3.1 数据并行架构

不同的处理器架构使用各种策略来避免延迟。CPU经过优化，可以处理各种数据结构和大型代码库。CPU可以有多个处理器，但每个处理器都以串行方式运行代码，有限的SIMD向量处理是次要的例外。为了尽量减少延迟的影响，CPU的大部分芯片都由快速本地缓存组成，内存中充满了接下来可能需要的数据。CPU还通过使用智能的技术来避免延迟，例如分支预测、指令重新排序、寄存器重命名和缓存预取[715]。

GPU采取不同的方法。GPU的大部分芯片区域专用于大量处理器，称为着色器核心，通常数量以千计。GPU是一个流处理器，依次处理有序的相似数据集。由于这种相似性——例如一组顶点或像素——GPU可以以大规模并行的方式处理这些数据。另一个重要元素是这些调用尽可能独立，这样它们就不需要来自相邻调用的信息，也不共享可写的内存位置。这条规则有时会被打破以允许新的和有用的功能，但这种例外是以潜在的延迟为代价的，因为一个处理器可能会等待另一个处理器完成它的工作。

GPU针对吞吐量进行了优化，吞吐量定义为可以处理数据的最大速率。然而，这种快速处理是有代价的。由于专用于高速缓存和控制逻辑的芯片面积较少，每个着色器核心的延迟通常比CPU处理器遇到的延迟要高得多[462]。

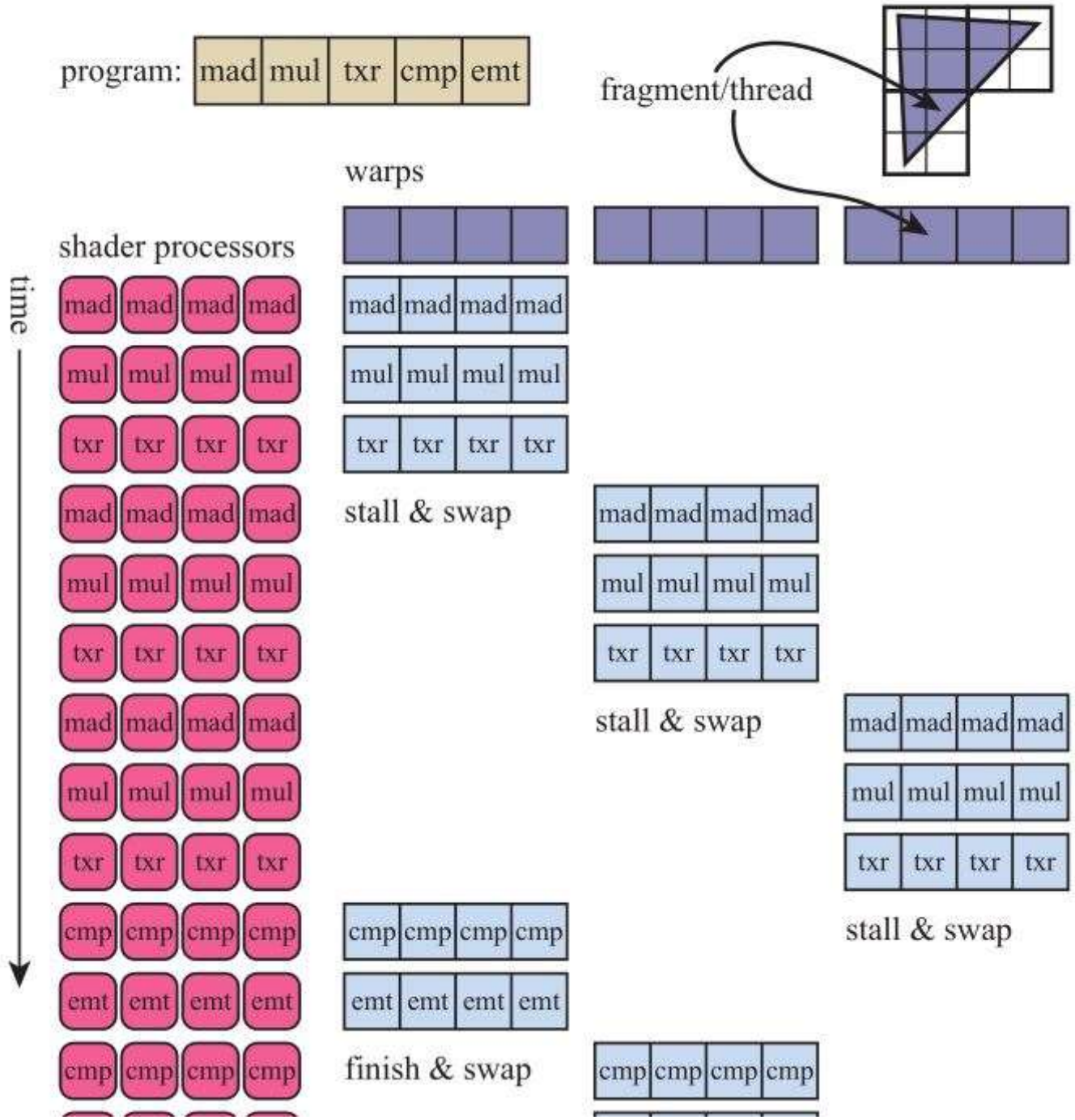
假设一个网格被光栅化，存在两千个像素有要处理的片元；一个像素着色器程序将被调用两千次。想象一下，只有一个着色器处理器，这是世界上最弱的GPU。它开始为两千个片段中的第一个片元执行着色器程序。着色器处理器对寄存器中的值执行一些算术运算。寄存器是本地的并且可以快速访问，因此不会发生延迟。然后着色器处理器接受了一条比如纹理访问的指令；一个例子是，对于给定的表面位置，程序需要知道应用于网格的图像的像素颜色。纹理是一个完全独立的资源，不是像素程序本地内存的一部分，纹理访问可能会在一定程度上非常耗时。一次内存获取可能需要数百到数千个时钟周期，在此期间GPU处理器什么也不做。此时着色器处理器将停止，等待返回纹理的颜色值。

为了让这个糟糕的GPU变得更好，给每个片元一个小的存储空间来存放它的本地寄存器。现在，着色器处理器不再停留在纹理获取上，而是允许切换并执行另一个片段，即2000个中的第二个片段。这个切换非常快，第一个或第二个片段中的任何东西都不会受到影响，除了注意哪个指令在第一个上执行。现在执行第二个片段。与第一个相同，执行一些算术函数，然后再次遇到纹理提取。着色器核心现在切换到另一个片段，第三个。最终所有两千个片段都以这种方式处理。此时，着色器处理器返回到第一个片段。此时纹理颜色已被获取并可使用，因此着色器程序可以继续执行。处理器以相同的方式继续执行，直到遇到另一条已知会暂停执行的指令，或者程序完成。单个片段的执行时间比着色器处理器专注于它的时间长，但片段的总体执行时间为整体大幅减少。

在这种架构中，通过切换到另一个片元让GPU保持忙碌，从而隐藏了延迟。GPU通过将指令执行逻辑与数据分离，使这种设计更进一步。这种设计方式被称为单指令多数据(SIMD)，能够在固定数量的着色器程序上以锁步方式执行相同的命令。SIMD的优势在于，与使用单独的逻辑和调度单元来运行每个程序相比，处理数据和交换所需的硅（和功率）要少得多。将我们的两千个片元示例转换为现代GPU术语，片元的每个像素着色器调用称为线程。这种类型的线程与CPU线程不同。它包含一些用于着色器输入值的内存，以及着色器执行所需的任何寄存器空间。使用相同着色器程序的线程被捆绑成组，NVIDIA称为warp，AMD称为wavefront。一个warp/wavefront被安排由一些GPU着色器内核执行，数量从8到64都行，使用SIMD处理。每个线程都映射到一个SIMD通道。

假设我们有两千个线程要执行。NVIDIA GPU上的warp包含32个线程。这产生了$2000/32 = 62.5$个warp，这意味着分配了63个warp，其中一个warp是半空的。warp的执行类似于我们的单GPU处理器示例。着色器程序在所有32个处理器上以锁步方式执行。当进行内存读取时，所有线程都会同时遇到它，因为对所有线程执行相同的指令。通常读取操作意味着这个线程warp将停止，所有线程都在等待它们的（不同的）结果。但是在GPU中却不会停顿，而是将warp交换为32个线程的不同warp，然后由32个内核执行。

这种交换与我们的单处理器系统一样快，因为在换入或换出warp时不会触及每个线程中的数据。每个线程都有自己的寄存器，每个warp都会跟踪它正在执行的指令。交换新的warp只是将一组内核指向一组不同的线程来执行；没有其他开销。如此warp执行或换出，直到所有工作完成。见图3.1。


program:
mad
mul
txr
cmp
emt
fragment/thread
warps
shader processors
time
stall & swap
stall & swap
stall & swap
finish & swap

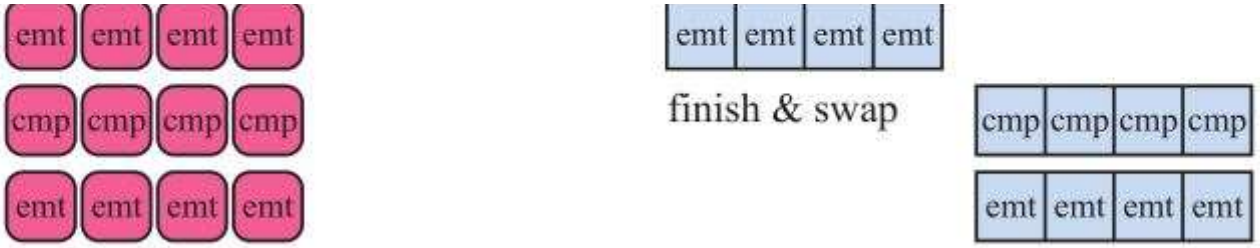


图3.1. 简化的着色器执行示例。三角形的片元，称为线程，组合成warp。每个warp显示为四个线程，但实际上有32个线程。要执行的着色器程序有5条指令长。四个GPU着色器处理器的集合为第一个warp执行这些指令，直到在“txr”命令上检测到停顿条件，这需要时间来获取其数据。第二个warp被换入并应用着色器程序的前三个指令，直到再次检测到停顿。在第三个warp被换入并停止后，通过交换第一个warp并继续执行来继续执行。如果此时其“txr”命令的数据尚未返回，则执行将真正停止，直到这些数据可用。每个warp依次完成。

在我们的简单示例中，纹理的内存获取延迟可能会导致warp被换出。实际上，遇到更短的延迟时都可以将warp换出，因为交换的成本非常低。还有其他几种技术用于优化执行[945]，但warp交换是所有GPU使用的主要延迟隐藏机制。这个过程的工作效率涉及几个因素。例如，如果线程很少，则可以创建很少的warp，从而使延迟隐藏成为问题。

着色器程序的结构是影响效率的重要特征。一个主要因素是每个线程的寄存器使用量。在我们的示例中，我们假设2000个线程可以同时驻留在GPU上。与每个线程相关联的着色器程序所需的寄存器越多，GPU中的线程就越少，因此warp也就越少。warp不足可能意味着无法通过交换来缓解卡顿。常驻的warp被称为“飞行中”，这个数字被称为占用率。高占用率意味着有许多warp可用于处理，因此空闲处理器的可能性较小。低占用率通常会导致性能不佳。内存获取的频率也会影响需要多少延迟隐藏。Lauritzen在文献[993]中概述了占用率如何受寄存器数量和着色器使用的共享内存的影响。Wronski在文献[1911, 1914]中讨论了理想的占用率如何根据着色器执行的操作类型而变化。

另一个影响整体效率的因素是由“if”语句和循环引起的动态分支。假设在着色器程序中遇到“if”语句。如果所有线程都评估并采用相同的分支，则warp可以继续而无需担心其他分支。然而，如果一些线程，甚至一个线程，采用备用路径，那么warp必须执行两个分支，丢弃每个特定线程不需要的结果[530, 945]。这个问题称为线程发散，其中一些线程可能需要执行循环迭代或执行warp中的其他线程不需要的“if”路径，从而使它们在此期间处于空闲状态。

所有GPU都实现了这些架构理念，从而导致系统具有严格的限制，但每功率的计算能力却非常庞大。了解该系统的运行方式将帮助你作为程序员更有效地利用它提供的功能。在接下来的部分中，我们将讨论GPU如何实现渲染管道、可编程着色器如何操作以及每个GPU阶段的演变和功能。