第三章 可扩展的并行执行

在第二章数据级并行中，我们学习了如何写一个能够加载kernel并操作一维网格的CUDA C程序。kernel指定了每个线程的C语言程序。当我们进行如此大规模的活动时，我们需要对这些活动加以控制来达到我们所需要的结果、效率和速度。在本章节中，我们将要学习控制并行程序执行的重要相关概念。我们将要从学习线程索引（下标）和区块索引如何帮助我们处理多为数组开始，然后我们将探讨资源的灵活分配和占用，再然后，我们将深入线程管理、等待时限和同步。精通这些概念的CUDA编程者才会具有编写和理解高性并行计算程序的能力。

CUDA线程机制

一个网格中所有的CUDA线程执行相同的kernel函数，他们依赖坐标来处理所要处理的部分数据从而实现相互分离。这些线程被组织成一个两层的结构：一个由一个或多个区块组成的网格、每个区块由一个或者多个线程组成。在同一个区块中的所有线程都有同一个区块索引，也就是kernel中blockIdx的值。每个线程都有一个线程索引，也就是kernel中的threadIdx值。当一个线程执行一个kernel函数，blockIdx和threadIdx的值反映了这个线程的坐标。kernel加载语句的执行配置参数制定了网格的维度以及每个区块的维度，这些维度是kernel函数中gridDim和blockDim的值。

> **层级结构**
>
> 与CUDA线程相似，许多现实世界的系统也是分层级架构的。美国的电话系统就是一个好的例子。在最顶层，电话系统是由*区域*构成，每个区域对应着真实世界地理上的区域。所有的在同一区域中的电话号码具有相同的*区域码*，一个电话区域可能会比一个城市大：比如在Central Illinois的许多县和城市都在同一个电话区域中，都使用217作为区域号码。在一个区域中，每个电话号码有一个7位数的本地号码，者能够让每个区域最多能够拥有千万个不同的电话号码。
>
> 每个电话号码可以看成是CUDA的线程，每个线程的区域号码就好比blockIdx，7位数的本地号码就好比threadIdx。这个层级结构允许系统能够容纳大量的电话号码并且实现了同一区域打电话的*本地化*。当给相同区域的电话号码打电话时，只需要拨打7位数的本地号码就行。如果我们大多数的电话都是打向本地的话，我们就会很少使用区域号码，如果我们偶尔需要给其他区域的号码打电话，我们只需拨打数字1加上区域号，在加上本地号码就行了。CUDA的线程层级结构同样也是本地化的一种形式。

总的来说，一个网格是一个区块的三维数组，每一个区块是一个线程的三维数组。当加载一个kernel的时候，程序需要指定每个维度上网格和区块的大小。编程者如果使用少于3的维度，可以通过将不使用的维度的大小设置为1，一个网格的具体结构是在kernel加载时在执行配置参数（<<<>>>中）指定的。第一个执行配置参数制定了网格上每个维度有多少个区块，第二个参数指定了每个区块的维度上有多少个线程。每个维度都是dim3类型，dim3类型是C的struct类型，三个属性都是无符号整形x,y,z。这些属性制定了三个维度的大小。

下面这个host代码能够加载vecAddkernel()并生成一维网格，网格中有32个区块，每个区块由128个线程组成。所以网格中所有的线程数是128*32 = 4096.

```
dim3 dimGrid(32, 1, 1);
dim3 dimBlock(128, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(…);
```

注意：dimBlock和dimGrid是编程者在host代码中定义的。这些变量能够取任何名字，只要这个变量名符合C的语法，并且这个变量是dim3类型。比如，下面这个生命完成了与上面的代码相同的事情。

```
dim3 dog(32, 1, 1);
dim3 cat(128, 1, 1);
vecAddKernel<<<dog, cat>>>(…);
```

网格和区块维度是能够从其他变量计算而来。图2.15中的kernel加载就是像下面这样写的。

```
dim3 dimGrid(ceil(n/256.0), 1, 1);
dim3 dimBlock(256, 1, 1);
vecAddKernel<<<dimGrid, dimBlock>>>(…);
```

区块的数量会随着向量的大小不同而不同，但需要网格中有足够多的线程来处理向量中的元素。在这个例子中，编程者选择将区块的大小定位256，在kernel加载时，变量n的值将决定网格的维度，如果n等于1000，网格将由4个区块组成，如果n是4000，网格将由16个区块组成。在每个例子中，都会有足够的线程来处理向量的元素。当vecAddkernel函数加载后，网格和区块维度将保持相同，知道整个网格完成了执行。

为了方便，CUDA C提供一个特殊的简便方法来加载一维的kernel。可以不适用dim3变量，而是用数学表达式来指定一维网格和区块的参数。这种情况下，CUDA C编译器把数学表达式当做x的维度并且嘉定y和z的维度是1.因此，图2.15中的kerne加载表达式是这样的。

```
vecAddKernel<<<ceil(n/256.0), 256>>>(…);
```

熟悉C的结构的读者会意识到这个“简写”表达式实际上是得益于x是dim3结构的第一个属性。这个简写允许编译器将x属性初始化给gridDim和blockDim变量。

在kernel函数中，gridDim和blockDim中x属性的值是根据执行配置参数而预先初始化的。如果n等于4000，vectAddkernel的gridDim.x和blockDim.x分别是16和256。不像host中的dim3变量，kernel函数中的这些变量作为CUDA C的特殊标明是不能够随意改变的，例如kernel中的gridDim和blockDim总是反映了网格和区块的维度。

在CUDA C中，gridDim的x、y、z的允许范围是1到65536。区块中的所有线程有着相同的blockDim.x、y、z值。在所有block中，blockIdx.x的值范围是从0到blockIdx.x-1，blockIdx.y的值范围从0到blockIdx.y-1，blockIdx.z的值范围从0到blockIdx.z-1。

关于区块的配置如下，每个区块的结构是一个线程构成的三维数组。可以通过将blockDim.z设置为1来创建一个两维的区块，可以通过将blockDim.y和blockDim.z设置为1来建立一个一维的区块，就像vectorAddkernel一样。如刚才所述，网格中所有的区块有着相同的维度和大小，区块每个维度的线程数量是被第二个执行配置参数在kernel加载时设置的，在kernel中，这个配置参数能够通过blockDim的x、y、z属性来访问到。

区块的大小不能多于1024个线程，只要总线程数不多于1024就总能把这些元素不多于1024。比如，blockDim(512,1,1), blockDim(8,16,4)和blockDim(32,16,2)是允许的blockDim值，但是blockDim(32,32,2)是不允许的因为总线程数超过了1024。

网格能够有比区块更高的维度，反之亦然。例如图3.1中展示了一个网格例子gridDim(2,2,1)，blockDim(4,2,2)。网格能够通过以下host代码生成。

```
dim3 dimGrid(2, 2, 1);
dim3 dimBlock(4, 2, 2);
KernelFunction<<<dimGrid, dimBlock>>>(…);
```

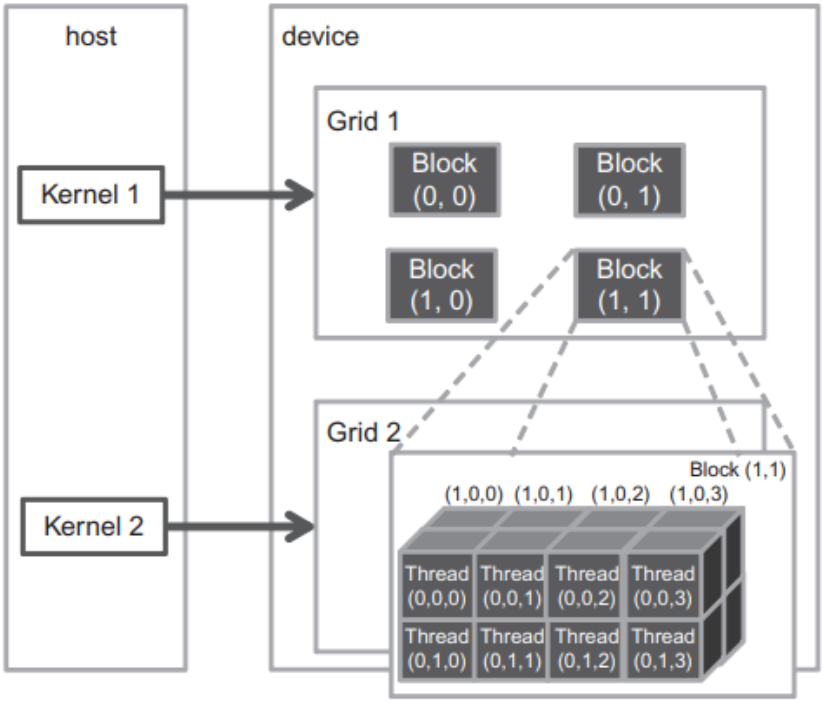

FIGURE 3.1

A multidimensional example of CUDA grid organization.

网格由四个区块组成2*2的数组，图3.1中每个区块都被标记为(blockIdx.y, blockIdx.x)，例如Block(1,0)的blockIdx.y=1,blockIdx.x=0.这么标记是因为最高的维度可以放在最前面，注意：这里的标注与C语言设置配置参数时的标注相反，设置配置参数时是将最低的维度放在最前面。这种反向的对block的标注使得当我们要表明线程的坐标与多维数据索引的关系时非常的高效。

每个threadIdx包含三个属性：x坐标threadIdx.x，y坐标threadIdx.y，z坐标threadIdx.z。在图3.1中展示了区块中线程的结构。在这个例子中，每个区块被架构成4*2*2的线程数组。网格中所有的区块都有着相同的维度，因此我们只需要知道其中一个block的参数。在图3.1中放大了Block(1,1)，展示了其中的16个线程。例如。Thread(1,0,2)的threadIdx.z=1，threadIdx.y=0，threadIdx.z=2。这个例子展示了4个区块，每个区块由16个线程，网格中一共有64个线程。我们用这个比较小的数字是我们的例子简单好理解，实际上CUDA网格包含上千到上百万个线程。

3.2线程到多维数据的映射

一维、二维或者三维具体选择那个是由数据的特点所决定的。图片是二维的像素点，因此使用二维的网格，网格中用2维的区块通常来说处理起图片中的像素来更加的方便。图3.2展示了这样一个要处理76*62个像素的图片的网格（水平或x方向76个像素，垂直或y方向62个像素。假设我们决定使用一个16*16的区块（x方向16个线程，y方向16个线程），我们将需要x方向5个区块，在y方向4个区块，总共5*4=20个区块。加粗的线表示区块的边界，阴影的部分秒回了处理像素的线程。用下面的公式，很容易就找到了block(1,0)的thread(0,0)处理的是哪一个图片像素：

$$P_{\text{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x}} = P_{1*16+0,0*16+0} = P_{16,0}.$$

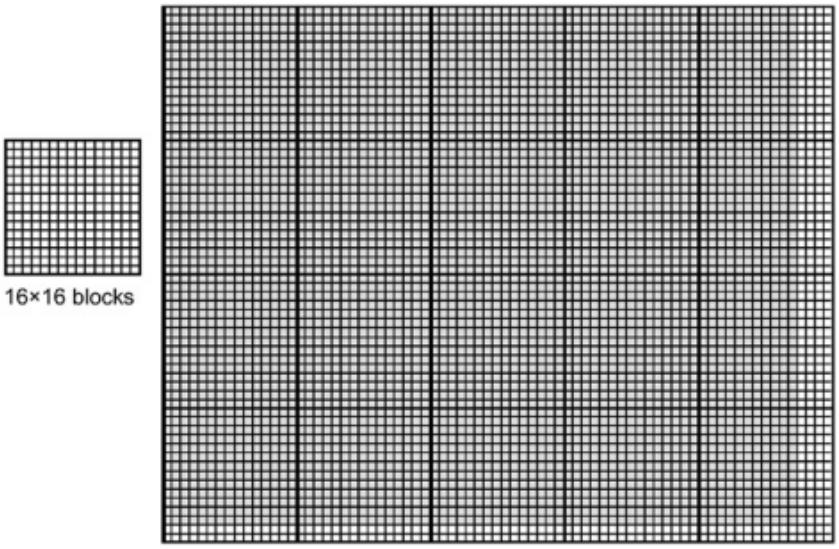

FIGURE 3.2

Using a 2D thread grid to process a 76 × 62 picture *P*.

注意：在x方向我们有4个额外的线程，在y方向我们有2个额外的线程，也就是说我们制造了80*64个线程来处理76*62个像素。这个情况与图2.11中用4个256线程区块来处理1000个元素向量的vecAddKernel很像。回忆当时我们用if表达式来防止最后24个线程的操作，这里相似，我们也应该用if表达式来判断线程的索引threadIdx.x和threadIdx.y是否在像素的索引范围内。

假设host代码用整形变量m来表示x方向的像素个数，整形变量n来表示y方向的像素个数。再假设输入的图片数据已经被复制到device内存中并且能够通过指针d_Pin来访问到。输出图片已经在device内存上分配了内存，并且能够通过指针d_Pout访问到。下面的这个host代码能够被用来加载一个二维的kernel函数colorToGrayscaleConversion来处理图片。

```
    dim3 dimGrid(ceil(m/16.0), ceil(n/16.0), 1);
    dim3 dimBlock(16, 16, 1);
colorToGreyscaleConversion<<<dimGrid,dimBlock>>>(d_Pin,d_Pout,m,n);
```

在这个例子中，我们假设区块的维度是固定的16*16。同时网格的维度取决于图片的维度，要处理2000*1500（300万像素）的图片，我们将会生成11750个区块——x方向125个，y方向94个。在kernel函数中，gridDim.x，gridDim.y，blockDim.x，blockD.y分别是125,94,16和16。

> **内存空间**
>
> 内存空间是描述现代计算机处理器访问内存的简化概念。它通常与每个正在执行的应用相关联。一个应用要处理的数据和要执行的指令都存储在这个应用的内存空间中。尤其是，每一个内存空间中的位置都有一个字节并且有一个地址，需要多个字节数据的变量（例如浮点数需要4个字节，双精度浮点数需要8个字节）在内存空间中存储在连续的位置中。当需要到内存空间访问数据时，处理器会生成开始地址（变量第一个比特的位置的地址）和所需的字节数。
>
> 内存空间中的位置和电话系统中的电话号码很像，内一个人都有唯一的电话号码。大多数现代计算机至少有4G比特的位置，G代表1073741824（2的30次方）。所有的位置都标记为一个0到最大数字的地址，每个位置都有唯一的地址，因此我们可以说内存空间是一个扁平的组织架构。因此，所有的多为数组最终都会被扁平化成为一维的数组。但是C编程者能够使用多维的语法来访问多维数组的元素，编译器将这些访问转换成一个指向数组开始元素的指针，并附带这多维索引计算出的偏移量。

在我们战术kernel代码之前，我们需要先来理解C语言是如何访问动态分配的多维数组的。理想情况下，我们想通过d_Pin[j][i]来访问一个二维数组的第j行和第i列的元素。但是，CUDA C所基于的ANSI C标准要求，如果d_Pin作为二维数组访问的话，必须要在编译时就知道这个二维数组的列数，但不幸的是，这个列数在编译时是不知道的，因为这是一个动态分配的数组。实际上，我们使用动态分配数组的一个

原因就是我们想根据运行时数据的不同而分配不同大小的数组，因此我们在设计时，这个二维动态分配的数组的列数就是在编译时不可知的。因此编程者需要将动态分配的二维数组线性化或者扁平化，从而在CUDA C中变成等价的一维数组。更新的C99标准允许多维的语法和动态分配的数组。未来的CUDA C版本可能会支持动态分配数组的多维语法。

实际上，C中的所有的多维数组都是线性化的，因为现代计算机的内存空间都是扁平化的。在静态分配数组中，编译器允许编程者使用高维索引语法，例如d_Pin[j][i]来访问元素，其实编译器背地里将这种数组转换成线性的一维数组，并将多维索引计算成一维的索引。在动态分配数组中，CUDA C编译器将这部分工作留给了编程者，因为在编译时缺少维度的信息。

一个二维的数组有两种方式线性化：第一种是把所有同行的元素放在连续的位置，然后不同的行一个接一个排布下去，这种方式叫做行主导的布局，在图3.3中描绘了出来。为了增加可读性，我们将使用Mj,i表示M的第j行第i列的元素，Pj,i等同于C语言中的M[j][i]，只是可读性更强。图3.3展示了一个4*4的矩阵M是如何线性化成为一个16个元素的一位数组的，显示第0行的所有的元素，然后紧接着是第1行所有的元素，等等。所以，M第j行第i列元素的一维等价索引是j*4+i。j*4跳过了所有第j行之前的元素，i跳过了第j行最前面的i个元素。M2,1的一维索引是2*4+1=9，如图3.3， M9的确是M2,1的一维表达。这个过程展示了C编译器是图和线性化二维数组的。

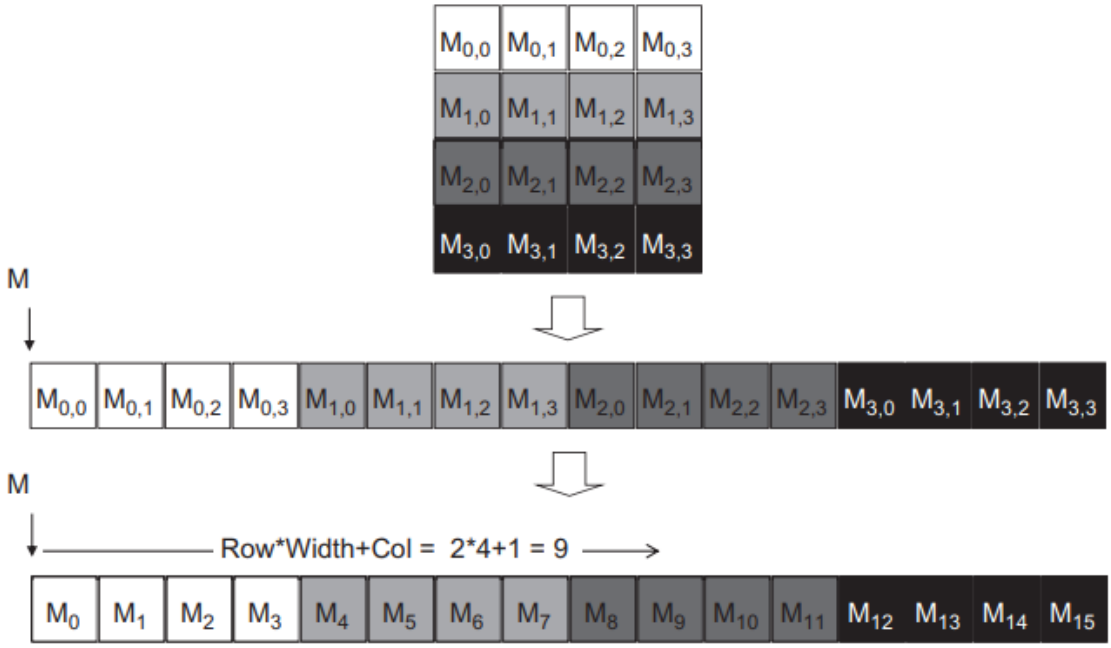

FIGURE 3.3

Row-major layout for a 2D C array. The result is an equivalent 1D array accessed by an index expression *j**Width+ *i* for an element that is in the *j* th row and *i* th column of an array of Width elements in each row.

另一种线性化的方法是将同一列的元素放在连续的位置，然后下一个列在内存空间中紧接着继续排列。这种方式成为列主导的布局，是FORTRAN编译器所采用的方式。二维数组的列主导布局与行主导的布局的转换方式是相同的。有过FORTRAN变成金利的读者应该知道CUDA C用的是行主导的布局而不是列主导的布局。此外，很多为FORTRAN程序使用的C库为了适应FORTRAN，使用的都是列布局，因此这些库的使用说明中会提到，如果在C程序中使用这个库，请自己手动转换，例如Basic Linear Algebra Subprograms(BLAS)。

> **线性代数函数**
>
> 线性代数操作被广泛使用于科学与工程应用中。BLAS，一种发行库的事实标准（de facto standard），它包含三个等级的线性代数函数，随着等级的提升，函数所执行的操作数量提升。level1函数实现形式为y=ax+y的向量操作，其中x和y是向量，a是标量。我们向量加法的例子是a=1的level1函数的特殊情况。level2函数实现形式为y=aAx+by的矩阵向量操作，其中A是矩阵，x和y是向量，ab是标量。我们将在稀疏线性代数中研究level2形式的函数。level3函数实现矩阵-矩阵操作，形式如C = aAB + bC，其中ABC是矩阵，ab是标量。我们矩阵-矩阵乘法的例子就是一个level3函数

> 的特殊例子，其中a=1， b=0。这些BLAS函数被用作高级代数函数的基础组成部分，例如线性系统求解器和奇异值分析。就像我们将要分析的一样，不同BLAS函数的实现方法（如顺序计算和并行计算）会导致性能的数量级的变化。

我们现在准备好了学习colorToGreyscaleConversion函数的源码了，如图3.4所示，kernel代码将每个像素转换成黑白像素所使用的公式如下：

$$L = r * 0.21 + g * 0.72 + b * 0.07$$

```
// we have 3 channels corresponding to RGB
// The input image is encoded as unsigned characters [0, 255]
__global__
void colorToGreyscaleConversion(unsigned char * Pout,  unsigned
            char * Pin, int width, int height) {,
int Col = threadIdx.x + blockIdx.x * blockDim.x;
int Row = threadIdx.y + blockIdx.y * blockDim.y;
if (Col < width && Row < height) {
   // get 1D coordinate for the grayscale image
   int greyOffset = Row*width + Col;
   // one can think of the RGB image having
   // CHANNEL times columns than the grayscale image
   int rgbOffset = greyOffset*CHANNELS;
   unsigned char r = Pin[rgbOffset    ]; // red value for pixel
   unsigned char g = Pin[rgbOffset + 2]; // green value for pixel
   unsigned char b = Pin[rgbOffset + 3]; // blue value for pixel
   // perform the rescaling and store it
   // We multiply by floating point constants
   Pout[grayOffset] = 0.21f*r + 0.71f*g + 0.07f*b;
 }
}
```

FIGURE 3.4

Source code of colorToGreyscaleConversion showing 2D thread mapping to data.

水平方向上总共能找到blockDim.x*gridDim.x这么多的线程。如vecAddKernel那个例子，Col=blockIdx.x*blockDim.x+threadIdx.x产生的每个整数范围是从0到blockDim.x*gridDim.x–1。我们知道gridDim.x*blockDim.x是大于等于宽度（host代码中传入的m值）的，所以我们至少在水平方向上有着和像素个数一样的线程数，垂直方向也一样。因此，只要我们测试并操作所有线程的Row和Col值在范围中（Col<width && Row<height）的线程，我们就能处理图片中的所有像素。

我们一直每个行有宽度width个像素点，因此我们可以我们可以讲第Row行第Col列的元素的索引转换成一维的形式——Row*width+Col。这个一维的索引greyOffset是输出的黑白图片的每个像素点Pout的索引，数据类型是无符号字符型，大小为1字节。76*62这个例子中，是通过下面这个公式计算thread(0,0)中block(1,0)的一维索引的：

$$Pout_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} = Pout_{1*16+0,0*16+0} = Pout_{16,0} = Pout[16 * 76 + 0] = Pout[1216]$$

对于Pin，我们将黑白像素的索引乘以3，因为每个像素的是存储为rgb的，每个元素都占一个字节。rgbOffset的结果提供了Pin数组中某个像素的起始地址，我们从这个地址开始连续的读出rgb三个元素的值，计算黑白像素的颜色值，并将最终结果用greyOffset索引写到Pout数组中。在76*62这个例子中，是通过如下公式计算thread(0,0)的block(0,0)的Pin像素一维索引的：

$$Pin_{blockIdx.y*blockDim.y+threadIdx.y,blockIdx.x*blockDim.x+threadIdx.x} = Pin_{1*16+0,0*16+0} = Pin_{16,0} = Pin[16 * 76 * 3 + 0] = Pin[3648]$$

通过这个一维索引访问的索引值是3648，访问的内存大小为3字节。

图3.5表示了当colorToGreyscaleConvention在处理76*62图片时的样子。假设我们使用16*16的区块，加载colorToGreyscaleConvention函数产生了84*64个线程。网格将会有5*4=20个区块，区块的执行特点将分为4大类，在图片3.5中用4中情况进行了描绘。

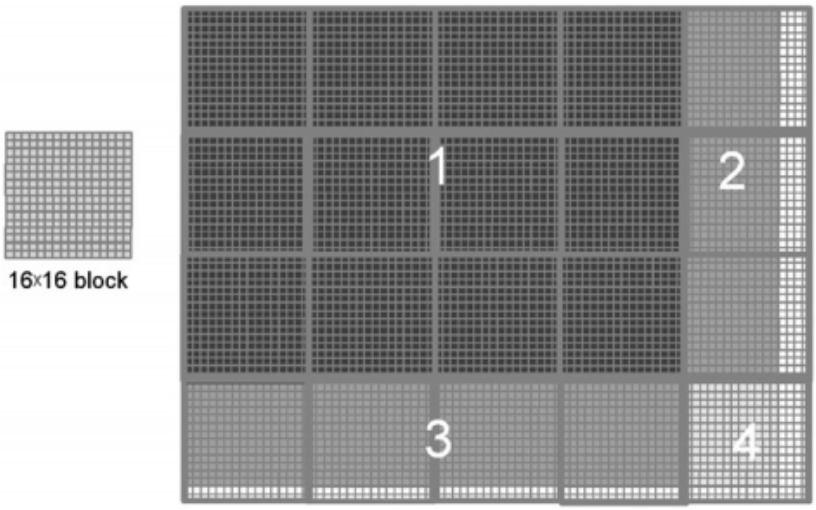

FIGURE 3.5

Covering a 76 × 62 picture with 16 × 16 blocks.

第一个区域，在图3.5中标记为1，是由12个区块组成，是处理图片的大部分区块。这里面所有的线程的Col的值和Row的值都在范围之内，所有的线程都会通过if表达式并处理图片中深色阴影所描绘的像素，也就是说每个区块中有16*16=256个线程来处理像素。在第二个区域在图3.5中标记为2，是由中等深度阴影所描绘的三个区块中，处理图片右上区域像素的线程构成。虽然这些线程的Row值总是在范围内，但是Col值有些会超出m值（76），原因是线程在水平方向上的个总是blockDim.x的整数倍，在本例子中，也就是16的整数倍，最小的超过76的16的背水是80,。因此这一区域中的一行中12个线程Col值在范围内，同时4个线程Col值在范围外，也就是说这四个线程无法通过if表达式，他们无法处理任何像素。总的来说每个区块中16*16=256个线程中的12*16=192个线程将会处理像素。

第三个区域在图3.5中标记为3，是由3个左下方的中等深度阴影秒回的区块组成。尽管每个线程的Col值总是在范围中，一些Row值超过了m值（62），原因是在垂直方向上的线程个数总是blockDim.x的整数倍，本例子中是16的倍数。大于62的最小的16的倍数是64，因此这每列的16个线程中14个线程Row值在范围之中，2个线程的Row将不能通过if表达式，因此不会处理任何像素。总共的256个线程中224个将会处理像素。第四个区域在图3.5中用4标记，由处理图片右下角的线程组成，用浅色阴影描绘。在最上14行中，每行有4个线程的Col值在范围之外，这点与区域2相似，最底下两行的线程Row值在范围之外，这点与区域3类似，因此，16*16=256个线程中只有14*12=168个线程处理像素。

我们能够在线性化数组的过程中简单的把二维数组线性化扩展成3维数组的线性化，可以通过计算Plane值而简单完成。假设编程者使用变量m和n来表示行数和列数，编程者在加载kernel时仍然需要决定blockDim.z和gridDim.z的值，在kernel中，数组的索引将包含另一个全局索引：

```
int Plane = blockIdx.z*blockDim.z + threadIdx.z
```

线性化的访问一个三维数组P将会是这种形式`P[Plane*m*n+Row*3+Col]`。一个处理三维P数组的kernel需要检查所有的三个全局索引Plane,Row,Col是否在合理的数组范围内。

3.3图片虚化：一个更复杂的kernel

我们已经学习了vecAddkernel和colorToGreyscaleConversion，这两个例子中的每个线程只处理一个像素的非常少量的代数操作。这两个例子完成了它们的任务：阐释CUDA C最基础的变成结构和数据级并行执行概念。这是，读者们应该问一个问题，是不是所有的CUDA线程都只是完成这样简单直接的操作并且相互独立？答案是——不！在真实的CUDA C程序中，线程通常是要完成更加复杂的数据计算并且相互之间有合作。接下来的介个章节，我们将处理复杂程度明显上升的例子，来展示这些特点。我们从图片虚化函数讲起。

图片虚化将像素之间的突然变化变得更加平滑，但任然保留一些轮廓使得我们能够识别关键的一些图片特点和信息。图3.6展示了图片虚化的效果。简单地说就是将图片弄得模糊。对于人眼来说，一个虚化的图片时将一些细节隐藏起来并呈现一种“大图片”的感觉，或者表现图像中主要对象的一种形式。在计算机图片处理算法中，一个常用的图片虚化作用是减少噪点的影响，通过早点周围的像素迷糊化的值来校正有问题的像素点。在计算机视觉中，图片虚化能被用作边界识别和对象识别算法从而找到图片的主题对象，而不是被大量的图片中其他对象所干扰。在实际中，通过对周围图片像素虚化，可被用作聚焦图片的特定部分。

FIGURE 3.6

An original image and a blurred version.

从数学上来讲，一个图片虚化函数是计算周围一定数量像素值得加权平均。我们将会在第七章学习卷积，像这种计算周围像素加权平均就是卷积模型。我们本章将使用一个简单的方法，知识简单地将周围N*N的像素值进行取平均来得到目标值。为了简单，我们并不会根据周围像素点距离中心的距离来设置权重值，这点与卷积虚化不同，卷积虚化通常使用高斯函数作为权重。

图3.7展示了一个利用3*3作为patch大小的虚化例子，当计算一个在（Row, Col）的输出像素的值时，我们可以看到patch的中心点被放在了点（Row, Col）上，因此这个3*3的patch跨越3行和3列，行为（Row-1, Row, Row+1），列为（Col-1, Col, Col+1）。图片中的九个像素的坐标分别是 (24, 49), (24, 50), (24, 51), (25, 49), (25, 50), (25, 51), (26, 49), (26, 50), 和(26, 51)。

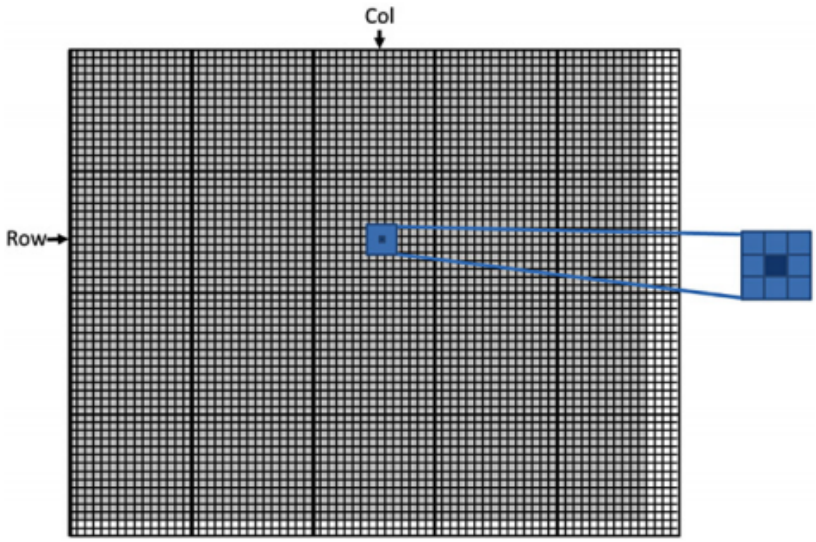

FIGURE 3.7

Each output pixel is the average of a patch of pixels in the input image.

图3.8展示了一个图片虚化kernel。相似于colorToGreyscaleConversion，我们每一个线程计算一个输出像素的值，因此线程与输出数据的位置映射是没有变的，因此在kernel的最开始我们看到了相似的Col和Row索引值，我们还看到了熟悉的if表达式，用来验证Col和Row是否在范围之内，只有Col和Row的值都在范围内的线程才能够处理像素。

```
__global__
void blurKernel(unsigned char * in, unsigned char * out, int w, int h)
    {
  int Col  = blockIdx.x * blockDim.x + threadIdx.x;
  int Row  = blockIdx.y * blockDim.y + threadIdx.y;

  if (Col < w && Row < h) {
1.    int pixVal = 0;
2.    int pixels = 0;

    // Get the average of the surrounding BLUR_SIZE x BLUR_SIZE box
3.    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
          }
        }
      }

    // Write our new pixel value out
10.   out[Row * w + Col] = (unsigned char)(pixVal / pixels);
  }
 }
```

FIGURE 3.8

An image blur kernel.

正如图3.7中所示，Col和Row值代表patch的中心的位置坐标。图3.8中的for循环遍历了patch中所有点的位置。我们假设程序已经有了一个定义好的常量BLUR_SIZE，因此2*BLUR_SIZE就是这个patch每个边上的像素点的数量。在这个3*3的patch中，BLUR_SIZE的值为1，在7*7的patch中BLUR_SIZE的值为3。外层的循环遍历了patch的所有行，每一行中，内层的循环有遍历所有的列。

在3*3的例子中BLUR_SIZE是1。对于计算（25,50）这个点的输出的线程，curRow变量的值是Row-BLUR_SIZE = (25 − 1) = 24，因此在第一个外层循环中，内层循环便利了第24行的所有patch中的像素，内层循环通过变量curCol从第Col-BLUR_SIZE = 50 − 1 = 49列遍历到 Col+BLUR_SIZE = 51 列。因此在第一个外层循环中处理的像素点是 (24, 49), (24, 50), 和(24, 51)。读者应当已经明白了，在第二个外层循环中遍历的是(25, 49), (25, 50), 和 (25, 51)，最终在第三个外层循环中遍历的是 (26, 49), (26, 50), and (26, 51)。

第八行使用了将curRow和curCol线性化之后的索引来访问当前位置的输入像素。代码将这些像素的值加和到变量pixVal中。第九行记录了总共有多少个像素被加入到了总和之中，存储为变量pixels。在所有patch中的像素被处理之后，第十行计算patch中所有像素值得加和平均值，通过用pixVal除以pixels得到。并用线性化的Row和Col坐标写入到输出像素位置。

第七行有一个if条件判断第九行和第十行是否要执行，对于靠近图片边缘的输出像素，patch坐标可能会超出合理范围。这点在图3.9中展示了出来。在1所示的情况中，在图片左上角的像素被虚化了，9个像素中只有5个是在图片中不真实存在的，这种情况下，输出像素的Row和Col的值分别为0和0。在执行循环的时候，CurRow和CurCol的值分别是 (−1,−1), (−1,0), (−1,1), (0,−1), (0,0), (0,1), (1,−1), (1,0), 和(1,1)。注意，这5个不存在的像素值，坐标中至少有一个值是小于0的，因此if表达式中curRow<0和curCol<0这两个条件将使得这些像素不被程序处理，跳过第八行和第九行。因此只有4个像素点的坐标能够通过if表达式并将像素点的值加和，并pixels变量加一，一共加了4次，这样第10行才能正确地计算平均数。

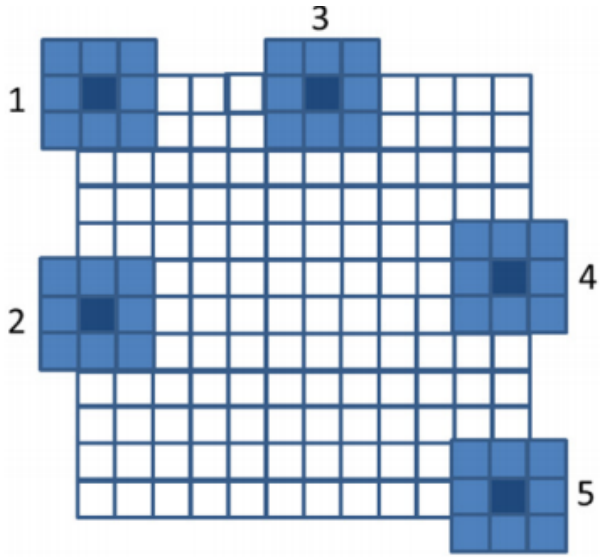

FIGURE 3.9

Handling boundary conditions for pixels near the edges of the image.

读者应当自行去思考图3.9中的其他情况并分析blurKernel的循环执行特点。注意，大多数线程都能找到对应的patch中3*3的全部像素，并会将9个像素全部加起来，但是对于边缘的4个像素点，只会加起来4个像素点的值，对于边缘上的其他patch，循环只会让6个像素点的值相加。pixels变量必须跟随这些变化而计算到底多少个点被加了进来。

3.4同步和明显的扩展性

我们至此已经讨论了如何加载kernel并让grid中的线程执行，还有如何将线程映射到对应的数据的一部分。但是我们还没有讲到多线程是如何被调度执行的。我们现在将要学习一个调度机制的概念。CUDA允许同一个区块中的线程通过使用一个同步阻碍函数（barrier synchronization function）__syncthreads()来相互协调合作。注意这个函数开头有两个下划线。当一个县城调用__syncthreads()函数是，该线程会停止在这个函数被调用的位置，直到这个区块中的所有的线程也达到了这个位置。这个过程可以保证一个区块中的所有线程在进行下面一些操作之前都完成了一段kernel执行代码。

同步阻碍函数是一个并行调度的简单且很流行的方法。在现实生活中，我们经常使用同步阻碍函数来进行并行调度，原因有很多。为了说明原因，假设4个朋友乘车去购物，他们可以都做不同的车，去不同的商店去买自己的衣服，这就是一个典型的并行模式，相比于顺序地一起去一个接一个的店铺，这种方式更加高效。但是在他们离开商店之前需要同步阻碍函数，最先买完的小伙伴要等待其他小伙伴都买满之后才可以走。没有同步阻碍函数的话，这个小伙伴离开的时候，可能会有1个或2个小伙伴还没买完，这无疑会让友谊的小船说翻就翻。

图3.10展示了同步阻碍函数的执行。区块中有N个线程，时间是从左到右，一些线程比其他线程先到达了同步阻碍函数的表达式，早到的线程等待其他线程，知道最晚的线程也到达了所有的线程都开始继续执行。有了同步阻碍函数之后，就没有线程会落后。

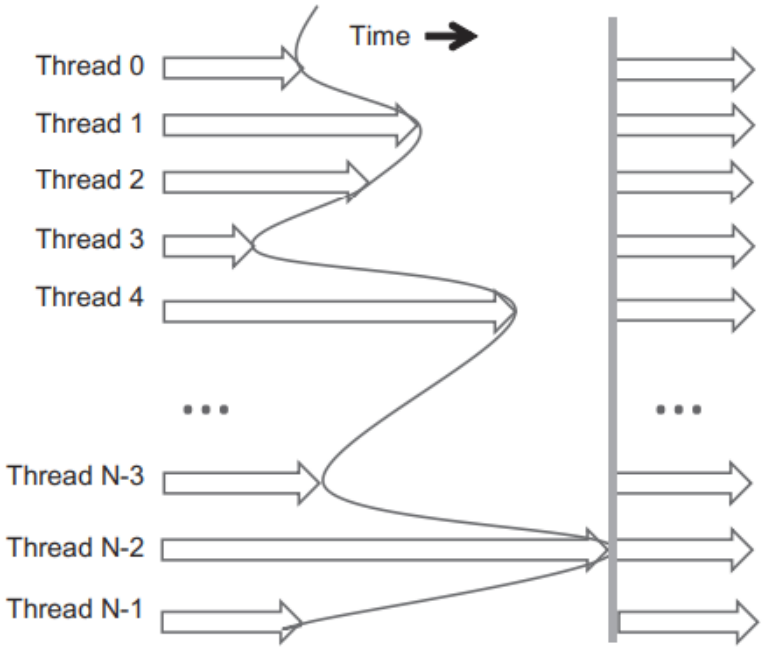

FIGURE 3.10

An example execution timing of barrier synchronization.

在CUDA中，如果声明了__syncthreads()表达式，那么所有这个区块中的线程都必须执行这一表达式，当一个__syncthreads()表达式被放在了if表达式中时，要么所有的线程都执行这句话，要么都不执行。对于一个if-then-else表达式，要么所有的线程都走then这一边，要么都走else这一边，不然的话then中的__syncthreads()与else中的__syncthreads()所停的位置不同，这是不允许的。如果一个线程停在了then中的阻碍函数，一个线程停在了else中的阻碍函数，二者因为停下来的点不同，会互相等待对方，但因为永远等不到，因此程序会停滞。编程者应当保证他们写的代码符合这一要求。

同步的功能还有就是对统一区块中的线程强加执行限制，这些线程应当在时间上彼此相近，从而避免了过长的等待时间。事实上，我们应当保证所有参与同步阻碍的线程最终会访问到所需的资源，从而最终能够到达同步阻碍函数的位置，否则一个永远到达不了同步阻碍函数的线程会导致所有其他线程停滞。CUDA运行时系统通过给一个区块中的所有线程作为一个单位分配执行资源，来满足这个条件。一个区块只有当运行时系统保证了其中所有线程都有足够的执行资源之后，才开始执行。当区块中的一个线程被分配资源后，这个区块中的其他线程都会被分配相同的计算资源，这样的话可以保证区块中线程在时间上很接近，防止同步阻碍等待时间过久和不确定性。

这就导致我们在设计CUDA程序是要为同步阻碍函数考虑一些事情。因为不同的区块中的线程不需要相互等待同步阻碍，因此CUDA运行时系统可以用任何顺序执行区块，这种灵活性就使得程序具有可扩展性，如图3.11所示。途中时间是由上到下，当程序在执行资源较少的系统中运行时，只有很少的区块在同时执行，如图3.11左侧所示。在执行资源较多的系统中运行时，会有较多的区块同时执行，如图3.11右侧。

这种执行相同的应用代码，却可以有着不同执行速度的性质，使得我们可以根据成本，能源，产品需求，特定的市场情况，从而有不同的实施方案。例如，一个移动端处理器可以较慢地执行代码同时功耗很低，一个笔记本处理器会增大功耗同时运行的更快，两种系统执行相同的代码。这种能够在不同的硬件设备上使用不同的执行资源的特点成为明显的可扩展性，这个特点减少了应用开发者的负担，并增加了可用性。

3.5资源分配

当一个kernel加载时，CUDA运行时系统生成对应的线程网格，正如之前讲到的那样，这些线程的执行资源是一个区块一个区块这么分配的。在当前的硬件系统中，执行资源是被架构成流式多处理器的（streaming multiprocessors 字太多了，以后简写SM）。图3.12展示了多个区块分配到每个SM中的情况。每个device设置一个数字表示每个SM最多能够分配多少个区块，例如我们假设一个CUDA的device最多允许8个区块分配到一个SM中，当有一种或者多种资源不足以支持8个区块所有线程同时执行时，CUDA运行时系统会自动减少分配到每个SM中的区块个数，直到使用的总资源数在限制之内。由于限制了SM的数量和每个SM中分配的区块的数量，CUDA的device能够启动的区块数量也是有限的。大多数网格能容纳比这多的多的区块。运行时系统有一个待执行的区块列表，当一个SM执行完之后，将列表中的区块分配给SM。

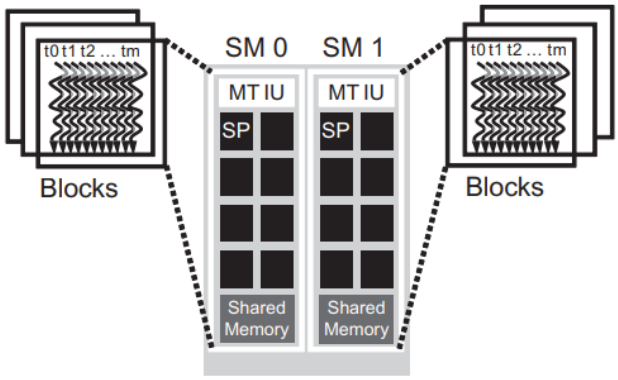

FIGURE 3.12

Thread block assignment to Streaming Multiprocessors (SMs).

图3.12展示了一个例子，每个SM分配三个线程区块，SM的其中一个资源限制是同时监视与安排的线程总数，硬件方面需要提供给SM资源（例如内置寄存器）从而维持线程和区块的索引、执行状态等信息。因此每一代硬件都会设置每个SM区块和线程的最大数量限制。例如Fermi架构，每个SM最多8区块和1536线程。这个参数还可以是像6区块，每个区块256个线程，或者3区块，每个区块512个线程等等。如果这个device只允许最多每个SM中8个区块，那么很明显你要用12个区块，每个区块128个线程肯定是不行的。如果一个CUDA的device有30个SM，每个SM能最多容纳1536个线程，device最多能够有46080个线程同时存在。

3.6查询设备的属性

将执行资源分配给区块产生了一个新的问题：我们怎么知道有多少资源？当一个CUDA应用在系统上执行的时候，如何确定一个device上的SM数量和每个SM可分配的区块数或者线程数？其他的资源由于与CUDA的执行相关我们还暂时没有讨论。总的来说，很多现代的应用被设计成在很多种类硬件上执行的，应用经常需要去查询有多少资源还有硬件层面有多大的能力，从而使得高性能的系统物尽其用，低性能的系统中收敛性能。

在CUDA C中，存在一个内置的机制，用来帮助host代码来查询系统中可用的device的属性。CUDA运行时系统（device驱动）有一个API叫做cudaGetDeviceCount，这个函数返回系统中有几个可用的CUDA的device。host代码能够通过以下代码确定有多少个能用的CUDA的device。

```
int dev_count;
cudaGetDeviceCount(&dev_count);
```

> **查询资源和能力**
>
> 在每日生活中，我们总是在查询环境中有多少可用的资源和能力。当我们要预定一个宾馆时，我们会查看宾馆房间中有多少生活用品和便利设施，如果房间中提供电吹风，我们就不用自己带一个了。大多数美国的宾馆都提供电吹风，但很多其他地区的宾馆不提供。
>
> 一些亚洲和欧洲的宾馆提供牙膏，有的还有牙刷，但是美国宾馆并不提供。许多美国宾馆提供洗发香波和护发素，但是很多其他地域的往往只提供洗发香波。
>
> 如果房间中有一个微波炉和一个冰箱，我们就能将剩饭剩菜保存起来，过一天再吃。如果宾馆有一个游泳池，我们可以带上泳衣，在完成工作后去游一会儿泳，如果宾馆没有游泳池但是有健身房，那我们可以带上跑鞋和健身衣。一些高端的亚洲酒店甚至还提供健身衣！
>
> 这些便利设施其实就是酒店的属性之一，或者说是酒店的资源和能力，经验丰富的旅行者可以在酒店网站上查看这些属性，选择更符合他们需求的酒店，并根据这些详细信息而打包行李。

但或许大家伙都不知道的是，很多现代PC都有两个或者多个显卡，原因始很多PC都有集成显卡。集成显卡是默认的图像处理单元，为现代的基于Windows的简单交互界面提供最最基础的能力与硬件资源，大多数CUDA应用在这些集成显卡上表现得很差，这就是为什么我们要在host代码上寻找所有的资源，并选择充足的资源让代码更好地执行。

CUDA运行时系统将所有可用的device从0至dev_count-1进行变化，并提供API函数cudaGetDeviceProperties，传入device的编号后，会返回这个device的属性信息。我们能够在host代码中这样遍历获取所有device的信息：

```
cudaDeviceProp dev_prop;
for (int i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties(&dev_prop, i);
  //decide if device has sufficient resources and capabilities
}
```

这个cudaDeviceProp是一个内置的C结构体，内有一些属性，代表着CUDA设备的属性，读者应当去看CUDA C Programming Guide从而了解所有的属性和类型。我们将继续讨论一些与分配线程执行资源的相关属性。我们假设属性dev_prop中的变量被cudaGetDeviceProperties函数赋值了，如果读者选择不同的变量名称，下面的内容中这个dev_prop属性名请读者自己替换掉。

正如名字所示，dev_prop.maxThreadsPerBlock这个属性表示这个device中每个区块最多能有多少个线程，一些device会是1024，还有一些device会低一些。未来的一些device可能会允许容纳多于1024个线程。因此，可用的device被找到了，并且去定了每个区块中最多容纳的线程数。

device中SM的数量在dev_prop.multiProcessorCount中给出。正如我们之前说过的一样，一些device只有少量的SM（比如只有2个），有的device有非常多的SM（比如30个）。如果应用需要很多的SM才能达到满意的效果，我们一定要检查这个属性的值。而且还有时钟频率（clock frequency），用dev_prop.clockRate表示，这两个属性相结合表示了这个device的硬件执行能力。

host代码能在dev_prop.maxThreadsDim[0], dev_prop.maxThreads Dim[1], 和dev_prop.maxThreadsDim[2]中得到区块的每个维度最大的线程数，这个信息能够在衡量区块维度设置成多少才能让硬件有充分的使用时，帮助我们调节系统参数。同样我们也可以从dev_prop.maxGridSize[0], dev_prop. maxGridSize[1], 和dev_prop.maxGridSize[2] 中得到网格的每个维度最多能有多少个区块。这个信息在我们判断一个网格是否有充足的线程来处理数据，或者判断是否需要循环的时候使用。

cudaDeviceProp还有许多的属性，我们将会在讲到其中的相关概念的时候在去讲。

3.7线程调度和时间容忍度

线程调度严格来说是一个与具体硬件实现相关的概念（implementation concept），因此在讨论它的时候必须指明具体硬件。迄今为止，在大多数硬件都是把分配给SM的区块进一步分为32个线程组成的最小单元，称为warps（这个东西不知道咋个翻译，就用英文吧）。warp的大小是根据硬件的实现不同而不同。warp并不是CUDA设定的，但是知道warp相关的知识能够帮助我们理解和优化特定CUDA的device上的CUDA应用。warp的大小是CUDA的device的属性，因此可以用dev_prop.warpSize查看。

warp是SM中线程调度的单位。在图3.13中展示了某硬件中区块分成warp的事例，每个warp由32个线程组成，这些线程有着连续的threadIdx值，第一个wrap中线程从0到31，第二个从31到63，以此类推。在这个例子中，Block 1, Block 2, 和Block 3被分配到了一个SM中，每个区块都被进一步分成了warp，为了方便调度。

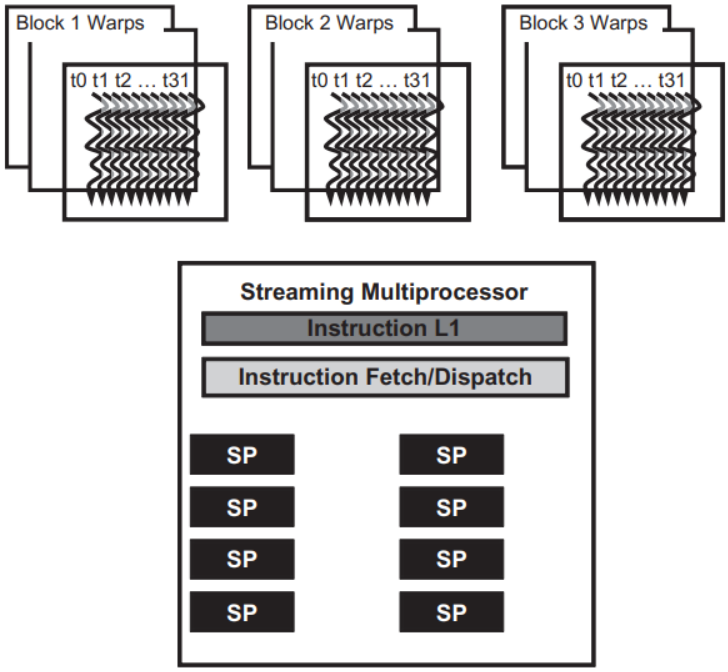

FIGURE 3.13

Blocks are partitioned into warps for thread scheduling.

我们通过区块的大小和SM中区块的个数能够计算一个SM中的warp的个数，图3.13中每个区块由256个线程，我们能够确定每个区块有256/32=8个warp。每个SM中有3个区块，一共有8*3=24个区块。

SM被设计成用统一指令执行所有warp中的线程，也就是SIMD(Single Instruction Multiple Data)，也就是说一个warp中的所有线程在同一时间获取唯一的指令，并做多个线程上对不同的数据执行。这个过程在图3.13中展示了出来，单个指令被获取到（或是说分发到）SM中的共同的执行单位中（SP）。这些线程会应用相同的指令到数据的不同部分。总之一个warp中所有的线程同一时间使用同一个指令。

图3.13还展示了一些硬件实际执行指令的的Streaming Processors（SPs）。总的来说，SM上的SP的个数要比线程数要少，例如同一时间每个SM只有足够的硬件去执行所有分配到SM的线程的一小部分的指令。在早期的GPU设计中，每个同一时间SM能够让一个warp执行1个指令，而在最近的设计中，每个SM能够在任何时间让一小部分warp执行指令。这两个例子中，硬件都能够让SM中一小部分warp执行指令。有个非常没毛病的问题：既然SM只能让其中一小部分warp执行指令，SM中为什么还需要辣么多的warp呢？答案就是，这就是CUDA处理高时间延迟的方法，比如访问全局内存。

当一个指令将要被一个warp执行的时候，需要等待上一个高时间延迟的操作，这是这个warp并没有被选中执行，而是另一个已经不需要等待结果的warp被选中执行指令，如果多个warp准备好执行了，会有一个优先机制来决定哪个warp会被选中执行，这个填充了其他线程延迟等待时间的机制被称为“延迟忍耐”或“延迟隐藏”。

Warp的调度同样是用来处理其他种类的操作延迟的，例如流水线浮点算术和分支指令，给了充足的warp数量后，硬件会很容易在任何时间点找到要执行的warp，因此使得尽管有着高延迟的操作，执行硬件也会被充分地利用。选择准备好的warp来执行会避免引入空闲时间或者浪费执行时间，避免又叫零开销线程调度。通过线程调度，那些长等待时间的warp指令被其他warp的执行隐藏了起来，这种能够处理高延迟的操作也是GPU不像CPU那样需要辣么大的缓存芯片和分支预测机制的主要原因，因此GPU能够将更多的芯片面积用于浮点数的运算。

> **延迟忍耐**
>
> 延迟忍耐在现如今的生活中管饭存在。例如在邮局，每个人都要邮寄包裹，邮寄之前都要填写一个表格上的信息，然后再去服务台。然而有的人会等待服务台排到他之后才被告知要填写表格。
>
> 当服务台前排起了长队之后，前台服务的工作人员需要提高效率，让顾客在服务台填写表单会导致效率低下，正确的做法是工作人员在等待这个人填写表单的这段时间去为其他顾客进行服务，这些其他顾客也应该是准备好了表单的人，而不应该是还没有填写表单的。
>
> 因此一个优秀的工作人员会礼貌的让第一个顾客去在一旁填写表单，然后继续服务其他顾客。大多数例子中，像这样的顾客在填完表单之后会回到服务台，而不是重新排队。
>
> 我们可以把邮局的顾客当做warp把硬件执行单元当做工作人员，需要填表单的顾客就像那些需要高时间延迟等待的warp。

我们现在准备好了一个简单地练习。假设一个CUDA的device允许每个SM最多8个区块，每个SM最多1024个线程，而且每个区块中最多512个线程。这种情况下，我们在图片虚化这个例子中应当使用 8×8, 16×16, 还是32×32 的线程区块？为了回答这个问题，我们应当分析每个选项的优点和缺点，如果我们使用8*8的区块，每个区块只有64个线程，我们需要1024/64 = 12个区块才能将SM中所有的线程占满，但是每个SM最多容纳8个区块，因此我们最终每个SM中只会有$64 \times 8 = 512$个线程，这也就是的每个SM中会有更少的等待调度的warp。

16*16的区块中有256个线程，因此每个SM需要1024/256 = 4个区块，这个数字没有达到最大限制数8，是一个很好的配置，因为这会让每个SM所有的线程工作并有着最大的待调度的warp。32*32的区块中有1024个线程，这超过了每个区块中最多512个线程的限制，因此只有16*16的区块能够让每个SM中分配最多的线程数。

3.8总结

kernel执行配置参数定义了网格的维度和区块的维度，blockIdx和threadIdx形成的唯一坐标使得网格中的线程能够找到对应的要处理的部分数据，编程者应当对每个kernel指明具体要处理哪部分数据，这种编程的模式强迫编程者将线程和对应的数据组织成相应层级和多个维度。

当一个网格加载后，网格中的区块能够被SM随机地执行，而且会体现出来CUDA应用的明显的可扩展性，但是这种扩展性有一个限制：不同区块之间的线程不能够相互同步。如果想在保持这种明显扩展性的前提下，想让不同区块的线程同步的最简单方法是在同步点终止kernel并在这个点之后重启kernel。

分配到SM的线程是以一块一块地执行的，每个CUDA的device都对每个SM中的资源有一些潜在的限制。每个CUDA的device对每个SM中的区块数和线程数设限，并且二者不一定谁先起到限制作用。对于每个kernel，一个或者多个资源限制会最终限制device中同时存在的线程总数的。

当一个区块被分配给一个SM之后，会被进一步分解成warp， warp中所有的线程同一时间执行同一个指令。任何时间SM只执行其中存在的warp中的一小部分的指令，这个情况使得其他warp等待那些需要高延迟的操作并不会减慢总的执行单元的通量。

3.9习题

各位看原书做吧。