第四章 内存和数据本地化

目前为止，我们已经学习了如何写一个CUDA的kernel函数还有如何配置和调度大量线程。在本章中，我们将学习如何让组织和安排数据的存储位置来使得大量线程的访问更加高效。我们在第二章中讲过，数据是先从host内存中传到device全局内存中的，在第三章中说过如何让线程通过他的区块索引和线程索引来访问全局内存中的部分数据，我们还讲了资源分配和线程调度。尽管我们现有的知识能够让CUDA应用跑起来，但是我们也仅仅能够施展CUDA的kernel的一小部分性能。性能不高的原因是因为很高的数据访问等待时延（需要消耗上百个时钟周期）和有限的全局内存访问带宽（全局内存基本都是DRAM动态内存构成）。虽然有着足够多的线程可以理论上减少等待时延，但是我们还是很容易遇到这种状况，当由于带宽的限制使得只有一小部分线程在工作，从而使得一些SM在空闲状态。为了避免这种状态，CUDA提供很多其他的资源和方法来使得访问全局内存的流量减少。在本章中，你会学到如何使用不同的内存类型来加速CUDA的kernel的执行效率。

4.1内存访问效率的重要性

我们可以通过计算图3.8中图片虚化的kernel主要部分的理论性能来体验内存访问效率的重要性。在图4.1中展示了这一部分。对执行时间影响最大的就是这个嵌套的for循环，循环中完成了patch中像素值得加和。

```
      for(int blurRow = -BLUR_SIZE; blurRow < BLUR_SIZE+1; ++blurRow) {
4.      for(int blurCol = -BLUR_SIZE; blurCol < BLUR_SIZE+1; ++blurCol) {

5.        int curRow = Row + blurRow;
6.        int curCol = Col + blurCol;
          // Verify we have a valid image pixel
7.        if(curRow > -1 && curRow < h && curCol > -1 && curCol < w) {
8.          pixVal += in[curRow * w + curCol];
9.          pixels++; // Keep track of number of pixels in the avg
          }
        }
      }
```

FIGURE 4.1

The most executed part of the image blurring kernel in Fig. 3.8.

在每个内循环中，每次浮点数加法运算都伴随一个全局内存访问，全局变量需要访问in[]数组元素，然后通过浮点数运算将picVal的值与in[]的值加和。因此，浮点数运算与全局变量访问的操作数量比为1:1=1.0，我们成这个比例为“计算与内存访问比”（compute-to-global-memory-access ratio）,反映了在一段代码中每次访问内存伴随着多少个浮点数运算。

这个“计算与内存访问比”反映了CUDA的kernel的性能，在高端device中，全局内存的带宽是 1,000 GB/s,或1 TB/s。对于占4字节的单精度浮点数而言，每秒最多能加载1000/4=250兆个单精度浮点数。对于1.0的“计算与内存访问比”而言，图片虚化kernel的执行将受到操作数（例如in []的元素）交付给GPU的速率的限制。我们将这种被内存访问通联限制的程序称之为“内存限制程序”。在我们的例子中，kernel最多只能达到250兆浮点数运算每秒（GFLOPS——giga floating point operations per second）。

虽然250GFLOPS是一个不错的新能了，但是对于高端的device性能极限而言，这只是12TFLOPS的2%左右。为了达到更高的kernel性能，我们需要增加通过减少全局内存的访问来增大这个“计算与内存访问比”，为了达到12TFLOPS，我们需要这个值达到48或者更高，总的来说，在过去几代的device发展来看，这个比例的期待值一直在增长，因为计算通量的增长属于要快于带宽的增长速度。本章的剩余部分将会介绍一个被广泛使用的用来减少全局内存访问次数的技术。

4.2矩阵乘法

矩阵与矩阵的乘法是一个i × j (i 行 j列)的矩阵与j × k的矩阵线程，结果为一个i × k 的矩阵P。矩阵乘法是BLAS的一个重要组成部分（第三章有讲到BLAS）。这个韩式是很多线性代数求解器的基础，比如说LU分解（LU decomposition）。矩阵乘法这个例子将会带领我们学习减少全局内存访问的一种简单方法。矩阵乘法的执行速度会根据矩阵大小不同而不同，而且还与全局内存访问的优化程度相关。因此，矩阵乘法是一个很好地例子。

矩阵乘法中，输出矩阵P的每个元素都是M中的一行与N中的一列的点乘，我们使用P(Row,Col)来表示P中的第Row行和第Col列的元素，如图4.2所示P(Row,Col)是P中的小方块，他的值是途中M的哪一个横条向量与N中的竖条向量的点乘的结果，点乘也叫做内积，是两个向量对应元素乘积的加和，公式如下：

$$P_{\text{Row,Col}} = \sum M_{\text{Row},k} * N_{k,\text{Col}}, \text{for } k = 0,1,\ldots \text{Width} - 1$$

举个例子：

$$P_{1,5} = M_{1,0} * N_{0,5} + M_{1,1} * N_{1,5} + M_{1,2} * N_{2,5} + \cdots + M_{1,\text{Width}-1} * N_{\text{Width}-1,5}$$

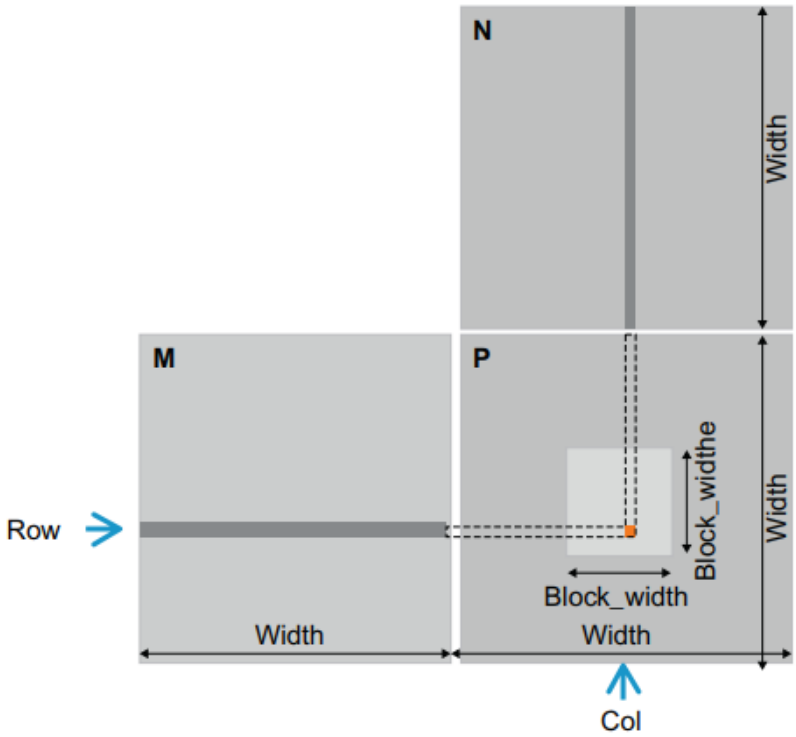

FIGURE 4.2

Matrix multiplication using multiple blocks by tiling P.

我们首先通过colorToGreyscaleConversion的方式来实现矩阵乘法，将每个线程映射到对应的那部分数据上。例如，每个线程都会计算一个P元素，每个P元素的行和列索引如下

```
Row=blockIdx.y*blockDim.y+threadIdx.y
```
and
```
Col=blockIdx.x*blockDim.x+threadIdx.x.
```

通过这个一对一的映射，Row和Col这两个线程索引也是输出数组的行和列的索引。在图4.3中展示了这个线程映射到数据的kernel函数的源代码，读者会立刻发现熟悉的Row,Col计算和if表达式来判断Row和Col是否在范围之内，这些语句与colorToGreyscaleConversion对应部分简直一毛一样，唯一明显的区别是我们的这个函数假设了矩阵是方阵，因此width和height都用Width表示。

```
__global__ void MatrixMulKernel(float* M, float* N, float* P,
  int Width) {
  // Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y+threadIdx.y;
  // Calculate the column index of P and N
  int Col = blockIdx.x*blockDim.x+threadIdx.x;
  if ((Row < Width) && (Col < Width)) {
    float Pvalue = 0;
    // each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k) {
      Pvalue += M[Row*Width+k]*N[k*Width+Col];
    }
    P[Row*Width+Col] = Pvalue;
  }

}
```

FIGURE 4.3

A simple matrix multiplication kernel using one thread to compute one P element.

线程对数据的映射有效地将P分成了一个有一个的小块，图4.2展示了其中一个小块，每个区块负责计算一个小块。

我们现在将注意力集中在每个线程是怎样的完成工作的，回忆一下当时我们说P(Row,Col)是M的第Row行与N的第Col行的内积，在图4.3中我们使用了一个for循环来站是这样的内积操作过程。在进入循环体之前，我们将Pvalue初始化为0，每次循环都从M的第Row行和N的第Col列来访问元素，然后将两个元素相乘，然后将结果累加到Pvalue中。

首先我们来看for循环中从M获取元素的过程，当时我们讲过M是被线性化之后的一维数组，在内存空间中一行接着一行的连续排列，因此第1行的第一个元素的索引是 M[1*Width]（因为我们要跳过第0行的所有元素），总的来说第Row行的第一个元素是 M[Row*Width]，因为所有的行是在内存空间中连续排列的。第Row行的第k个元素是M[Row*Width+k]。在图4.3中可见这个表达式。

我们现在再来看N，如图4.3，第Col列的第一个元素是N[Col]。获取这一列中下一个元素的话需要跳过整整一行，正好是下一行的同一列的元素，因此第Col列中的第k个元素是 N[k*Width+Col]。

在执行完for循环之后，所有的线程在Pvalue中都有P的元素的值，每个线程再用一维等效索引 Row*Width+Col 将值写入到输出中，这点与colorToGreyscaleConversion很相似。

我们现在使用一个例子来说明矩阵乘法kernel的执行。在图4.4中展示了一个4*4矩阵P，BLOCK_WIDTH=2。之所以选用这么小的区块是因为这样一个图能放的下。矩阵P现在被分成了4个小块，每个区块计算一个小块，小块是由2*2的线程组成的，每个线程计算一个P的元素。这个例子中区块(0,0)的线程(0,0)计算P(0,0)，区块(1,0)的线程(0,0)计算P(2,0)。

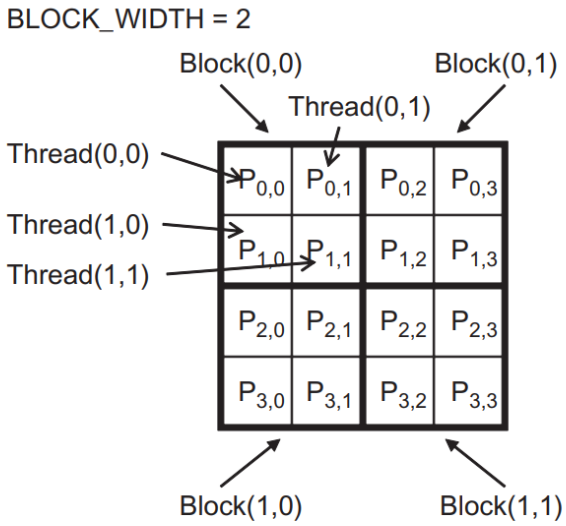

FIGURE 4.4

A small execution example of matrixMulKernel.

matrixMulKernel中的Row和Col表示了P中要被线程所计算的元素位置，Row还表示了M的行，Col还表示N的列，在图4.5中展示了每个线程区块的乘法操作。区块(0,0)中的线程(1,0)的Row和Col的值分别是0*0 + 1= 1 和 0*0 + 0= 0，这对应着M中的1行与N中的0列的点乘，结果位于P(1,0)。

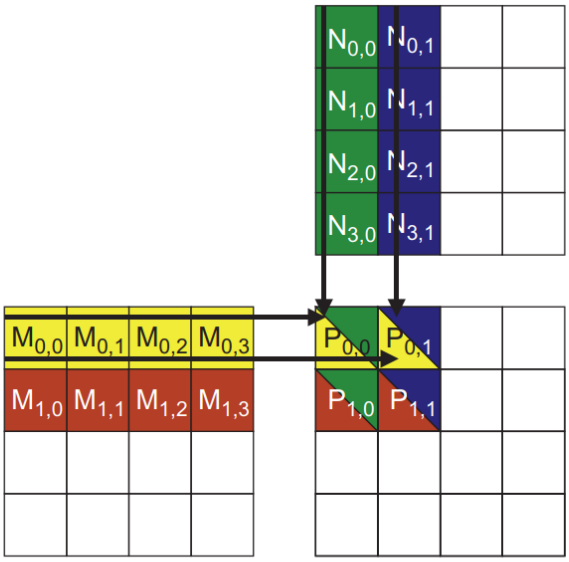

FIGURE 4.5

Matrix multiplication actions of one thread block.

我们在看一下图4.3中的for循环中区块(0,0)线程(0,0)的执行。在第0个循环中(k=0),Row*Width+k=0*4 + 0 = 0,k*Width+Col=0*4 + 0= 0。因此我们在访问M[0]和N[0]，也就是M(0,0),N(0,0)的一维等效索引。注意这些实际上是M的第0行的元素和N的第0列的元素。在第1个循环中(k=1),Row*Width+k=0*4+1=1, k*Width+Col=1*4+0=4，我们访问的M[1]和N[4]实际上是M(0,1)与N(1,0)的一维等效索引，他们是M第0行的元素与N的第0列的元素。

在第2个循环中(k=2), Row*Width+k=0*4+2=2，k*Width+Col=8，等等，第3个循环k=3, Row*Width+k=0*4+ 3 ,k*Width+ Col= 12。我们现在知道了内层循环是怎么将M的第0行与N的第0列处理的了，在循环完成之后，这个线程会写入到 P[Row*Width+Col]中，也就是P[0]，也就是P(0,0)。至此，区块(0,0)的线程(0,0)成功地计算了M的第0行与N的第0列的内积结果，然后将结果写入到P(0,0)中。读者请自己去思考区块(0,0)中其他线程是怎么执行的，就当做练习。

注意matrixMulKernel能够处理每个维度最多16*65536个元素的矩阵，这种情况下，比这个限制还要大的矩阵要相乘的话，我们可以将大的矩阵分解成小的子矩阵，每个小矩阵大小能够被网格所容纳，然后我们可以用host代码来循环加载kernel来完成矩阵P的乘法。我们还可以改变kernel代码，让每个kernel线程处理更多的P元素。

我们可以通过计算图4.3的kernel代码的预期性能来估算内存访问效率，kernel中主要影响执行时间的是这个内层for循环

```
for(int k = 0;k < Width;++k)Pvalue +
  = M[Row * Width + k] * N[k * Width + Col];
```

在这次循环中，都要访问两个全局内存变量，然后计算一次加法浮点数运算和一次乘法浮点数运算。两次访问全局变量分别是获取M中的元素与N中的元素，两个浮点数运算分别是，M和N获取到的元素的乘法与将结果累加到Pvalue变量中。因此“计算全局内存访问比”是1.0。从我们第三章讲的知道，这会导致我们的GPU性能只发挥了2%左右，为了使现代device的计算吞吐量达到良好的利用率，我们需要将比率至少增加一个数量级。在下一个章节中，我们将介绍如何通过使用CUDA特殊种类的内存来完成这一小目标。

4.3CUDA内存种类

一个CUDA的device中有很多种不同的内存，他们能帮助我们编程者增加“计算全局访问比”并为CUDA程序提速。在图4.6中展示了这些CUDA的device内存，在图片的底部是全局内存和常量内存。这些种类的内存能够被host代码的API函数读写，在第二章中，我们已经介绍了全局内存，全局内存能够被device读写，常量内存能提供高带宽，支持低访问时延，能够被device只读。

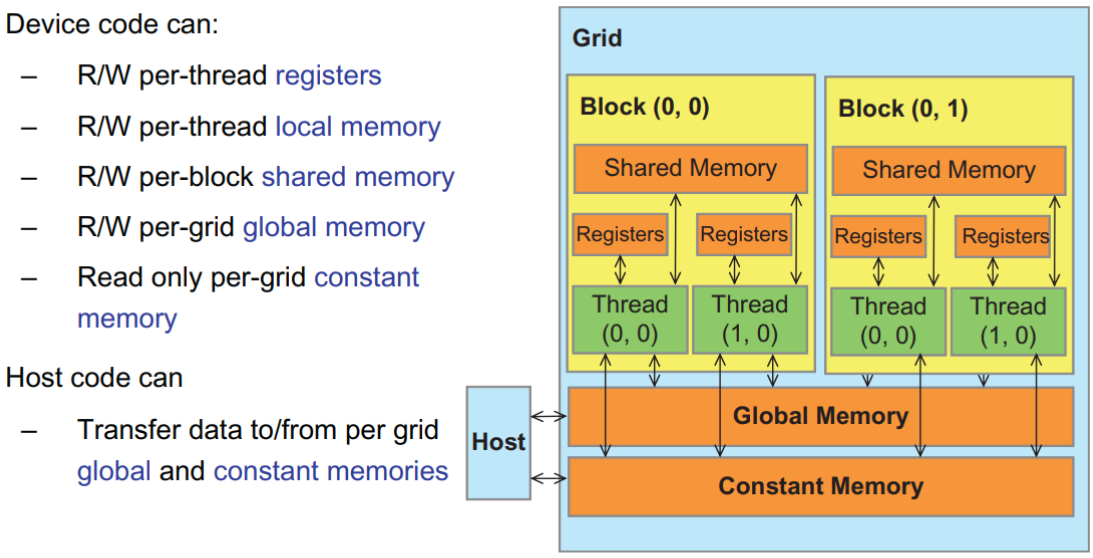

FIGURE 4.6

Overview of the CUDA device memory model.

图4.6中展示了寄存器和共享内存，这两个是在芯片上的内存，在这些内存中的变量能够以非常非常快的速度被并行访问，寄存器位于在线程中，每个线程只能访问自己的寄存器，一个kernel函数会将频繁访问的变量放到每个线程的私有寄存器中存储，共享内存是在线程区块中的，同一区块中的所有线程能够访问共享内存中的变量。共享内存可以存放其他线程的共享输入数据，或者是中间环节计算结果，方便了线程之间的合作。编程者通过声明存储变量的CUDA的内存种类，可以决定这个数据是否可访问以及访问的速度。

为了能够解释共享内存和全局内存的去和，我们需要来研究一下现代处理器中是如何实现这些不同种类的内存的。所有的现代处理器都是根据根据1945年冯诺依曼的模型设计的，如图4.7所示。CUDA的device也不例外，CUDA 的device的全局变量对应着途中的Memory，途中的Processor对应着芯片。全局变量使用DRAM技术实现的，这种技术有高等待时延和相对低的带宽，寄存器对应着图中的Register file，是在芯片上的，相比于全局内存能提供很低的访问实验和较高带宽，在特定的device中，寄存器的总访问带宽至少比全局内存的带宽大两个数量级，而且当一个变量被存到寄存器中之后，对这个变量的访问不再消耗全局内存的带宽，这减少的带宽消耗可帮助提高“计算全局访问比”。

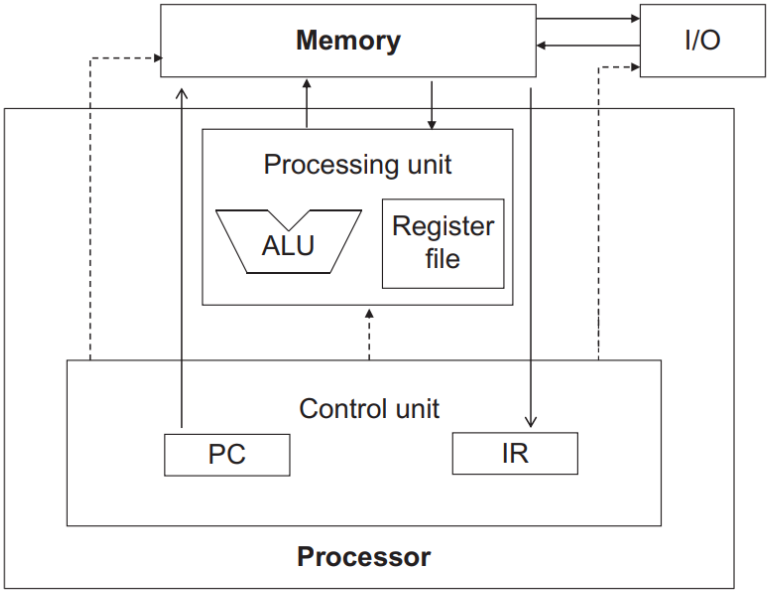

FIGURE 4.7

Memory vs. registers in a modern computer based on the von Neumann model.

一个很微妙的地方在于，访问寄存器相比于访问全局内存需要更少的指令。现代处理器都有寄存器的内置算数操作，举个例子，一个浮点数加法的运算如下：

```
fadd r1, r2, r3
```

其中r2和r3是寄存器代号，表示输入的数据，加法的结果存储在寄存器r1中，因此当进行一个寄存器中就存在的算术运算时，我们就并不需要向ALU提供操作数（Arithmetic and logic unit，是完成数学计算的单元）。

> **冯诺依曼模型**
>
> 约翰·冯·诺依曼（John von Neumann）在1945年的一个开创性报告中，描述了一种建立电子计算机的模型，该模型基于开创性的电子离散可变自动计算机（EDVAC）计算机的设计而建立。 现在几乎所有现代计算机的基础蓝图都通常称为von Neumann模型。
>
> 冯诺依曼模型如图4.7所示。计算机有一个输入和输出功能，使得人们能够向计算机传递程序和数据，计算机也能生成程序和数据，为了执行程序，计算机需要先把程序写入到内存之中。
>
> 该程序包含一组指令。 控制单元维护一个程序计数器（Program counter），其中包含要执行的下一条指令的存储器地址。 在每个“指令周期”中，控制单元都使用PC将指令提取到指令寄存器（Instruction register）中。 指令位然后用于确定计算机的所有组件要采取的操作，这就是为什么该模型也称为“存储程序”模型的原因。 该术语意味着用户可以通过将不同的程序存储到计算机的内存中来更改计算机的行为。

如果操作数在全局变量之中，处理器需要将操作数通过指令转移到ALU之中。例如，如果第一个浮点数加法的操作的操作数在全局变量之中，会产生如下的指令。

```
load r2, r4, offset
fadd r1, r2, r3
```

其中load指令将生成一个偏移量offset，并将这个值与r4中的地址相加，然后回到这个加和后的地址去访问数据，并将数据存储到r2寄存器中。下一步是将r2和r3中的值加和，结果存储到r1中，因为处理器每个时钟周期只能够获取并执行有限个数的指令，这个需要额外加载数据的版本会消耗更多的时间，因此，将操作数放到寄存器中能够加快执行速度。

将数据存储在寄存器中还有一个很关键的原因，在现代的计算机中，向全局内存获取数据要比从寄存器获取数据所消耗的能耗至少要高一个数量级，我们将会对比这两种内存访问的速度和能耗，但是，我们一会儿就会学到，每个县城中的能用的寄存器在现如今的GPU中非常的少。我们需要注意不要超过资源限制。

图4.8展示了CUDA的device中的共享内存和寄存器。尽管二者都是在芯片上的内存，他们在功能上和在访问成本上非常不同，共享内存被设计成一种位于芯片内部的存储空间，当处理器需要访问芯片内部的共享内存时，它需要一个内存加载操作，与访问全局内存非常相似，但是因为共享内存是位于芯片内的，因此可以以更低的时延和更高的通量来访问，但相比于寄存器中数据的访问，还是高时延低通量的，毕竟还是有加载的操作。在计算及结构的学术名词中，共享内存是一种暂存器（scratchpad memory）。

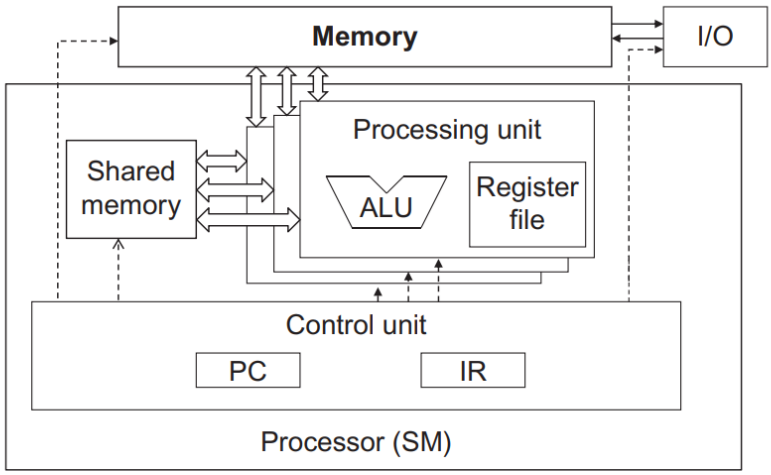

FIGURE 4.8

Shared memory vs. registers in a CUDA device SM.

共享内存与寄存器的一个很关键的区别是区块中的所有线程都能访问共享内存中的数据，但是寄存器中的不行，共享内存设计的初衷是支持高效，高带宽，同区块之间线程的共享数据。如图4.8所示，一个CUDA device的SM调用很多处理单元，使得多线程同时工作，一个区块中的线程能被分配到这些处理单元中，因此，这些CUDA device中共享内存的硬件实现通常设计为允许多个处理单元同时访问其内容，以支持块中线程之间的有效数据共享。我们将学习几个重要的能够有效利用共享内存的并行算法。

> **处理单元和线程**
>
> 现在我们已经介绍了冯诺依曼模型，我们已经讨论了这些线程是如何实现的。现代计算机的一个线程就是冯诺依曼处理器上的一个程序执行状态，一个线程由程序代码、正在执行的代码和变量中的值和数据结构组成。
>
> 在冯诺依曼模型计算机中，程序代码是保存在内存中的，PC跟踪正在执行的程序的特定点，IR(Instruction Register)将特定点获取到的指令保存起来，寄存器和内存保存了变量和数据结构。
>
> 现代处理器设计为允许内容切换的（context-switching），也就是说多线程是在时间上共享的，处理器轮流对他们进行处理，通过保存和恢复PC(Program count)、内存和寄存器中的值，我们可以将一个线程暂停，并能够正确地恢复线程并继续执行。

> 一些处理器提供多线程单元，允许多线程同时执行。图4.8展示了Single-Instruction Multiple-Data的设计风格，其中多个处理单元共享PC(program count)和IR。在这种设计下，所有的线程执行同一个指令，同时起作用。

现在应当清楚了寄存器，共享内存，全局内存的不同功能，带宽，时延。因此需要声明一个变量的过程让变量存到对应种类的内存中。表格4.1展示了CUDA声明不同种类内存的语法，每个声明还给出了它的CUDA域和存在时间，域表示有哪些线程能够访问到这个变量：是只有一个线程，区块中的所有线程，还是所有网格中的所有线程。如果一个变量的域是单线程，那么每个线程都会生成独自生成该变量，每个线程只能访问自己线程上的该变量。换句话说就是，如果kernel声明一个变量的域是单线程，那么如果有1亿个线程的话，每个线程在初始化的时候都会产生一个私有变量，一共产生1亿个。

Table 4.1 CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
`__device__ __shared__ int SharedVar;`	Shared	Block	Kernel
`__device__ int GlobalVar;`	Global	Grid	Application
`__device__ __constant__ int ConstVar;`	Constant	Grid	Application

存在时间表明了程序执行中的哪些部分可以使用这个变量，要么是在kernel执行的过程中，要么是整个CUDA应用中。如果一个变量的存在时间是在kernel执行过程中，这个变量必须是在kernel中声明的，而且只能被kernel代码使用。如果这个kernel函数被引用过很多次，这个变量的值并不会也被引用过去，每次引用必须初始化这个变量后才能够使用。如果一个变量的存在时间是整个应用的话，这个变量必须在函数外声明，这个变量的值在整个执行过程中存在且能被任何kernel访问。

我们称这种不是数组或者矩阵的变量为标量变量（scalar variables），如表格4.1所示，所有的kernel中和device函数中声明的标量变量都存储在寄存器中，这些变量的域是单个线程，当一个kernel函数声明一个自动变量的时候，每个执行这个kernel函数的线程都会在执行时复制一下这个变量到自己的线程中，当一个线程终结时，所有的自定变量都消失。在图4.1中，变量blurRow, blurCol, curRow, curCol, pixels, 和pixVal都是这种自动变量。注意获取这些变量是非常的快且并行的，但是要注意不能超过寄存器在硬件层面的限制。使用大量的寄存器变量能够使得分配到每个SM的活跃线程数（active threads）产生负面影响，我们将在第五章将这个问题。

数组形式的自动变量不存储在寄存器中。他们被存储到了全局变量中，这会导致长等待时延和潜在的访问堵塞。与自动标量变量相似，这些数组的域是单个线程，也就是说，每个线程都有自己的数组变量，当一个线程终结时，数组变量的值也被释放掉。根据我们的经验，自动数组变量很少在kernel函数中使用。

如果一个变量在声明前有“__shared__”（每个“__”中有两个下划线）关键词，这个变量声明了一个CUDA共享变量。在“__shared__”前面可选择加上“__device__”关键词，加或者不加都表示一个意思。这样的声明必须在kernel函数中或者device函数中。共享变量存储在共享内存中，共享变量的域是在一个线程区块中，也就是说区块中的所有线程都能够访问到同一个变量。每个线程区块在执行kernel的时候会生成一个单独的变量。当一个kernel终结时，这个变量的内容被释放。我们之前讲到过，共享变量是一种同一个区块中不同变量的很好的合作方法。访问共享内存中的共享变量非常快速且并行度很高，CUDA编程者经常将全局变量最常被访问的部分变量存储到共享变量中，由于着重于全局变量中的小部分数据的大量访问，因此有一些算法会重新设计，我们将在4.4节用矩阵乘法来讲解。

如果一个变量用“__constant__”关键字声明，那它声明的就是一个CUDA中的常量变量。同样前面可以加“__device__”关键字，也可以不加，效果一样。常量变量的声明必须在任何函数体的外部，常量变量的域是所有的网格，也就是说所有的网格中的所有的线程可以访问到同一个常量变量。长两变量的存在时间是整个应用的执行时间。常量变量常用于给kernel函数提供输入值，常量变量存储在全局内存中但可以更高效地访问。通过正确的访问模式，访问常量内存能够非常快速且并行度很高，目前常量变量在CUDA应用中的总大小不能超过65536字节。输入数据的大小需要符合这一限制，我们将在第七章详细介绍。

一个只用“__device__”声明的变量是一个全局变量并会被保存在全局内存中，访问全局变量是非常慢的，最近的device改进了访问全局变量的延迟和通量。全局变量的一个很重要的有事就是能被所有的kernel和线程访问，全局变量的内容会存在于整个执行过程，因此全局变量能被用作跨区块线程的合作上，但是，将不同区块上的线程同步还要保证不同线程中数据一致性的唯一简单方法就是终结正在执行的kernel。因此全局变量常用语在kernel调用之间传递信息。

在CUDA中，指针用于指向全局内存中的数据对象。指针在kernel函数中用一下两种方式出现：（1）当对象是由host函数分配的，指向对象的指针是cudaMalloc函数初始化的，这个指针能够传递给kernel函数作为参数，例如图4.3中的变量M,N,P。（2）一个在全局内存中声明的变量地址被赋值给一个指针变量。例如kernel函数中的 `float* ptr = &GlobalVar;` 将GlobalVar的地址赋给了自动指针变量ptr。读者应当参考CUDA Programming Guide来使用其他内存类型的指针。

4.4 用分块法减少内存流量

我们使用CUDA的device内存有一个固有的优缺点平衡，全局变量很大但是速度慢，共享内存很小但是速度快。一种常用的将数据拆分成多个部分的策略叫做分块法（tiling），将数据分成不同的块，从而使每一块各自存到合适的内存中。tile（瓦片）这个词的意思是用作类比，一面墙能被瓦片一块一块地覆盖住，类比全局变量能够由不同内存类型的变量一块一块地组成，重点是这些块上的kernel计算能够互不影响。注意不是任何kernel函数的所有数据结构都能被分成块。

分块的概念能够用矩阵乘法这个例子来阐释，图4.5展示了图4.3的kernel函数，为了方便读者，我们在图4.9中重复展示。为了简便，我们使用Py,x, My,x, 和Ny,x分别 代表 P[y*Width+ x], M[y*Width+ x], 和N[y*Width+ x]。这个例子假设我们使用2*2=4个区块来计算P矩阵。图4.9特别展示了block(0,0)的P0,0, P0,1, P1,0, 和P1,1四个线程的计算过程， block(0,0) 中的 thread(0,0) 和thread(0,1) 用黑色的箭头指出了他们对M和N的元素的访问过程，举个例子，其中thread(0,0)访问M0,0和N0,0，然后M0,1和N1,0然后M0,2和N2,0，然后M0,3和N3,0。

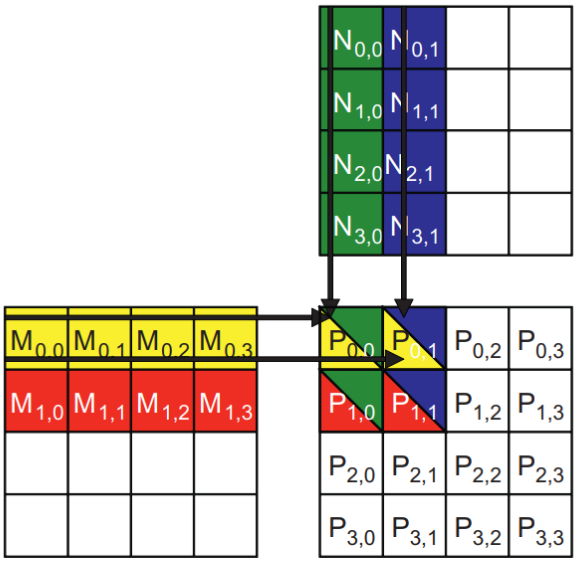

FIGURE 4.9

A small example of matrix multiplication. For brevity, we show *M*[*y**Width+ *x*], *N*[*y**Width + *x*], *P*[y*Width+ *x*] as $M_{y,x}$, $N_{y,x}$ $P_{y,x}$.

图4.10展示了block0,0中的所有线程访问全局内存的过程，所有的线程用垂直方向列出来，时间的增加用水平向右的方向表示。每个线程在执行过程中获取M的四个元素和N的四个元素。在图中高亮的四个线程我们可以看到在访问M和N的元素时有明显的重合部分，例如thread0,0和thread1,0都需要访问M中的row0行，相似的，thread0,1和thread1,1都需要访问N的column1列。

Access order

$thread_{0,0}$	$M_{0,0} * N_{0,0}$	$M_{0,1} * N_{1,0}$	$M_{0,2} * N_{2,0}$	$M_{0,3} * N_{3,0}$
$thread_{0,1}$	$M_{0,0} * N_{0,1}$	$M_{0,1} * N_{1,1}$	$M_{0,2} * N_{2,1}$	$M_{0,3} * N_{3,1}$
$thread_{1,0}$	$M_{1,0} * N_{0,0}$	$M_{1,1} * N_{1,0}$	$M_{1,2} * N_{2,0}$	$M_{1,3} * N_{3,0}$
$thread_{1,1}$	$M_{1,0} * N_{0,1}$	$M_{1,1} * N_{1,1}$	$M_{1,2} * N_{2,1}$	$M_{1,3} * N_{3,1}$

FIGURE 4.10

Global memory accesses performed by threads in $block_{0,0}$.

图4.3中的kernel的写法其实是让thread0,0和thread0,1都从全局内存中访问第0行元素，如果thread0,0和thread0,1能够合作使得M的这些元素只需要从全局变量中访问一次，那么总的全局内存访问量就减少了一半，每个M和N的元素在执行block0,0的时候被访问了两次，因此如果所有的四个线程能够合作访问全局内存，对全局内存访问的流量就减少了一半。

读者应当知道在这个例子中全局内存流量的潜在减少量是与使用的区块维度成比例的。如果是Width*Width的区块，潜在的全局内存流量的潜在减少量是Width。因此，如果我们使用16*16的区块，那全局内存的流量会通过线程间的合作大约减少到1/16。

交通拥堵就好比流量拥塞，如图4.11所示，高速路交通拥堵的原因是大量的车挤在为少量车行驶而设计的路上，当交通拥堵发生时，每辆车的行驶时间都显著上升。交通用时会轻易翻倍，甚至翻三倍。

FIGURE 4.11

Reducing traffic congestion in highway systems.

大多数的减少交通拥堵的方法是减少路上的车辆总数，假设总的通勤者的数量是恒定的，人们需要共享车辆从而减少在路上的车辆数，一个常见的方法是拼车，就是说一群通勤者轮流开车送所有人去上班，政府应当颁布政策鼓励拼车。在有的国家中，政府只是简单地每日禁用特定的部分车辆上路，比如，单双号政策，周一周三周五只允许单数号牌的车辆上路，这个规则鼓励了人们在不同日子选择拼车上班。在有的国家，汽油的价格过高，导致很多人拼车只为了省些油钱，很多国家都是对减少路上的车总数有激励措施的，在美国，一些拥堵的高速线路变成了拼车线路，只有车上有两个或者更多的人的车，才允许在这条线路上行驶。所有这些鼓励拼车的措施都旨在克服拼车需要额外努力的事实，如图4.12所示。

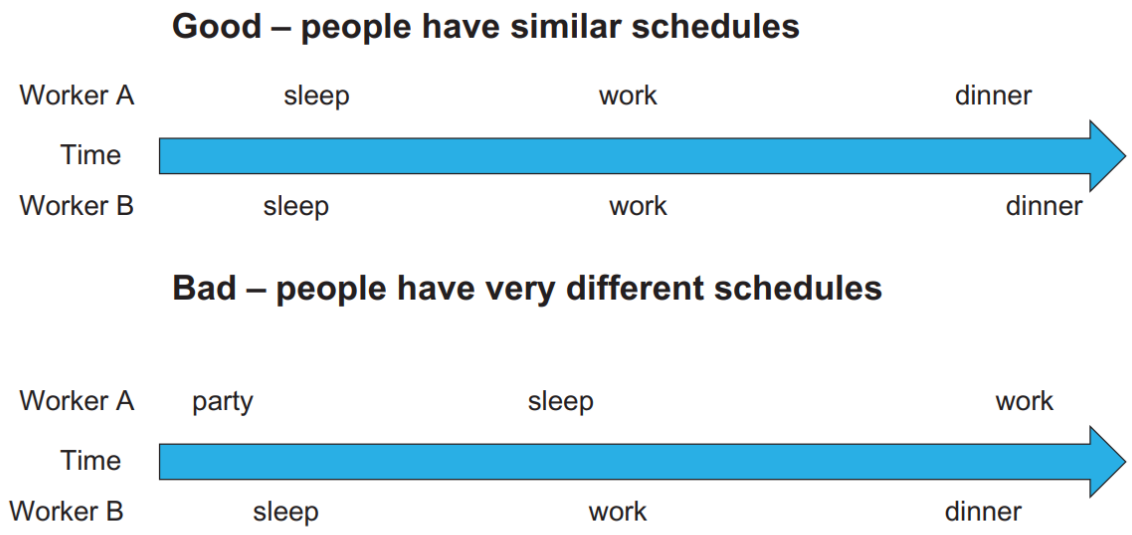

FIGURE 4.12

Carpooling requires synchronization among people.

拼车需要上班的乘客们有着相同的时间安排，在图4.12的上半部分展示了一个很适合拼车的时间安排，时间从左到右，工人A和工人B有着相同的睡觉、工作和晚饭时间。这个安排允许这两个工人做一辆车上下班。与之对比，在图中下半部分的时间安排不同，工人A和工人B有着不同的习惯，工人A都要多人运动到天亮，然后白天睡觉，晚上上班，工人B在晚上睡觉，在白天工作，下午6点回家恰饭。这个时间安排相差过多导致这两个工人不能上下班做一辆车。如果想拼车，就要乘客们有像上半个图所示的时间安排。

分块算法与拼车安排很相似，我们可以把访问数据的线程当做通勤者，把DRAM访问请求当做车辆，当DRAM请求率超过了带宽的限制时，交通堵塞发生了，此时算法单元空闲了下来，如果多线程访问DRAM中同一位置的数据，他们能够自行地组成“拼车小队”，然后一起去DRAM去请求，但是这需要一个相似的线程执行调度来让他们对数据的访问结合起来，这个情景在图4.13中展示了出来，其中一个个小格子代表DRAM的不同位置，有很多从DRAM位置指出来的箭头，指向了一个线程，这个箭头代表线程在不同的时间访问不同的内存位置，时间从左向右，箭头指向线程的不同时间点。上半部分展示了两个线程在不同时间访问同一内存，其中第二个线程与第一个相比明显慢了半拍。之所以下面这种安排的方式不好是因为：从DRAM中拿回来的数据需要存储在芯片上的内存中一段时间来让线程二来执行，这样的话需要在芯片内存上存储大量的数据元素，需要很大的芯片内存。

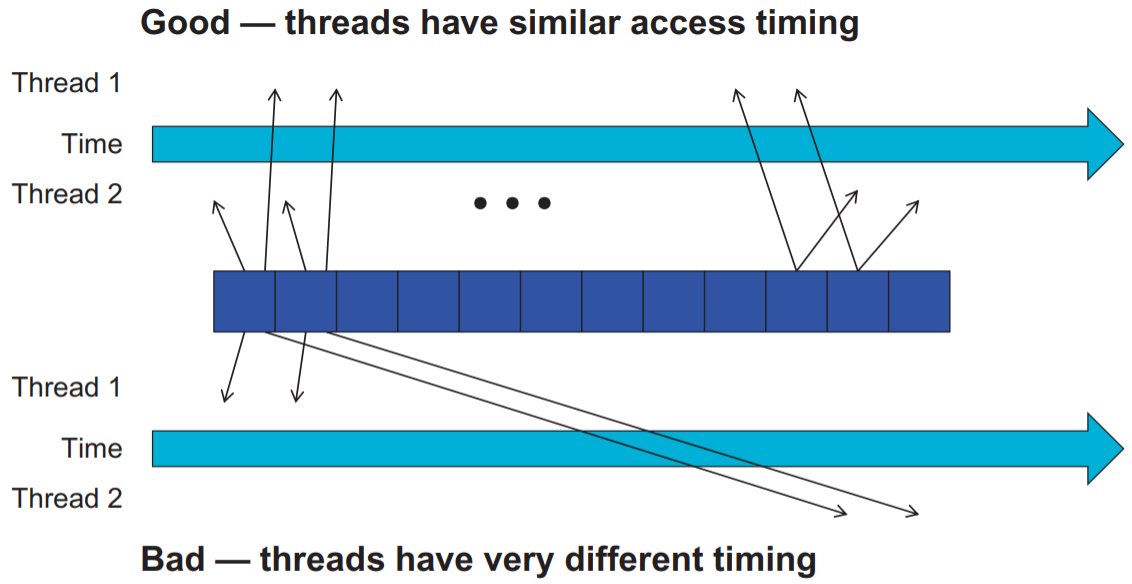

FIGURE 4.13

Tiled Algorithms require synchronization among threads.

在并行计算领域，分块是一种编程转换技巧，用来本地化不同线程的内存位置，并规划访问时间。它将每个线程的长访问序列划分为多个阶段，并使用同步阻碍函数使访问每个部分的时间间隔保持紧密。这个技术通过在时间和空间上本地化访问，从而控制了芯片上内存数量的需求。用我们拼车的例子对比，我们强迫“拼车小队”的线程们用同一个时间安排表执行程序。

我们现在展示分块矩阵乘法算法。基础的想法是：那些合作的线程们将加载M和N的一小部分元素，在进行乘法计算之前将之存储在共享内存中，共享内存的大小并不大，在加载M和N的元素的时候不能超过共享内存的限制，这个情况应当将M和N分成小块，然后将小块放到共享内存中去。最简单的形式是，块的维度与区块的维度相同，如图4.11。

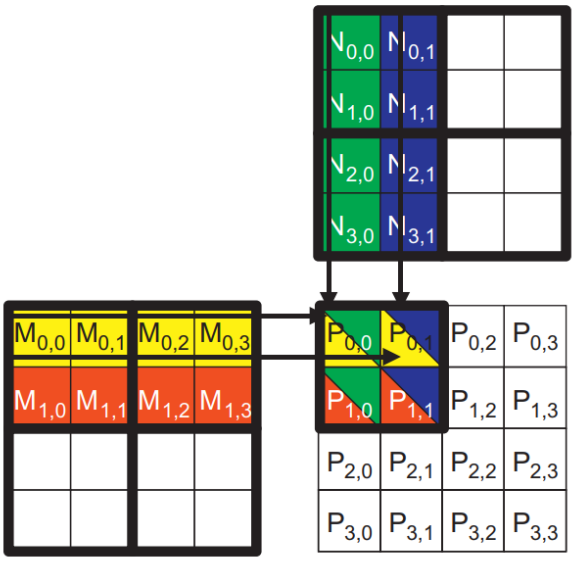

FIGURE 4.14

Tiling *M* and *N* to utilize shared memory.

图4.11中，我们将M和N分成2*2的小块，途中用粗线表示小块之间的分隔，每个线程的乘法现在被分成的多个阶段，每个阶段中，区块中所有的线程合作加载M的一块和N的一块，可以通过使一个块中的每个线程将一个M元素和一个N元素加载到共享内存中来完成这种合作，如图4.15所示，每个行表示一个线程的执行活动，时间从左至右，我们只需要展示block0,0中的线程活动，其他区块都有相似的行为，M元素的共享内存数组称作Mds，N的称为Nds。在第一阶段开始时，block0,0四个线程合作加载M的一个小块进入共享内存：thread0,0加载M0,0至Mds0,0，thread0,1加载M0,1至Mds0,1，thread1,0加载M1,0至Mds1,0，thread1,1加载M1,1至Mds1,1（如图4.15第二列所示）。N的一小块的加载也类似，如图4.15第三列所示。

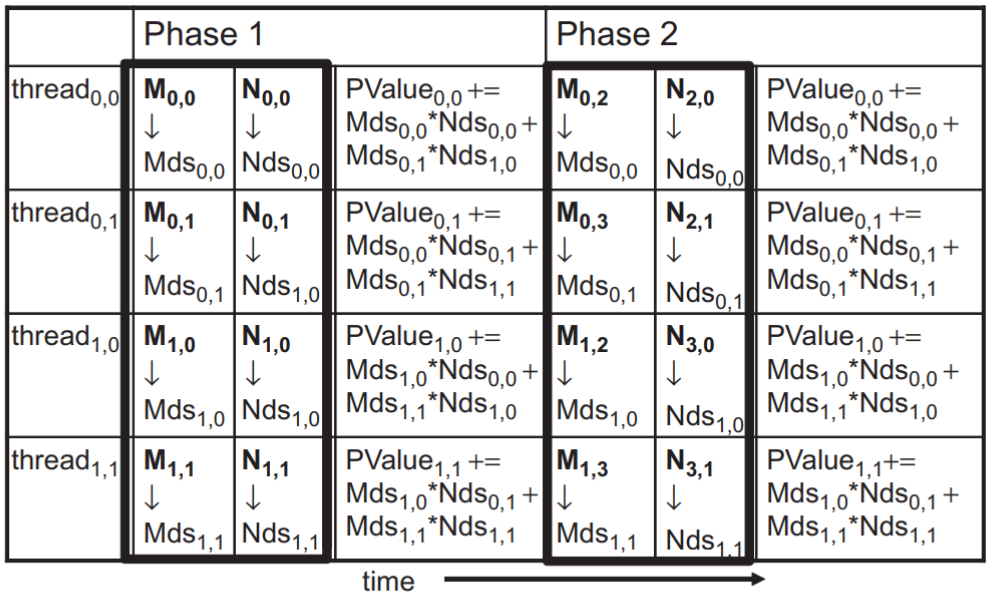

FIGURE 4.15

Execution phases of a tiled matrix multiplication.

在M和N的两个小块加载到共享内存中之后，这些元素被用来计算乘法，每个在共享内存中的值被使用两次，例如M1,1的值在thread1,1家在城Mds1,1，这个值被用了两次，第一次thread1,0使用，第二次thread1,1使用，通过加载每个全局内存值到共享内存，每个值再多次使用，我们减少了全局内存的访问，在这个例子中我们减少了一半的访问数量。读者应当验证如果小块是N*N的那么就减少了N的倍数。

图4.3中的每个点乘现在被分成了图4.15的两个阶段，在每个阶段中，两个输入矩阵元素的乘积被累加到Pvalue中。Pvalue是一个自动变量，在每个线程中都有一个这样的变量，我们使用下标来表示不同的线程的Pvalue值。在第一阶段和第二阶段中的计算在图4.15中的第四列和第七列展示了出来。总的来说，如果一个输入矩阵的维度是宽Width，小块的宽度是TILE_WIDTH，那点乘就将被分成Width/TILE_WIDTH个阶段，这些阶段的创建是减少全局内存访问的关键。每个阶段着重于输入矩阵值的一小部分，线程之间能够合作加载不同部分进入共享内存，使用共享内存中的值来满足阶段中相同的输入需求。

Mds和Nds被复用于承载输入值，每个阶段中，相同的Mds和Nds被用来承载M和N的部分元素，从而使得更小的共享内存为大多数全局内存的访问服务。每个阶段只需要一小部分的输入矩阵元素，这种行为被称为本地化（locality 感觉也可以翻译成局域化）。当一个算法有本地化的特点时，就能用更小，更快的内存来完成大多数的访问，从而减少全局内存的访问数量。本地化对多核CPU和多线程GPU的高性能非常重要。我们将在第五章再讲本地化的概念。

4.5小块矩阵乘法kernel

我们现在可以开始展示小块矩阵乘法的kernel，其中使用共享内存来减少全局内存的流量。图4.16的kernel实现了图4.15的执行过程。在图4.16中，第一行和第二行声明了Mds和Nds为共享内存变量，我们之前讲过，共享内存变量的域是区块。因此，一堆Mds和Nds在生成在每个区块之中，区块中的所有的线程能够访问相同的Mds和Nds。这点非常重要，因为只有在一个线程的小伙伴将所需要的M和N的元素全都加载到共享内存中后，才能够使用这些值从而完成自己的需求。

```
__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_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 d_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 d_M and d_N tiles required to compute d_P element
8.  for (int ph = 0; ph < Width/TILE_WIDTH; ++ph) {

      // Collaborative loading of d_M and d_N tiles into shared memory
9.    Mds[ty][tx] = d_M[Row*Width + ph*TILE_WIDTH + tx];
10.   Nds[ty][tx] = d_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. d_P[Row*Width + Col] = Pvalue;
  }
```

FIGURE 4.16

A tiled Matrix Multiplication Kernel using shared memory.

第三行和第四行将threadIdx和blockIdx值保存到一个自动变量中，也就是保存到了能够快速访问的寄存器中。我们之前讲过，自动标量变量是保存在寄存器中的，域单个线程，也就是说运行时系统为每个线程都生成了tx,ty,bx和by变量，并将它们存在寄存器中可被每个线程单独地快速访问。这些变量被初始化为threadIdx和blockIdx的值，并在下面的线程存在时间中被多次访问，当线程终结时，变量的值被释放。

第五行和第六行决定了线程计算的P元素的row和col索引。这个代码假设了每个线程都能够计算一个P元素，如第六行所示，P元素的水平（x）位置，或者说列的索引，能计算为bx*TILE_WIDTH+ tx，因为每个区块在水平方向覆盖TILE_WIDTH个元素，一个bx的区块的线程，至少有bx个区块，或是说bx*TILE_WIDTH个线程在它前面，这些线程占P的bx*TILE_WIDTH个元素，另外同个区块中还有tx个线程在其前面，因此bx和tx应被用来计算P的x为bx*TILE_WIDTH+tx的元素。水平索引被存储到Col变量中，图4.17也展示了这一情况。

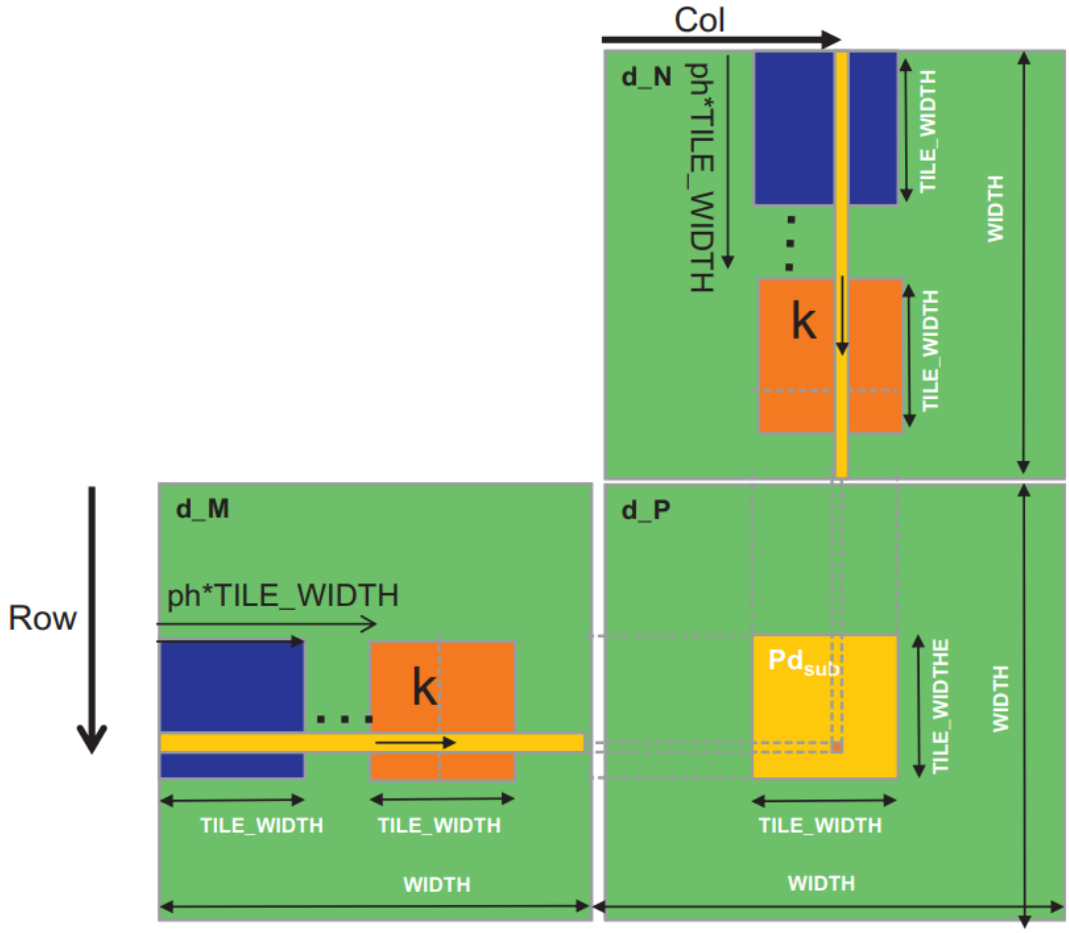

FIGURE 4.17

Calculation of the matrix indexes in tiled multiplication.

图4.14中，P元素的x索引是通过block1,0的thread0,1计算而来，0*2+1 = 1。相似的，y索引能用by*TILE_WIDTH+ ty计算。这个垂直索引被保存在每个线程的Row中，因此每个线程计算Col列和Row行的P元素，如图4.17。我们在4.14中说过，P元素的y索引能够通过block0,1的thread1,0计算而来，1*2+0=2。因此这个线程计算的是P2,1。

图4.16中的第八行标记了遍历计算P元素所有阶段的循环的开始，每个循环对应图4.15中的一个阶段。ph值代表已经完成点乘的阶段数，之前说过每个阶段使用M的一小块和N的一小块元素，因此在每个阶段的最开始，ph*TILE_WIDTH对的M和N元素已经被预先处理了。

在每个阶段中，第9行加载对应的M元素进入共享内存，因为我们已经知道M的行和N的列要被线程处理，我们现在讨论M的列索引和N的行索引。正如图4.17所示，每个区块有TILE_WIDTH^2个合作的线程来加载TILE_WIDTH^2个M元素到共享内存中，因此我们只需要分配每个线程加载一个M元素，我门能通过blockIdx和threadIdx很轻易地完成这件事。加载的M部分元素列索引的开始是ph*TILE_WIDTH，因此一种简单的方法是让所有的线程加载一个距离开始位置tx（threadIdx.x）的元素。

第九行表示的就是这种情况，其中每个线程加载M[Row*Width + ph*TILE_WIDTH + tx]，其中线性化的索引是通过行索引Row和列索引ph*TILE_WIDTH + tx计算而来。既然Row的值是ty的线性函数，每个TILE_WIDTH^2个线程将会加载唯一的M元素进入共享内存中。总的来说，这些线程将会加载一个图4.17中的一个M的一部分。读者应当使用图4.14和图4.15中的例子来验证每个线程的地址计算结果无误。

第十一行的阻碍函数保证了所有的线程在完成对M和N的加载后再进行下一步操作。第十二行展示了基于这一小块的元素的一个阶段的点乘操作。thread（ty,tx）的循环执行过程展示在了图4.17中，箭头k指向M和N元素的访问方向，第十二行中的k变量就是箭头k。这些元素将会从Mds和Nds中访问，第十四行的阻碍函数__syncthreads()保证了所有的线程在进行下一步的操作之前已经完成了对M和N需要进行的所有操作，通过这种方式，没有线程会过早地加载或破坏其他线程的输入值。

第八行到第十四行的嵌套循环展示了一种叫做strip-mining的技术，这种技术将长循环转换成多个阶段。每个阶段都包含一个内部循环，该循环执行原始循环的多个连续迭代。原始循环变为外部循环，其作用是迭代调用内部循环，以便原始循环的所有迭代均按其原始顺序执行。 通过在内循环之前和之后添加同步阻碍函数，我们强制同一块中的所有线程将其工作完全集中在其输入数据的一部分上。 剥离可以通过在数据并行程序中分块来创建所需的阶段。

在所有的阶段的点乘结束后，第八行的循环执行结束。所有的线程用Row和Col线性化过来的索引将结果写入P中。

分块算法提供了一个潜在的好处。对于矩阵乘法，全局内存的访问量减少了TILE_WIDTH的倍数，如果我们使用16*16的块，我们可以减少16次全局变量的访问。这将“计算全局内存访问比”增加到1:16。这个增长允许CUDA的device的内存带宽达到它的性能极限，例如，一个device有150GB/s的全局内存带宽，那算力可达到600GFLOPS。

虽然分块矩阵乘法kernel的性能增长是很显著的，但是我们简化了很多因素。首先，矩阵的宽度是区块宽度的整数倍，这个假设使得我们的kernel不能正确处理任意大小的矩阵。其次，我们还假设了矩阵是方阵，这点在真实情况中不多见，在下一小节中，我们将展示一个有边界检测的kernel来消除这些假设条件。

4.6边界检测

我们现在将分块矩阵乘法扩展到任意宽度，扩展将会使得kernel正确地处理宽度不是小块宽度整数倍的矩阵。图4.18是将图4.14中的例子转换成M,N,P都是3*3的图。矩阵有的宽度为3，并不是2的倍数，图4.18显示了block0,0阶段1的内存访问模式。Thread0,1和thread1,1将尝试加载不存在的M个元素。类似地，thread1、0和thread1、1将尝试访问N个不存在的元素。

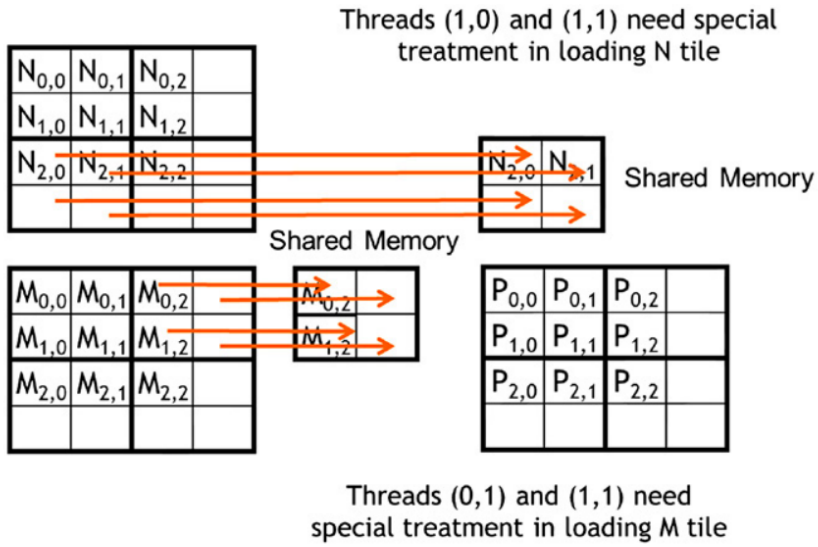

FIGURE 4.18

Loading input matrix elements that are close to the edge–phase 1 of $Block_{0,0}$.

访问不存在的元素有两方面的问题。访问超过行尾的不存在的元素(图4.18中thread1、0和thread1、1的M次访问)将对不正确的元素执行。在我们的示例中，线程将尝试访问不存在的M0、3和M1、3。这种情况下内存将会怎样加载？回答这个问题我们要先回到二维数组的线性化的问题，M0、2之后的元素是M1、0，尽管thread0、1尝试访问M0、3，但是实际访问到的缺失M1、0，使用这个值进行内积计算将会明显破坏输出值。

当访问超过最后一列元素的时候会产生一个相似的问题。这些访问的地址是超出数组所分配的地址的，一些系统会从其他数据结构中返回随机的值，但是还有的系统会拒绝这个访问并让程序停止和报错。任意一种方式都会不能得到正确的输出。

我们讨论了这么久，感觉问题只会出现在最后一个阶段线程的执行中。我们可以通过对最后一个阶段进行特殊处理来解决这个问题。但是不幸的是，任何一个阶段都有可能出现问题，图4.19展示了block1、1在第0阶段的内存访问模型。我们可以看到thread1、0和thread1、1尝试访问不存在的M3、0和M3、1元素，同时thread0、1和thread1、1尝试访问N0、3和N1、3，这些元素都是不存在的。

注意，通过把相应线程不包括进来是不能解决这些有问题的访问的。举例来说，block1、1的thread1、0不计算任何P元素，但是它需要在第一阶段加载M2,1，还有一些计算P元素的线程会尝试访问不存在的M和N元素，如图4.18所示，block0、0的thread0、1计算P0、1，但是这个线程在阶段1尝试访问不存在的M0、3。这些现象告诉我们，需要对M小块加载、N小块加载和P元素的计算使用不同的边界检测模型。

我们从加载输入小块的边界检测模型开始，当一个线程尝试访问一个输入小块元素时，这个线程应当先测试这个输入元素是否符合限制条件，可以通过检测y和x的索引轻易做到，在图4.16的第九行中，线性化的索引是从值为Row的x索引和值为ph*TILE_WIDTH + tx的y索引计算而来。边界条件测试将会检测两个索引是否都小于Width：(Row<Width) && (ph*TILE_WIDTH+tx)<Width。如果条件满足，那么线程将会加载M元素，读者应当自行验证N元素的边界测试为： (ph*TILE_WIDTH+ty)<Width && Col<Width。

如果这个情况不能满足，线程不应当加载元素，在这个例子中还有个问题，就是应当把什么元素存储到共享内存中呢？答案是0.0，这个值不会对内积的计算产生任何影响。

最终，一个线程如果计算合法P元素的话，应当只存储最终内积值。这个条件的边界测试是(Row < Width) && (Col < Width)。有额外边界条件的kernel代码如图4.20所示。

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

        // Collaborative loading of M and N tiles into shared memory
 9.     if ((Row< Width) && (ph*TILE_WIDTH+tx)< Width)
            Mds[ty][tx] = M[Row*Width + ph*TILE_WIDTH + tx];
10.     if ((ph*TILE_WIDTH+ty)<Width && Col<Width)
            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.   if ((Row<Width) && (Col<Width)P[Row*Width + Col] = Pvalue;
```

FIGURE 4.20

Tiled matrix multiplication kernel with boundary condition checks.

有了边界条件检测，这个分块矩阵乘法kernel距离一个总的矩阵乘法kernel只差一步了。总的来说，矩阵乘法是为矩形方阵设计的，一个大小为j*k的矩阵M与一个k*l的矩阵N相乘，结果是一个大小为j*l的矩阵P。目前我们的kernel只能够处理方形矩阵。

幸运的是，我们的kernel能被轻易的扩展为一个普通矩阵乘法的kernel。首先Width参数被三个整数参数j、k和l取代，其中Width原来是代表P矩阵的高度或M矩阵的高度，这个值会被j取代，用Width表示的M的宽度和N的高度，会被k取代，用Width表示的N和P的宽度，会被l取代。这个kernel的详细解释过程留给读者作为练习。

4.7内存作为并行的限制因素

虽然CUDA寄存器和共享内存能够有效地减少全局内存的访问数量，但是我们必须小心不要超过这些内存的能力限制。这些内存是线程执行的必要资源之一，每个CUDA的device只提供有限的资源，因此限制一个SM中的同时存在的线程的总数。总的来说，每个线程需要越多的资源，每个SM中的同时存在的线程数就越少，反之亦然。

为了说明kernel寄存器的使用与device所支持的并行等级之间的关系，假设现在有一个device叫D，每个SM能够容纳最多1535个线程和16384个寄存器，虽然16384个寄存器是一个很大的数字，但是每个线程允许使用的只是其中很小的一部分寄存器。为了支持1536个线程，每个线程只能只用16384/1536=10个寄存器，如果每个线程使用11个寄存器，那么每个SM中的执行的线程总数将会减少，这个减少的量是以区块单位的，例如每个区块由512个线程，线程的减少将会是一次就减少512个。因此，比1536更小的线程数是1024个线程，这样一下子就减少了1/3的同时存在的线程数。这个过程能够潜在地减少可被调度的warp总数，从而减少了在高时延操作时的处理器能被找到了可执行warp。

每个SM可用的寄存器每个device都不一样，一个应用能够动态地为每个device的SM指定可用的寄存器，并选择特定的device使用合适数量的寄存器数。寄存器的数量能通过cudaGetDeviceProperties函数得到，我们在3.6节讲过这个函数。假设&dev_prop的值传递给了函数，函数将device的性质写入到这个变量中，那么dev_prop.regsPerBlock就会展示每个SM中的可用寄存器总数。对于device D，这个值会返回16384，应用能够将这个数分到SM中的线程中去，然后每个线程能用多少个寄存器就确定了下来。

共享内存的使用仍然可以限制每个SM中的线程数，我们可以假设device D有16384字节（16k）的共享内存，这些内存在区块中，区块在SM中，我们可以假设D中每个SM能最多有8个区块，这样的话，为了达到最多的区块数，每个区块最多可疑使用不能超过2k字节的共享内存，不然的话，每个SM中的区块的数量将会减少，直到所有的区块的共享内存的总数不超过16k字节，例如，如果每个区块使用5k字节的共享内存，每个SM中最多有3个区块。

对于矩阵乘法的例子，共享内存可以变成一个限制条件。对于大小为16*16的小块，每个区块需要16*16*4 = 1k的存储空间，用来存储Mds。注意每个元素都是浮点数，因此每个元素占4字节，还需要为Nds留1kb的存储空间。因此，每个区块使用2k字节的共享内存，16k字节的共享内存最多能够允许8个区块同时在SM中存在，该硬件最多允许16k共享内存，我们使用了16k，因此共享内存不会成为这个例子的限制。此时，真正的限制是每个SM最多允许1536个线程，这就使得我们最多每个SM只能有6个区块，因此最多有6*2kb = 12kb个共享内存被使用了，这些限制每个device都不同，但是却可以通过运行时查询到。

每个SM的共享内存大小会根据device的不同而不同，每一代device或每个device的模型都有不同的共享内存总量限制（在每个SM中）。我们肯定希望kernel会根据不同的可用共享内存总数来尽可能多地使用，我们希望host代码能够动态的指定共享内存的大小并调整kernel使用的共享内存的总量，这点我们还是可以通过CUDAGetDeviceProperties函数做到。还是假设&dev_prop传给了函数，那么我们就能通过dev_prop.sharedMenPerBlock得到每个SM中共享内存的总数（原文这里写的是寄存器总数，明显不对）。编程者可通过总数决定每个区块分多少共享内存。

不幸的是，图4.16的kernel不支持这么做，图4.16中使用的声明将其共享内存使用量的大小硬性地编译为编译时常量：

```
__shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
__shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

也就是说，无论TILE_WIDTH在编译时是多大，Mds和Nds的大小被设定为了TILE_WIDTH^2个元素，我们假设

```
#define TILE_WIDTH 16.
```

Mds和Nds都有256个元素，如果我们想要改变Mds和Nds的大小，我们需要改变TILE_WIDTH的大小并重新编译代码。kernel不能轻易地在不重新编译的情况下改变共享内存的使用。

我们可以在CUDA中使用不同的声明风格来实现这样的调整。我们可以在共享内存声明前添加C " extern "关键字，并在声明中省略数组的大小。这样，Mds和Nds的声明可写为：

```
extern __shared__ Mds[];
extern __shared__ Nds[];
```

注意，数组现在是一维的。我们将需要使用基于垂直和水平索引的线性化索引。在运行时，当我们启动内核时，我们可以根据设备查询结果动态地确定要使用的共享内存数量，并将其作为第三个配置参数提供给内核启动。修改后的内核可以通过以下语句启动

```
size_t size=
  calculate_appropriate_SM_usage(dev_prop.sharedMemPerBlock,...);
matrixMulKernel<<<dimGrid, dimBlock, size>>>(Md, Nd, Pd, Width);
```

其中，size_t是一个内置类型，用于声明一个变量来保存动态分配的数据结构的大小信息。大小以字节表示。在我们的矩阵乘法示例中，对于一个16×16的小块，我们有一个16×16×4=1024字节的大小。省略了在运行时设置大小值的计算细节。

4.8总结

总的来说，现代处理器程序的执行速度会被内存的速度所限制。为了达到CUDA的device更好的性能发挥，我们需要kernel代码有更高的“计算全局内存访问比”。如果这个比值很低，kernel就会被内存限制，例如，程序的执行速度受到从内存中访问其操作数的速度的限制。

CUDA定义了寄存器，共享内存和常量内存。这些内存相比全局内存更小，但是能够更快地访问。有效地使用这些内存需要良好的算法设计。我们用矩阵乘法来演示分块算法，这是一种被广泛使用的曾倩数据获取本地化的一种算法，能够高效地利用共享内存。在并行编程中，分块迫使多个线程在执行的每个阶段共同着重于输入数据的子集，以便子集数据可以被放置到这些特殊的内存类型中，从而提高访问速度。我们证明，在16×16平铺的情况下，全局内存访问不再是矩阵乘法性能的主要限制因素。

然而，CUDA编程者需要意识到这些类型的内存的大小有限。他们的能力是根据不同的代码应用所不同的。一旦超出了它们的容量，它们就会限制可以在每个SM中同时执行的线程数量。在开发应用程序时，判断硬件限制的能力是计算思维的一个关键方面。

尽管我们介绍了CUDA编程的分块算法，这种分块技术实际上被用于各种高性能并行计算系统之中。原因是应用程序必须通过将数据本地化访问之后才能够达到对内存的告诉访问。在多核CPU系统之中，数据本地化允许应用高效的使用芯片上的数据缓存来减少内存访问延迟，从而提高性能。因此，读者会发现分块算法在其他的编程模型的并行计算系统中也非常好用。

我们本章的目的是介绍本地化的概念，分块，和不同的CUDA内存类型，寄存器和常量内存在分块算法中的使用还没有讲到，这些内存种类在分块算法中如何使用我们将在并行算法模式中讲解。

4.9习题

看书做吧，各位。