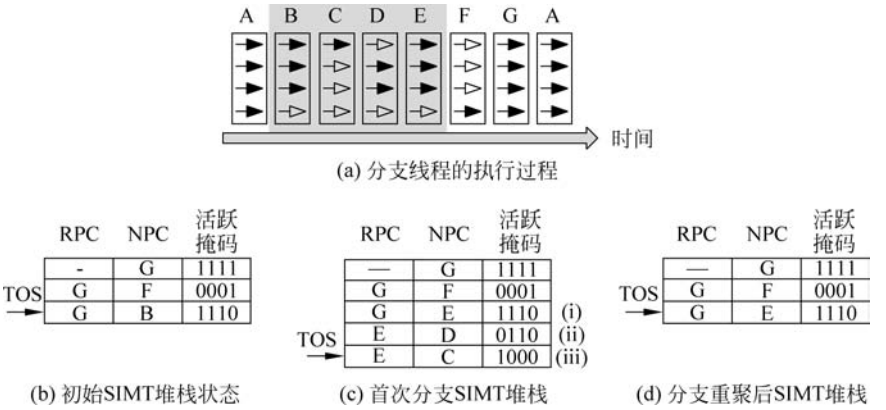


图 3-6 SIMT 堆栈实现对图 3-5 例子的管理

率。例如可以根据线程束中各个线程的执行情况动态地避免无效分支的执行，当所有线程都选择一个分支方向时，另一个方向的活跃掩码全为 0 便可以省略对应的分支路径，而不必以空流水的方式执行，提高执行效率。

3.3.3 分支屏障

基于 SIMT 堆栈的线程分支管理方式简单高效，但在特殊情况下可能会存在功能和效率上的问题，例如文献[5]就指出在原子操作下，SIMT 堆栈可能会产生线程死锁的问题。本节将结合文献[5-7]来具体分析这一问题，并讨论利用分支屏障和 Yield 指令解决这个问题的方法。

1. SIMT 堆栈可能的死锁

图 3-7 展示了一个 SIMT 堆栈可能会产生死锁的代码示例。

```
A: *mutex = 0
B: while( atomicCAS(mutex, 0, 1));
C: // critical section
   atomicExch(mutex, 0);
```

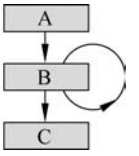


图 3-7 一个 SIMT 堆栈可能产生死锁的例子

这段代码中，代码块 A 首先初始化了一个公共锁变量 mutex，它可以被线程束内所有线程读取和修改。在 B 块中，每个线程试图对 mutex 执行 atomicCAS 操作，读取 mutex 的值并和 0 进行比较，如果两者相等，那么 mutex 的值将和第三个参数 1 进行交换，设置 mutex 的值为 1。该函数的返回值是 mutex 未交换之前的值。由于 atomicCAS 是一个原子操作，同一个线程束内多个线程需要串行化地访问存储器中的 mutex 锁变量，这也就意味着只有一个线程可以看到 mutex 的 0 值，进而获得锁退出循环执行 C 操作，而其他线程都只能看到 1 值而不断地循环。在块 C 中，获得锁的线程执行关键区操作，然后再通过

atomicExch 原子交换将 mutex 赋值为 0，即由获得 mutex 锁变量的线程来释放这把锁。

同时，考虑 B 中 SIMT 堆栈的执行过程。图 3-8(a)显示了该线程束执行 A 操作时 SIMT 堆栈的状态，其活跃掩码为 1111。当 B 执行完 atomicCAS 操作并返回后，B 中线程发生了分支，其重聚点为 C。由于 B 操作是原子的，在线程束内只有一个线程能够离开循环。假设只有第一个线程获得了锁变量而退出循环，那么 C 的活跃掩码为 1000，而 B 的活跃掩码为 0111。根据前文 SIMT 堆栈的描述，需要将 C 和 B 及各自的活跃掩码压入 SIMT 堆栈，此时 SIMT 堆栈的状态可能如图 3-8(b)所示。那么只有等待后三个线程先完成 B 对应的指令，才能去执行 C。但 B 是一个死循环，需要等待 C 指令完成后释放锁才能脱离循环，因此这里将产生死锁。

(a) 执行A时SIMT堆栈状态　　(b) 分支发生后SIMT堆栈状态

图 3-8　SIMT 堆栈发生死锁的具体过程

2. 分支屏障和 Yield 指令

针对 SIMT 堆栈管理线程束分支时存在死锁的问题，文献[7]提出了一种利用分支屏障和 Yield 指令来解决死锁的方式。相比于 SIMT 堆栈，这种方式允许屏障中的某些线程进入让步状态，从而允许其他线程先能够通过屏障执行下面的指令，避免死锁。

为此，分支屏障专门设计了增加屏障和等待屏障指令，例如 ADD 和 WAIT，并保存必要的信息使得分支屏障能够实现类似于 SIMT 堆栈的功能。当程序开始进入分支的时候，编译器会插入 ADD 指令来产生一个屏障，线程执行 ADD 指令时会与给定编号的屏障绑定而进入屏障。进入屏障的线程会沿着一条分支执行程序，直到到达 WAIT 指令时等待，等到所有绑定了这个屏障的线程到达这个 WAIT 指令屏障才能解除。线程重新进入活跃状态，以 SIMT 方式执行接下来的指令。值得注意的是，每个线程束内可能会有多个分支屏障，参与分支屏障的线程执行的分支也不相同。

仍然采用图 3-5 中的例子，但利用分支屏障实现条件分支的管理。在本例中，在块 A 和 B 中增加了专门的 ADD 指令用来初始化分支屏障。此时线程束内所有的活跃线程，会根据 ADD 指令修改自己对应的屏障参与掩码，以确定哪些线程会进入哪个分支屏障中，如 A 块的分支屏障为 B0，B 块的分支屏障为 B1。进入分支时，线程调度器会选择一组线程执行。对应于 ADD，WAIT 指令用于分支屏障内部的线程相互等待，一般存在于分支重聚点处，比如块 E 和 G。执行到 WAIT 的线程会修改线程状态，表明线程已经挂起。一旦屏障参与掩码中的所有线程都执行了相应的 WAIT 指令，线程调度器就可以将分支屏障中的线程切换到活跃状态。利用分支屏障管理图 3-5 中条件分支的例子如图 3-9 所示。

针对 SIMT 堆栈可能出现的死锁问题，分支屏障还设计了 Yield 指令，使得某些线程可以进入让步状态。进入让步状态的线程会退出占用的资源暂缓执行，其他分支路径的线程

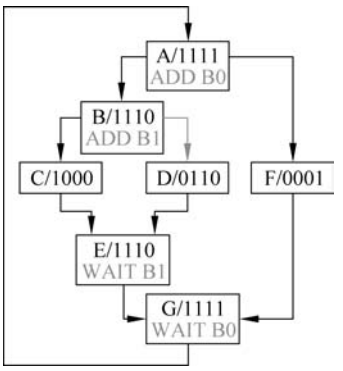


图 3-9 利用分支屏障管理图 3-5 中条件分支的例子

也无须在屏障处等待那些已经进入让步状态的线程。在具体实现中，可以采用不同的方式来判定分支屏障中的线程是否需要进入让步状态。

(1) 执行了编译器在分支路径中显式插入的 Yield 指令。

(2) 运行超时或执行了某一固定次数的跳回操作，即从数值较大的 PC 跳回数值较小的 PC。这相当于硬件会判定跳回操作超过某一数值后，认为线程执行出现了死循环。这正是本节例子中出现的情况。

基于分支屏障实现的线程调度器可以自由切换不同的线程执行，而无须按照 SIMT 堆栈的方式提取栈顶的线程束执行。配合 Yield 状态的线程不再执行，这样可以解锁其他线程，从而避免死锁。

3. 死锁问题的解决

下面介绍如何利用新的分支屏障和 Yield 指令来解决图 3-7 中所示的死锁问题。如图 3-10 所示，由于块 A 是分支的开始，C 为重聚点，因此添加分支屏障的指令应该在 A 中，而 C 中第一条指令应为对应的 WAIT 指令。图 3-10(a)中插入了分支屏障指令实现了与 SIMT 类似的分支管理。由于没有 Yield 指令，分支屏障依然要求屏障前的所有分叉线程都重聚到屏障后才能继续执行，因此第一个线程会一直等待在屏障处，从而无法执行 C 块代码，也无法释放锁资源。图 3-10(b)中块 B 插入了 Yield 指令来避免死锁的发生。Yield 指令会让 B 中的部分线程让步而放弃执行循环，从而使第一个线程能够不等待 B 中的其他线程，跨越分支屏障而先执行 C，释放锁资源。然后让步状态会解除，让 B/0111 的一个线程再次得到锁而退出循环。最终，B 中所有线程都会走出死循环，不会再发生死锁，完成代码的既定功能。

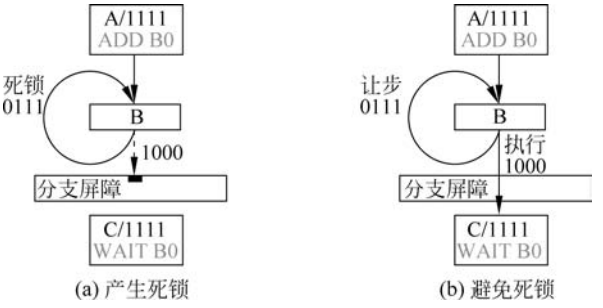


图 3-10 采用分支屏障和 Yield 指令避免死锁

为了便于理解，下面详细分析线程在刚进入循环及第一次解除死锁的过程，如图 3-11 所示，图 3-11(a)为马上进入循环时分支屏障的状态。由于 4 个线程都会参与屏障，因此屏

障参与掩码是不变的。而屏障状态为 4 比特，每一位代表一个线程，若线程已经执行到了屏障，则对应位置标为 1，否则为 0。由于还没有执行 B 中程序，因此图 3-11(a)中屏障状态为 0000。线程状态有 3 个，其中 00 为就绪状态，01 为挂起状态，10 为让步状态。线程 RPC 为接下来要执行的指令，每个线程对应一个 RPC。此时，所有线程都是活跃的，准备进入循环 B 中。

屏障参与掩码	屏障状态	线程状态	线程RPC	线程活跃
1111	0000	00 00 00 00	BBBB	1111

(a) 马上进入循环

屏障参与掩码	屏障状态	线程状态	线程RPC	线程活跃
1111	1000	01 00 00 00	CBBB	0111

(b) 第一次执行完循环

屏障参与掩码	屏障状态	线程状态	线程RPC	线程活跃
1111	1000	00 10 10 10	CBBB	1000

(c) 线程进入让步状态

屏障参与掩码	屏障状态	线程状态	线程RPC	线程活跃
1111	1100	01 01 00 00	CCBB	0011

(d) 第一次摆脱死锁

图 3-11　采用 Yield 指令避免死锁的具体过程

图 3-11(b)为线程执行完一次循环后的结果。此时，其中一个线程可以开始执行 C 中的指令，即到达了屏障，而其他的线程必须再次执行循环。假设第一个线程到达了屏障，它由于 WAIT 指令被挂起，其他线程继续执行循环。因此第一个线程的 RPC 被调度器修改为 C，并且状态转为非活跃，等待其他线程一起执行 C 中指令。而其他线程仍然活跃，等待执行 B 中指令。

图 3-11(c)为线程进入让步状态。根据第二个触发条件，线程执行 Yield 指令。这时，三个线程进入让步状态(即 10)，第一个线程进入活跃状态(即 00)，因此第一个线程不需要等待其他三个线程就可以穿过屏障执行 C 中的指令。

图 3-11(d)为第一次离开死锁。由于第一个线程执行了指令 C，可以释放 B 循环三个线程中的一个进入屏障。假设第二个线程被释放出来，到达了屏障，这样前两个线程被挂起，而剩下两个线程继续执行 B 循环并适时做出让步。重复图 3-11(c)和图 3-11(d)，最终所有线程都能走出死锁。

代码 3-2 给出了图 3-7 示例在 NVIDIA Volta 架构下所对应的 SASS 代码，其中第 1 行～第 6 行显示了块 A 的代码，第 7 行～第 11 行显示了块 B 的代码，第 12 行～第 14 行显示了块 C 的代码。第 3 行的 BSSY 指令可以认为是增加分支屏障中的 ADD 指令，这条指令增加了一个屏障 B0。第 12 行的 BSYNC 即为分支屏障中的 WAIT 指令，表示线程必须在 B0 屏障中等待其他分支的线程执行完毕后，才能一起继续执行。第 8 行 Yield 指令是 Volta 架构新增加的代码，使得执行 B 中的线程进入让步状态，防止死锁。

代码 3-2　SASS 代码中采用 Yield 指令避免死锁的示例

```
1    /*0020*/          STS [RZ], RZ;
2    /*0030*/          BMOV.32.CLEAR RZ, B0;
3    /*0040*/          BSSY B0, 0xe0;
4    /*0050*/          MOV R3, 0x1;
5    /*0060*/          NOP;
6    /*0070*/          BAR.sync 0x0;
7    /*0080*/          IMAD.MOV.U32 R2, RZ, RZ, RZ;
8    /*0090*/          YIELD;
9    /*00a0*/          ATOMS.CAS R0, [RZ], R2, R3;
10   /*00b0*/          ISETP.NE.AND P0, PT, R0, RZ, PT;
11   /*00c0*/          @!P0 BAR 0x80;
12   /*00d0*/          BSYNC B0;
13   /*00e0*/          ATOMS.EXCH RZ, [RZ], RZ;
14   /*00f0*/          EXIT;
```

3.3.4　扩展讨论：更高效的线程分支执行

从前文的介绍可以看到，GPGPU 架构支持条件分支的基本思想就是串行执行发生分叉的线程，但这种方式会损失 SIMT 硬件的执行效率，成为影响 GPGPU 性能的重要因素之一。单纯的 SIMT 堆栈管理方式虽然基本保证了分支执行的正确性，但在某些情况下的 IPC 并不能达到最优。

为了提高线程分支执行的效率，通过分析线程分支的执行过程可以发现，架构设计者可以从以下两个来角度来进行优化。

（1）寻找更早的分支重聚点，从而尽早让分叉的线程重新回到 SIMT 执行状态，减少线程在分叉状态下存续的时间。实际上，前面提到的直接后继重聚点（IPDOM）是一种直观的重聚点位置。它以两条分支路径再次合并的位置作为重聚点，符合对称分支代码的结构，但在多样的分支代码结构下未必是最优的重聚点选择方案。

（2）积极地实施分支线程的动态重组和合并，这样即便线程仍然处在分叉状态，能够让更多分叉的线程一起执行来提高 SIMT 硬件的利用率。例如，将不同分支路径但相同的指令进行重组合并就可以改善分支程序的执行效率。但这往往需要打破原有线程束的静态构造等限制，需要微架构的支持。

为了提高线程分支的执行效率，研究人员基于以上两种思想开展了广泛的研究。本节将挑选其中具有代表性的技术和方法进行介绍，深入理解 GPGPU 架构设计的权衡和考量。

1. 分支重聚点的选择

重聚点的选择有利于让线程尽早脱离分支状态，恢复到 SIMT 执行状态，但重聚点的选择会根据代码结构的不同而有所不同。本节将介绍一种不同于 IPDOM 的分支重聚点。

程序控制流可分为结构化控制流和非结构化控制流。诸如顺序执行的基本块、条件分支和循环，如 if…then…else、for 循环、do…while 循环等，被称为结构化控制流；而 goto、break、短路优化、长跳转和异常检测等被称为非结构化控制流。这里的短路优化是指布尔运算中只有当第一个参数不能确定表达式的值时，才会执行或评估第二个参数。例如，在与(AND)逻辑中，如果第一个参数为 false 则无须判断后面的参数，表达式结果必然为 false；在或(OR)逻辑中，如果第一个参数为 true 则无须判断后面的参数，表达式结果必然为 true。编译器在为复合的布尔逻辑生成代码时，有时会利用短路优化尽快地给出条件判断的结果，确定分支路径。

对于非结构化控制流，常常存在早于 IPDOM 的局部重聚点，可以提前对部分线程进行重聚，从而提高 SIMT 硬件的资源利用率，改善程序的执行效率。为便于理解，图 3-12(a)给出了一段由复合布尔运算构成的控制流程序及其分支流图。由于编译器使用了短路优化，处理器无须完全执行 4 个条件判断，因此至多存在 7 条不同的控制路径，如图 3-12(b.1)所示。考虑一种最糟糕的情况，假设一个线程束包含了 7 个线程且运行时分别选择了 7 种控制路径。如果使用前面介绍的 SIMT 堆栈方式，可能会出现图 3-12(b.2)的情况，其中横轴表示各个线程，纵轴表示时间，灰色方块表示执行单元处于停顿状态。从图 3-12(b.2)中可以看到，同一个基本块的不同线程会被安排在不同的周期执行，例如 B3、B4 和 B5 的线程在多个执行路径下被多次拆分执行，使得程序执行效率十分低下。

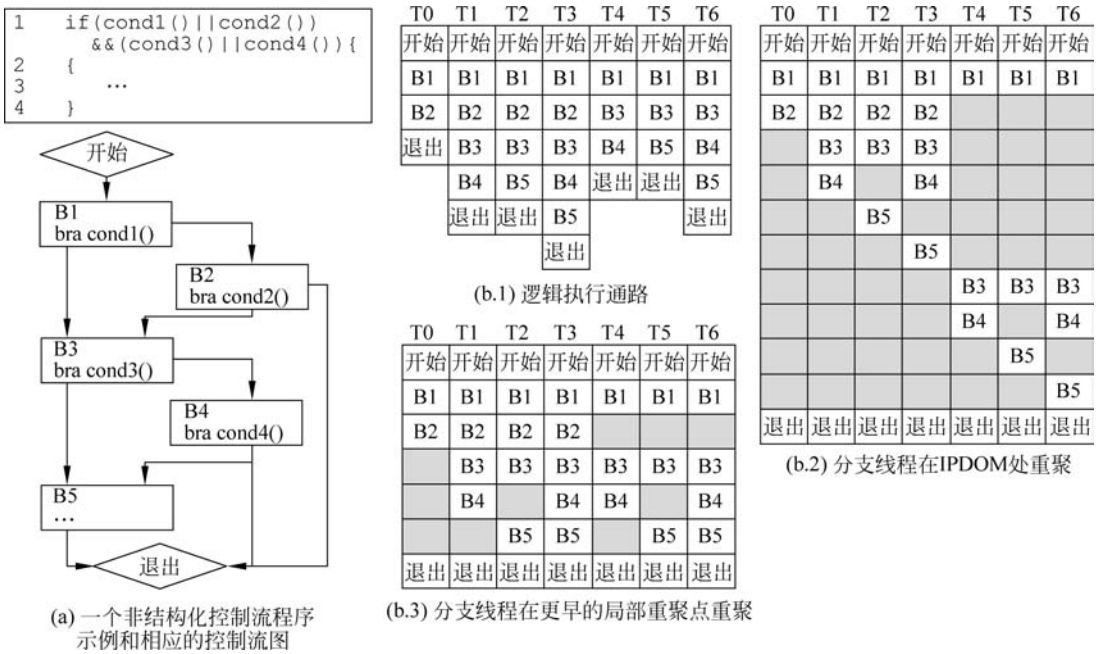
针对这个问题，如果能让 T1～T3 线程在执行 B3 时尽早地与互补路径的 T4～T6 线程执行 B3 重聚，那么将有效地提升并行度，如图 3-12(b.3)所示。但现有 SIMT 堆栈下无法实现这样的控制，原因是 B1 块的分支路径会将 B3(T4～T6)和 B2(T1～T3)压栈。一旦选择执行 B2 块，就需要将 B2 块后面的分支 B3、B4、B5 完全压栈和出栈后，才能将 B2 退栈回到 B1 块分支的另一个路径 B3(T4～T6)来执行。因此，为了能够这样执行就需要不同于 SIMT 堆栈的管理方式，利用不同于 IPDOM 的局部重聚点来发现这个可能性。在文献[8]中，将这种新的局部重聚点称之为 TF(Thread Frontiers)，并且给出了一种基于编译器和硬件协同管理 TF 的机制。

TF 可理解为在任一时间点，分叉的线程可能执行的所有基本块。换句话说，就是当一部分线程进入一个分支执行某个基本块时，其他线程可能等待执行另一个分支的基本块即为该基本块的 TF。比如，当线程 T1～T3 将要执行 B3 时，非活跃的 T4～T6 也可能会执行 B3。由于 T1～T3 和 T4～T6 可以同时执行 B3，因此可以形成一个 TF 重聚点，两个线程分片可以合并执行。为了实现线程在 TF 重聚，需要两种支持。

(1) 当程序出现分支时，非活跃线程需要在活跃线程的 TF 中等待。比如，在 T1～T3 将要执行 B3 时，非活跃线程 T4～T6 在 B3 的 TF 中等待。

(2) 如果部分线程进入 TF，需要进行重聚合检查判断是否有线程可以合并。比如，当 T1～T3 进入 B3 时，发现 B3 包含在其 TF 中，这时需要进行重聚合检查，即检查 T1～T3 中执行 B3 的线程和等待在 TF 中 B3 的线程 T4～T6 能否合并。

这些功能可以由编译器和硬件调度器共同完成。编译器通过算法分析出每个基本块的



(a) 一个非结构化控制流程序示例和相应的控制流图

(b.1) 逻辑执行通路

(b.2) 分支线程在IPDOM处重聚

(b.3) 分支线程在更早的局部重聚点重聚

图 3-12　利用 TF 重聚点对非结构化分支进行优化的示例

TF 信息，并且在适当的位置插入线程重聚合检查。编译器还需要为每个基本块分配一个优先级，帮助硬件按照指定的优先级顺序对线程进行调度，以最大化 TF 合并的可能。在具体的实现中，可以使用 PC 值作为优先级判断的依据：PC 值越小的指令，执行的优先级越高。对于硬件调度器，保证线程按照优先级顺序执行基本块，同时在执行期间遇到编译器放置的重聚合检查时，在可能的情况下进行线程合并。例如，在本例中基本块的优先级顺序应为(B1、B2、B3、B4、B5、Exit)，调度器按照这种优先级顺序执行基本块，有利于 T0～T3 在执行完 B2 之后，T1～T3 与 T4～T6 尽早合并。

2. 线程动态重组及合并

提高线程分支执行效率的另一种方式就是打破原有静态线程束的限制，对特定互补的线程分片进行重组及合并，以便在分支存续期间提高 SIMT 硬件的利用率。这个优化可以在 IPDOM 重聚点下进行，也可以在局部重聚点如 TF 下进行。

根据 SIMT 线程分支的特点，可以从不同线程和不同 PC 所构成的多个维度进行动态重组和合并。如图 3-13 所示，假设有 8 个线程(T0～T7)，分成 2 个线程束 W0 和 W1，运行时可能分别执行不同分支的指令，那么不同线程束中相同 PC 的指令很可能存在互补，例如，如果 W0 中的 T0/T1 和 W1 中的 T6/T7 都执行到了 true 分支路径，就可以在相同的 PC 值处进行互补合并。这种可能性主要来源于 GPGPU 中同时存在大量线程，大概率有不同的线程在执行同样的分支路径。然而其难点在于这些线程可能跨度很大，未必在同一个线程束内，也可能这些线程在特定 SIMT 通路上存在冲突。

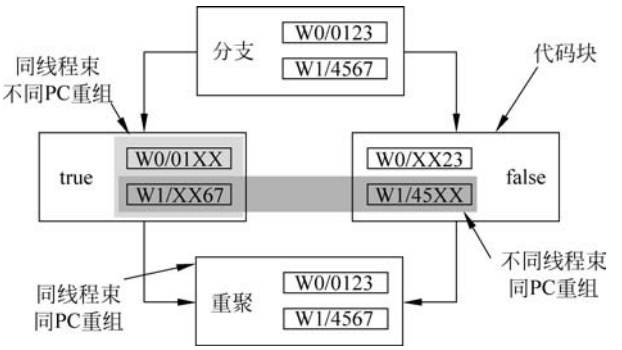


图 3-13　分支线程在多个维度进行重组和合并的可能性

另一种可能性是相同的线程束在不同的 PC 处进行重组和合并，例如 W1 在 true 路径中的 T6/T7 和在 false 路径中的 T4/T5。这种可能性主要来源于同一线程束在不同分支路径中的线程往往存在互补性，即发生分支的 W1 在 true 路径中的线程必然和 false 路径中的线程存在互补性。然而其难点在于同一时刻需要发射执行不同的指令，一定程度上呈现出 MIMD 执行的特性。

根据图 3-13 分支线程在多个维度进行重组和合并的可能性，本节将探讨四个维度下的线程重组和合并技术，即同线程束同 PC 合并、不同线程束同 PC 合并、同线程束不同 PC 合并和不同线程束不同 PC 合并。

1）同线程束同 PC 线程的重组和合并

实际上，前文介绍的 IPDOM 重聚点可以看成是相同线程束同 PC 合并。如图 3-13 所示，当部分线程先到达重聚点，还需要等待同线程束内其他线程也执行到重聚点才能完成分支，相当于相同线程束内的线程在相同的 PC 处，即重聚点 PC 处合并。但是这种合并方式并不能有效利用 SIMT 通道，因为重聚点前的分支路径不具有相同 PC，有必要打破线程束和 PC 的限制，提升程序的整体运行效率。

2）不同线程束同 PC 线程的重组和合并

不同线程束同 PC 线程可以重组和合并。理想情况下，多个相同路径的线程可以重组为一个更为“完整”的线程束来执行。这种合并的好处是不会破坏 SIMT 执行，而更多体现在调度方面的设计。这里以图 3-14(a)所示的线程分支流图为例，8 个执行 if…else…分支的线程 T0～T8 分别组织在 2 个线程束 W0 和 W1 中。图 3-14(b)显示了基于 SIMT 堆栈的分支执行过程，W0 和 W1 的分支线程在代码块 B、C 中串行执行。图 3-14(c)显示了线程进行“原位合并”后的结果，其中 W0 的 T0 和 W1 的 T6/T7 合并执行，W0 的 T1/T2/T3 和 W1 的 T4 合并执行。不过由于 W0 的 T1 和 W1 的 T5 在合并时存在冲突，导致 W1 的 T5 需要单独执行。但即便如此，后者的执行时间仍然少于前者。

为了实现线程的重组合并，文献[8]提出了动态线程束原位合并的思路，并在硬件上设计了如图 3-15 中的 PC-Warp LUT(查找表)以建立 PC 和线程束之间的映射关系。它通过

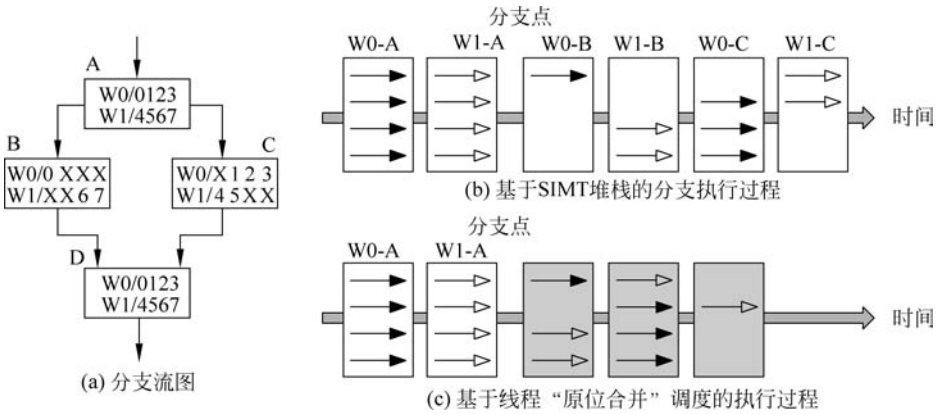


图 3-14　使用与未使用不同线程束同 PC 的线程合并的执行过程对比

哈希运算 H 为相同 PC 值的不同线程匹配到一个 PC-Warp LUT 表项，然后根据“错位”或“原位”的规则尽可能与表项中已有的线程束合并以填充 SIMT 通道。为了平衡线程束产生和消耗存在的速率差，还在中间引入了一个线程池，重组完成的线程束会先进入池中等待。根据线程束优先级的高低，发射部件采取特定的机制选取池中的线程束进行调度，从而达到较高的 SIMT 通道利用率。

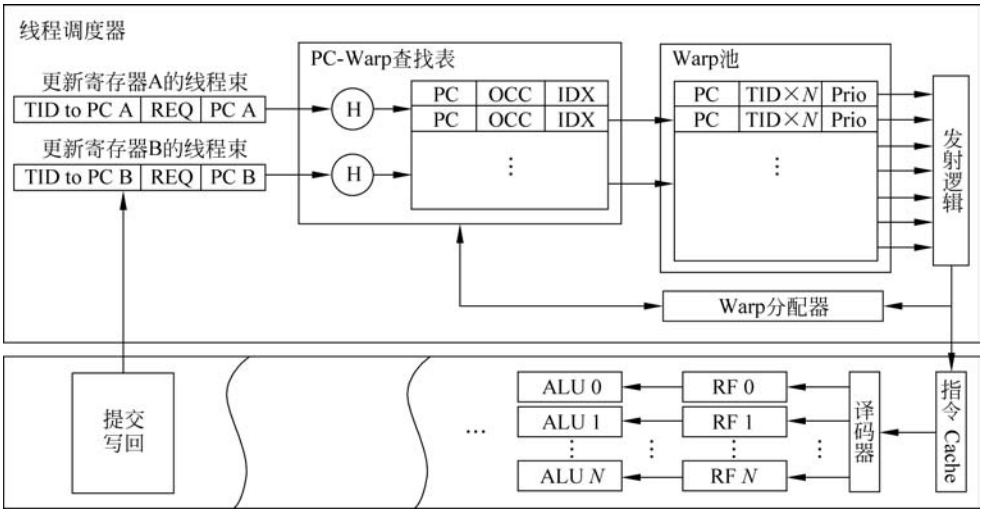


图 3-15　不同线程束同 PC 线程合并的硬件实现

为了支持“原位合并”，文献[9]提出了一种基于原有 SIMT 堆栈的实现方法，称为线程块压缩。为此调度器需要维护一个与 SIMT 堆栈类似的结构，如图 3-16 所示，栈中每项元素包含四个属性，其中 RPC 和 NPC 与 SIMT 堆栈中相同，活跃掩码相比于 SIMT 堆栈有所扩展，表示了线程块中的所有线程。而 WCnt 表明执行该指令的活跃线程束数量，说明线程块中有多少线程束已经准备好执行该指令。栈初始化如图 3-16(a)所示，线程块将要执行 A

指令。由于线程块内两个线程束都要执行 A，因此活跃掩码都被写入其中且 WCnt 设为 2。接下来如图 3-16(b)所示，第一个线程束 W0 执行完 A 后，其线程发生分支，不同的分支 C 和 B 被压入栈中。由于活跃的线程束数量减少，TOS 的 WCnt 的值减 1。由于 W0 需要等待 W1 执行完 A，因此栈顶指针 TOS 维持不变，B 和 C 中的 WCnt 的值为 0。如图 3-16(c)所示，当线程束 W1 也执行完 A 后，其分支被压入栈中，扩展了活跃掩码位并与 W0 的活跃掩码合并，TOS 将指向下一个需要执行的分支 B。此时线程 T0/T6/T7 可以合并执行，两个线程束可以合并成为一个，WCnt 的值为 1。当 B 被执行完毕后，C 和 D 也相继被弹出，如图 3-16(e)和图 3-16(f)所示，最终所有的指令都执行完毕。

图 3-16　线程块压缩技术下 SIMT 堆栈的更新过程

同样，为了实现不同线程束在同一 PC 处线程的重组合并，文献[10]提出了一种基于大线程束的线程管理和重组策略，期望可以发现更多线程合并的机会。每个大线程束由若干线程束的连续线程组成，当出现分支时，根据分支情况生成多个子线程束。为了实现这种管理，大线程束将其线程的活跃掩码统一组织为一个二维结构，如图 3-17(a)所示。矩阵的列数等于大线程束的宽度，每行表示子线程束的活跃掩码，重组时会尽可能从不同列选择活跃线程，因而更利于实现线程的“原位合并”，避免线程冲突现象。图 3-17(b)展示了在连续 4 个周期内从 1 个大线程束生成 4 个子线程束的过程。线程调度器每个周期从矩阵各列中搜索到 1 个活跃掩码并找到对应的线程将其加入子线程束中，然后清除对应掩码位。重复此过程，直到当前掩码矩阵中所有非 0 位被清空，标志着大线程束在当前分支路径中的线程已经处理完毕。

大线程束处理分支和重聚的方法与 SIMT 堆栈方式类似。在一个大线程束执行分支指

令时，只有当其最后一个子线程束执行完毕才能确定是否发生了分支。当所有分支子线程束完成执行后，一方面更新当前大线程束的 PC 值和活跃掩码，另一方面将重聚点 PC、活跃掩码和待执行 PC 等信息压入大线程束的 SIMT 堆栈中。每当一个分支执行完毕便将对应的分支项从栈顶弹出。这个大线程束间的调度应由更高阶的调度策略和调度器来决定。这种机制不仅能够解决单一分支问题，还能应对嵌套分支的情形。另外，对于无条件分支，如 jump 指令，大线程束的方式也适用。无条件分支指令只需要更新一次 PC 值，仅一次跳转即可使下一个大线程束提前开始执行，减少不必要的指令发射时间。

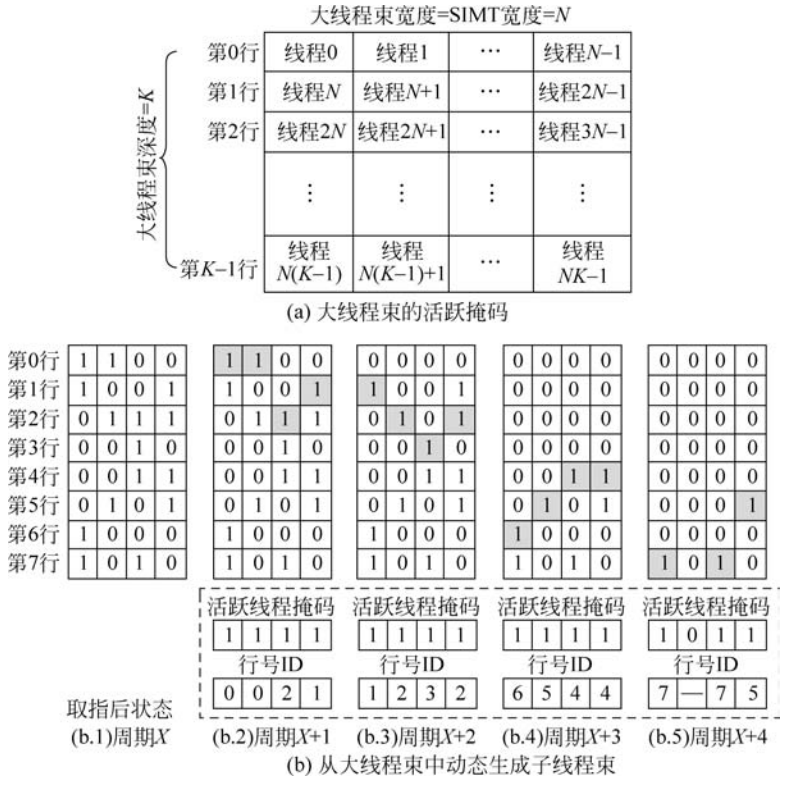


图 3-17 大线程束重组

本节介绍的动态线程束重组、线程块压缩和大线程束重组等方法，都属于跨线程束同 PC 线程的重组和合并方法，本质上是类似的，只是具体实现方法上有所不同。然而，这种重组和合并也可能影响到架构设计的其他方面，例如：

（1）这种方式并不总是能够减少线程束的数量而获得性能收益。以图 3-14(c)为例，执行 C 块时也试图进行线程合并，但线程束的数量并没有减少。

（2）当新线程束访问寄存器时，要避免线程错位访问或多个线程访问寄存器出现冲突，以便新线程束内的线程可以高效地获取寄存器文件中的数据。

（3）这种方式还会导致线程之间的同步问题，原则上不允许线程脱离线程块单独调度。

(4) 这种方式虽然能够提高 SIMT 通道的利用率,但也可能会导致更多的高速缓存缺失。如图 3-18 所示,W0 需要的数据在缓存中而 W1 的数据不在。如图 3-18(a)所示,如果没有线程合并只会有一个线程束发生缓存缺失。如图 3-18(b)所示,如果进行了线程合并可能导致两个合并后的线程束都发生缓存缺失。为解决这个问题还可以采用预测的方式,通过预测合并后是否会提升性能来避免不合理的压缩带来的性能影响。

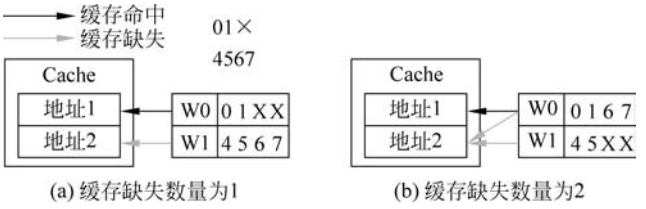


图 3-18　线程合并较未合并带来的负面影响

(5) 这种方式倾向使用轮询的调度策略(参见 3.4 节的内容),这样不同线程束的执行进展大致相同,利于在线程分支时找到同 PC 互补的线程。其他的线程束调度策略是否会破坏这种可能性或重组后的线程是否会影响到整体的调度,则需要进一步思考和研究。

3) 同线程束不同 PC 线程的重组和合并

与不同线程束同 PC 线程的重组和合并相对应,还可以在同线程束不同 PC 的维度进行线程合并。这种方式的可能性主要源于分支路径往往存在互补性,即发生线程分叉后,true 路径的线程必然和 false 路径的线程存在互补性。然而,相比于同线程束不同 PC 的合并,不同线程束同 PC 线程的合并难度更大,因为其从本质上改变了 SIMT 计算模型,不同通道上执行了不同的指令,这更加倾向于 MIMD 执行。

文献[12]提出了一种允许同一个线程束内互补线程的不同指令同时执行的方式,称为 Simultaneous Branch Interweaving(SBI)。它主要针对 if…else…这一对称结构的分支进行优化,因为执行 if 指令的线程和执行 else 指令的线程是互补的,不会产生冲突。SBI 允许 if 和 else 中的指令被一个线程束内的线程同时执行,提高 SIMT 通道的利用率。

以图 3-19(a)中的分支流图为例。假设两个线程束 W1 和 W2 各有 4 个线程。指令块 1～6 旁标注了线程束有哪些线程执行该指令块。图 3-19(b)～(d)对比了采用 SIMT 堆栈与 SBI 执行结果的区别。在 SIMT 堆栈下,如图 3-19(b)所示,一个调度器调度一个线程分片执行,而 SBI 则提供了一个副调度路径允许同线程束中互补线程的不同指令进入功能单元执行。如图 3-19(c)所示,主调度在为 W1 线程 T0/T3 调度指令块 I2 和 I3 的同时,副调度则为 W1 线程 T1/T2 调度指令 I5 和 I6。

为了实现这种重组的方式,需要修改原有的 GPGPU 中指令读取和调度器的结构。该研究基于经典的 NVIDIA Fermi 架构 SM 进行设计。图 3-20(a)显示了 Fermi 架构 SM 的基本结构,它包含两条独立的指令流水线可以同时调度两个线程束,所以可以将其中一个确定为主调度器,另一个为副调度器。副调度器接收主调度器的线程束 ID(Wid)来跟随发射某个线程束的指令。如图 3-20(b)所示,修改后的架构可以支持同线程束两条不同指令 I1

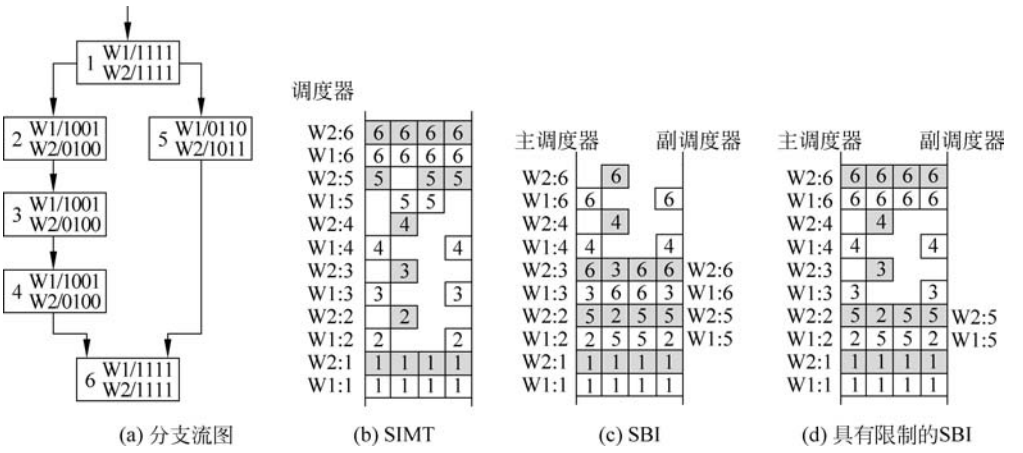


图 3-19 同线程束不同 PC 线程合并示例

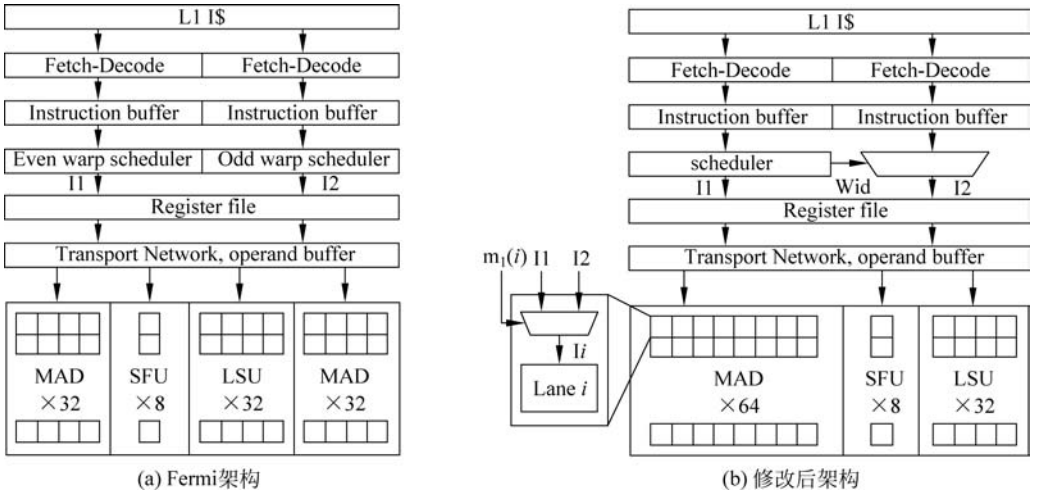


图 3-20 SBI 模式下指令调度器的结构

和 I2 的执行。为了使两条通道接收两条不同的指令，每个通道前设置了一个多路选择器，选择执行 I1 或 I2 中的一条。

选取两条指令流水线的 PC 是实现线程束内不同 PC 线程合并的关键。在执行阶段当线程分片进入了某个分支时，会计算互补的线程分片将执行哪些 PC(互补路径的 PC)，且这些 PC 的优先级会高于接下来指令的 PC，以便增加重组的可能性。比如当 W1/1001 进入 I2 时，W1 的互补线程 0110 执行 I5 的优先级会高于 W1 接下来要执行的 I3。

不过，SBI 也存在一定问题。比如，如果没有遇到资源冲突，两个线程分片之间会彼此不同步运行，这样可能会忽略原有的重聚点，推迟线程束恢复 SIMT 执行的时机。如图 3-19(c)所示，副调度器没有等待线程重聚就“提前”调度了 W1 的 I6 指令，造成 I6 处本

应重聚的线程被打乱。针对这一问题，可以对 SBI 进一步限制，不允许分支点和重聚点的指令参与线程重组，如图 3-19(d)所示。这可以通过记录分支指令的 PC_{div} 和重聚点指令的 PC_{rec} 来判断副调度器是否应该被开启。在该示例中，PC_{div} 为 I1，PC_{rec} 为 I6。当副调度器调度完 I5 到达指令 I6 时，如果主调度器还在调度 I2、I3 或 I4，副调度器可以调度。当主调度器开始调度 I6，副调度器停止调度，这样 I6 就不会被副调度器提前调度。

4) 不同线程束不同 PC 线程的重组和合并

上述介绍的同线程束不同 PC 线程的重组和合并主要针对互补的分支结构，比如执行 if…else…指令产生的分支时，两条分支路径的线程分片有机会重组为同一个线程束。但还有很多时候分支是不平衡的。比如只有 if 而没有 else 语句下的非结构化分支，或 if 和 else 分支的路径长度不对等，这会导致同一线程束中并没有互补的线程分片能够执行不同的 PC 指令来填充 SIMT 通道。为了解决这个问题，可以选取其他线程束中执行不同 PC 的线程来填充 SIMT 通道，这就是不同线程束不同 PC 线程进行重组和合并的思想。

文献[12]在 SBI 的基础上又提出了 SWI(Simultaneous Warp Interweaving)来解决这个问题。SWI 的硬件架构与 SBI 相同，如图 3-20(b)所示，可以看到每个 SIMT 通道上有一个多选器用于选择执行哪个调度器发射的指令。当一条指令 I 被发射之后，另一个调度器可以获取指令 I 对应的活跃掩码来寻找通道上不冲突的另一条指令 I′。这个 I′可以来源于同线程束内的指令即 SBI，也可以来源于不同线程束的指令即 SWI。图 3-21 展示了 SWI 是如何工作的。SWI 选择 W1 中的 I3 和 W2 中的 I2 同时执行，及 W1 中的 I4 和 W2 中的 I3 同时执行。同时，SWI 和 SBI 技术也并不冲突，两者也可能结合起来进一步提高 SIMT 通道的利用率来改善性能。不过 SWI 由于允许了 MIMD 执行会使得线程分支的硬件设计更为复杂。

图 3-21　不同线程束不同 PC 线程重组和合并示例

3.4　线程束调度

调度在计算机中是一个常见的概念。笼统地讲，调度是指分配工作所需资源的方法。在计算机中这个“资源”可以涵盖各种层次的资源，既可以是虚拟的计算资源，如线程、进程或数据流，也可以由硬件资源，如处理器、网络连接或 ALU 单元。调度的目的是使得所有资源都处于忙碌状态，从而允许多个工作可以有效地同时共享资源，或达到指定的服务质量。调度的工作可以由软件程序完成，称之为调度算法或策略；也可以由硬件单元来完成，称之为调度器。调度算法和调度器可能会针对不同目标而设计，例如，吞吐率最大化、响应时间最小化、最低延迟或最大化公平。这些目标在同一系统中往往是相互矛盾的，因此调度算法和调度器要实现一个权衡利弊的折中方案，这取决于用户的需求和目的。

CUDA 和 OpenCL 编程模型可以定义任意数量的线程块和线程，线程块会被分配到可编程多处理器上，由内部的流处理器提供线程并行度。但毕竟硬件资源是有限的，每个周期只能执行若干线程。当有多个线程处于就绪状态时，应该选取哪一个来执行呢？这其实就是一个调度问题。早期的 GPGPU 多采用轮询策略来保证调度的公平性。尽管这种策略简单可行，但很多时候执行效率并不高，因此人们提出了多种改进和优化的调度策略。本节将针对上述问题展开讨论。

3.4.1 线程束并行、调度与发射

在编程人员看来，线程是按照线程块指定的配置规模来组织和执行的。从硬件角度来看，当一个线程块被分配给一个可编程多处理器后，GPGPU 会根据线程的编号（TID），将若干相邻编号的线程组织成线程束。线程束中所有线程按照锁步方式执行，所有线程的执行进度是一致的，因此一个线程束可以共享一个 PC。线程束中每个线程按照自己线程的 TID 和标量寄存器的内容来处理不同的数据。多个线程聚集在一起就等价于向量操作，多个线程的标量寄存器聚集在一起就等价于向量寄存器，向量宽度即为线程束大小。如同 2.2.3 节的分析，这种基于线程 TID 的向量构造方式与传统的 SIMD 不同，它不需要编程人员的参与，因此可以看成是基于硬件的隐式 SIMD 或向量化。这种方式提供了相当的灵活性，例如线程块可以配置为 256×1、16×16 等多种维度，硬件都会自动地构造出线程束来对线程块进行切分并执行。

大量的线程束提供了高度的并行性，使得 GPGPU 可以借助零开销的线程束切换来掩藏如缓存缺失等长延时操作。原则上线程束越多，并行度越高，延时掩藏的效果可能会越好。但实际上这个并行度是由一个可编程多处理器中可用的硬件资源及每个线程的资源需求决定的，如最大线程数、最大线程块数及寄存器和共享存储器的容量。例如，在 NVIDIA V100 GPGPU 中，一个可编程多处理器最多同时执行 2048 个线程，即 64 个线程束或 32 个线程块，并为这些线程提供了 65 536 个寄存器和最多 96KB 的共享存储器。如果一个内核函数使用了 2048 个线程且每个线程使用超过 32 个寄存器，那么就会超过一个可编程多处理器内部寄存器数量；如果每个线程束占用的共享存储器超过 1536B，那么共享存储器的资源无法支撑足够多的线程束在可编程多处理器中执行。最终执行时可达到的线程并行度是由线程块、线程、寄存器和共享存储器中允许的最小并行度决定的。由于并不是所有资源都能够同时达到满载，因此对于非瓶颈的资源来说存在一定的浪费。

当可编程多处理器中有众多线程束且处于就绪态（或活跃）时，需要调度器从其中挑选出一个。这个被选中的线程束会在接下来的执行周期中根据它的 PC 发射出一条新的指令来执行。从整个可编程多处理器角度看，由于调度器每个周期都可以切换它所选择的线程束，不同线程束的不同指令可能会细粒度地交织在一起，而同一个线程束的指令则是顺序执行的，如图 3-22 所示。调度器需要根据 GPGPU 的架构特点设计合适的策略来做出这个选择，尽可能保证 SIMT 执行单元不会空闲。

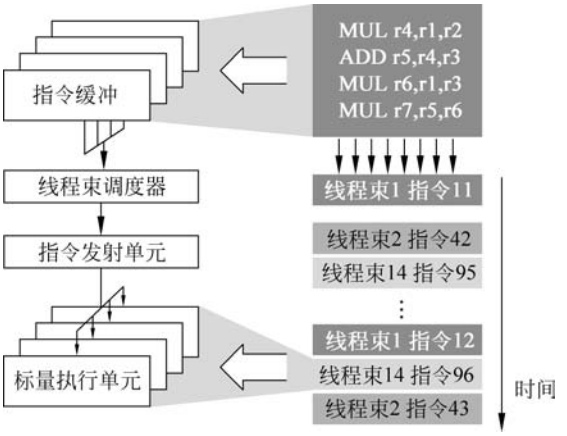


图 3-22　调度器的工作原理和线程束指令交错执行

3.4.2　基本的调度策略

简单来讲，GPGPU 线程束调度器的职责是从就绪的线程束中挑选一个或多个线程束发送给空闲的执行单元。这个过程看似简单，但由于连接了指令取指和执行两个关键步骤，调度器的选择会涉及整个 GPGPU 执行过程的多方面，对 GPGPU 的性能有着重要的影响。

首先一个问题是，什么样的线程束可以认为是就绪的？在处理器中，一条就绪的指令一般需满足以下三个基本条件：下一条指令已经取到，指令的所有相关性都已解决，以及指令需要的执行单元可用。在 GPGPU 架构中，就绪的线程束也类似。根据 NVIDIA 对发射停顿原因的描述，主要有以下一些原因。

(1) Pipeline busy，指令运行所需的功能单元正忙。

(2) Texture 单元正忙。

(3) Constant 缓存缺失，一般说来会在第一次访问时缺失。

(4) Instruction Fetch，指令缓存缺失，一般只有第一次运行访问容易缺失。例如，跳转到新的地方或是达到指令缓存行的边界。

(5) Memory Throttle，有大量存储访问操作尚未完成，为了不加剧性能损耗导致存储指令无法下发。这种原因造成的停顿可以通过合并存储器事务来缓解。

(6) Memory Dependency，由于请求资源不可用或满载导致 load/store 无法执行，可以通过存储访问对齐和改变访问模式来缓解。

(7) Synchronization，线程束在等待同步指令，如 CUDA 中的_syncthreads()要求线程块中的所有线程都到达后才能统一继续执行下一条指令。

(8) Execution Dependency，输入依赖关系还未解决，即输入值未就绪。这与 CPU 中的数据相关是类似的。

只有消除了上述原因的线程束才可能被认为是可以发射的。通过对这些停顿的动态统

计分析(profiling),架构设计者可以获知特定的内核函数性能损失的原因。例如,在存储受限型的应用中,存储依赖(memory dependency)的占比往往会很高,此时 GPGPU 的性能大幅受限于访存。当然,问题溯源是为了改进。一方面,编程人员可以根据分析的结果来优化内核函数的代码,另一方面,架构设计者可以获得对微架构进一步的优化方向。

在获知线程束就绪后,调度器又是如何工作的呢?根据指令流水线的介绍,从指令缓存中读取到的指令一般会被存放在一个小的指令解码缓冲区中。如图 3-23 所示,这个指令解码缓冲区可以采用一个简单的表结构,表项数目与可编程多处理器所允许的最大线程束的数量相关。每个表项包含了一个线程束的基本信息,包括线程块 ID、线程束 ID 和线程 ID。由于所有线程使用同一个内核函数,所以这些信息主要用来判断线程执行的进度、是否已经完成及生成逻辑寄存器到物理寄存器的映射等,以便能够访问到各自线程束的物理寄存器。在指令解码缓冲区中,每个线程束可能会存储几条待执行指令,减少发射停顿的可能。

ID	PC	解码后指令	Ready	Valid

图 3-23 线程束调度器条目的基本结构

当一条线程束指令解码完成后,会设置有效字段(valid)表明该指令有效,然后实施就绪检查以决定是否可以发射。如果可以,设置就绪字段(ready)表明该指令已经就绪,等待调度器的选择和发射;否则该指令就一直在指令缓冲区中等待直到就绪字段被设置。当一条就绪的线程束指令发射后,线程束调度器会将该表项清除,并通知取指单元加载新的指令进来,对指令解码并重复上述的操作。不同的线程束指令可能执行的进度并不一致,因此会导致每个线程束的 PC 字段并不相同,因此需要在指令解码缓冲区设置足够多的条目,保留每个线程束执行的进度。

那么在众多就绪的线程束中,调度器是如何做出选择的呢?早期的线程束调度器往往采用基本的轮询(Round-Robin,RR)调度策略。如图 3-24 所示,它在调度过程中,对处于就绪状态的线程束 0、1、3、4、5 都赋予相同的优先级,并按照轮询的策略依次选择处于就绪状态的线程束指令进行调度,完成后再切换到下一个就绪线程束,如线程束 0、1、3、4、5 都执行完成第 1 条指令(指令 0)后再重复上述过程直到执行结束。与之相对应的另一种策略称为 GTO(Greedy-Then-Oldest)。该策略允许一个线程束按照贪心策略一直执行到不能执行为止。例如,当线程遭遇了缓存缺失,此时调度器再选择一个最久未调度的线程束来执行,如果再次停顿再调度其他线程束,直到执行结束。图 3-24 对比了两种策略的不同。在该例子中,GTO 调度器首先选择了线程束 0 的前 3 条指令执行,直到无法继续执行指令 3,此时再切换到线程束 1 的前 3 条指令执行。在这个过程中,线程束 2 由于某种原因始终未能就绪,因此它的就绪字段不会被设置,无论哪种调度器都不会调度它。线程束的生命周期起始于它被分配到可编程多处理器上的时刻,因此一个线程块内的线程束具有相同的生命周期。实际上,GTO 调度与轮询调度可以认为是两种极端情况。但两者都存在一定的问题,后面的内容将对这些不足和改进方法展开更为细致的讨论。

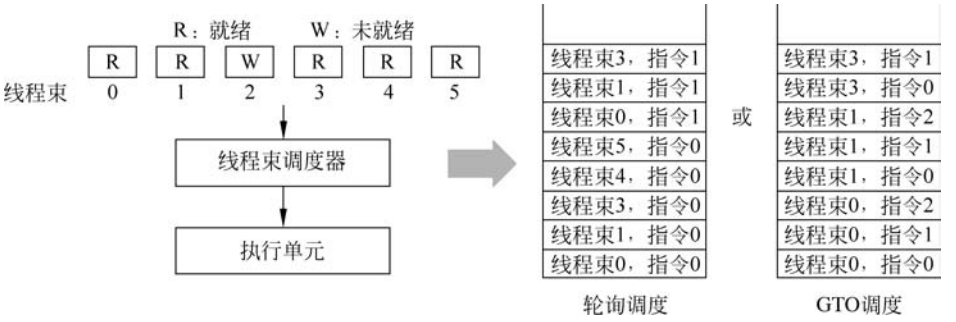


图 3-24　基本的轮询调度和 GTO 调度策略

3.4.3　扩展讨论：线程束调度策略优化

GPGPU 的执行性能与线程束的调度之间关系密切。线程束调度的主要功能是选择合适的线程束发射执行，但这个“合适”却很难给出具体的定义，总体上以改善性能和功耗为目标。合适的调度策略和调度器设计需要综合考虑硬件设计的复杂度、开销及代码执行过程中多种复杂的情况，比如，掩藏长延时操作以提高吞吐率、发掘数据局部性以降低延时、线程执行进度的平衡等，从而获得性能和功耗的最优化。

在 GPGPU 架构中，数据的访存延时仍然是影响性能的主要因素，而发掘数据的局部性则是改善访存延时最有效的手段之一。由于 SIMT 架构的特点，一般来讲内核函数中往往存在两种数据局部性：线程束内局部性(intra-warp locality)和线程束间局部性(inter-warp locality)。当数据被一个线程访问后，如果不久之后还会被同一线程束中的其他线程再次访问则称为线程束内局部性，而如果再次访问的是其他线程束中的线程则称为线程束间局部性。注意这里所谓的“再次访问”可能是这个数据本身，也可能是相邻地址的数据，因此是包含了时间和空间两种局部性而言的。线程束内局部性主要源于线程束内的线程往往是连续分配的，会线性地访问连续的地址空间，因而易于通过合并操作命中同一个缓存行；而线程束间局部性往往是由于线程块中的线程束也具有类似的地址连续特性。回顾轮询和 GTO 两种调度策略，其实两者就是发掘线程束内局部性和线程束间局部性的不同体现：轮询策略通过执行不同线程束的同一条指令，较好地获得了线程束间局部性；而 GTO 策略则更多地考虑了线程束内的局部性。到底哪种因素更重要、对性能的影响更大则要根据实际运行程序的特点决定。

GPGPU 架构采用大规模线程的设计初衷就是希望能够利用线程的快速切换达到掩藏访问延时的目的，从而保证或提高吞吐率。然而，基本的轮询调度策略并不能很好地达到这一效果。考虑图 3-25 的情形，使用轮询策略对 16 个线程束 W0，W1，…，W15 进行调度，每个线程束中 $I_0 \sim I_{k-1}$ 均为运算指令，I_k 为访存指令。首先，调度器依次调度各线程束的指令 I_0，16 个周期后再依次调度各线程束执行指令 I_1。重复上述过程，直到 $16\times(k-1)$ 个周期后，所有线程束先后进入访存指令 I_k，执行长延时操作，假设相邻线程束执行 I_k 指令仅相

差一个周期，显然线程束 W0 的访存操作几乎不可能在 16 个周期内完成，而此时也没有更多可供调度的线程束来隐藏访存延迟，这导致流水线陷入了一段较长时间的空闲。考虑到存储访问的延时往往需要几十甚至上百个时钟周期，除非有大量线程束可供调度，否则很容易导致延时不能被有效地掩盖。这反映出轮询调度策略对长延时操作的容忍度还不够高。

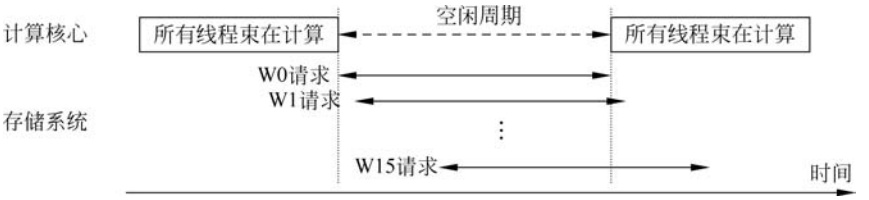


图 3-25 基本的轮询调度存在的问题

与之相对，GTO 策略则倾向于让一个线程束尽快执行。当遭遇了长延时操作时，其他线程束还可以有更多的指令（如上例中的 $I_0 \sim I_{k-1}$）用于掩藏延时，从而提供一定程度的改进。但 GTO 策略可能会破坏线程间局部性，在高速缓存很小的情况下可能会导致缓存数据重用不足甚至抖动现象，这反而拉长了平均访存延时，使得这些本来可以避免的访存缺失反而需要更高的线程并行度来掩盖。

因此，GPGPU 线程束的调度往往需要解决两方面的问题。

（1）调度策略需要能够首先甄别出执行过程中影响性能的主要因素。

（2）调度器能够以简单的硬件逻辑运用轮询和 GTO 策略或两者的结合取得更好的性能。

两者相互依赖，因为调度器需要专门的硬件对某些指标进行动态统计，反馈给调度器进行策略的调整。线程束调度作为线程切换的最小粒度，在这方面也有很大的空间。本节介绍了对缓存缺失、指令停顿等指标的统计，并调整调度策略的案例，来帮助读者理解线程束调度策略和调度器设计的要点。

1. 利用并行性掩藏长延时操作

在 GPGPU 架构中，轮询和 GTO 调度策略都比较理想化，面对复杂情况时显示出诸多不足。基于两者的基本思想，针对长延时操作的掩藏有如下几种改进的调度策略。

1）两级轮询调度

基于轮询的调度策略可以保证线程束调度的公平性，允许相邻的线程、线程束和线程块相继执行。如果程序具有良好的空间局部性，这种方式利于挖掘数据的空间局部性。但数据局部性特征只能一定程度上改善访存延时，并不能改善 GPGPU 对长延时操作的掩藏能力。

为了解决轮询调度策略在应对长延时操作时表现不佳的问题，文献[10]设计了一种两级线程束调度（two-level warp scheduling）策略，它将所有线程束划分为固定大小的组（fetch group），组间基于优先级顺序的策略进行调度，但本质上还是轮询策略。在初始条件下，第 0 组优先级最高，第 1 组次之……。第 0 组将优先得到调度，当该组中所有线程束依

次执行到访存指令时，将该组的优先级降至最低。同时赋予第 1 组最高优先级，组内每个线程束权重相等，同样按照轮询策略调度，以此类推。调度器通过修改各组的优先级，切换到下一个优先级最高的组并执行，达到了隐藏延时、缩短流水线空闲时间的目的。图 3-26 仍然采用图 3-25 中 16 个线程束的例子，分为 2 组，每组包含 8 个线程束。第 0 组拥有较高优先级，第 1 组优先级次之。调度器优先选取第 0 组调度，组内按照轮询策略依序执行各个线程束的指令 I_0、指令 I_1……直到组内 8 个线程束都执行到访存指令 I_k 时，将第 0 组的优先级置为最低，然后赋予第 1 组最高优先级并调度执行。此时还有 8 个线程束，每个线程束也有足够的指令（$I_0 \sim I_{k-1}$）可供调度，从而更好地掩藏第 0 组访存操作带来的长延时，因此两级轮询调度总用时更短。在这个例子中，理想情况下节省的时间约为组 1 计算的时间。

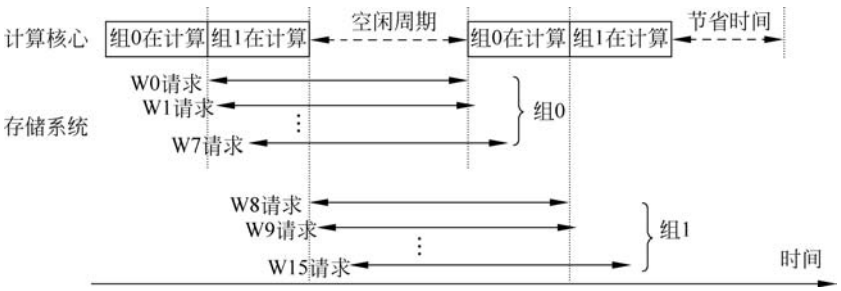


图 3-26　两级轮询调度改善了长延时操作掩藏的能力

这种两级调度策略是对基本轮询调度策略的一种改进，实现起来也相对简单。调度策略通过将各个组的长延时操作分隔开，使访存指令可以分批次更早地发射执行，将后继组的运算阶段和前置组的访存阶段重叠起来，提高了长延时操作的掩藏能力。同时组内和组间仍采用轮询的方式，让相邻线程束相继执行，尽可能地保证数据的空间局部性。

组规模的设置也存在影响：当组内线程束数量过少时，加载到 DRAM 行缓冲区的数据不能得到充分利用，且线程束级并行度过低；如果偏向另一个极端，即组内线程束数量过多时，最坏情况退化到基准轮询调度策略，则两级调度的优势将被弱化，对长延时操作的容忍度降低。这个阈值的选择针对具体的案例也可能有所不同，这在调度器中也要有所考虑。

2）线程块感知的两级轮询调度

与两级轮询调度策略类似，文献[16]同样从提高 GPGPU 对长延时操作的容忍能力出发，提出了另一种两级调度策略——线程块感知的两级线程束调度（CTA[①]-aware two-level warp scheduling）。与前者不同的是，该策略兼顾了线程块间数据的分布特点而试图利用线程块间的数据局部性，所以将第一级（对应于两级轮询调度中的“组”级）设置为线程块级，即分组时将若干存在数据局部性的线程块分配到同一组中。相应地，下一级（对应于两级轮询调度中的线程束级）仍设置为线程束级，每个线程块内包含若干线程束，以此作为该策略下的第二级进行调度。该策略之所以沿用“两级轮询调度”的名称，是因为它在两个级别的

① 线程块有时也被称为 CTA（Cooperative Thread Array，协作线程组）。

调度方面仍然选取了轮询策略。组一级线程块之间按照轮询策略进行调度，当前置组中所有线程束都因长延时操作而被阻塞时，调度器切换到下一组线程块并继续执行。线程块内的线程束具有相等的优先级，线程束级同样按照轮询策略调度执行。这种方法在理念上与两级轮询调度相同，可以认为是在具体操作层面上的改进。

3）结合数据预取的两级调度策略

预取作为一种掩盖长延时访存操作的技术，被广泛应用于 CPU 中。在 GPGPU 中，如果线程束调度和预取策略配合不当，会导致需要预取线程束的调度时机与当前正在执行的线程束过于接近，使得延时不能被充分掩藏。如图 3-27(a)所示，假定有 8 个线程束 W0～W7 需要从 2 个 DRAM 板块中读取不同的数据块 D0～D7。一般情况下，连续线程束的数据在 DRAM 中往往具备空间局部性。假设 W0～W3 的数据块存储在板块 0 中，W4～W7 的数据块存储在板块 1 中。基本的轮询调度策略较好地保留了这种空间局部性和板块并行性，即当 W0～W3(W4～W7)需求的数据在板块 0(板块 1)的同一行时，一次读取就可以读出 W0～W3(W4～W7)所有所需的数据，并且板块 0 和板块 1 的读取可以并行。但轮询调度有时并不能很好地与预取策略结合，如图 3-27(b)所示。当一个线程束，如 W0 访问全局存储器时，预取器会预取下一个连续的数据块 P1，可能就是下一个将要被调度的线程束所需要的数据。但因为轮询调度中连续线程束的调度时间非常接近，在发出预取请求不久后需要该数据的线程束就会被调度，导致这个预取并不能有效地减少该线程束等待数据的时间，降低了预取的质量。类似的情况在基于轮询的两级调度中依然存在。

文献[17]对于这种线程束调度和预取策略配合不当的问题提出了一种预取感知的调度方式(prefetch-aware warp schedule)。它采用两级调度策略，将线程束分组以便将连续的线程束隔开，比如 W0/W2/W4/W6 为组 0，W1/W2/W5/W7 为组 1。假设按照两级轮询优先调度第 0 组配合简单预取策略，如图 3-27(c)所示，在 W0 请求访问全局存储器时，对 W1 的预取 P1 也一样被发出，而 W1 真正被调度时已经在组 0 的线程束全部进入停顿之后，一部分预取时间已经被执行时间掩盖，从而能够提高预取质量。等到组 0 的数据和预取数据返回后，所有的线程束都可以计算。

2. 利用局部性提高片上数据复用

虽然 GPGPU 强调线程并行性和计算的吞吐率，但利用数据的局部性对提升性能来讲也至关重要，因为当数据被加载到片上存储尤其是 L1 数据缓存后，如果能有效地重用这些数据提高缓存的命中率，既可以减少访存的延时，又可以减少重复的访存操作，相当于减少了长延时访存操作的次数，这也是 GPGPU 中增加了缓存部件的一个重要原因。

虽然缓存在通用场景下对数据重用很重要，但一般来讲，可编程多处理器内部的 L1 数据缓存容量往往都很小，只有几十到几百 KB 规模。考虑到一个可编程多处理器中巨大的线程数目，每个线程能够分配到的 L1 数据缓存容量往往只有几个字节。根据缓存的 3C 模型，这显然会带来严重的缓存冲突问题。为了能够利用缓存降低访存延时，一种方法就是通过降低同时活跃的线程块或线程束数目来提高每个线程块或线程束所分配到的缓存容量，进而提高 L1 缓存的命中率，获取 GPGPU 整体运行效率的提升。这种技术也称为“限流”

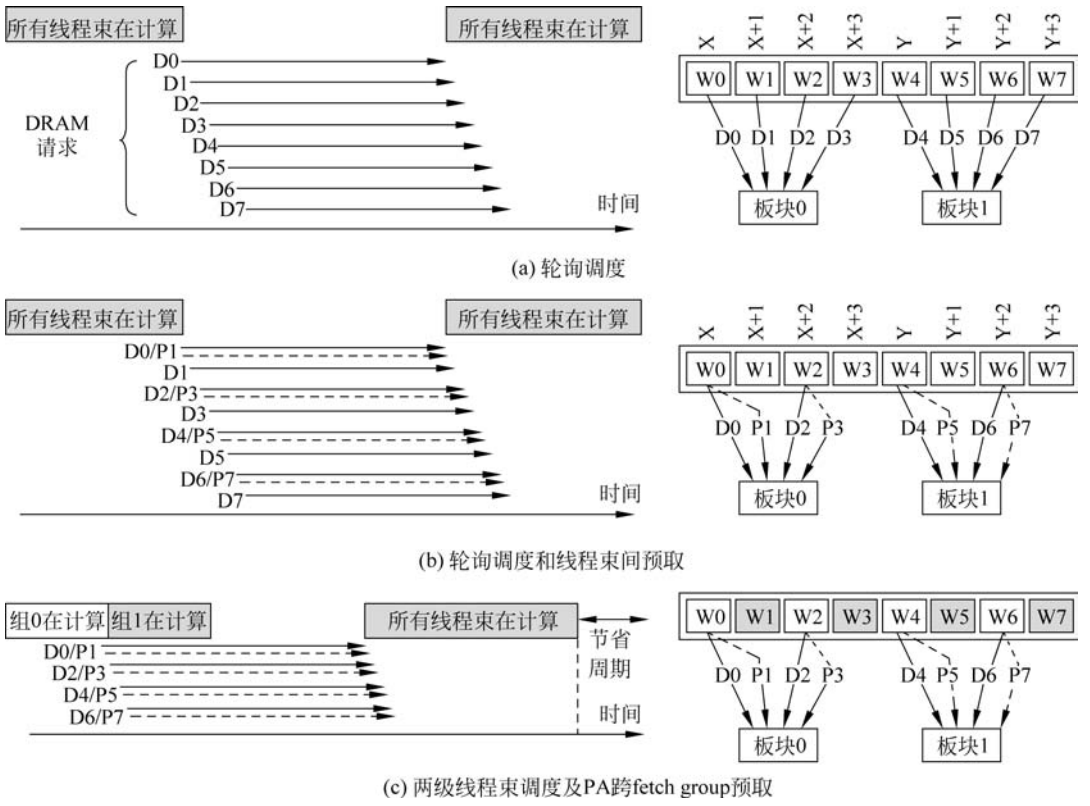


图 3-27　轮询调度和预取

(throttling)技术。对于缓存敏感的内核函数来说,限流技术通过提高 L1 数据缓存的命中率,可能会带来良好的性能提升,同时还可以与线程束调度很好地结合。

针对本节开始提到的两种典型的数据局部性,有研究对缓存敏感型应用的访存行为进行了统计分析,发现线程束内局部性现象比线程束间局部性更为普遍,因此可以充分利用线程束内局部性改善缓存敏感型应用的性能。传统的 GTO 策略虽然一定程度上利用了线程束内的局部性,但它缺少访存情况的主动反馈,无法指导调度器根据实际的访存情况进行策略的调整。文献[18]针对这一问题提出了一种缓存感知的调度策略(Cache-Conscious Wavefront Scheduling,CCWS)。CCWS 通过限制可编程多处理器中可以发射访存指令的活跃线程束数量,保证 L1 缓存中的数据得到更为充分有效地复用,提高访存命中率。

CCWS 是一种带有反馈机制的、可动态调整的线程束调度方案,其核心设计思想是,如果线程束发生缓存局部性缺失,则为它提供更多的缓存资源,以降低可能复用的数据被替换出缓存的可能性。为此,该方法设计了一套评分系统用以量化局部性丢失的情况。图 3-28 显示了这个评分系统在运行时实施线程束限流的一种可能状态:在初始 T0 时刻,线程束 W0～W3 的局部性分值(Lost-Locality Score,LLS)相同,因此具有平等的优先级。T0 至 T1 时刻,W2 执行中发生了局部性丢失,则为其赋予一个更高的分值,并将分值最高的 W2

置于栈底优先考虑。另外，虚线表示允许发射访存指令的累积分数上限。可以看到，此时分值更低的 W3 被“顶”出了上界，因而不能发射访存指令，从而该 L1 数据缓存所支持的线程束数量就从 4 个减少为 3 个，让 W2 获得了更充裕的缓存资源，达到了限流的作用。

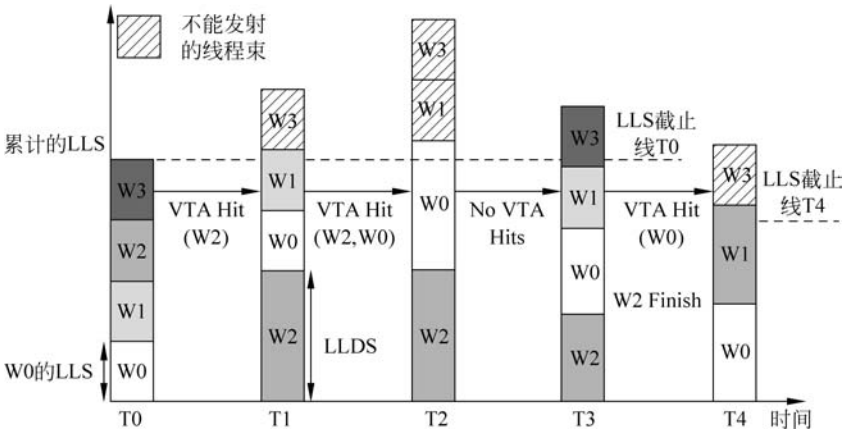


图 3-28 局部性评分系统运行时实施线程束限流的一种可能状态

为了实现 CCWS 的调度策略，该文献设计了线程束内局部性丢失检测器（Lost Intra-Wavefront Locality Detector，LLD）和局部性评分系统（Locality Scoring System，LSS）两个主要的部件。LLD 用于检测丢失局部性的线程束。它本质上是一个仅存储缓存标签（tag）的受害者缓存（victim cache）：每个线程束都拥有一个受害者标签列（Victim Tag Array，VTA），当 L1 缓存中某一缓存行被逐出时，将该行的标签写入对应线程束的 VTA 中。若此后这个线程束发射的访存指令在 L1 缓存中再次发生缺失，又恰好被 VTA 所“捕获”，即表明访问的数据已被逐出 L1 缓存，该线程束发生了一次局部性丢失。如果能够为该线程束提供更多的独占 L1 缓存资源，则可能避免上述情形的发生，从而寻找到丢失局部性的线程束，为 CCWS 调度提供优化的对象。LSS 属于 LLD 的“下游”模块，它接收 LLD 发现的局部性丢失线程束并将其反馈到线程束的评分上。该模块通过累计分值和边界阈值的大小关系来实现限流。如图 3-28 所示的例子，当接收到来自 LLD 的判断信号后，将对应线程束的分数提高到 LLDS（Lost-Locality Detected Score），例如将 W2 的分值置为 LLDS。此后若该线程束在短期内不再发生局部性丢失，则每个周期降低分数，直到减为初始分值为止。当然，若在恢复过程中又发生了一次局部性丢失，则其分数将被重新置为 LLDS。为了限制流多处理器内可以发射访存指令的线程束数量，还需要设置一个“上限”，称为累计分数截止线（cumulative LLS cutoff），它可以将超过上限的线程束过滤掉，即屏蔽这些线程束的访存机会，为丢失局部性的线程束提供更多的独占访存机会。由于 W2 增加的分值将 W3 推出了累积分数截止线，W3 的“Can Issue”（可发射）位被清空为 0，因而 W3 被暂时屏蔽不可以发射访存指令。在 CCWS 中，各种参数的设计对于性能有着直接的影响。对于这些参数的量化设计细节，可以参见文献中的详细论述。

CCWS 调度策略通过对线程数据的合理限流达到增加 L1 缓存容量的目的，有效地提升了 L1 数据缓存的命中率，从另一个角度提高了访存指令的执行效率。但也可以看到，CCWS 的方法需要较多的存储资源来记录 LLD 和 LSS 的各种信息，硬件结构也相对复杂，而且仅考虑了 L1 数据缓存的局部性问题，这在实际设计中需要仔细权衡。

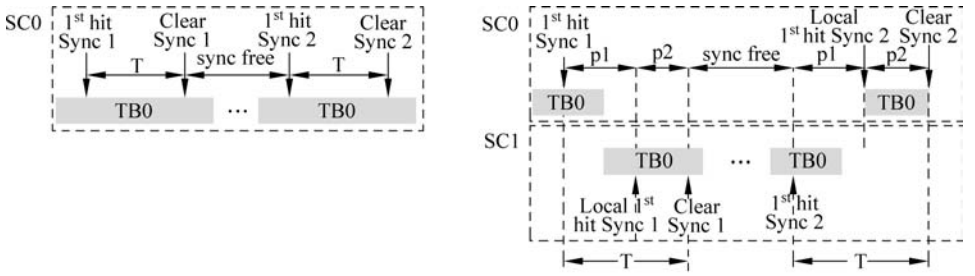
3. 线程束进度分化与调度平衡

在理想情况下，GPGPU 中不同线程束的执行路径完全相同，执行时间也类似。但有些时候，线程束的执行进度也会表现出较大的差别。例如，在遭遇同步栅栏或线程分支时，不同的线程束执行出现分化。由于 GPGPU 依照线程块为粒度分配处理器资源（如寄存器文件、调度表项等），如果一个线程块中不同线程束之间执行进度差异很大，先执行完成的线程束就会一直等待后完成的线程束而长期占用处理器资源。这不仅会导致资源闲置，还会造成可用资源不足、并行度降低等问题。因此，在线程束执行进度差异较大时，平衡不同线程束的执行进度对于改善 GPGPU 的性能来说也是一个重要的因素。

1）多调度器协同策略

前面介绍的调度策略都是针对一个线程束调度器的情况。当可编程多处理器中有多个调度器时，如果缺乏相互协同也可能会导致执行过程不够高效。图 3-29(a)展示了一个单调度器的情形，记为 SC0。当一个线程块 TB0 被分配到可编程多处理器上时，其内部线程束需要到达同步栅栏点（Sync1 和 Sync2）后继续执行。其中，1^{st} hit 表示 TB0 的第一个线程束到达同步点，clear 表示 TB0 的最后一个线程束到达同步点即可“清除”，这时 TB0 所有线程束可以继续执行。图 3-29(b)则展示了双调度器（SC0 和 SC1）的情形，此时 TB0 的线程束可能会被分配到两个调度器上且两边调度顺序并不相同。在调度器 SC0 上，一个线程束首次到达同步点 1，记为 1^{st} hit Sync1，而后 SC1 上的线程束也遇到了同步点 1，记为 local 1^{st} hit Sync1。SC0 和 SC1 分别执行 TB0 剩余的线程束并在其中一个调度器完成执行后等待，直到另一个调度器清除同步点 1 为止。由于两个调度器之间彼此独立，1^{st} hit 和 clear 之间的时间间隔可能会很长，这可能导致流水线停顿。更糟糕的是，由于 SC1 并不知道 SC0 调度了 TB0 的线程束，SC1 将自由地调度其他线程块，使得 SC0 上 TB0 的等待时间更长。为便于分析，这里将从 1^{st} hit 到 clear 等待的总时长划分为 2 个阶段：p1 和 p2，local 1^{st} hit 作为两段时间的分割点。通过对不同调度策略下 p1 和 p2 时长的统计发现平均情况下 p1 占据总时长的比例高达 85%～90%。

根据以上分析不难发现，p1 和 p2 两个阶段是相互独立的，二者不存在相关性。对于 p1，它反映的是线程块间调度的开销，其主要原因是调度器之间缺乏协作，即当线程块 TB_x 的某个线程束在调度器上首次运行到同步点时，其他调度器无法感知也无法及时将 TB_x 提前执行。针对这个问题，文献[19]提出为每个线程束调度器设计一个优先级队列，不同线程块的线程束按照优先级从高到低的顺序排列，其中同一线程块内的线程束优先级相等。当 TB_x 中的某个线程束首次执行到同步点时，将其优先级降至最低并移到队列尾部，同时提高所有调度器中 TB_x 的线程束优先级，在不抢占执行的情况下将 TB_x 的线程束移动到各队列的首部，这样可以更快地被调度执行以达到减小 p1 的目的。对于 p2，迟滞当前线程块

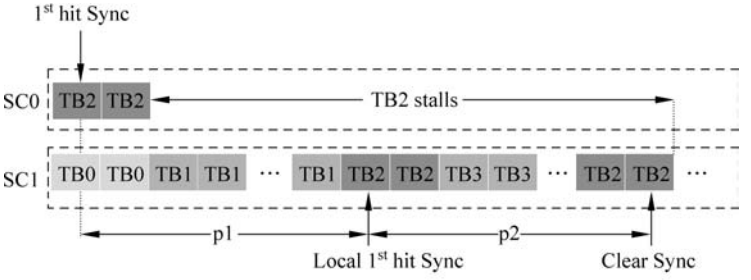


(a) 单线程束调度器　　(b) 两个线程束调度器

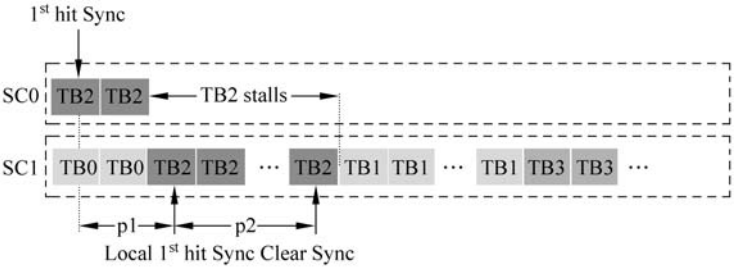
图 3-29　调度器处理同步栅栏时产生空闲的例子

清除同步点的主要原因在于阻塞的线程束恢复后没能及时得到调度。为此，调度器应保持 TB_x 的优先级不变。一旦线程束恢复到准备状态，则立即恢复对 TB_x 的调度。

图 3-30 展示了多线程束调度器协同的优势，其中图 3-30(a)为 GTO 策略下调度器未协同的情形，当 SC0 上 TB2 首次遇到同步点 Sync 时，SC1 中 TB2 的线程束仍位于 TB0 和 TB1 的线程束后面，同时 SC1 上 TB2 的线程束并不连续，使得 p1 和 p2 都比较长。如图 3-30(b)所示，采取了多线程束调度器协同策略后，SC1 中 TB2 的线程束被提前到 TB0 之后执行(因为此时 TB0 的线程束尚未执行完毕)，这大大缩短了 p1。此外 TB2 的所有线程束都被提前，使得原本不连续的线程束彼此相邻，p2 也被明显缩短，改善程序的执行性能。



(a) GTO策略下调度器未协同的情形



(b) GTO调度策略下采用多线程束调度器协同

图 3-30　多线程束调度器协同对程序执行性能的影响

2）线程束动态均衡调度策略

实际上，线程执行进度的分化在一个调度器下也会出现。一个进度分化的原因在于单一线程块内的线程束由于存储系统访问的不确定性，即便采用轮询调度，不同线程束的执行进度也可能存在较大差异。另外一个进度分化的重要原因就是在线程同步栅栏处或在分支线程重聚的位置，执行快的线程束先到达，等待执行慢的线程束。在此期间先到达的线程束并不会释放其占有的硬件资源（如寄存器文件），导致大量资源被闲置浪费。而且当越来越多的线程束到达栅栏或重聚点而不得不等待时，活跃线程束的数量也变得越来越少而不足以掩藏长延时操作，导致流水线的吞吐量明显降低。因此，需要一种动态协调线程束执行进度的方法，缩小最快和最慢线程束之间的执行差距。

文献[20]提出了一种基于运行时动态感知线程束进度的调度策略——关键性感知的线程束协调加速（Coordinated criticality-Aware Warp Acceleration，CAWA）。其中，线程束的“关键性”（criticality）反映的就是线程束执行时间的长短，执行时间最长（即执行最慢）的一个线程束被称为关键线程束（critical warp），因为这个线程束往往决定着当前整个线程块的执行时间。一个简单的方法就是给予关键线程束更高的调度优先级，分配更多的硬件和时间资源给它，最大限度满足其执行需求。

首先，为了在运行时判定一个线程束是否关键，文中提出了一种称为关键度预测的度量方法来为每个线程束维护一个关键性度量值（criticality counter）。影响线程束关键性的因素主要来自两方面：线程分支导致的工作负载差异和共享资源竞争引入的停顿。对于前者，当指令执行遇到分支时，不同分支路径内指令数量多数情况下是不相等的，可以直观地用指令数目作为判据之一。哪个路径的指令多，其对关键度的影响越大。对于后者，即访问共享资源发生竞争造成线程束空闲等待，也增加了其跃升为关键线程束的可能。综合以上两方面影响因素，可以得到对线程束关键性的度量。

基于对线程束关键度的判别，该文献提出了一种基于 GTO 的关键性感知线程束调度策略，称为 greedy Criticality-Aware Warp Scheduling（gCAWS）。在 GTO 策略中，调度器会尽可能选择同一个线程束执行，其他就绪的线程束需要等待，这种策略没有考虑线程束的关键性问题。gCAWS 策略改进了调度选取线程束的机制，每次选择关键度最高的一个线程束执行，即给予关键线程束以更高的调度优先级。当关键度最高的线程束有多个时，按照 GTO 策略选择生命周期最长的线程束执行。在执行阶段，不断更新关键度的值，以便发现新的更为关键的线程束进行调度。可以看到，gCAWS 在调度上同时满足了关键线程束和生命周期最长线程束的急迫需求，有利于在线程束执行分化时的调度平衡。

3.5 记分牌

在 GPGPU 指令流水线中，为了防止由于数据相关而导致的流水线执行错误，GPGPU 需要在指令发射阶段检查待发射的指令是否与正在执行但尚未写回寄存器的指令之间存在数据相关。一般会采用记分牌或类似技术避免指令间由于数据相关带来的竞争和冒险。本

节将重点讨论记分牌技术及它在 GPGPU 架构下的设计方法。

3.5.1 数据相关性

在流水线执行中，指令之间的数据相关会对指令级并行产生直接影响。例如，当程序中两条相近的指令访问相同的寄存器时，指令的流水化会改变相关操作数的访问顺序，可能会导致流水化执行得到不正确的结果。为保证程序正确执行，存在数据相关的指令必须按照程序顺序来执行。在通用处理器中，寄存器数据可能存在三种类型的相关：写后读、写后写和读后写，都可能会导致冒险。

(1) 写后读(Read After Write，RAW)，也称真数据相关(true dependence)。按照程序顺序，某个特定寄存器的写指令后面为该寄存器的读指令。若读指令先于写指令执行，则读指令只能访问到未被写指令更新的寄存器旧值，从而产生错误的执行结果。因此，为保证读指令可以获取正确的值，必须保持程序顺序，即先写后读。

(2) 写后写(Write After Write，WAW)，也称名称相关(name dependence)。按照程序顺序，写指令 1 后为写指令 2，并且都会更新同一个目的寄存器。若写指令 2 先于写指令 1 执行，则最后保留在目的寄存器中的是写指令 1 的结果，这与程序顺序执行的语义不符。为了避免这一问题，可以要求指令按照程序顺序执行，即先执行写指令 1，后执行写指令 2。

(3) 读后写(Write After Read，WAR)，也称反相关(anti-dependence)。按照程序顺序，对某个特定寄存器的读指令后面为该寄存器的写指令。若写指令先于读指令执行，则读指令将获取到更新后的寄存器值，产生错误的执行结果。为了避免这一问题，可以要求指令按照程序顺序执行，即先执行读指令，读取后执行写指令。

实际上，存在 WAW 相关和 WAR 相关的两条指令之间并没有真正的数据传递，而是由于采用了相同的寄存器编号，将两条不相关的指令人为地联系到一起。因此，除保守地维持指令原来的顺序之外，还可以通过寄存器重命名(register renaming)技术消除 WAW 和 WAR 相关。代码 3-3 展示了寄存器重命名的代码示例。可以看到，add. s32 使 r8 和 sub. s32 使用 r8 存在 WAR 相关，可以将 sub. s32 指令中的 r8 重命名为 t；ld. global. s32 使用 r6 和 mul. s32 使 r6 存在 WAW 相关，可以将 ld 指令中的目标寄存器重命名为 s。通过分配不同的寄存器，就可以消除流水执行中可能发生的 WAW 和 WAR 相关，也使得指令的动态调度成为可能。

代码 3-3 采用寄存器重命名技术消除 WAW 和 WAR 相关的示例

```
1  // 采用寄存器重命名技术之前
2  div. s32          %r0,%r2,%r4
3  add. s32          %r16,%r4,%r8
4  ld. global. s32   %r6,array[r1]
5  sub. s32          %r8,%r10,%r14
6  mul. s32          %r6,%r10,%r12
```

```
1  // 采用寄存器重命名技术之后
2  div. s32          %r0,%r2,%r4
3  add. s32          %r16,%r4,%r8
4  ld. global. s32   s,array[r1]
5  sub. s32          t,%r10,%r14
6  mul. s32          %r6,%r10,%r12
```

在流水线的硬件结构中，指令调度阶段需要增加专门的硬件来检测和处理数据相关性问题，以避免流水线执行错误。一般来讲，CPU 设计中经典的记分牌和 Tomasulo 算法可以实现这一目标。经典的记分牌技术通过标记指令状态、功能单元状态和寄存器结果状态，控制数据寄存器与功能单元之间的数据传送，实现了乱序流水线下指令相关性的检测和消除，保证了程序执行的正确性，同时提高了程序的执行性能。Tomasulo 算法也支持乱序流水线调度，其核心思想与积分牌类似，并引入了保留站(reservation station)结构，实现对寄存器的动态重命名，消除了 WAW 和 WAR 冒险。同时它引入公共数据总线(common data bus)，允许操作数可用时立即存储在保留站中触发指令执行，而不用等待寄存器写回，从而将写后读相关的损失降至最低。关于记分牌和 Tomasulo 算法可以参考文献[3]中的介绍。

然而不管是记分牌还是 Tomasulo 算法，其复杂度和硬件开销都相对较高。一方面，在 GPGPU 架构中，由于寄存器和功能单元的数量众多，记录它们运行时状态信息的硬件开销也将显著增加。除此之外，大量连线的成本也不容忽视。另一方面，对于传统的 CPU 设计来说，由于数据相关性导致的流水线停顿会显著影响指令的发射效率，大幅降低指令级并行性会对性能带来不利的影响。对于 GPGPU 架构来说，其指令并行度本身就很高，大量不同的线程束可以提供无相关性的指令供调度器选择。即便某个线程束由于数据相关而导致发射停顿，利用线程束调度器还可以从其他的线程束中找到合适的指令填充流水线，降低数据相关对流水线性能的影响。因此，对于 GPGPU 架构来说，利用乱序执行进行指令调度提高指令级并行性并非必要，也不需要复杂的记分牌和 Tomasulo 设计，但 GPGPU 流水线仍然需要数据相关性的检测和处理。

3.5.2 GPGPU 中的记分牌

为了提高 SIMT 运算单元的硬件效率，GPGPU 一般会采用顺序执行的方式，避免乱序流水线带来的指令管理开销。但 GPGPU 指令的执行仍然可能需要多个周期才能完成，而且不同指令存在不等长执行周期的情况。因此，为了让同一线程束的后续指令在发射时减少等待时间而尽早发射，仍然要保证前后指令之间不存在数据相关，从而提高指令的发射和执行效率。假设采用经典的五级顺序流水线设计，3 种数据相关性冒险如图 3-31 所示。在 GPGPU 架构中，重点是要避免发生 RAW 和 WAW 冒险。对于 WAR 冒险，在顺序流水线下一般不会发生，因为后续指令的寄存器写回一般不太可能会超前于前序指令对同一寄存器的读取。

记分牌的机制可以避免由数据相关导致的冒险情况发生。相比于乱序执行流水线中的记分牌，顺序执行中的记分牌设计会相对简单。在 GPGPU 顺序执行下，一个简单的记分牌方案可以设计如下：记分牌为每个线程束寄存器分配 1 个比特用于记录相应寄存器的写完成状态。如果正在执行的线程束指令将要写回的目标寄存器为 R_x，则在记分牌中将寄存器对应的标识置为 1，表示该指令尚未写回完成。在此之前，如果同一线程束中的后续指令不存在数据相关，则可以尽早进入流水线执行。否则，如果同一线程束存在后续指令需要读取或修改 R_x，由于设置了标识位，后续指令将会受到限制而处于非就绪状态，不能被调度或发

图 3-31 顺序流水线下的 3 种数据相关性冒险

射，从而避免了 RAW 和 WAW 相关性冒险。直到前序指令写回 R_x 完成，寄存器 R_x 对应的标识会被重置为 0，后续存在数据相关的指令才可以被调度进入执行单元。在该线程束指令流水线因数据相关而被停顿过程中，其他线程束的指令仍然可以被调度执行，因为不同线程束的寄存器 R_x 实际上物理位置并不相同(参见 4.2 节寄存器文件的结构)。

这一记分牌设计方案虽然简单，但主要存在两方面的问题。

(1) GPGPU 中存在大量的寄存器，如果为每个寄存器都分配 1 比特标识，记分牌将占用大量的空间。假设每个可编程多处理器最多支持 64 个线程束，每个线程束分配最多 128 个寄存器，那么每个可编程多处理器需要 8K 比特的记分牌存储空间。

(2) 所有待发射的线程束指令在调度时需要一直查询记分牌，直到所依赖的指令执行完毕，更新寄存器对应的标识位后，后续指令才能发射。假设每个可编程多处理器最高支持 64 个线程束，每个线程束指令最多需要访问 4 个操作数，那么每个周期要同时检查所有 64 个线程束指令的数据相关性，记分牌需要 256 个端口读取状态提供给线程束调度器。这种设计会带来巨大的硬件开销，显然是不现实的。

3.5.3 扩展讨论：记分牌设计优化

前面提到的这种简单记分牌设计方案在 GPGPU 架构下的硬件开销很高。本节将针对适合 GPGPU 架构的记分牌设计进行讨论，介绍几种优化硬件开销的设计思路。

1. 基于寄存器编号索引的记分牌设计

NVIDIA 的专利中提到了一种新的基于硬件的记分牌实现方法和处理过程。如图 3-32(a)所示，首先配备一块记分牌的存储空间，并将这块空间划分成若干区域。考虑到记分牌主要是对指令缓冲(I-Buffer)中已解码的指令进行相关性检查才可能发射，因此可以将记分牌存储空间划分为与指令缓冲中指令数目相同的区域。每个区域中包含若干条目，每个条目包含两个属性：寄存器 RID(Register ID)和尺寸指示器。寄存器 RID 记录了该区域所对应的线程束目前正在执行的若干指令中，将要写回的目的寄存器编号。如果指令中将要写回的

寄存器为一个序列，尺寸指示器则负责记录该寄存器序列的长度，而 RID 只需要记录这个序列中的第一个寄存器 RID。例如，假设某个线程束运行了一个纹理读取指令，并且结果将会写入 r0、r1、r2、r3 4 个寄存器中。这时，记分牌中该线程束对应区域中一个条目的 RID 将被设为 r0，尺寸设置为 4。采用这种记录方式的好处是，如果目的寄存器是连续分配和使用的，可以避免采用多个条目来记录，减少了记分牌存储空间的使用量，而这可以通过编译器中的寄存器分配算法来最大化这一可能性。

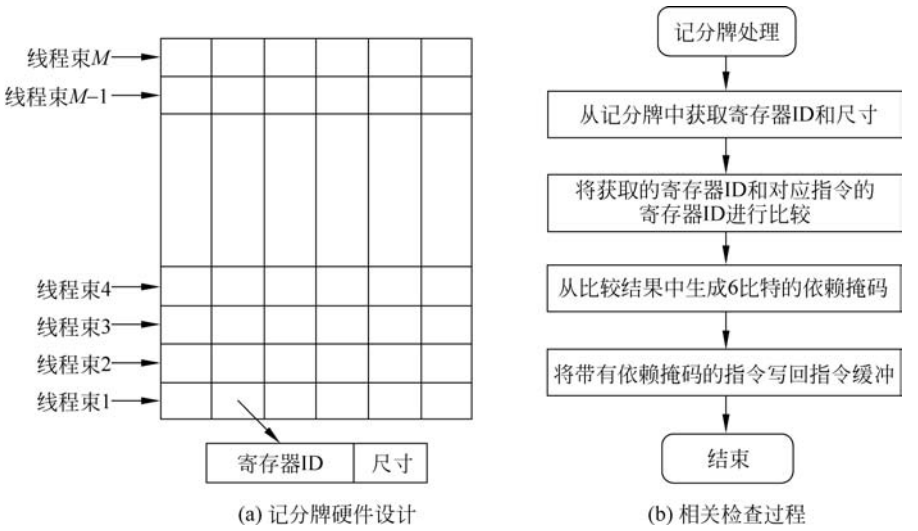


图 3-32　一种基于寄存器编号索引的记分牌硬件设计及相关性检查的过程

每个区域条目的数量也不是越多越好。通常，如果记分牌存储空间中的条目数量过多，就可能造成存储资源的浪费，导致类似简单记分牌的设计冗余。如果条目个数不足，那么能够同时处理的相关性冲突的寄存器数量就会减少，造成编译器寄存器分配的困难。条目个数不足也可能会导致为了保证没有相关性违例，后续指令在运行时需要等待前面指令来清空记分牌的某个条目才能发射，产生不必要的发射停顿。在上述 NVIDIA 的专利中，每个区域设定最多可以存储 6 个条目，而在文献[22]的研究中也发现，3～4 个条目基本可以满足大多数应用在实际运行中的需求。

图 3-32(b)显示了这一记分牌算法进行寄存器依赖性检查的过程。假设在某个时刻，有许多指令正在执行，则会有多个目的寄存器在记分牌存储空间中留有记录。为了发射下一条指令，它需要将该指令的源寄存器或目的寄存器 RID 及尺寸信息与记分牌记录的信息对比，如果相同则存在 RAW 和 WAW 冲突的风险，依赖性掩码的对应位会置为 1。得到的依赖性掩码会连同指令写入指令缓冲中，直到依赖性掩码全部清 0 才能发射该条指令，避免发生数据冲突。当执行单元完成某条指令后，对应的目的源寄存器 RID 信息也会在记分牌中消除，从而释放出所有具有相关性的指令。

相比之前基本的记分牌设计需要为每个寄存器分配 1 比特的标志位，这种基于 RID 编

码比对的记分牌设计避免了提到的两个问题。假设每个线程束最多拥有 128 个寄存器，那么需要 7 比特记录 RID。假设尺寸指示器最多支持 4 个连续寄存器写回，那么仅需要 2 比特。记分牌每个区域假设有 6 个条目，这样记分牌的一个区域只要(7+2)×6=54 个比特。每个可编程多处理器内部记分牌占用空间与指令缓冲的深度有关，因此这个方案需要的记分牌存储空间会小于之前的记分牌方案，也可以提高访问的并行度。如果每个线程束拥有更多的寄存器，那么这种方案将会更加节省开销。

实际上，这种记分牌编码方式主要通过寄存器编码的方式替代了原来的独热码(one-hot)方式来识别寄存器，同时限制未完成写回的寄存器数量(如 6 个)，从而减少了记分牌存储空间的开销。

2. 基于读写屏障的软件记分牌设计

在上述基于寄存器编号索引的硬件记分牌中，当一条新的指令准备发射时，需要搜索记分牌存储空间里对应线程束区域中的所有项目，以便根据寄存器编号确定寄存器之间是否存在相关性。事实上，这个过程还可以通过软硬件结合的方式进一步优化。研究人员对 NVIDIA 的 GPGPU 分析发现，其架构可以采用这样的软件记分牌设计：首先设计一定数量的读写屏障，借助编译器分析，显式地将存在相关性的寄存器绑定到某个读写屏障上；在运行时，目的寄存器的写操作可以直接设定绑定的读写屏障，而源寄存器的读操作需要读取绑定的读写屏障来获知该寄存器的写操作是否完成。由于这些信息由编译器提供，可以节省硬件开销，并降低搜索的代价，从而快速定位到绑定的读写屏障。

代码 3-4 给出了 NVIDIA Turing 架构下数据归约内核函数的一段 SASS 代码，可以帮助理解这种基于读写屏障的软件记分牌工作方式。根据 2.5 节所述，Volta 和 Turing 架构下每条指令的长度为 4 个字，即 128 比特。其中，64 个比特为本条指令的机器码，还有 64 个比特为控制码。编译器在 SASS 指令中通过控制码直接控制硬件的读写屏障，以解决数据冲突。本节基于文献[23]和文献[24]对控制码的分析和研究来解释这一过程。

代码 3-4 利用读写屏障实现记分牌功能的代码及其控制码

```
1   0X00000110   --:-:-:Y:1   IMAD.IADD R5, R0, 0x1, R7
2   0X00000120   --:-:-:Y:5   BAR.SYNC 0x0
3   0X00000130   --:-:-:Y:1   ISETP.GE.U32.AND P0, PT, R5, c[0x0][0x160], PT
4   0X00000140   --:-:-:Y:3   BSSY B0, 0x210
5   0X00000150   --:-:-:-:4   ISETP.GE.U32.AND.EX P0,PT,RZ,c[0x0][0x164],PT,P0
6   0X00000160   --:-:-:Y:2   ISETP.GE.U32.OR P0, PT, R8, R7, P0
7   0X00000170   --:-:-:-:4   SHF.R.U32.HI R7, RZ, 0x1, R7
8   0X00000180   --:-:-:-:6   ISETP.NE.AND P1, PT, R7, RZ, PT
9   0X00000190   01:-:-:Y:6   @P0 BRA 0x200
10  0X000001a0   --:-:-:Y:1   LEA R4, P0, R5, c[0x0][0x180], 0x2
11  0X000001b0   --:-:2:Y:3   LDG.E.SYS R6, [R2]
```

```
12  0X000001c0    --:-:-:-:8    LEA.HI.X R5, R5, c[0x0][0x184], RZ, 0x2, P0
13  0X000001d0    --:-:2:Y:2    LDG.E.SYS R5, [R4]
14  0X000001e0    04:-:-:-:8    IMAD.IADD R9, R6, 0x1, R5
15  0X000001f0    --:0:-:Y:2    STG.E.SYS [R2], R9
16  0X00000200    --:-:-:Y:5    BSYNC B0
17  0X00000210    --:-:-:Y:5    @P1 BRA 0x110
```

代码 3-4 左边一列代表了每条指令的地址，中间一列为 64 位的控制码，最右边是 SASS 指令的汇编形式。中间的控制代码又可以分割为 5 个字段：Wmsk:Rd:Wr:Y:S。

(1) S 称为停顿计数(stall counts)。在该版本 SASS 中占用了 4 位，表示 0～15 个时钟周期的停顿计数。停顿计数的主要目的是指导调度器多长时间才能调度下一条指令。对于许多指令，流水线深度为 6 个时钟周期。也就是说一般情况下，如果一条指令需要使用上一条指令的运算结果，需要在两条指令之间插入 5 条指令，否则就需要停顿 5 个时钟周期以避免 RAW 冲突。

(2) Y：称为让步标识(yield hint flag)，占用 1 位，主要用于指导调度器进行指令发射。如果这个标志位置为 1，意味着调度器会更加倾向发射其他线程束的指令。如果调度器已经准备好了其他线程束的指令，线程束指令间的切换在 GPGPU 中是不需要代价的。

(3) Wr：称为写依赖屏障(write dependency barriers)，占用 3 位，以编号形式代表 6 个屏障，用于解决 RAW 和 WAW 数据冒险。由于很多指令可能没办法预知延迟周期的数目，比如共享存储器和全局存储器操作的延迟数目就不固定，那么仅使用停顿计数可能无法保证一定能够消除指令间的数据冲突。因此，通过将该指令的目的寄存器绑定到某个写屏障并设置其状态，可以保护这个待写回的寄存器不会被提前读取，直到该寄存器写回完成才会解除绑定关系，将寄存器移出屏障，后续指令才能再次访问该寄存器的值，从而避免 RAW 和 WAW 数据冲突。例如，第 11 行及第 13 行的 LDG 指令，分别将 R6 和 R5 寄存器绑定到 2 号写屏障中，后续指令通过 Wmsk 字段标识查询到 2 号屏障的状态就可以决定是否能够读取 R6 和 R5 的值。

(4) Rd：称为读依赖屏障(read dependency barriers)，占用 3 位，以编号形式代表 6 个屏障，用于解决 WAR 数据冒险。与写屏障类似，控制码会将对应指令需要读取的寄存器绑定到某个读屏障中。在没有读取完成该寄存器的值之前，不允许其他指令对其进行修改，从而避免 WAR 数据冲突。例如，第 15 行的 STG 指令将寄存器 R9 绑定到 0 号读屏障中，后续向 R9 中写入数据的指令需要查询 0 号屏障就能知道读取 R9 的操作是否已经完成。值得注意的是，读依赖屏障实际上与写依赖屏障共享 6 个屏障。

(5) Wmsk：称为等待屏障掩码(wait barrier mask)，用于标明该指令需要查询哪个屏障。该掩码共有 6 位，每一位对应一个读写屏障。指令会等待处于置位状态的屏障，直到该屏障被清空，才能继续执行指令。例如，在第 14 行 IMAD 指令中，04(即 000100)对应了 2 号屏障(屏障号从零开始)。这行的控制码要求检测 2 号屏障是否被置位，即检测 R5 和 R6

寄存器中的值是否准备完毕才能继续执行指令，这样就避免了 RAW 数据冒险。

相比之前基于寄存器编号的硬件记分牌设计，这种软件记分牌设计节省了存储空间。原则上，每个线程束只要维护 6 个写屏障和读屏障就可以避免数据竞争和冒险。编译器通过将屏障编号编码到指令中，使得硬件记分牌只需要少量的解码逻辑就可以在运行时确定寄存器究竟在哪个屏障中。在运行时通过读取屏障状态，确定感兴趣的寄存器状态是否合适。相比于纯硬件实现的记分牌，这种方式避免了查询属于该线程束的所有条目。实际上，屏障的设立充当了寄存器编号的桥梁。由于屏障数目在内核函数中并不需要很多，因此这种方式能以较少的比特达到数据相关性检测的目的。

3.6 线程块分配与调度

在 GPGPU 编程模型中，线程块是一个重要的层次，有时也称为协作线程组(Cooperative Thread Array，CTA)。它是由一组线程或多个线程束构成的，是 CUDA 或 OpenCL 程序将任务分配给可编程多处理器(SM 或 CU)的基本任务单元。

3.6.1 线程块并行、分配与调度

线程块是由一个或多个线程束组成的，同一个线程块内部的线程束可以在块内进行同步操作。按照经典的 CUDA 和 OpenCL 编程模型，线程块之间应该是相互独立的，不应存在依赖关系[①]。因此，线程块可以自由地分配到任意一个可编程多处理器上，也可以在可编程多处理器上自由地被调度执行。线程块在编程模型上的独立性保证了它们的执行顺序不会影响到程序执行的结果。

为了能够执行线程块，GPGPU 架构首先应该关注的是线程块如何分配到各个可编程多处理器上。如图 3-2 所示，GPGPU 架构中的线程块调度器负责管理所有线程块的分配。当线程块调度器能够在某个可编程多处理器上分配一个线程块所需的所有资源时，它会创建一个线程块。这些资源包括线程空间和寄存器，还包括为其分配的共享存储器和同步栅栏等。这些资源的需求都由内核函数声明，线程块调度器会根据需求等待足够的资源，直到在某个可编程多处理器上可以分配这些资源运行一个线程块。然后每个线程块创建各自的线程束，等待可编程多处理器内部的线程束调度器开始调度执行。线程块调度器同时需要监控何时一个线程块的所有线程和线程束全部执行完毕退出，释放线程块共享资源和它的线程束资源，以便分配下一个线程块。

分配到可编程多处理器后，线程块的调度与线程束的调度之间存在密切的联系。线程束调度作为基本的调度粒度，会影响到一个可编程多处理器中线程束的执行情况，进而影响到线程块局部的执行。线程块的执行情况会反馈给全局的线程块调度器，进而影响线程块

① 如 2.4.2 节所述，CUDA 9.0 之后引入了协作组(cooperative groups)，允许在线程之间重新定义新的同步协作关系。

全局的执行速度。线程块的调度与初始线程块的分配也密切相关，因为调度的对象就是分配到给定可编程多处理器的线程块，因此分配方式也会影响调度的质量。例如，可以通过建立线程束调度器和线程块调度器之间的交互，改进每个可编程多处理器中线程块的分配方式和最大可分配的数量等。

线程块的分配和调度以最大化 GPGPU 的处理性能为主要目标，因此与线程束调度在策略上有很多相同之处。但总体来讲，两者支持的计算粒度不同，访存操作的考虑也有所不同。例如，线程束调度重点考虑的是可编程多处理器内部 L1 数据缓存的空间局部性。由于线程块中线程数目更高，空间局部性尺度更大，因此还会考虑 DRAM 的空间局部性。

3.6.2　基本的线程块分配与调度策略

线程块的分配和调度是 GPGPU 硬件多线程执行的前提。线程块的分配决定了哪些线程块会被安排到哪些可编程多处理器上执行，而线程块的调度决定了已分配的线程块按照什么顺序执行。两者关系密切，对于 GPGPU 的性能有着直接的影响。

1. 线程块的分配策略

在线程块分配方面，GPGPU 通常采用轮询作为基本策略。首先，线程块调度器将按照轮询方式为每个可编程多处理器分配至少一个线程块，若第一轮分配结束后可编程多处理器上仍有空闲未分配的资源（包括寄存器、共享存储器、线程块分配槽等），则进行第二轮分配，同理，若第二轮分配后仍有资源剩余，可以开始下一轮资源分配，直到所有可编程多处理器上的资源饱和为止。对于尚未分配的线程块，需要等待已分配的线程块执行完毕并将占有的资源释放后，才可以分配到可编程多处理器上执行。由于 GPGPU 执行的上下文信息比较丰富，为了方便管理并简化硬件，GPGPU 一般不允许任务的抢占和迁移，即当一个线程块分配给一个可编程多处理器之后，在其完成之前不会被其他任务抢占或迁移到其他可编程多处理器上执行。

图 3-33 描述了一个基于轮询的线程块分配示例。假设一个 GPGPU 中有 3 个可编程多处理器，分别为 SM0、SM1 和 SM2，每个 SM 允许最多同时执行 2 个线程块。一个内核函数声明了 12 个线程块 TB0～TB11。根据轮询的原则，TB0～TB2 被分配到 SM0～SM2。由于每个 SM 可以同时执行 2 个线程块，TB3～TB5 也被分配到 SM0～SM2 中。此时，SM 的硬件资源已经被完全占用，剩下的线程块暂时无法分配到 SM 中执行，必须等待有线程块执行完毕释放硬件资源，才能继续分配。一段时间后，SM2 中 TB5 率先执行完毕释放硬件资源，TB6 被分配到 SM2 中执行。之后 SM0 中 TB3 执行完毕，TB7 被分配到 SM0 中执行。最终线程块执行的

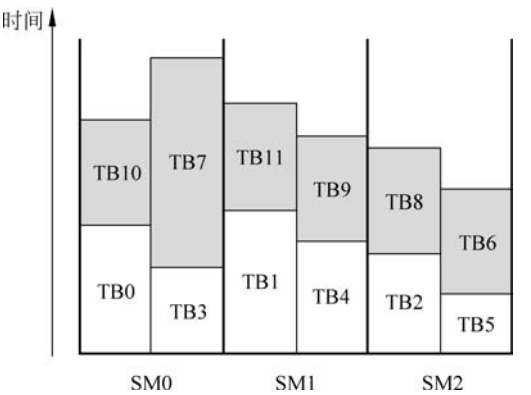


图 3-33　基于轮询的线程块分配示例

流程如图 3-33 所示。可以看到,初始一轮的线程块分配顺序还比较有规律,但第二轮的线程块分配完全是按照执行进度来安排的。

在 NVIDIA 的 GPGPU 中,线程块的分配由千兆线程引擎(giga thread engine)来管理,大体遵循轮询策略,但并不完全是朴素的轮询。例如,有研究对 M2050 GPGPU 上的线程块分配情况进行了实验分析。运行一段简单的向量加法的内核函数,通过内嵌汇编语句获得可编程多处理器的编号并输出。M2050 具有 14 个 SM,每个 SM 最多分配 6 个线程块。运行这段代码获得的分配结果如图 3-34 所示。大多数线程块按照轮询的方式分配到了相邻的 SM 上,但又并非朴素的轮询。出现这种情况的原因可能是在早期架构中,两个 SM 组成了一个纹理处理簇(Texture Processor Cluster,TPC)。实际 GPGPU 中线程块的分配可能还需要考虑 TPC,从而和轮询策略有些许不同。即便如此,大部分研究仍然以轮询策略作为线程块分配的基本策略,并基于此进行不同角度的研究和优化。

代码 3-5　线程块分配和调度顺序的测试代码

```
1    __global__ void
     vectorAdd(const float * A, const float * B, float * C, int numElements)
2    {
3        unsigned int ret;
         //将执行该线程块的 SM ID 写入变量 ret 中
4        asm("mov.u32 %0, %smid;" : "=r"(ret));
5        if (threadIdx.x == 0)
6            printf("BlockID: %d, SMID: %d\n", blockIdx.x, ret);
7        int i = blockDim.x * blockIdx.x + threadIdx.x;
8        if (i < numElements){
9            C[i] = A[i] + B[i];
10       }
11   }
```

	不同的SM													
分配的线程块	0	1	2	3	4	5	6	7						
	8	9	10	11	12	13	14	15	16	17	18	19	20	21
	22	23	24	25	26	27	28	29	30	31	32	33	34	35
	39	40	41	42	43	44	45	46	36	37	38	47	48	49
									50	51	52	53	54	55
	56	57	58	59	60	61	62	63	64	65	66	67	68	69
	70	71	72	73	74	75	76	77	78	79	80	81	82	83

图 3-34　NVIDIA M2050 上的线程块分配情况

基于轮询的线程块分配策略简单易行,而且保证了 GPGPU 中不同可编程多处理器之间的负载均衡,尽可能公平地利用每个可编程多处理器的资源。然而,轮询的分配策略也存在一定问题,比如可能会破坏线程块之间的空间局部性。一般情况下,相邻线程块所要访问

的数据地址由于与其线程 ID 等参数线性相关，很大可能会存储在全局存储器中连续的地址空间上，因此 ID 相近的线程块所需要的数据在 DRAM 或缓存中也相近。如果将它们分配在同一个可编程多处理器上，就可以访问 DRAM 中的同一行或缓存的同一行，利用空间局部性减少访存次数或提高访存效率。轮询的分配策略反而会将它们分配到不同的可编程多处理器上，导致相邻数据的请求会从不同的可编程多处理器中发起。如果随着执行时间的推进，线程块的执行进度有明显的差别，可能会降低访存合并的可能性，对性能造成不利的影响。

2. 线程块的调度策略

线程块的调度与线程束的调度策略有很高的关联性。两者对 GPGPU 的执行性能都有着重要的影响，所关注的问题也类似，只是调度的粒度有所不同。因此可以看到两者所采用的策略有很多相似之处，比如轮询调度策略，GTO 调度策略对于线程块的调度也同样适用。很多线程束调度的改进设计思想也可以应用在线程块调度问题上，或将两者联系起来作为一个整体来考虑。例如，通过建立线程束调度器和线程块调度器之间的交互，调度器更好地协调多个可编程多处理器之间的线程执行。

线程块的调度与线程块的分配策略也密切相关，分配方式也会影响到调度的质量。例如，每个可编程多处理器中线程块最大可分配的数量就与调度策略和执行性能相关。轮询的分配策略虽然具有公平性，但按照可编程多处理器允许的最高并行度将尽可能多的线程块分配执行，并不一定会提升应用的性能。很多研究统计表明，随着可编程多处理器中运行的线程块数目的增加，一些应用的性能只会缓慢提升甚至下降。

图 3-35 的例子对这个问题给出了直观的解释。假设有 4 个线程块 TB0～TB3 被分配到一个可编程多处理器上。图 3-35(a)中假设线程块和各自的线程束都按照 GTO 的方式

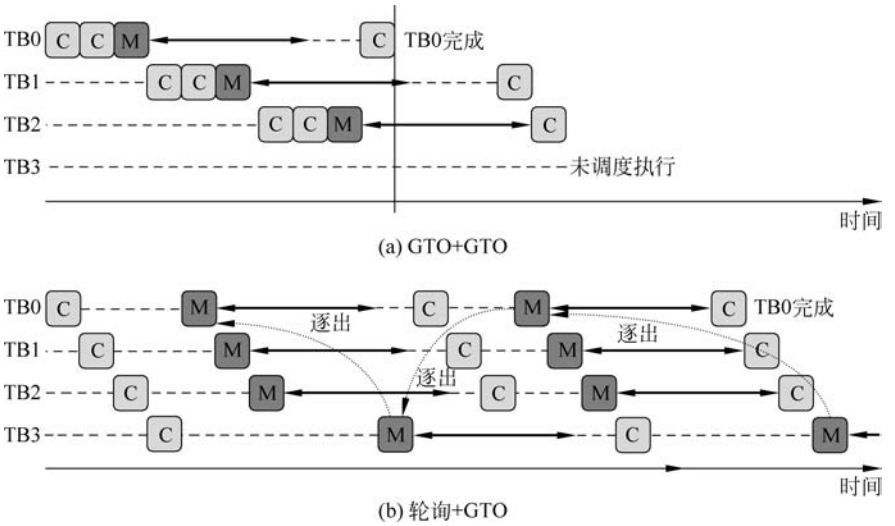


图 3-35　线程块采用不同调度可能出现的问题

进行调度。那么当一个线程块，如 TB0 执行遭遇停顿，此时会去调度其他线程块如 TB1、TB2 或 TB3 执行。由于线程块的计算执行相对较长，假设在 TB3 被调度之前，TB0 的长延时操作就已经完成，那么遵循 GTO 策略的调度器会倾向于重新执行 TB0，使得 TB3 不会得到调度。此时将 TB3 分配到这个可编程多处理器上其实对性能是没有帮助的，反而可能会由于分配了过多的线程块而导致资源紧张，因此可能会发生随着线程块数目的增加性能反而下降的情况。如果改变线程块的调度策略为轮询策略也同样存在问题，如图 3-35(b)就显示了这样一种情况，假设 TB3 和 TB0 读取的数据都存放在同一缓存行中，就会导致 TB3 和 TB0 在数据缓存上存在竞争。此时线程块的轮询调度会调度 TB3 执行，使得 TB0 刚刚访问返回的数据受到影响，因冲突缺失导致缓存抖动问题，增加了缓存缺失率和访问开销，也会导致随着线程块数量的增加性能反而下降的情况。因此独立的调度策略设计并不能解决这个问题，需要与线程块分配策略协同优化。例如，类似于线程束节流的方法，通过减少可编程多处理器中线程块的数量，也可以缓解这个问题。

3.6.3 扩展讨论：线程块分配与调度策略优化

线程块的分配和调度策略与 GPGPU 性能关系密切。本小节将针对简单的线程块分配和调度算法所暴露出的问题介绍几种设计优化的思路。这些优化的出发点主要是围绕 SIMT 线程地址所展现出的连续特性，进而在缓存和 DRAM 的局部性上寻求更优化的访存操作及在线程块分配进行限流等方面提高 GPGPU 资源利用率。

1. 感知空间局部性的调度策略

1) 感知 L1 缓存局部性的块级线程块调度

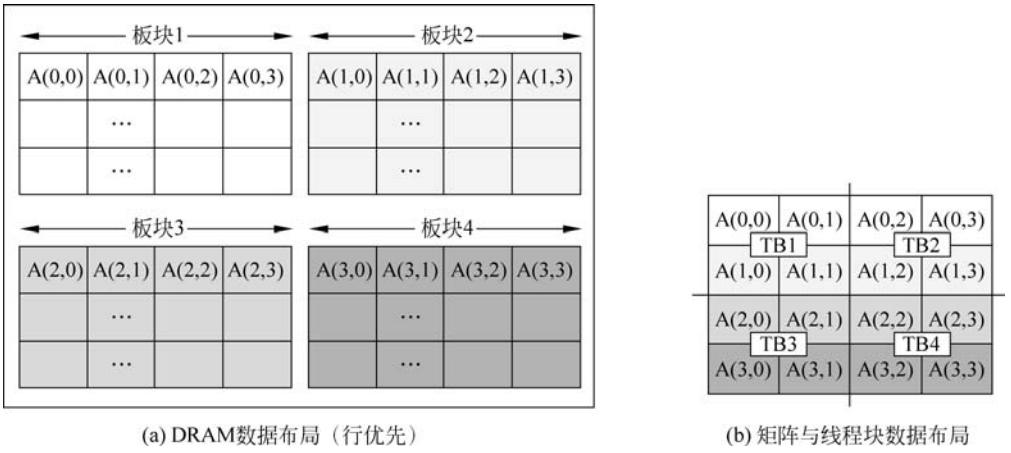
基本的轮询调度策略将连续的线程块分配到不同可编程多处理器上，可能导致线程块之间的数据局部性遭到破坏。针对这个问题，文献[26]提出了块级线程块调度(Block CTA Scheduling，BCS)和连续线程块感知(Sequential CTA-Aware，SCA)的线程束调度相配合的策略。前者意在将若干连续的线程块分配到同一个可编程多处理器上以充分利用线程块间的数据局部性，后者在线程束调度时兼顾线程块的调度，保持缓存的空间局部性。

为便于理解，假设内核函数中线程块按照二维结构配置，即每个线程块中包含 16×16 个线程，每个线程访问 1 个字(4 字节)的数据，因此线程块中一行访问的数据量为 16×4＝64 字节。一般情况下，L1 数据缓存行容量为 128 字节，由此可以得出相邻两个线程块的行数据可以共享一个缓存行，即线程块之间会存在空间局部性。但相邻的线程块由于会被轮询策略分配到不同的可编程多处理器上，破坏了这一空间局部性。即便将相邻的线程块分配到同一个可编程多处理器上，这一空间局部性也很难保证，原因在于分配到一个可编程多处理器上的两个连续线程块不一定具有相同的执行进度，二者执行结束的时间也各不相同。当其中一个线程块执行完成并释放资源后，简单地再调度一个新的线程块“补位”可能会导致后续线程块调度“错位”，也无法保证线程块间数据局部性得到有效利用。为此不得不采用一种“延迟”的调度策略，即等待连续的两个线程块都执行完毕后才调度新的线程块进入可编程多处理器。这便是块级线程块调度 BCS 策略的初衷。

与之对应，线程束的调度也应该考虑这种数据的空间局部性，有意识地调度连续线程块中的线程束以最大限度提高缓存行复用的可能，这便是连续线程块调度 SCA 策略设计的初衷。它结合了轮询和 GTO 调度：在连续两个线程块之间和一个线程块内部采用轮询策略进行调度，保证了数据的空间局部性。而在线程束执行过程中，采用 GTO 策略贪心地执行选中的线程束，直到其中一个线程束因长延时操作而停滞，才切换调度下一组线程束继续执行，后者保证了线程束原有的时间局部性得到有效利用。

2）感知 DRAM 板块的线程块协同调度

基本的轮询调度策略还可能增加 DRAM 板块访问冲突的风险。以矩阵数据的存储为例。假定采用行主序的方式存储矩阵数据，那么连续的数据会被存储在 DRAM 连续的地址。为提高访问的并行性，不同行可能会被存放在 DRAM 的不同板块中。如图 3-36(a)所示，假设 DRAM 配置有 4 个板块，矩阵第 1 行会存储在板块 1 中，第 2 行存储在板块 2 中，以此类推，第 5 行会再次存储在板块 1 中。当矩阵规模比较大时，编程人员往往会对矩阵进行分块处理，如图 3-36(b)所示。根据不同的分块规则，即便是矩阵同一行的数据，也很可能会被分配到不同的线程块中进行处理。当连续的线程块被分配到不同的可编程多处理器上并行访问时，它们可能会同时访问相同板块中不同位置的数据，由此引发板块冲突造成访存延迟增大、效率降低等问题。



(a) DRAM数据布局（行优先）　　(b) 矩阵与线程块数据布局

图 3-36　可能的 DRAM 数据布局和线程块数据布局

图 3-37(a)直观地说明了上述问题。其中连续的线程块 TB1 和 TB2 被分配到不同的可编程多处理器(SM1 和 SM2)上，二者可以并行执行。当发生访存操作时，理想情况下它们可以访问到某一板块中同一行的数据，以充分利用 DRAM 行缓冲区获得较高的命中率。为了证明这种现象的普遍性，文献[16]对 38 种典型的 GPGPU 应用，包括 SAD(Sum of Abs. Differences)、JPEG(JPEG Decoding)、SC(Stream Cluster)和 FFT(Fast Fourier Transform)等应用的访存行为进行了统计，发现相同 DRAM 行被连续线程块访问的频率为 64%，其中一些应用则更为突出，如 JPEG 解码中这一频率达到了 99%。但在实际执行

中，线程块执行的进度很可能发生失配，导致当 TB1 和 TB2 访问 DRAM 时就可能产生板块冲突的现象且造成行缓冲区无法发挥作用。值得注意的是，板块 3 和板块 4 始终处于空闲状态，连续线程块的访存并没有充分利用全局存储器的板块。

为了提高 DRAM 访问的并行度，应尽量防止板块冲突和读写资源闲置。以图 3-37(b)所示的情形为例，如果 SM1 和 SM2 分别选择不连续的 TB1 和 TB4 来执行，由于二者访问的数据存放在不同的板块内(TB1 的数据存储在板块 1 和 2 中，TB4 的数据存储在板块 3 和 4 中)，访存操作可以充分利用 DRAM 提供的 4 个板块提高读写的板块级并行度。

尽管这样的策略提高了 DRAM 访问并行度，但也破坏了线程块间的空间局部性，牺牲了数据复用的可能。为了弥补这一损失，还可以将那些已经被加载到 DRAM 行缓存区中却未被访问到的数据预取读入 L2 缓存中，以备后续的连续线程块读取使用。以图 3-37(c)所示的情形为例，在响应 TB1 和 TB4 发出的访存请求时，将 DRAM 激活行中的数据预取到 L2 缓存中。假设 TB2 和 TB3 的数据分别与 TB1 和 TB4 存放在 DRAM 的相同行中，那么 TB2 和 TB3 发出的访存请求完全可以被 L2 缓存捕获，而无须进一步访问下一级存储器，如图 3-37(d)所示。

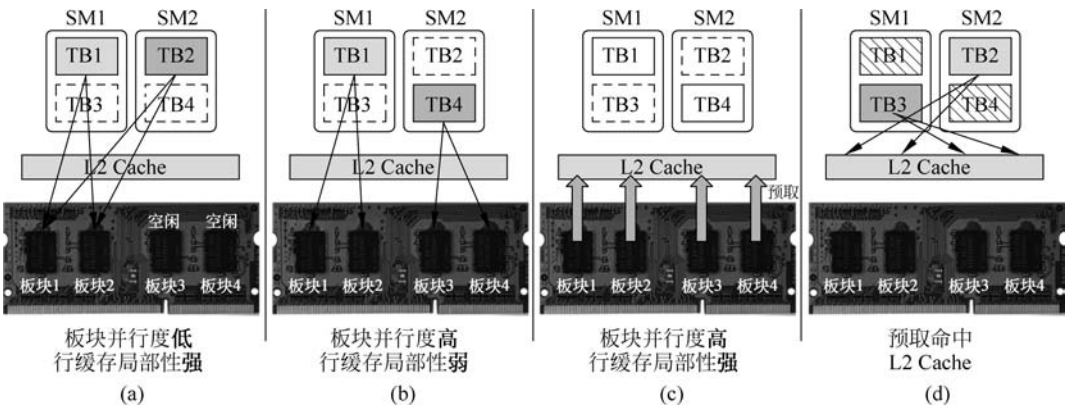


图 3-37 线程块轮询调度可能造成的 DRAM 板块冲突及解决方案

2. 感知时间局部性的抢占调度策略

轮询调度策略在 L1 数据缓存命中率和复用率方面也存在一定问题。GPGPU 中 L1 数据缓存的容量往往只有几十或几百 KB，远远无法满足大量线程块并行执行所产生的数据缓存需求，容易导致缓存冲突、抖动和缺失现象。

图 3-38(a)的例子展示了这样的情况：一个可编程多处理器上运行了 4 个线程块 TB1、TB3、TB5、TB7。按照轮询策略，调度器先选择 TB1 中的线程束调度执行，当其中线程束因长延时操作而陆续阻塞后，调度器调度 TB3 继续执行。若执行到 TB5 时，TB1 所需的数据刚好返回，此时 TB1 再度进入就绪状态。如果按照严格的轮询策略，TB1 需要等待 TB7 执行完成后才能再次得到调度。考虑到 L1 数据缓存容量十分有限，TB1 之前加载到缓存的数据很可能被后续执行的线程块替换掉。这些数据可能还没有得到有效复用，由此需要引

入反复的访存操作造成执行效率下降。

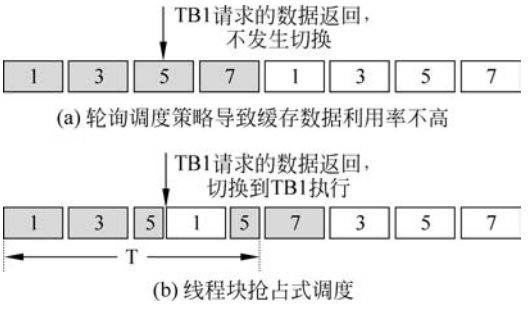


图 3-38　线程块轮询调度与抢占式调度

为此，文献[16]提出了一种解决方案：允许再次进入就绪状态的线程块抢占正在执行的线程块，即只要有一组线程块转为就绪状态，便赋予其最高优先级并立即开始执行，执行完成后再调度下一组线程块继续执行。以图 3-38(b)所示的情形为例：一旦 TB1 转为就绪状态，便抢占正在执行的 TB5，直到 TB1 中所有线程束完成后才将执行的优先权交还给 TB5。通过统计时间间隔 T 内执行的线程块数量，系统可以发现只有 3 个线程块被调度执行，少于轮询调度策略在相同时间内执行的线程块数量。这意味着更少的线程块可以更加充分地利用 L1 数据缓存，降低了未复用数据被提前替换的风险。该文献对 38 种应用进行了仿真实验，在抢占策略下 L1 数据缓存的命中率平均提高 18%，个别应用如 PVC(Page View Count)、IIX(Inverted Index)的命中率提升均达到 90%以上，对于这些访存密集型应用显著降低了缓存冲突发生的概率。

3. 限制线程块数量的怠惰分配和调度策略

保持较高的线程并行度有利于提高对长延时操作的容忍度，因此调度器倾向于给可编程多处理器分配更多的线程块。但前面的例子已经表明，当可编程多处理器所分配的线程块越来越多时，整个性能可能会呈现出“先上升、再平缓、后下降”的趋势，其原因主要包括资源竞争和缓存抖动等。因此，并不是分配的线程块越多越好，需要一种更加合理的线程块分配和调度策略，保证向可编程多处理器分配的线程块数量不但满足高资源利用率的需求，而且也能避免资源竞争所引发的负面影响。

3.6.2 节介绍的抢占式调度实际上是通过优先级的转换减少可编程多处理器上活跃的线程块数量。而文献[26]提出了怠惰线程块调度(Lazy CTA Scheduling，LCS)策略，动态地调整每个可编程多处理器上最多可承载的线程块数量达到类似的目标。LCS 策略主要包括以下三个步骤。

(1) 监视(monitor)。首先按照 GTO 策略对分配到可编程多处理器上的线程块进行调度，并全过程地监视第一个执行的线程块，在其完成所有指令的执行并退出时，记录每个线程块所发射的指令数量。

(2) 节流(throttle)。监视阶段结束后调度器会获得每个可编程多处理器内每个线程

块发射指令的数量。计算所有线程块执行指令的数量除以执行最多指令的线程块所发射的指令数量，得到每个可编程多处理器更为合理的线程块数量上限。

(3) 怠惰执行(lazy execution)。对于一个内核函数，当每个可编程多处理器中第一个线程块退出后，根据计算阈值限制每个可编程多处理器上最多可分配的线程块数量。由于同一个内核函数可能存在相同的计算特征，出于简化硬件设计的考虑，实际使用中可以只计算一个可编程多处理器的阈值并将其推广到所有可编程多处理器上。而对于不同的内核函数，由于彼此的计算特征存在差异，当新内核函数的线程块被分配到可编程多处理器上时，阈值需要重新计算。

进入怠惰执行阶段后，该文献还提出调度器对 1D 负载和 2D 负载实施不同的调度策略。其中，1D 负载指内核函数中线程块的组织形式为一维，而 2D 负载指内核函数中线程块的组织形式为二维。对于前者，对线程块和线程束分别采用轮询和 GTO 的策略进行调度。对于后者，基于前文介绍的块级线程块调度 BCS 与连续线程块感知 SCA。BCS 能够在线程块调度层面利用线程块间数据局部性，而 SCA 策略则在线程块内部的线程束之间利用数据局部性。实际上，怠惰调度方法的核心思想是通过怠惰线程块数量的计算保证资源的充分配给，在此基础上再利用线程块和线程束两级调度对空间局部性的感知来提升访存密集型应用的执行效率。

4. 利用线程块重聚类感知局部性的软件调度策略

GPGPU 中线程块调度策略的研究普遍以挖掘线程块间的局部性为主要目的。前面介绍的策略都是通过调度器硬件直接改变线程块及线程束执行的顺序来感知和优化局部性，而文献[27]则从另一个角度，提出利用软件手段通过线程块的聚类(clustering)或重构(shaping)来改善局部性。

该文献通过分析认为，线程块之间存在数据可复用的原因可以分为以下 5 种类型。

(1) 算法相关。即由特定算法引入的线程块间数据复用，例如 k-means 聚类算法、矩阵乘法及离散余弦变换等。

(2) 缓存行相关。这一类数据复用由缓存设计引入，类似于空间局部性。例如，当一个线程请求一个整型数据(4B)发生缓存缺失时，将会从存储器读取一整行缓存行(如 128B)数据送入 L1 数据缓存中。当存在其他线程块的线程访问其余 31 个整型数据时，L1 缓存即可完全“捕获”而无须访问下一级存储器。这类复用常发生于存在未合并的访存请求或访问数据没有对齐缓存行边界的情况。

(3) 数据相关。这一类局部性来源于不规则数据结构，如图、树、哈希表、链表等在存储器中的组织方式和访存规则。由于数据的不规则性属于数据本身的特性，而数据的来源是多样的，因此这类数据复用具有偶然性。典型的数据相关应用包括广度优先搜索、直方图和 B+树操作等。

(4) 写相关。这类应用可能存在线程块间的数据复用，然而若某个不相关的线程块修改了可能被复用的缓存行数据，旧的缓存行将被替代成新的缓存行，从而无法实现数据的复用。这种情况一般发生于一个内核函数读写相同一段数据，且访问距离小于一个缓存行的

长度时，由此会导致可复用数据被逐出的现象。

(5) 流。流应用的访存请求通常是经过合并和对齐的，然而其数据复用却仅存在于线程块内部(如通过共享存储器实现)。这类应用几乎没有线程块间的局部性。

根据数据复用的难易程度和概率，诸如算法相关(程序决定)和缓存行相关(架构决定)的应用可以在执行前判断出来，这两类应用的局部性是“可利用(exploitable)”的；而数据相关(数据决定)、写相关(存在局部性但难以被利用)和流(几乎没有局部性)的复用性并不显著，只能在运行时决定是否“可利用”或“不可利用”。

为了使线程块间的数据复用在 L1 缓存上发挥到最大限度，针对不同的数据复用类型应该采用不同的方法。图 3-39 展示了该文献提出的线程块调度框架，其中最左侧的 O 表示原内核函数，最右侧的 N 表示重聚类或重构出来的新的内核函数。

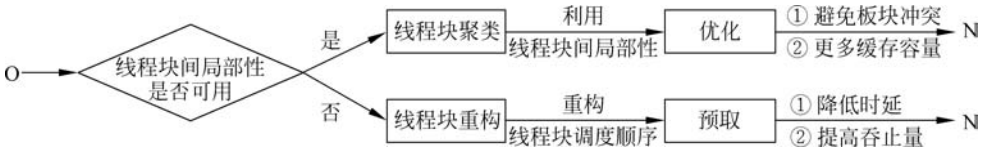


图 3-39　优化的线程块调度框架

(1) 对于“可利用”的局部性，利用聚类的方法发掘线程块间的局部性，让新内核函数 N 尽可能避免缓存冲突，获得更高的缓存容量。聚类的目的就是找到一个从 O 到 N 的映射。一种简单的策略就是假定 N 中线程块数量与 O 中线程块数量相等(即|N|=|O|，为 1 对 1 映射)的条件下，对 O 中的每一个线程块 u 重定向到 N 中的线程块 v。换句话说，将线程块 u 经过一系列变换操作转换到新内核函数 N 的线程块 v，用新生成的坐标 bx 和 by 分别代替原来线程块索引 blockIdx.x 和 blockIdx.y，从而实现重聚类下线程块到硬件的映射。

(2) 对于“不可利用”的局部性，采用重构模式为线程块重新规定一种特定的执行顺序，然后配合数据预取，实现降低访问延时、提高吞吐率，改进这些“不可利用”程序的执行性能。该文献提出一种软件线程块调度策略实现方法，实现从 O 到 N 的映射。

总之，通过聚类或重构形成的新内核函数保持了原有内核函数的功能，由于进行了面向线程块调度和数据复用的多种优化，这些内核函数具备了更好的数据复用的可能性。

参 考 文 献

[1] Nvidia. Guide D. Cuda C programming guide[Z]. (2017-06-01)[2021-08-12]. https://eva. fing. edu. uy/pluginfile. php/174141/mod_resource/content/1/CUDA_C_Programming_Guide. pdf.

[2] Tor M Aamodt, Wilson W L Fung, I Singh, et al. GPGPU-Sim 3. x manual[Z]. [2021-08-12]. https://gpgpu-sim. org/manual/index. php/Main_Page.

[3] Hennessy J L, Patterson D A. Computer architecture: a quantitative approach[M]. 5th ed. 北京：机械工业出版社，2012.

[4] Rogers T G, Johnson D R, O'Connor M, et al. A variable warp size architecture[C]. Proceedings of the

42nd Annual International Symposium on Computer Architecture(ISCA). IEEE,2015: 489-501.

[5] ElTantawy A,Aamodt T M. MIMD synchronization on SIMT architectures[C]. 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO). IEEE,2016: 1-14.

[6] Aamodt T M,Fung W W L, Rogers T G. General-purpose graphics processor architectures[J]. Synthesis Lectures on Computer Architecture,2018,13(2): 1-140.

[7] Diamos G F,Johnson R C,Grover V,et al. Execution of divergent threads using a convergence barrier: U. S. Patent 10, 067, 768[P]. (2018-09-04)[2021-08-12]. https://www.freepatentsonline.com/y2016/0019066.html.

[8] Fung W W L,Sham I, Yuan G, et al. Dynamic warp formation and scheduling for efficient GPU control flow[C]. 40th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO). IEEE,2007: 407-420.

[9] Fung W W L,Aamodt T M. Thread block compaction for efficient SIMT control flow[C]. 2011 IEEE 17th International Symposium on High Performance Computer Architecture(HPCA). IEEE, 2011: 25-36.

[10] Narasiman V,Shebanow M,Lee C J,et al. Improving GPU performance via large warps and two-level warp scheduling[C]. Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO). 2011: 308-317.

[11] Rhu M, Erez M. CAPRI: Prediction of compaction-adequacy for handling control-divergence in GPGPU architectures[C]. Proceedings of 39th Annual International Symposium on Computer Architecture (ISCA). IEEE 2012: 61-71.

[12] Brunie N,Collange S,Diamos G. Simultaneous branch and warp interweaving for sustained GPU performance[C]. 2012 39th Annual International Symposium on Computer Architecture(ISCA). IEEE,2012: 49-60.

[13] NVIDIA. RTX on the NVIDIA Turing GPU[Z]. [2021-08-12] https://old.hotchips.org/hc31/HC31_2.12_NVIDIA_final.pdf.

[14] NVIDIA. VOLTA: PROGRAMMABILITY AND PERFORMANCE[Z].[2021-08-12].https://old.hotchips.org/wp-content/uploads/hc_archives/hc29/HC29.21-Monday-Pub/HC29.21.10-GPU-Gaming-Pub/HC29.21.132-Volta-Choquette-NVIDIA-Final3.pdf.

[15] NVIDIA. NVIDIA Nsight Visual Studio Edition 4.1 User Guide[Z]. [2021-08-12]. https://docs.nvidia.com/nsight-visual-studio-edition/4.1/Nsight_Visual_Studio_Edition_User_Guide.htm.

[16] Jog A,Kayiran O,Chidambaram Nachiappan N, et al. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance[J]. ACM SIGPLAN Notices,2013,48(4): 395-406.

[17] Jog A,Kayiran O,Mishra A K, et al. Orchestrated scheduling and prefetching for GPGPUs[C]. Proceedings of the 40th Annual International Symposium on Computer Architecture(ISCA). 2013: 332-343.

[18] Rogers T G,O'Connor M, Aamodt T M. Cache-conscious wavefront scheduling[C]. 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO). IEEE,2012: 72-83.

[19] Liu J,Yang J,Melhem R. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers[C]. 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO). IEEE,2015: 383-394.

[20] Lee S Y,Arunkumar A,Wu C J. CAWA: Coordinated warp scheduling and cache prioritization for

critical warp acceleration of GPGPU workloads[C]. 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture(ISCA). IEEE,2015: 515-527.

[21] Coon B W,Mills P C,Oberman S F,et al. Tracking register usage during multithreaded processing using a scoreboard having separate memory regions and storing sequential register size indicators: U. S. Patent 7, 434, 032[P]. (2008-10-07)[2021-08-12]. https://www. freepatentsonline. com/7434032. html.

[22] Lashgar A,Salehi E,Baniasadi A. Understanding outstanding memory request handling resources in gpgpus[C]. Proceedings of 6th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies(HEART),IEEE,2015: 15-21.

[23] Nervana. Control Codes[Z]. [2021-08-12]. https://github. com/NervanaSystems/maxas/wiki/Control-Codes.

[24] Paweł Dziepak. On GPUs,ranges,latency,and superoptimisers[Z]. [2021-08-12]. https://paweldziepak. dev/2019/09/01/on-gpus-ranges-latency-and-superoptimisers/.

[25] Jia Z,Maggioni M, Staiger B, et al. Dissecting the NVIDIA volta GPU architecture via microbenchmarking[J]. arXiv preprint arXiv:1804. 06826,2018.

[26] Lee M,Song S,Moon J,et al. Improving GPGPU resource utilization through alternative thread block scheduling[C]. 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA). IEEE,2014: 260-271.

[27] Li A, Song S L, Liu W, et al. Locality-aware CTA clustering for modern GPUs[C]. 22nd International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). IEEE,2017: 297-311.