第五章 性能的影响因素

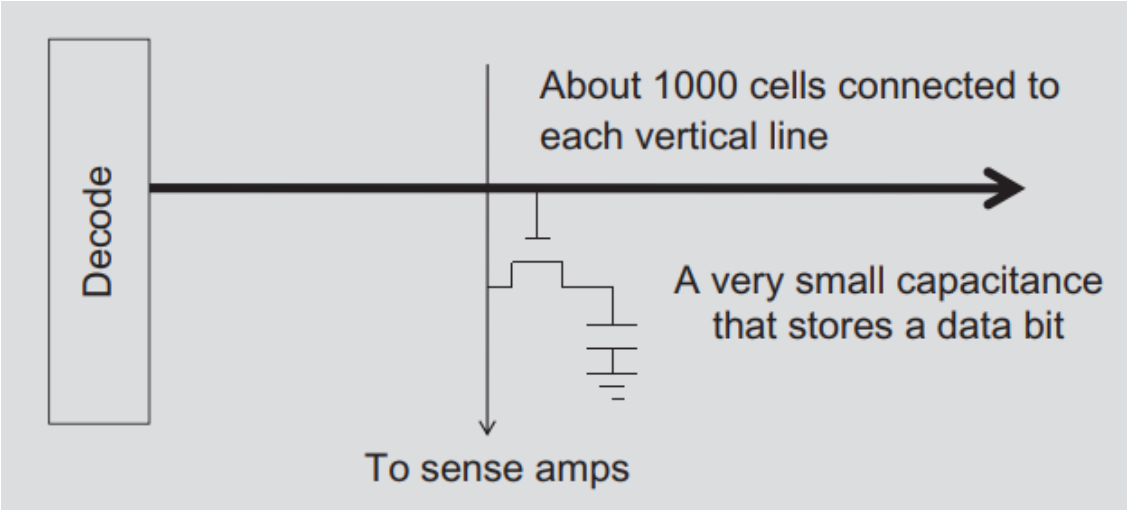
并行程序的执行速度很大程度上受到计算硬件的限制。虽然学习并行代码与硬件资源限制之间的关系对于高性能的编程模型十分重要，但是学习这个技巧最好的方法还是上手练习。在本章中，我们将要讨论CUDA device中的主要资源限制和它们是如何影响kernel执行性能的。为了达到性能目标，我们在设计程序时通常要让带程序的一次执行就高过预期的性能。在不同的应用中，不同的限制因素将成为更高性能的阻碍，我们称之为瓶颈。我们可以经常通过不同资源之间的制衡，从而动态地改进特定CUDA device上应用的性能。当缓解的资源是主要限制的资源，同时恶化的资源不产生负面影响时，这是一个很好的方法。如果没有这样的理解，性能调优就只能是猜测;合理的策略可能会也可能不会提高性能。除了深入了解这些资源约束之外，本章还进一步提供了一些原则和案例研究，旨在培养对能够带来高性能执行的算法模式类型的直觉。它还建立了一些习惯用法和思想，这些习惯用法和思想可能会在性能调优过程中带来良好的性能改进。

5.1全局内存带宽

CUDA kernel性能最重要的影响因素之一就是全局内存的带宽。CUDA kernel利用大量的数据级并行。自然，CUDA应用程序倾向于在短时间内处理来自全局内存的大量数据。在第4章内存和数据局部性中，我们研究了分块算法，它利用共享内存来减少每个线程区块中的一组线程必须从全局内存中访问的数据总量。在本章中，我们将进一步讨论内存合作的方法，从而更有效地将数据从全局内存移动到共享内存和寄存器中。内存合并技术通常与分块技术一起使用，从而使CUDA设备通过更有效地利用全局内存带宽来达到其性能潜力。

CUDA device的全局内存是通过DRAM实现的，数据按位被存储在DRAM单元中，这些单元是小型电容器，用存在或不存在少量的电荷来表示0和1。从DRAM中读数据需要小的电容器使用其微小的电荷来驱动通向传感器的高电容线，并启动其检测机制，该机制确定电容器中是否存在足够的电荷量才能符合“1”。这个过程在现代DRAM中需要10纳秒。这与半纳秒始终周期的现代device形成鲜明的对比。因为这与我们期待的数据访问速度相比太慢了，因此现代DRAM使用并行的方法来增加数据方位率，我们通常称之为内存访问通量。

> **为什么DRAM这么慢？**
>
> 下面这个图片展示了一个DRAM单元和访问其的路径。解码器是一种电子电路，它使用晶体管驱动一条线，连接到成千上万个电池的出口闸。它可能需要很长时间的线路，以充分充电或放电到所需的水平。



对于单元来说，更大的挑战是如何将垂直线驱动到检测放大器，并允许检测放大器检测其内容。 这基于电荷共享。 门释放出储存在电池中的极少量电荷。如果单元格含量为“ 1”，则少量电荷必须将长位线的大电容的电势升高到足够高的水平，从而可以触发读出放大器的检测机制。 一个很好的类比是，某人在长长的走廊的一端拿着一小杯咖啡，而另一个人闻到通过走廊传播的香气，以确定咖啡的味道。

通过在每个单元中使用更大，更强的电容器可以加快这一过程。 但是，DRAM的发展方向相反。 为了在每个芯片中存储更多的位，每个单元中的电容器尺寸逐渐减小，因此其强度逐渐降低。 这就是为什么DRAM的访问延迟没有逐渐随着发展减少的原因。

每当一个DRAM地址被访问，包括被访问地址的一段连续的地址被访问了。每个DRAM提供很多的感受器，并且这些感受器是并行工作的。每个DRAM芯片中提供了许多传感器，它们并行工作。 每个都在这些连续位置中感测到位的内容。 一旦被传感器检测到，来自所有这些连续位置的数据就可以以非常高的速度传输到处理器。这些被访问和传递的连续位置称为DRAM加速。 如果应用程序集中使用了这些加速数据，则DRAM可以以比访问真正随机位置序列更高的速率提供数据。

认识到现代DRAM的加速的组织情况，当前的CUDA设备采用了一种技术，该技术允许程序员通过将线程的内存访问组织为有利的模式来实现较高的全局内存访问效率。 该技术利用了以下事实：线程warp中的线程在任何给定的时间点执行相同的指令。 当warp中的所有线程都执行加载指令时，硬件将检测它们是否访问连续的全局内存位置。 即，当warp中的所有线程访问连续的全局存储位置时，将获得最有利的访问模式。

这种情况下，硬件建辉结合或合并所有这些线程，并对DRAM连续的位置进行连续的访问。举个例子，一个warp的加载指令，如果thread0访问全局内存位置为N， thread1的地址为N+1， thread2的地址为N+2，等等。所有的这些访问都会被合并，或者结合成单个访问来访问DRAM的连续位置，这种合并访问允许DRAM加速发送数据。

为了理解如何高效的合并硬件，我们需要复习一下在C语言中多维数组是如何产生连续的内存地址的。回忆第三章，图3.3，为了方便，图5.1对其复制了一份。在C和CUDA中，多维数组的线性化是基于行的连续内存空间分布的。“基于行”指的是数据保持行内元素的相互位置关系不变， 行内连续的元素在内存空间中也是连续分布的，在图5.1中，row0的4个元素先首先被按顺序放置到了内存中，然后row1被放置，然后row2，最后row3。应当清楚知道M0、0和M1、0虽然在二维数组中是相邻的，但是在线性化的内存空间地址中，而这中间空了4个元素。

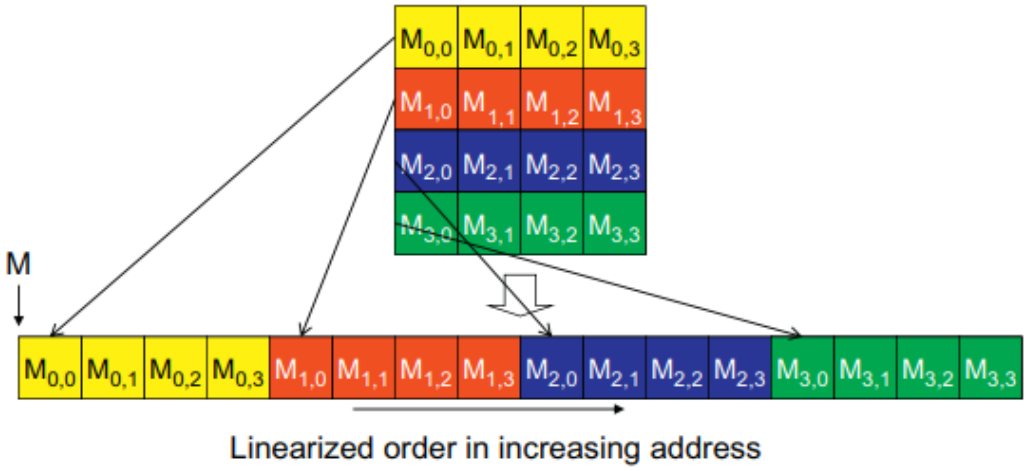


FIGURE 5.1

Placing matrix elements into linear order.

图5.2展示了CUDA喜欢的和不喜欢的基于行的二维数组的访问模式。回忆图4.7（这里应该是原文引用错了，译者觉得应该是图4.3），在我们的简单矩阵乘法kernel中，每个线程访问M的一行元素和N的一列元素，读者在继续看下去之前应当复习4.3节。图2.5(A)展示了对M元素的数据访问模式，其中warp中的线程读取连续的行，也就是说在第0个循环中，warp中的线程读取第0行的第0个元素，然后第1个循环，再一起读取第0行的第1个元素，因此并没有合并访问。而在图5.2B中，所有的线程访问第0行的相邻位置的元素，因此合并访问。

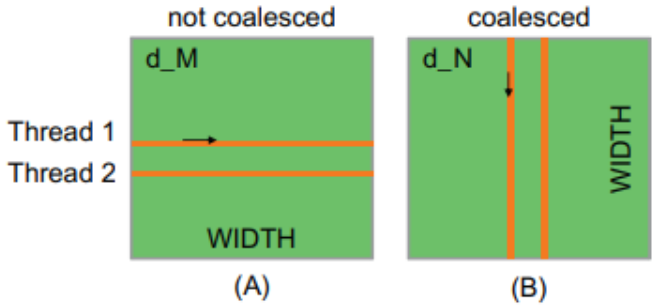


FIGURE 5.2

Memory access patterns in C 2D arrays for coalescing.

为了理解为什么图5.2B的访问模式相比于图5.1A更好，我们需要复习矩阵元素是如何访问的。图5.3展示了一个小的4*4的矩阵访问的例子。图5.3上面部分的箭头展示了kernel代码的访问模式。这个访问模式是图4.3中为访问N产生的。

```
N[k*Width + Col]
```

在给定的循环k中，k*Width的值都是相同的。 Col=blockIdx.x*blockDim.x+threadIdx.x。因为blockIdx.x和blockDim.x的值在所有相同区块中的线程中都是相同的，所以区块中不同线程只有threadIdx.x不同。因为连续的线程有着连续的threadIdx.x的值，所以他们访问的元素地址也是相连的。举个例子，在图5.3中，假设我们正在使用4*4的区块，warp的大小是4。也就是说，这种情况下，我们只是用1个区块来计算整个P矩阵，Width、blockDim.x和blockIdx.x的值分别是4、4和0。在第0个循环中，k的值是0，每个线程访问N的索引是：

```
N[k*Width+Col]=N[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
              =N[0*4 + 0*4 + threadidx.x]
              =N[threadIdx.x]
```

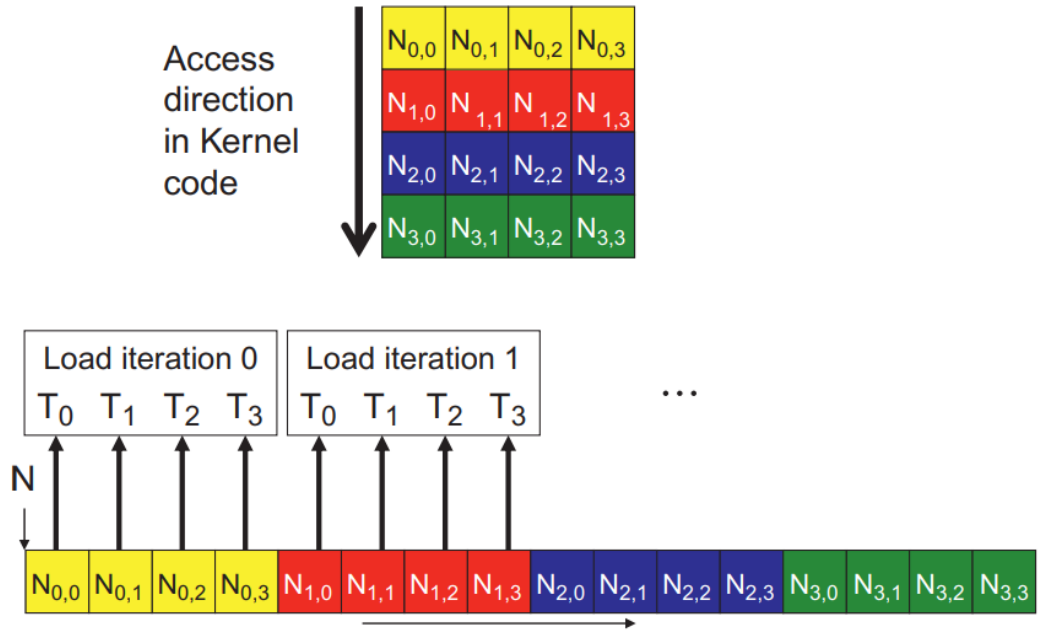


FIGURE 5.3

A coalesced access pattern.

也就是说，每个区块中，访问N的索引就是threadIdx.x的值，访问N元素的T0、T1、T2、T3分别访问的是N[0]、N[1]、N[2]、N[3]。这个在图5.3中的“Load iteration 0”中所展示。这些元素在全局内存中是连续的，硬件会检测到这些访问是同个warp中的不同线程对连续内存位置的访问。硬件会合并这些访问，形成综合访问，这使得DRAM支持更高的访问率。

在下一个循环中，k的值是1，每个线程对N元素的访问索引变成了：

```
N[k*Width+Col]  =N[k*Width+blockIdx.x*blockDim.x+threadIdx.x]
                =N[1*4 + 0*4 + threadidx.x]
                =N[4+threadIdx.x]
```

T0, T1, T2, T3所访问的N元素分别是 N[5], N[6], N[7], 和N[8]，如图5.3中的“Load iteration 1”所示。所有这些访问再次合并成综合访问，并改进了DRAM的带宽使用。

图5.4展示了一个不合并的矩阵数据访问模式。上部分的箭头展示了kernel代码每个线程访问行内元素的顺序。图5.4的上部分的箭头展示了kernel代码中一个线程的访问模式。这个访问模式是图4.3的对M元素访问所生成的。

```
M[Row*Width+k]
```

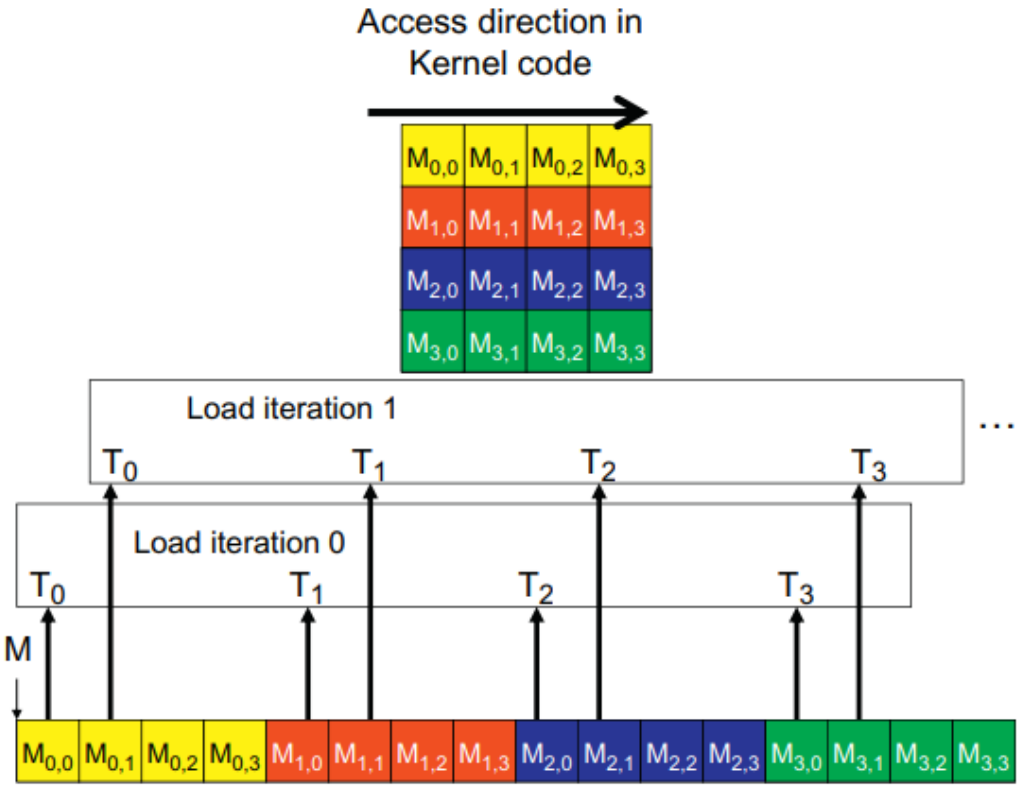


FIGURE 5.4

An un-coalesced access pattern.

在给定的循环k中，所有线程的k*Width值是相同的。回忆图4.3中，Row = blockIdx.y * blockDim.y + threadIdx.y。相同区块中所有的线程都有相同的blockIdx.y和blockDim.y， Row*Width+k中只有threadId.y的值不同。在图5.4中，我们在此假设使用4*4的区块，warp的大小是4， Width, blockDim.y, blockIdx.y的大小分别是4,4,和0。在第0个循环中，k的值是0.每个线程访问M元素的索引是：

```
M[Row*Width+k]  =M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
                =M[((0*4+threadIdx.y)*4 + 0]
                =M[threadIdx.x*4]
```

也就是说，访问M的元素的序偶因是threadI.x*4。访问M元素的T0, T1, T2, T3访问的分别是M[0], M[4], M[8], 和M[12]，如图5.4中“Load iteration 0”所展示的那样。这些元素在全局内存中并不是连续的位置。硬件不能将这些访问合并成综合访问。

在下一个循环中，k的值是1，每个线程用来访问M元素的索引是：

```
M[Row*Width+k]  =M[(blockIdx.y*blockDim.y+threadIdx.y)*Width+k]
                =M[(0*4+threadidx.x)*4+1]
                =M[threadIdx.x*4+1]
```

访问M元素的T0, T1, T2, T3访问的分别是 M[1], M[5], M[9], 和M[13]，如图5.4中的“Load iteration 1”所示。这次一样，这些访问不能合并成综合访问。

对于一个现实中的矩阵，每个维度通常会有上百甚至上千的元素。相邻线程在每次迭代中访问的M个元素可能相隔数百甚至数千个元素。底部的“Load iteration 0”框显示了线程如何在第0次迭代中访问这些非连续的位置。硬件将判断出这些元素的访问距离很远，并且不能合并。因此，当kernel循环遍历一行时，对全局内存的访问效率远低于kernel遍历列时的效率。

如果一个算法本质上需要kernel代码沿着行来遍历数据，我们可以用共享内存使得内存合并，这种技术叫做corner turning。如图5.5所示，每个线程从M读一行，这个模式不能合并访问。幸运的是tile算法能够让访问合并，正如我们在第四章所说的内存和数据本地化，同一区块中的线程会合作将内存的一小块加载到共享内存中。一定要注意保证这些小块是合并模式加载的数据。当数据在共享内存中是，他们可以

基于行或者基于列进行性访问，二者不会造成很大的差异，因为共享内存的本质是芯片上的高速内存，因此不存在通过合并访问提高数据访问率。

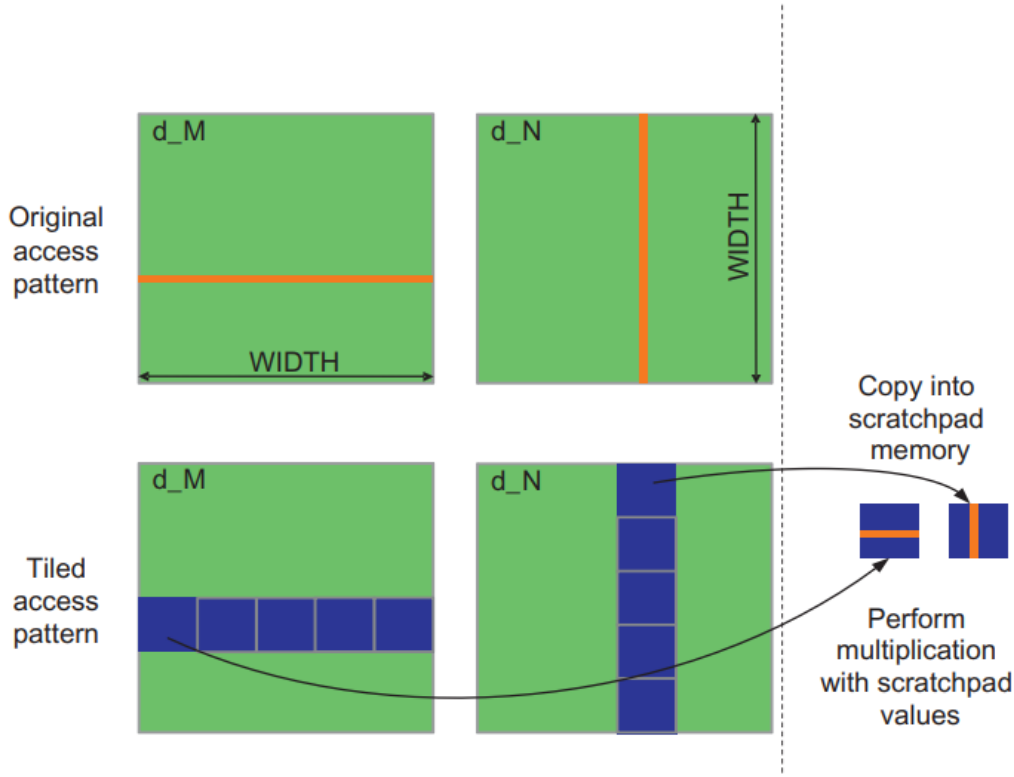


FIGURE 5.5

Using shared memory to enable coalescing.

我们为了方便，将图4.16重新放在这里作为图5.6，展示将M和N的元素加载到共享内存中的矩阵乘法kernel。回忆在每个阶段的开始（第9行到第11行），区块中的每个线程都负责加载M中的一个元素和N中的一个元素到Mds和Nds中。注意，每个小块中有TILE_WIDTH^2个线程，使用threadIdx.y和threadIdx.x确定具体加载哪个元素。

```
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
 1.   __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
 2.   __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

 3.   int bx = blockIdx.x;  int by = blockIdx.y;
 4.   int tx = threadIdx.x; int ty = threadIdx.y;

      // Identify the row and column of the P element to work on
 5.   int Row = by * TILE_WIDTH + ty;
 6.   int Col = bx * TILE_WIDTH + tx;

 7.   float Pvalue = 0;
      // Loop over the M and N tiles required to compute the P element
 8.   for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

        // Collaborative loading of M and N tiles into shared memory
 9.     Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     Nds[ty][tx] = N[(ph*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.       for (int k = 0; k < TILE_WIDTH; ++k) {
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
          }
14.       __syncthreads();
      }
15.   P[Row*Width + Col] = Pvalue;
    }
```

FIGURE 5.6

Tiled Matrix Multiplication Kernel using shared memory.

第9行M元素加载，每个线程使用ph来计算tile左侧元素的索引。线性化的索引计算与二维数组访问M[Row][ph*TILE_SIZE+tx]是一样的。注意线程所使用的列索引只有threadIdx不同。行索引是由blockIdx.y和threadIdx.y(第五行)所计算的，也就是说同一区块中的，相同blockIdx.y/threadIdx.y和相邻threadIdx.x的线程会访问相邻的M元素。TILE_WIDTH个线程加载小块的一行，这些线程threadIdx在y方向相同、x方向上的连续索引。硬件会将这些内存访问进行合并加载。

N的情况如下，对于有着相同threadIdx.y值的线程，ph*TILE_SIZE+ty相同。问题是相邻的threadIdx.x的值是否会访问N中连续的元素。注意每个线程的列索引是 Col=bx*TILE_SIZE+tx(第6行)。第一个部分bx*TILE_SIZE对于区块中所有的线程来说都是一样的。第二个部分tx就是threadIdx.x的值。因此，threadIdx.x连续的线程访问N中连续的行元素，硬件会合并这些访问。

注意，在简单算法中，threadIdx.x连续的那些线程访问垂直连续的元素，分块算法将之转化为另一个访问内存的模式，其中threadIdx.x连续的线程访问水平连续的元素，也就是说我们将垂直访问模式转换成了水平访问模式，我们通常称之为“Corner turning”。Corner turning还可以将垂直访问的模式转换成水平访问的模式，水平访问模式在FORTRAN的二维数组中很有用，因为FORTRAN的二维数组是基于列的线性化的。

在分块算法中，加载M和N元素都是合并的。因此，分块矩阵乘法算法相比于简单算法有两个好处：首先，因为使用了共享内存内存加载的数量减少了。其次，剩余的全局内存加载因为合并了，所以大大提高了DRAM带宽的使用。这两个优势互相影响，从而大大提高了分块算法的性能提升。在一个现代的device上，分块kernel内比简单算法快30倍左右。

图5.6的5,6,9,10行是分块算法中常用的将矩阵元素加载到共享内存中的编程模式。我们鼓励读者通过11行与12行的点乘循环俩分析数据访问模式。注意，warp中的线程不会访问Mds中的连续位置，这并不是有问题，因为Mds是共享内存，并不需要通过合并来提升数据访问效率。

5.2再讲讲内存并行

正如我们在5,1章节中所介绍的，DRAM加速是一种并行架构：多个相邻位置在DRAM中并行访问。但是对于现代处理器来说，单单加速并不能达到DRAM访问带宽的需求。DRAM系统还使用了另外两种形式的并行架构，banks和channels。在最高级的视角来看，一个处理器有一个或者多个channel，每个channel是内存控制者，通过一条总线链接一组DRAM bank。图5.7展示了一个有着4个channel的处理器。每个channel有一个总线连接着4个DRAM bank。在真实系统中，一个处理器一般有一个到八个channel，每个链接这大量的bank。

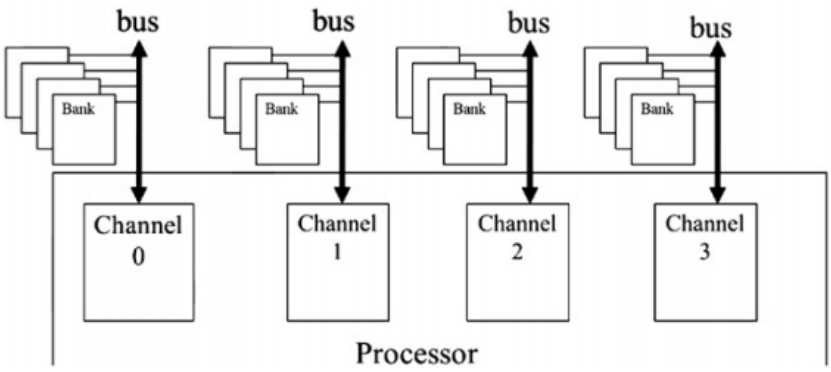


FIGURE 5.7

Channels and banks in DRAM systems.

一个总线的数据传输带宽取决于宽度和时钟频率。现代的DDR（double data rate）总线的性能是每个时钟周期两次数据传输，每个时钟周期的上升沿执行一次，而在每个时钟周期的下降沿执行一次。举个例子，一个64位的DDR总线用一个1GHZ的时钟周期，带宽是8B*2*1GHz = 16GB/sec。这看上去很大，但是对于现代CPU和GPU来说太小了。一个现代CPU需要至少32GB/s的带宽，一个现代GPU需要128GB/s的带宽，对于这个例子，CPU需要2个channel，GPU需要8个channel。

对于每个channel，它连接的bank的数量取决于要多少个bank才能将数据传输带宽用满。这个在图5.8中所示。每个bank由DRAM单元阵列、用于访问这些单元的传感放大器以及用于将数据突发传送到总线的接口组成。

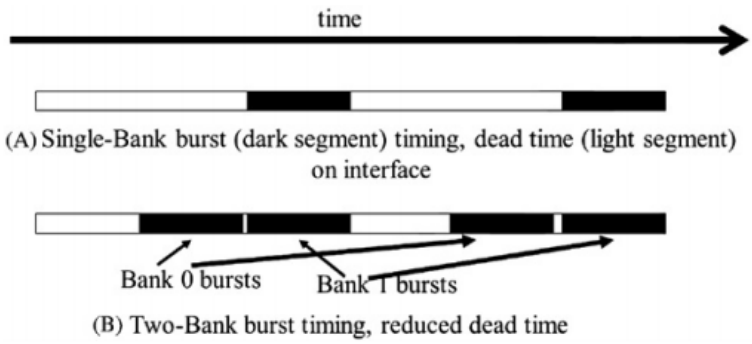


FIGURE 5.8

Banking improves the utilization of data transfer bandwidth of a channel.

图5.8A展示了当单个bank被连接到channel时的数据传输时间。可以看到，两个相连的DRAM内存单元读取访问的时间。回顾5.1章节中，每个访问要包括解码器启用单元，并使单元与传感放大器共享其存储的电荷，需要很长的等待时延。这个等待时间在图中用浅灰来表示，在时间轴线的左端。当传感放大器完成了它的工作，加速数据通过总线进行传输。传输的时间在途中用黑色表示。在第二个内存读取访问中，也会在数据传输之前有长时间的等待延迟。

在真实情况中，访问的延迟会比数据传输时间长，很显然一个bank的数据传输不能将channel总线完全利用。举个例子，如果DRAM单元阵列访问延迟与数据传输时间的比率是20:1，那么channel总线的使用率为1/21 = 4.8%，也就是一哥16GB/s的channel将会向处理器传输数据，速度不超过0.76GB/s。这是完全不能接受的。这个问题可以通过链接多个bank到同一个channel总线进行解决。

当两个bank连接到一哥channel总线时，当第一个bank为另一个访问服务时，第二个bank可以开始访问数据了，因此，我们可以将访问DRAM单元阵列的延迟覆盖起来。图5.8B展示的就是用两个bank的架构的时间线。我们假设bank0比图5.8B的时间轴开始得早一些，在第一个bank开始访问它的单元阵列之后，第二个bank同样开始访问它的单元阵列，当bank0的访问完成后，它将加速数据传输出去（左侧的黑色部分第一个），当bank0完成了它的数据传输之后，bank1能够传输它的加速数据（左侧第二个黑色部分）。这个模式在下一次访问是重复出现。

从图5.8B中可以看出通过两个bank，我们能够潜在地使用两倍的channel总线的数据传输带宽。总的来说，若单元阵列访问延迟与数据传输时间比率为R，我们需要有至少R+1个bank来保证完全利用channel总线的数据传输带宽。举个例子，如果比率是20，那么我们至少要21个bank连接到总线上。在真实情况中，每个总线上链接bank的数量需要比R大，有两个原因：首先，更多的bank减少了相同bank的多个同时访问的概率，也就是避免了bank冲突，因为每个bank同一时间只能服务一个访问，单元阵列就不能将这些冲突的访问等待时延给覆盖掉，有大量的bank增加了这些访问被分给大量bank的概率。其次原因是每个单元阵列的大小是为了达到合理的延迟和特定的制作工艺而设计的，这就限制了每个bank所能提供的单元个数。为了满足指定的内存大小，我们可能需要大量的bank。

在线程的并行执行与DRAM系统的并行架构之间有一个很重要的关系。为了达到device指定的内存访问带宽，就必须要有充足数量的线程同时访问内存。而且，这些内存访问必须被等分到channel和bank中。当然，对bank的每个访问都必须向我们在5.1章节中学的那样，是合并访问。

图5.9展示了将M元素分配到channel和bank的简单例子。我们假设一个加速传输两个元素，也就是8字节。这种分布是硬件设计的，channel和bank的地址是：数组最开始的8个字节（M[0]和M[1]）被存储在channel0的bank0中，再下面8个字节(M[2]和M[3])在channel1的bank0，在下一个是channel3的bank0

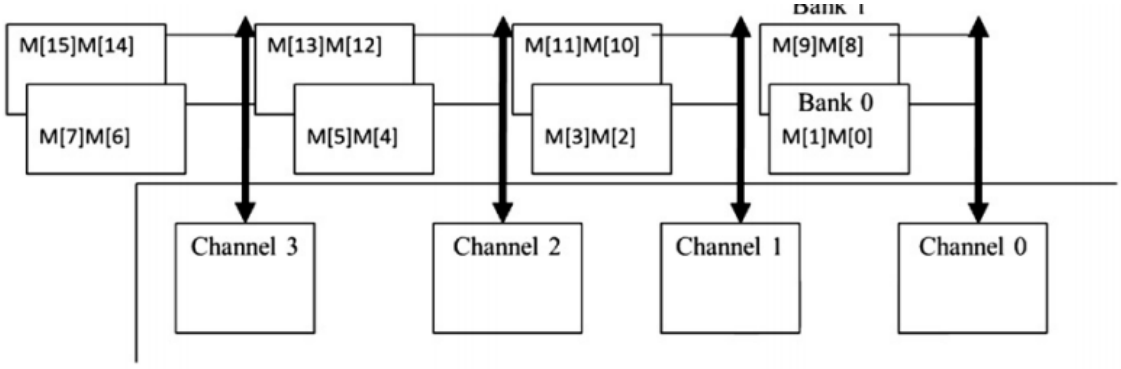


FIGURE 5.9

Distributing array elements into channels and banks.

再然后，像换行一样，重写再给channel0分配，但是不用bank0了，而是用bank1来访问下面8个字节。用这种方式，M[10] 和M[11]会在channel1的bank1中，M[12] 和M[13]会在channel2的bank1中，M[14] 和M[15]将会在channel3的bank1中。尽管在途中没有显示，我们应当知道，再增加元素的话，应当是在重新分配给channel0的bank0.举个例子，如果有M[16] 和M[17]的话，那将会被存储到channel0的bank0中，M[18] 和M[19]将会存储到channel1的bank0中，如此类推。

图5.9中所展示的分配方法通常被称为交错数据分配（interleaved data distribution）。将数据分配给系统中不同的bank和channel。这个方法保证了即使小的数组也能够分配分很好。也就是说，我们只是在充分利用完channel0的bank0的DRAM加速后，再转到channel11的bank0。在我们的例子中，只要我们的元素个数大于16个们就能将所有的channel和bank都利用上。

我们现在要讲解并行线程执行与并行内存架构之间的关系。我们使用图4.9的例子，我们在这里用5.10展示。我们假设这是一个2*2区块和2*2小块的矩阵乘法。

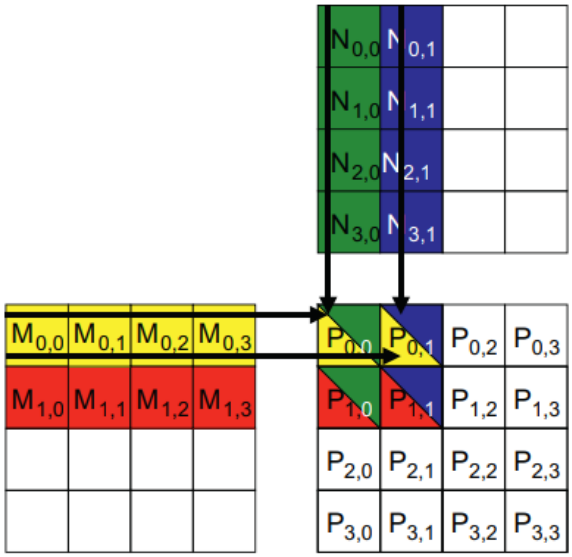


FIGURE 5.10

A small example of matrix multiplication (replicated from Fig. 4.9).

在kernel执行phase0时，所有的线程都加载第一个小块。每个小块中的M元素在图5.11中展示了出来。第二行展示了Phase0的访问的M元素，用二维索引展示了出来。第三行展示了相同而M元素的线性化索引。假设所有的线程区块都是并行执行的，我们可以看到每个区块将会产生两个合并访问。

Tiles loaded by	Block 0,0	Block 0,1	Block 1,0	Block 1,1
Phase 0 (2D index)	M[0][0],M[0][1], M[1][0], M[1][1]	M[0][0], M[0][1], M[1][0], M[1][1]	M[2][0], M[2][1], M[3][0], M[3][1]	M[2][0], M[2][1], M[3][0], M[3][1]
Phase 0 (linearized index)	M[0], M[1], M[4], M[5]	M[0], M[1], M[4], M[5]	M[8], M[9], M[12], M[13]	M[8], M[9], M[12], M[13]
Phase 1 (2D index)	M[0][2],M[0][3], M[1][2], M[1][3]	M[0][2],M[0][3], M[1][2], M[1][3]	M[2][2], M[2][3], M[3][2], M[3][3]	M[2][2], M[2][3], M[3][2], M[3][3]
Phase 1 (linearized index)	M[2], M[3], M[6], M[7]	M[2], M[3], M[6], M[7]	M[10], M[11], M[14], M[15]	M[10], M[11], M[14], M[15]

FIGURE 5.11

M elements loaded by thread blocks in each phase.

根据图5.9的分布，这些合并访问将会分配给channel0的两个bank和channel2的两个bank。这四个访问将会并行执行，从而使用两个channel，并增加每个channel数据传输的带宽。

我们还看到，Block0、0和Block0、1将会加载相同的M元素，大多数现代device都有缓存，只要二者的执行时间相差不多，就能够将这些访问合并成一个。实际上，GPU device的缓存主要是为了结合这些访问并减少对DRAM系统的访问而设计的。

第4行和第5行展示了在phase1的M元素加载的kernel执行。我们看到这些访问被channel1的bank和channel3的bank执行了。这些访问还是并行的，读者应当体会到了线程的并行执行和DRAM系统的并行结构之间的关系。一方面，DRAM系统带宽的充分利用需要大量的线程通过不同的bank和channel进行同时的访问。另一方面，device执行通量又需要更好的使用DRAM系统的并行架构。举个例子，如果同时执行的线程都访问相同的channel，内存访问通量和总的device执行速度都会大幅减小。

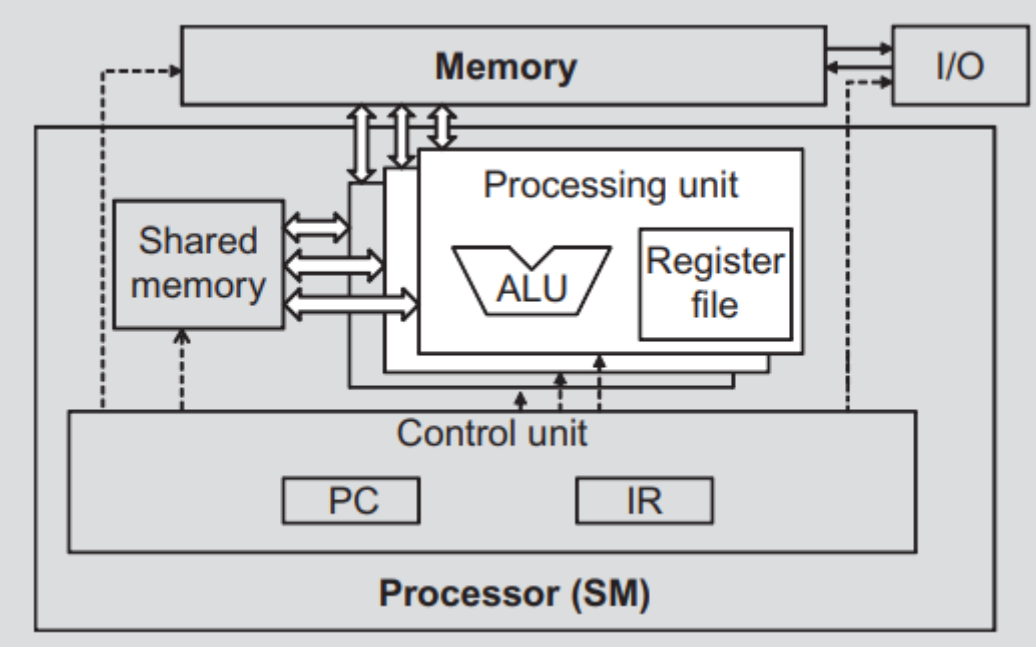
鼓励读者证明8*8的矩阵乘法，同样是2*2的线程区块，会使用图5.9中的所有四个channel。同样，更大的DRAM加速大小，会需要更大的矩阵乘法，从而完全使用所有channel的数据传输带宽。

5.3Warp和SIMD硬件

我们现在讲注意力转移到线程的执行是如何影响性能的。回忆当加载一个CUDA kernel的时候，会产生一网格的线程，这个网格又由两层结构组成。最上层是网格由一维、二维或者三维的区块阵列组成，在底层，每个区块又是由一维、二维或者三维的线程阵列组成。在第三章中，我们知道区块是完全不按顺序执行的，从而使得不同的device上有跨设备的透明可扩展性。但是我们没有详细讲每个区块中线程的执行时间。

从概念上讲，应该假设一个块中的线程可以相对于彼此以任何顺序执行，在分阶段的算法中，应当使用同步阻碍函数，来保证所有的线程完成的特定的阶段的执行再进行下一阶段。kernel执行的正确性不应该依赖于某些线程的同步执行。说到这里，我们还想指出，由于各种硬件成本的考虑，当前的CUDA设备实际上绑定了多个线程来执行。这种实现策略会导致某些类型的kernel代码受到性能限制。对于应用程序开发人员来说，我们需要将这些受到限制的代码改造成性能更好的类型。

正如我们第三章讲到的，每个区块都分成warp。Warp是通过SIMD执行的，这些实现技术帮助我们减少硬件的制造成本，减少了执行时的电力消耗，并能够合并内存访问。在可见的未来，我们期待warp技术将还会是流行的技术。但是warp的大小会根据应用的不同而不同。到目前为止，所有的CUDA device使用相似的warp参数，也就是每个warp32个线程。

> **Warp和SIMD硬件**
>
> 下图展示了将线程打包成warp执行的过程。处理器只有一个控制单元，用来获取和解码指令。相同的控制信号传递给多个执行单元，每个都执行单元都执行warp中的一个线程，既然所有的执行单元都通过相同的指令控制，他们执行的区别就只有寄存器文件中的执行数据。这个设计叫做单指令多数据（SIMD：single-instruction-multiple-data）。举个例子，尽管所有的执行单元被一个指令控制：`add r1, r2, r3`。但每个执行单元中r2和r3的值是不同的。
>
> 
>
>
> 现在的处理器中控制单元十分的复杂，包括十分复杂的逻辑来获取指令并访问指令寄存器的端口，还包括芯片上的指令缓存来减少指令获取的延迟。多个处理单元来共享控制单元能够减少硬件制作成本和电力消耗。
>
> 因为处理器越来越被能源消耗所限制了，新的处理器越来越喜欢使用SIMD设计。实际上，我们在未来可以看到更多的处理单元共享一个控制单元。

区块根据线程被分解成warp，如果一个区块是一个一维的阵列，也就是说只用了threadIdx.x的值，分解过程就很简单明了，一个warp中的threadIdx.x是连续的并且逐渐增加的，对于32的warp大小，warp0 从thread0开始，到thread31结束，warp1从thread32开始，到thread63结束。总的来说warp n 从thread32*n开始，到thread 32(n+1)-1结束，对于一个大小不是32整数倍的区块来说，最后的warp将被填充至32个线程。举个例子，如果一个区块有48个线程，它将会被分解成两个warp， warp1将会补充16个额外的线程。

对于多维线程组成的区块，在分解成warp之前先要将多维线性化转换成基于行的一维。线性顺序是：将y和z坐标较大的行放在y和z坐标较低的行之后。也就是说，如果一个区块包含两个维度的线程，那么就要将threadIdx.y为0的线程放在threadIdx.y为1的线程的前面，同理threadIdx.y为2的线程要放在threadIdx.y为1的线程的后面，以此类推。

图5.12展示了将二维区块放置成线程顺序的例子。上面的部分展示了区块的二维情况，读者应当看出来这是一个基于行的二维数组。线程标记为Ty、x，其中x表示threadIdx.x， y表示threadIdx.y。图5.12的下半部分展示了区块的线性化情况，最开始的4个线程是threadIdx.y为0的线程，按照threadIdx.x升序排列，再下4个是threadIdx.y为1的线程，同样按照threaIdx.x的升序排列。对于这个例子，所有的16个线程形成了一个warp，这个warp会与另外的16个warp合并组成一个32线程的warp。假设一个二维的区块由8*8的线程，64个线程将形成2个waro。第一个warp从T0、0到T3、7，第二个warp从T4、0到T7、7。读者可以画出这个图来理解这一过程。

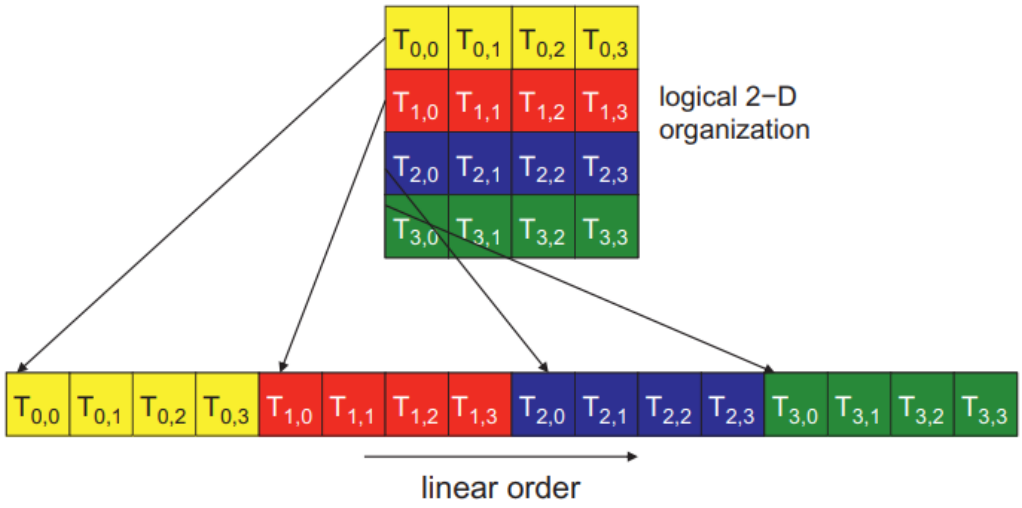


FIGURE 5.12

Placing 2D threads into linear order.

对于一个三维的区块，我们先把所有的threadIdx.z值为0的线程线性化，这些线程被当做如图5.12的二维区块。threadIdx.z为1的所有线程排在这之后，以此类推。对于一个三维大小为2*8*4的线程区块，64个线程将会被分解成两个warp，第一个从T0、0、0到T0、7、3，第二个从T1、0、0到T1、7、3。

SIMD硬件将打包执行warp中的所有的线程。一个指令在同个warp中的所有的线程中执行。当warp中所有的线程都组训相同的执行路径，或者更官方的说法是控制流时，它能够很好的工作。举个例子，对于一个if-else表达式，当所有的线程都通过if或者都通过else时，执行的会很好。当warp中的线程通过if-else的不同部分时，SIMD硬件将通过多个路径来多次通过。遍执行在if部分之后的那些线程，另一遍执行在else部分之后的那些线程。 在每次通过期间，走另一条路径的线程均不生效。 这些路径是顺序排列的，因此会增加执行时间。

多路径的方法执行不同warp扩展了SIMD硬件的完成完整CUDA线程语义的能力，当硬件对warp中所有线程执行相同指令时，它有选择地让线程仅在每次通过时才生效，从而允许每个线程采用自己的控制流路径。这样既可以保持线程的独立性，又可以利用SIMD降低硬件成本。

当同一个warp中的线程执行不同的路径时，我们说这些线程异化了它们的执行。在if-else的例子中，异化发生于部分线程通过了if而部分线程通过了else。异化的代价是硬件需要采取的额外操作，以便允许线程束中的线程做出自己的决定。

异化还发生于其他的构造中，例如：如果warp中的线程执行一个for循环（不同的线程可能迭代六次，七次或八次），所有的线程将一起完成前6个遍历过程。两个路径将会用来处理第7个遍历过程，其中一个通过了第七层遍历，其中一个没有通过。两个路径来处理第8个遍历过程，其中一个通过了第八层遍历，其中一个没有通过。

可以通过检查控制构造的决策条件来确定它是否会导致线程异化。如果决策条件是基于threadIdx值的，那么控制语句可能会导致线程异化例如，语句if (threadIdx)。x > 2){}使线程遵循两个不同的控制流路径。线程0、1和2的路径与线程3、4、5等不同。类似地，如果循环条件基于线程索引值，则循环可能导致线程异化。

使用带有线程异化的控制构造的一个普遍用途是在将线程映射到数据时处理边界情况。这通常是因为线程总数需要是块大小的倍数，而数据大小可以是任意数。从图2.12中的向量加法核函数开始，我们在addVecKernel中有一个if (i<n)语句。这是因为并不是所有的向量长度都可以表示为块大小的倍数。例如，假设向量长度为1003。假设我们选择64作为区块大小。需要启动16个区块来处理所有1003个向量元素。但是，这16个区块将有1024个线程。我们需要禁用线程区块15中的最后21个线程来执行原程序不期望/不允许的工作。请记住,这些16块划分为若干个32warp。只有最后的warp才会有控制异化发生。

注意，控制异化对性能的影响随着被处理向量的增大而减小。对于长度为100的向量，四个warp中的一个将具有控制异化发生，这将对性能产生重大影响。对于一个大小为1000的向量，32个warp中只有一个具有控制散度。也就是说，控制异化只会影响大约3%的执行时间。即使使warp的执行时间增加一倍，对总执行时间的净影响也只有3%左右。显然，如果矢量长度为10,000或更多，那么在313个warp中只有一个具有控制异化发生。控制异化的影响将大大小于1%!

对于二维的数据，例如颜色到灰度转换示例，if语句还用于为处理数据边缘操作的线程的判断边界条件。在图3.2中，对于76×62的图像，我们使用20 = 5*4个二维区块，每个区块由16×16个线程组成。每个区块将被分割成8个warp，每个warp由一个区块的两行组成。总共有160个warp(每个区块8个warp)。

控制异化的影响分析见图3.5。在区域1的12区块中，没有一个warp会有控制异化。1区有12*8 = 96个warp。对于区域2，所有24个warp都有控制异化。对于区域3，请注意，所有底部的warp都映射到完全在图片外部的数据，因此没有一个能通过if条件（如果图片在垂直维度上的像素数是奇数的话，读者应该知道这些warp会有控制异化），因为它们都遵循相同的控制流路径，所以这32个warp中没有一个具有控制异化！在第4区域，前7条warp将有控制异化，但最后一条不会。总的来说，160个warp中的31个将会有控制异化。

同样，控制异化对性能的影响随着水平维度中像素数目的增加而减小。举个例子，如果我们处理一张200×150的图片，用16×16大小的区块，总共会有130 = 13*10个区块、1040个warp。从区域1到区域4的warp数将为864(12*9*8)、72(9*8)、96(12*8)和8(1*8)。这些warp中只有80个会有控制异化。因此，控制异化对代码性能的影响将小于8%。显然，如果我们处理水平维度超过1000个像素的真实图像，控制异化对性能的影响将小于2%。

在一些重要的并行算法中，随着时间的推移，参与计算的线程数量会发生变化，控制异化也会自然出现。我们将使用一个reduction算法来说明这种现象。

reduction算法指的是从一组值中生成一个值。这个值可以是所有元素的和、最大值、最小值等。所有这些类型的reduction算法都使用相同的计算结构。通过按顺序遍历数组的每个元素，可以很容易地实现reduction。当访问一个元素时，要采取的操作取决于所执行哪种reduction。如果要求和，那将当前步骤中访问的元素的值或当前值添加到之前运行的和中。对于求最大值，那就将当前值与到目前为止访问的所有元素的运行最大值进行比较，如果当前值大于正在运行的最大值，则当前元素值将成为正在运行的最大值。如果要求最小值，那就将当前正在访问的元素的值与正在运行的最小值进行比较，如果当前值小于正在运行的最小值，则当前元素值将成为正在运行的最小值。顺序执行的算法在访问完所有元素时结束。

顺序执行的reduction算法的工作最高效的方法是每个元素只访问一次，并且在访问每个元素时只执行少量的工作。它的执行时间与所涉及的元素的数量成正比。即算法的计算复杂度为O(N)，其中N为reduction所涉及的元素数量。

访问一个大数组的所有元素所需的时间促使并行执行。并行reduction算法类似于足球比赛的赛制。事实上，世界杯的淘汰过程就是“取最大值”的reduction，其中“取最大值”被定义为“击败”所有其他球队的球队。比赛的“reduction”过程是在多个回合中完成的。队伍分成两人一组。在第一轮中，所有的对战都并行进行。第一轮的获胜者进入第二轮，其获胜者进入第三轮，以此类推。有16支队伍参加比赛，第一轮将有8支队伍胜出，第二轮有4支队伍胜出，第三轮有2支队伍胜出，第四轮有1支队伍胜出。

很容易看出，即使有1024支队伍，也只需要10轮就能确定最终的获胜者。关键是要有足够的足球场来同时举办第一轮的512场比赛、第二轮的256场比赛、第三轮的128场比赛，以此类推。有了足够的场地，即使有6万个团队，我们也能在16轮比赛中决定最终的获胜者。当然，需要有足够的足球场和足够的官员来容纳第一轮的3万场比赛，等等。

图5.13显示了执行并行加和的reduction的kernel函数。原始数组在全局内存中。每个线程区块通过将片section的元素加载到共享内存并对这些元素执行并行reduction来减少数组部分元素。代码将输入数组X的元素从全局内存加载到共享内存中。reduction操作已经完成意味着共享内存中的一些元素的值将被部分元素的加和替换。kernel函数中for循环的每次迭代都实现了一轮reduction。

```
1. __shared__ float partialSum[SIZE];
   partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];
2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
4. {
5.   __syncthreads();
6.   if (t % (2*stride) == 0)
7.     partialSum[t] += partialSum[t+stride];
8. }
```

FIGURE 5.13

A simple sum reduction kernel.

for循环中的__syncthreads()语句(第5行)确保在允许任何一个线程开始当前迭代之前，生成了前一次迭代的所有部分元素之和。这样，所有进入第二次迭代的线程都将使用第一次迭代生成的值。第一轮结束后，偶数元素将被第一轮产生的部分元素的和所取代。第二轮后，索引为4的倍数的元素将被部分元素的和取代。在最后一轮之后，整个部分的总和将是partialSum[0]。

在图5.13中，第3行将stride变量初始化为1。在第一次迭代中，第6行中的if语句用于仅选择要在两个相邻元素之间执行加法的偶数线程。kernel的执行如图5.14所示。线程和数组元素值在水平方向显示。随着时间从上到下的推移，线程在垂直方向上发生一次又一次的迭代。图5.14中的每一行显示了for循环迭代后数组元素的内容。

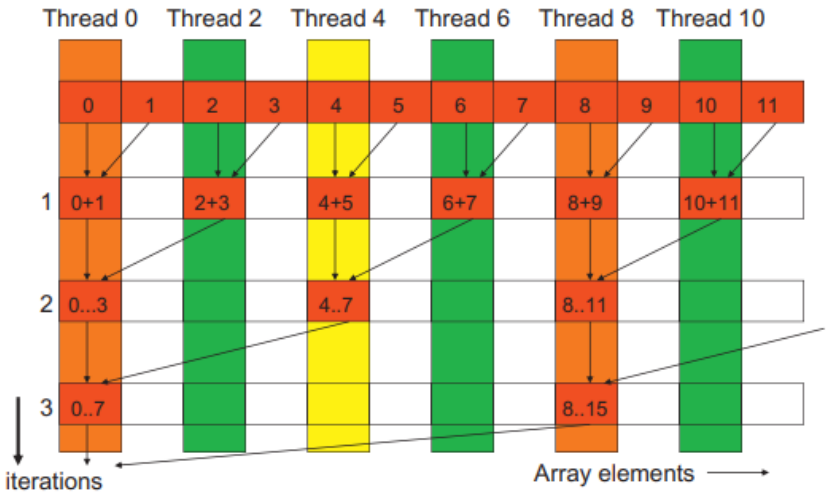


FIGURE 5.14

Execution of the sum reduction kernel.

如图5.16所示，数组的偶数元素在迭代一次后保持两个元素的和。在第二次迭代之前，stride变量的值加倍为2。在第二次迭代中，只有索引为4的倍数的线程才会执行第7行中的加法操作。每个线程生成四个元素的和，如第2行所示。每个section中有512个元素，kernel函数将在9次迭代后生成整个section的和。通过使用blockDim。x作为第4行中的循环边界，内核假设启动它的线程数与部分中的元素数相同。也就是说，对于大小为512的section，需要用512个线程加载kernel。

我们来分析一下kernel做了什么事。假设需要做reduction的元素总数为N，第一轮需要做N/2个加法。第二轮需要N/4个加法。最后一轮只有一个加法。有log2(N)轮。kernel函数执行的相加总数为N/2 + N/4+ N/8 +…+ 1 = N−1。因此，reduction算法的计算复杂度为O(N)。该算法是有效的。但是，我们还需要确保在执行内核时高效地利用了硬件资源。

图5.13中的kernel明显存在线程异化。在循环的第一次迭代中，只有threadIdx.x是偶数的线程才会执行加法操作。执行加法这些线程将需要一个路径，没有执行第七行加法的那些线程将需要一个额外的路径。在后续的迭代中，执行第7行的线程会更少，但是在每个迭代中执行所有线程仍然需要两个路径。只要对算法稍加修改，就可以减少这种异化。

图5.15展示了一个改进的kernel函数，其和reduction算法略有不同。它不是在第一轮中对相邻的元素加和，而是添加彼此相隔半个section的元素。它通过将stride初始化为section的一半大小来实现。所有在第一轮中加和的两个元素彼此之间的距离都是一半section的长度。第一次迭代后，所有的两个元素之和存储在数组的前半部分，如图5.16所示。在进入下一个迭代之前，循环将stride除以2。因此，对于第二次迭代，stride变量值为section大小的四分之一。也就是说，在第二次迭代中，彼此相加的元素相聚四分之一个section的长度。

```
1.  __shared__ float partialSum[SIZE];
    partialSum[threadIdx.x] = X[blockIdx.x*blockDim.x+threadIdx.x];

2.  unsigned int t = threadIdx.x;
3.  for (unsigned int stride = blockDim.x/2; stride >= 1; stride = stride>>1)
4.  {
5.    __syncthreads();
6.    if (t < stride)
7.      partialSum[t] += partialSum[t+stride];
8.  }
```

FIGURE 5.15

A kernel with fewer thread divergence.

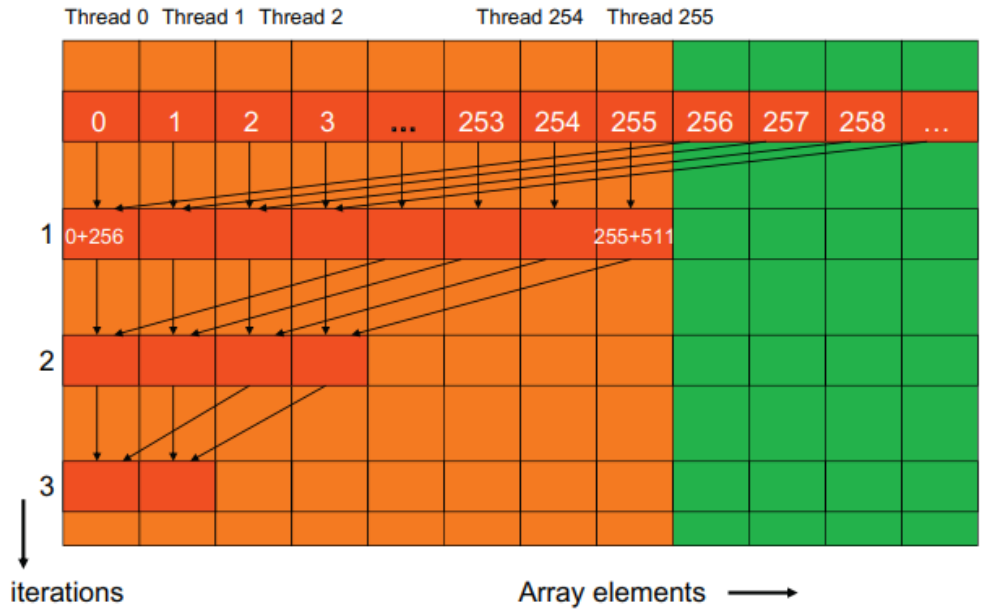


FIGURE 5.16

Execution of the revised algorithm.

注意，图5.15中的kernel在循环中仍然有一个if语句(第6行)。在每次迭代中执行第7行线程的数量与图5.13中相同。那么，为什么两个kernel之间会有性能差异呢?答案在于执行第7行的线程与不执行第7行的线程的相对位置。

图5.16展示了图5.15中修改后的kernel的执行情况。在第一次迭代中，所有threadIdx.x小于section大小一半的线程都执行第7行。对于包含512个元素的section，线程0到255在第一次迭代中执行add语句，而线程256到511则不执行。在第一次迭代之后，两个元素的和分别存储在元素0到255中。因为warp由32个连续threadIdx.x值的线程组成，warp 0到warp 7中的所有线程都执行加法操作，而warp 8到warp 15中的所有线程都跳过加法操作。因为每个warp所有的线程都通过相同的路径，因此没有线程的异化发生！

图5.15中的kernel并没有完全消除if语句引起的异化。读者应该自行验证：从第5次迭代开始，执行第7行线程的数量将降到32以下。也就是说，最后五个迭代将只有16、8、4、2和1个线程执行添加操作。这意味着内核执行在这些迭代中仍然会有异化。然而，具有异化的迭代次数从10次减少到5次。

图5.13和5.15之间的差异很小，但对性能有非常重要的影响。只有一个对SIMD device上的线程如何高效执行有清晰理解的人，才能自信地做出这样的调整。

5.4资源的动态分配

SM中的执行资源包括寄存器、共享内存、线程区块槽和线程槽。这些资源被动态分区并分配给线程以支持它们的执行。在第3章中，我们已经看到Fermi这一代的device有1536个线程槽。这些线程槽在运行时被分成区块。如果每个线程区块由512个线程组成，那么1536个线程槽就会分成3个块。在这种情况下，由于线程槽的限制，每个SM最多可以容纳三个线程区块。

如果每个线程区块包含256个线程，则1536个线程槽会分成6个线程区块。在线程区块之间动态划分线程槽的能力使SMs具有通用性。它们要么可以执行许多线程区块，每个线程区块都有少量的线程，或者执行少量线程区块，每个线程区块都有多个线程。这与固定划分区块的方法相反，在固定区块划分的方法中，不管实际需要如何，每个块都接收固定数量的资源。当一个区块具有较少的线程并且无法支持比固定分区更多的线程槽时，固定分区将导致线程槽的浪费。

资源的动态分配可能会产生资源限制之间的微妙关系，从而导致资源的利用率不足。这种关系可以发生在区块槽和线程槽之间。例如，如果每个区块有128个线程，那么1536个线程槽可以被分成12个块。然而，由于在每个SM中只有8个区块槽，所以只允许8个区块。这意味着最终只有1024个线程槽被使用。因此，为了充分利用区块槽和线程槽，每个区块中至少需要256个线程。

正如我们在第4章中提到的那样，在CUDA kernel中声明的自动变量被放入寄存器中。 一些kernel可能会使用大量的自动变量，而其他kernel可能会使用少量自动变量。 因此，人们应该期望某些kernel大量使用寄存器，而有些则需要使用少量寄存器。 通过在区块之间动态划分寄存器，如果kernel需要较少的寄存器，则SM可以容纳更多的区块；如果kernel需要更多的寄存器，则SM可以容纳更少的块。 但是，确实需要意识到寄存器限制和其他资源限制之间的潜在相互关系。

在矩阵乘法示例中，假设每个SM具有16,384个寄存器，并且kernel代码每个线程使用10个寄存器。 如果我们的线程区块是16×16的，那么每个SM上可以运行多少个线程？ 我们可以首先通过计算每个块所需的寄存器数量（10 * 16 * 16 = 2560）。六个区块所需的寄存器数量为15,360，低于16,384的限制。 添加另一个区块将需要17,920个寄存器，这超出了限制。 因此，寄存器限制允许在每个SM上运行总共有1536个线程的6个区块，这也符合8个区块槽和1536个线程槽的限制。

现在假设程序员在kernel中声明了另外两个自动变量，并将每个线程使用的寄存器数量增加到12。假设相同的16×16的区块，每个区块现在需要12 * 16 * 16 = 3072个寄存器。 现在，六个块所需的寄存器数量为18432，超过了某些CUDA硬件的寄存器限制。 CUDA运行时系统通过将分配给每个SM的区块数减少一个，从而将所需的寄存器数减少到15,360来处理这种情况。 但是，这将SM上运行的线程数从1536减少到1280。也就是说，通过使用两个额外的自动变量，程序在每个SM中的warp并行数量降低了1/6。有时这被称为“性能断崖”，其中资源使用的轻微增加会导致并行性和性能的显着降低。

共享内存是另一个在运行时动态分配的资源。分块算法通常需要大量共享内存才能奏效。不幸的是，大量共享内存的使用会减少在SM上运行的线程区块的数量。正如我们在5.3节中所讨论的，线程并行性的降低会对DRAM系统内存访问带宽的利用产生负面影响。减少的内存访问通量又会进一步降低线程执行通量。这是一个陷阱，可能会导致分块算法的性能降低，应该小心避免。

读者应该很清楚，所有动态分配资源的约束以复杂的方式相互作用。准确地确定在每个SM中运行的线程数量是困难的。读者可以参考CUDA占用计算器[NVIDIA]（CUDA Occupancy Calculator），这是一个可下载的Excel表格，可以根据内核的资源使用情况，计算特定device上运行的每个SM上的实际线程数。

5.5线程粒度

在性能调优中一个重要的决策算法是线程的粒度。有时，将更多的工作放到每个线程中而使用更少的线程是有利的。当线程之间存在一些冗余工作时，就会体现出这种优势。在当前一代的device中，每个SM都有有限的指令处理带宽。无论是浮点计算指令,装入指令或分支指令，每一个指令都要消耗指令处理带宽。消除冗余工作可以缓解指令处理带宽的压力和提高整个kernel的执行速度。

图5.17展示了矩阵乘法中的这种机会。 图5.6中的分块算法使用一个线程来计算输出P矩阵的一个元素。这需要在M的一行与N的一列之间形成点积。

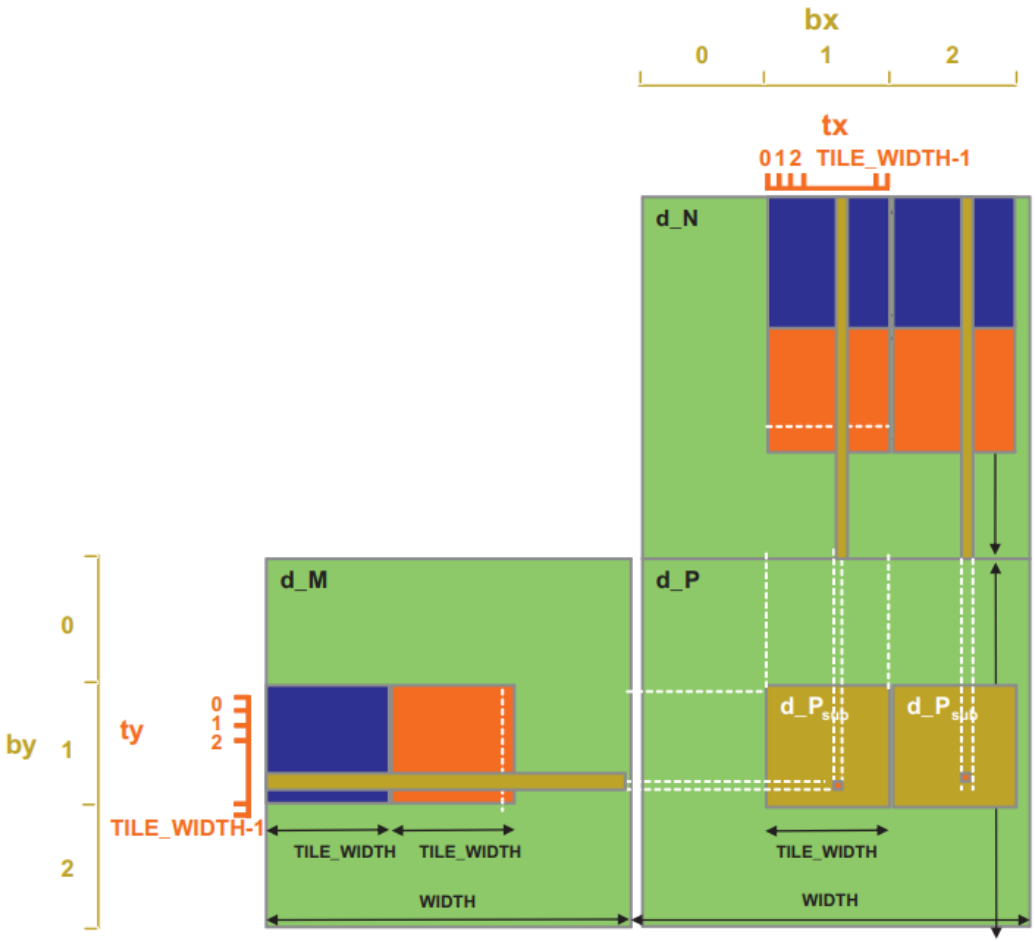


FIGURE 5.17

Increased thread granularity with rectangular tiles.

线程粒度调整的机会来自于多个区块冗余地加载M小块这一事实。图5.11也演示了这一点。如图5.17所示，计算两个相邻小块中相同位置的P元素会使用M中相同的一行。在原来的分块算法中，这两个P小块冗余的加载了M中相同的行。可以通过将两个线程区块合并为一个来消除这种冗余。新线程区块中的每个线程现在都要计算两个P元素。这是通过修改kernel来实现的，此时kernel的最内层循环计算两个点

乘。两个点乘使用相同的Mds行但不同的Nds列。这将减少四分之一的全局内存访问。我们鼓励读者编写新的kernel作为练习。

潜在的不利之处在于，新kernel现在使用甚至更多的寄存器和共享内存。 正如我们在上一节中讨论的那样，每个SM上可以运行的区块数可能会减少。 对于给定的矩阵大小，这还会将线程区块的总数减少了一半，这可能导致较小尺寸的矩阵的并行度不足。 实际上，四个相邻的水平区块的合并可显著提高大型（2048×2048或更多）矩阵乘法的性能。

5.6总结

在本章中，我们回顾了CUDA device上影响应用程序性能的主要方面：全局内存访问合并、内存并行、控制流异化、动态资源分配和指令混合。这些方面都受制于device的硬件限制。有了这些知识，读者应该能够推断出你遇到的任何kernel代码的性能。

更重要的是,我们需要能够将低性能的代码调优。我们为高性能程序模式的实用技术开了个头。我们将在接下来的几章中继续研究这些技术，讨论这些技术在并行计算模式和应用案例研究中的实际应用。

5.7练习

自行看书