第七章 并行卷积模式——模板计算简介

在接下来的几章中，我们将讨论一系列重要的并行计算模式。这些模式是出现在许多并行应用程序中的各种并行算法的基础。我们将从卷积开始，卷积是一种常用的阵列运算，以各种形式应用于信号处理、数字记录、图像处理、视频处理和计算机视觉。在这些应用领域中，卷积通常作为一种滤波器来执行，它将信号和像素转换为更理想的值。我们的图像模糊kernel也是这样一个过滤器，使信号值平滑，以便人们可以看到整体趋势。另一个例子，高斯滤波器是卷积滤波器，可用于锐化图像中对象的边界和边缘

在高性能计算中，卷积模式通常被称为模板计算，广泛出现在求解微分方程的数值方法中。它还构成了仿真模型中许多求解力的算法的基础。卷积通常涉及对每个数据元素进行大量的算术运算。对于高清晰度图像和视频等大型数据集，计算量可能非常大。每个输出数据元素可以彼此独立地计算，这是并行计算的一个理想特征。另一方面，输出数据元素之间存在大量的输入数据共享，边界条件具有一定的挑战性。这使卷积成为复杂的分块方法和输入数据暂存方法的重要用例。

7.1背景介绍

卷积是一种数组操作，其中每个输出数据元素是相邻所有输入元素的加权和。加权和计算中使用的权重由输入卷积模板数组（input mask array）定义，通常称为卷积核（kernel）。由于CUDA kernel函数和卷积核之间有名称冲突，我们将这些卷积模板数组称为卷积模板，以避免混淆。相同的卷积模板通常用于数组的所有元素。

在音频数字信号处理中，输入数据采用一维形式，将采样信号值表示为时间的函数。图7.1显示了一维数据的卷积示例，其中将5个元素卷积模板数组M应用于7个元素输入数组N。我们将遵循C语言约定，其中N和P元素的索引从0到6，M 元素从0到4进行索引。我们使用5元素卷积模板M的事实意味着，每个P元素都是由对应位置的N元素，左边的两个N元素和右边两个N元素的加权和生成的。

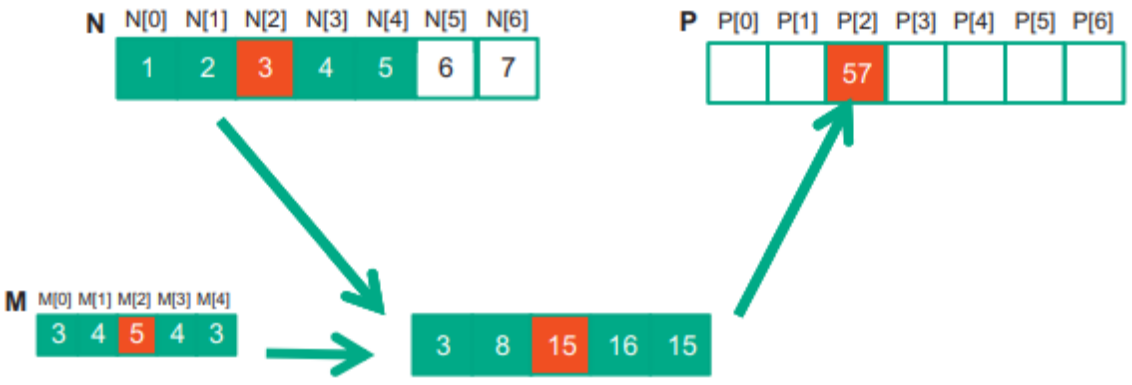

FIGURE 7.1

A 1D convolution example, inside elements.

例如，P[2]的值是N[0](即 N[2-2])到N[4](即N[2+2])的加权和。在这个例子中，我们假设N的元素的值是1、2、3、…、7。M中的元素定义权重，在本例中权重为3、4、5、4、3。将每个权重值与对应的N个元素值相乘，然后将乘积相加。如图7.1所示，P[2]计算如下:

```
P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4]
     = 1*3 + 2*4 + 3*5 + 4*4 + 5*3
     = 57
```

一般来说，卷积模板的大小往往是一个奇数，这使得加权和计算围绕被计算的元素对称。也就是说，奇数个卷积模板元素使得加权和是包含被计算元素的每边相同数量的元素的。在图7.1中，卷积模板大小为5个元素。每个输出元素都是由对应的输入元素左边两个元素加右边两个元素的加权和计算出来的。

在图7.1中，可以将P [i]的计算视为从N [i-2]开始的N子数组与M数组之间的内积。 图7.2显示了P [3]的计算。 与图7.1的计算相比，计算结果偏移了1个元素。 即，P [3]的值是N [1]（即，N [3-2]）到N [5]（即，N [3 + 2]）的加权和。 我们可以想到对P [3]的计算如下：

```
P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4]
     = 2*3 + 3*4 + 4*5 + 5*4 + 6*3
     = 76
```

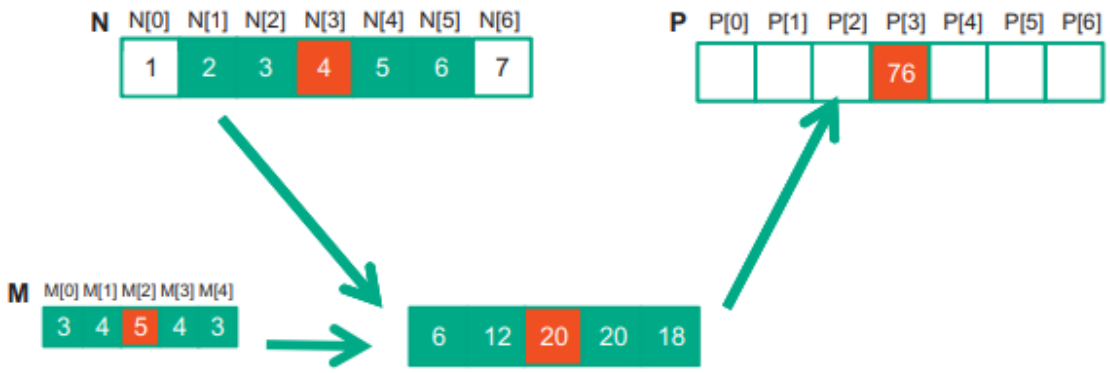

FIGURE 7.2

1D convolution, calculation of P[3].

因为卷积是根据相邻元素定义的，所以边界条件自然会出现在接近数组末端的输出元素上。如图7.3所示，计算P[1]时，在N[1]的左边只有一个N元。也就是说，根据卷积的定义，没有足够的N个元素来计算p[1]。处理这种边界条件的一种典型方法是为这些缺失的N个元素定义一个默认值。对于大多数应用程序，默认值是0，这就是我们在图7.3中使用的值。例如，在音频信号处理中，我们可以假设在录音开始前和结束后，信号的音量为0。此时，P[1]的计算如下:

```
P[1] = 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4]
     = 0 * 3 + 1*4 + 2*5 + 3*4 + 4*3
     = 38
```

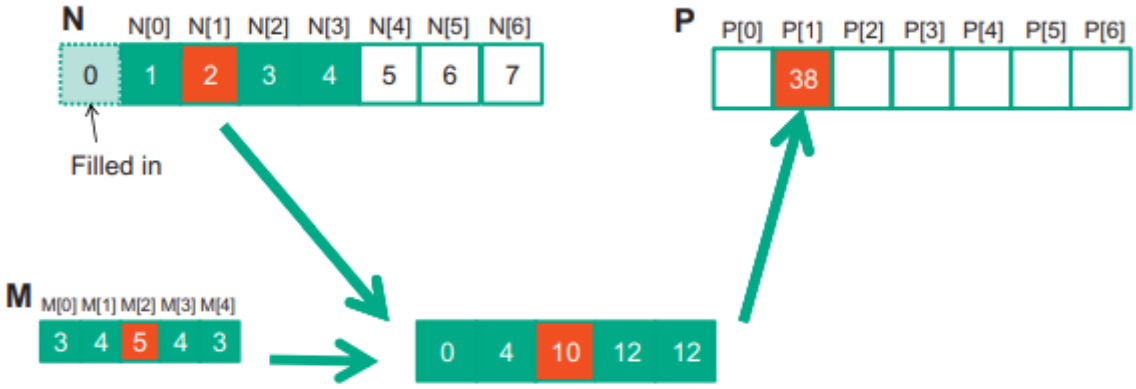

FIGURE 7.3

1D convolution boundary condition.

本计算中不存在的N元素如图7.3中的虚线框所示。应该清楚的是，P[0]的计算将涉及到两个缺失的N元素，对于这个例子，两者都将被假定为0。我们把P[0]的计算留作练习。这些缺失的元素在文献中通常被称为“神魔鬼单元”（ghost cells）或“光环元素”（halo cells）。由于在并行计算中使用了分块算法，还存在其他类型的缺失元素。这些缺失元素可以对分块算法的有效性和/或效率产生重大影响。我们很快会回到这一点上。

而且，并不是所有应用程序都假设缺失元素为0。例如，一些应用程序可能会假设缺失元素的值与最近的有效数据元素相同。

对于图像处理和计算机视觉，输入数据通常是二维数组，像素位于x-y空间。因此，图像卷积是2D卷积，如图7.4所示。在二维卷积中，卷积模板M是一个二维数组。它的x和y维大小决定了加权和计算中包含的邻居的范围。在图7.4中，为了简单起见，我们使用了一个5×5的卷及模板。通常，卷积模板不必是一个正方形数组。为了生成一个输出元素，我们取输入数组n中中心在对应位置的子数组，然后对卷积模板数组元素和图像数组元素进行两两相乘。 对于我们的示例，结果显示为图7.4中N和P下方的5×5乘积数组。 输出元素的值是乘积数组中所有元素的总和。

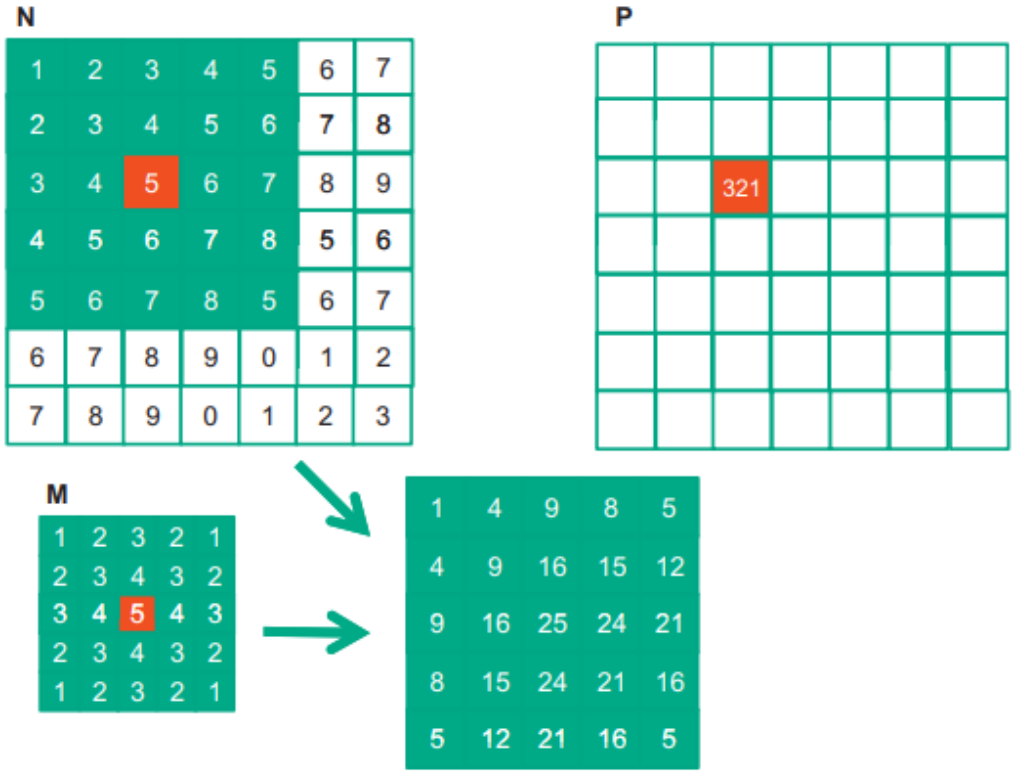

FIGURE 7.4

A 2D convolution example.

图7.4中的例子显示了P2,2的计算。为了简便起见，我们将使用Ny,x来表示在C数组寻址时的N[y][x]。因为N和P很有可能是动态分配的数组，所以我们将在实际的代码示例中使用线性化的索引。用于计算P2,2值的N子数组在x或水平方向上跨度从N0,0到N0,4，在y或垂直方向上跨度从N0,0到N4,0。计算如下：

$$
\begin{aligned}
P_{2,2} &= N_{0,0}*M_{0,0} + N_{0,1}*M_{0,1} + N_{0,2}*M_{0,2} + N_{0,3}*M_{0,3} + N_{0,4}*M_{0,4} \\
&\quad + N_{1,0}*M_{1,0} + N_{1,1}*M_{1,1} + N_{1,2}*M_{1,2} + N_{1,3}*M_{1,3} + N_{1,4}*M_{1,4} \\
&\quad + N_{2,0}*M_{2,0} + N_{2,1}*M_{1,1} + N_{2,2}*M_{2,2} + N_{2,3}*M_{2,3} + N_{2,4}*M_{2,4} \\
&\quad + N_{3,0}*M_{3,0} + N_{3,1}*M_{3,1} + N_{3,2}*M_{3,2} + N_{3,3}*M_{3,3} + N_{3,4}*M_{3,4} \\
&\quad + N_{4,0}*M_{4,0} + N_{4,1}*M_{4,1} + N_{4,2}*M_{4,2} + N_{4,3}*M_{4,3} + N_{4,4}*M_{4,4} \\
&= 1*1 + 2*2 + 3*3 + 4*2 + 5*1 \\
&\quad + 2*2 + 3*3 + 4*4 + 5*3 + 6*2 \\
&\quad + 3*3 + 4*4 + 5*5 + 6*4 + 7*3 \\
&\quad + 4*2 + 5*3 + 6*4 + 7*3 + 8*2 \\
&\quad + 5*1 + 6*2 + 7*3 + 8*2 + 5*1 \\
&= 1 + 4 + 9 + 8 + 5 \\
&\quad + 4 + 9 + 16 + 15 + 12 \\
&\quad + 9 + 16 + 25 + 24 + 21 \\
&\quad + 8 + 15 + 24 + 21 + 16 \\
&\quad + 5 + 12 + 21 + 16 + 5 \\
&= 321
\end{aligned}
$$

与一维卷积一样，二维卷积也必须处理边界条件。在x和y维度上都有边界的情况下，存在更复杂的边界条件:输出元素的计算可能涉及水平边界、垂直边界或两者都有的边界条件。图7.5说明了涉及两个边界的P元的计算。由图7.5可知，P1,0的计算涉及到N子数组中缺失的两列和一横行，就像在1D卷积中一样，不同的应用对缺失的N个元素的默认值是不同的。在我们的示例中，我们假设默认值为0。这些边界条件也会影响分块算法的效率。我们很快会回到这一点上。

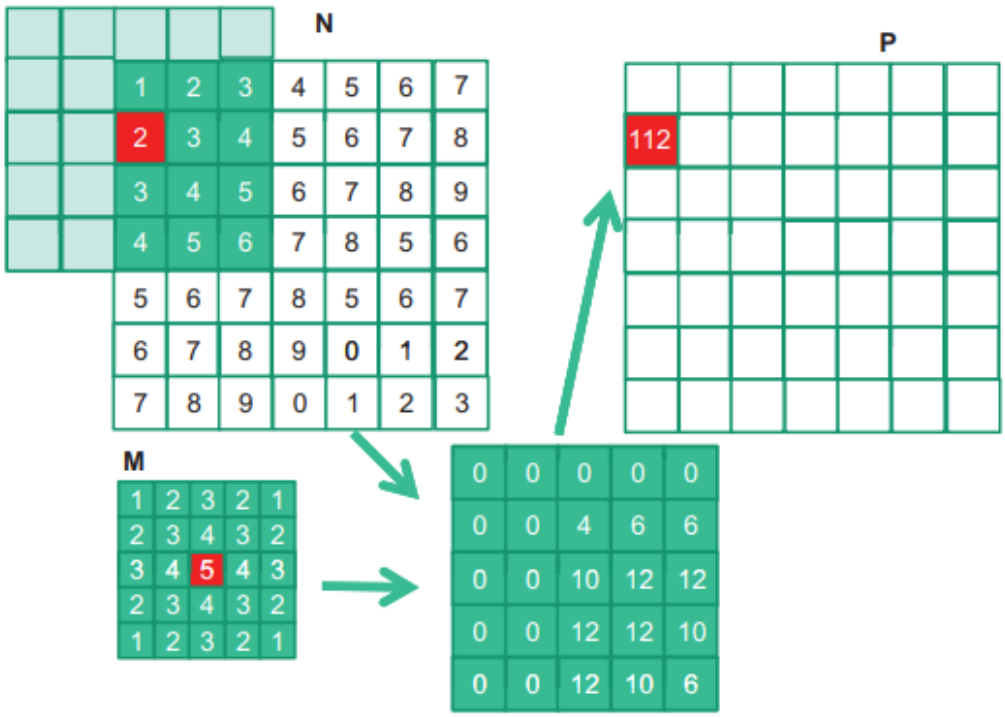

FIGURE 7.5

A 2D convolution boundary condition.

7.2一维并行卷积基础算法

正如我们在7.1节中提到的那样，所有输出（P）元素的计算都可以并行进行。 这使卷积成为并行计算的理想问题。 根据我们在矩阵-矩阵乘法中的经验，我们可以快速编写一个简单的并行卷积kernel。为简单起见，我们将从一维卷积开始。

第一步是定义kernel的主要输入参数。 我们假设一维卷积内核接收五个参数：指向输入数组N的指针，

指向输入卷积模板M的指针，指向输出数组P的指针，卷积模板Mask_Width的大小以及输入和输出数组Width的大小。 因此，我们有以下设置：

```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
float *P,
    int Mask_Width, int Width) {
    // kernel body
}
```

第二步是确定并实现线程到输出元素的映射。 由于输出数组是一维的，因此一种简单的好方法是将线程网格设计成一维的，并让网格中的每个线程计算一个输出元素。 读者应该认识到，就输出元素而言，这与矢量加法示例相同。 因此，我们可以使用以下语句根据每个线程区块索引，区块大小和线程索引来计算输出元素索引：

```
int i = blockIdx.x*blockDim.x + threadIdx.x;
```

一旦确定了输出元素索引，就可以使用对输出元素索引的偏移量来访问输入N的元素和卷积模板M的元素。 为简单起见，我们假设Mask_Width为奇数且卷积是对称的，即Mask_Width为2 * n + 1（其中n为整数）。 P [i]的计算将使用N [in]， N [i-n + 1]， …， N [i-1]， N [i]， N [i + 1]， N [i + n- 1]， N [i + n]。我们可以使用一个简单的循环在kernel中进行此计算：

```
float Pvalue = 0;
int N_start_point = i - (Mask_Width/2);
for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
}
P[i] = Pvalue;
```

变量Pvalue将所有中间结果累加到寄存器中，以节省DRAM带宽。 for循环累积了从相邻元素到输出P元素的所有输出。 循环中的if语句测试在N数组的左侧还是右侧的元素是否为范围内单元格。 由于我们假定将0值用于范围外元素值，因此我们可以简单地跳过范围外元素及其对应的N元素的乘法和累加。 循环结束后，我们将Pvalue释放到输出P元素中。 现在在图7.6中有一个简单的kernel。

```
__global__ void convolution_1D_basic_kernel(float *N, float *M, float *P,
  int Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

FIGURE 7.6

A 1D convolution kernel with boundary condition handling.

我们可以对图7.6进行观察。首先，这里存在控制流异化。计算P数组左端或右端附近的输出P元素的线程将处理范围外元素。正如我们在7.1节中展示的那样，这些相邻线程中的每一个都会遇到不同数量的范围外元素。因此，它们在if语句中的决策都会有所不同。计算P [0]的线程大约一半的几率将跳过乘法累加语句，而计算P [1]的线程将跳过较少的次数，依此类推。控制异化的成本将取决于Width（输入数组的大小）和Mask_Width（卷积模板的大小）。对于较大的输入矩阵和较小的卷积模板，控制偏差仅在输

出元素的一小部分中发生，这将使控制异化的影响较小。由于卷积通常应用于大型图像和空间数据，因此我们通常预估异化的影响适度或微不足道（Since convolution is often applied to large images and spatial data, we typically expect that the effect of convergence to be modest or insignificant.）。

一个更严重的问题是内存带宽。在kernel中，浮点算术计算与全局内存访问的比率仅约为1.0。正如我们在矩阵-矩阵乘法示例中所看到的，只能期望这个简单的内核以峰值性能的一小部分运行。在接下来的两节中，我们将讨论两种减少全局内存访问次数的关键技术。

7.3常量内存和缓存

卷积模板数组M在卷积中的使用有三个有趣的特性。首先，M数组的大小通常较小。大多数卷积模板在每个维数中小于10个元素。即使在3D卷积的情况下，模板通常只包含少于1000个元素。其次，M的内容在整个内核执行过程中都没有改变。第三，所有线程都需要访问模板元素。还有就是，所有线程都以相同的顺序访问M的元素，从M[0]开始，在图7.6中for循环的迭代中，每次移动一个元素。这两个属性使得模板数组非常适合用于常量内存和缓存(图7.7)。

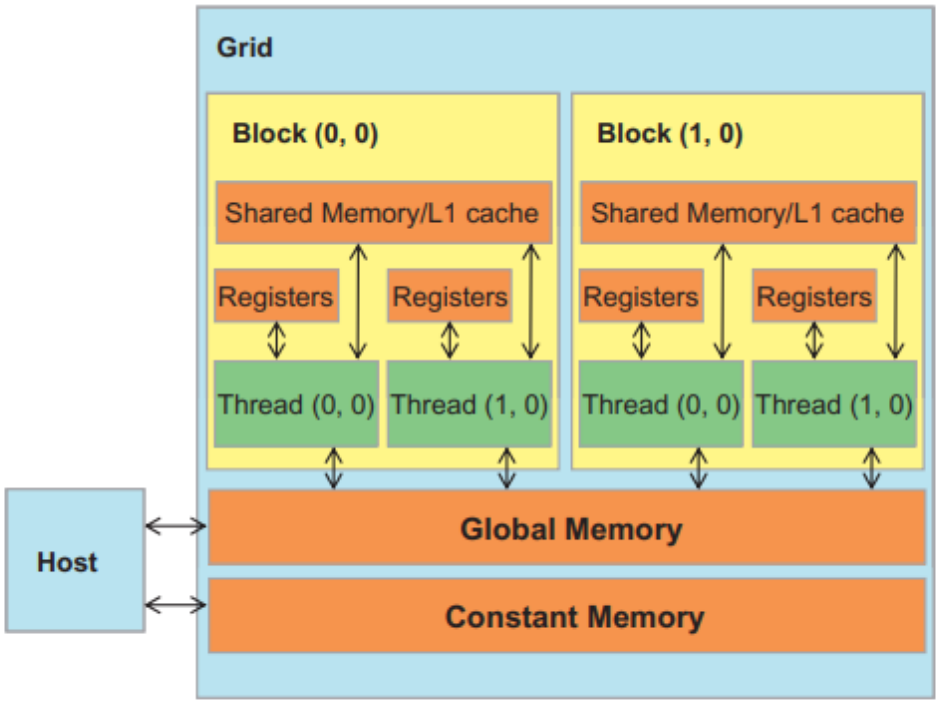

FIGURE 7.7

A review of the CUDA Memory Model.

正如我们在第5章中讨论的，CUDA编程模型允许程序员在常量内存中声明一个变量。与全局内存变量一样，常量内存变量对所有线程区块也是可见的。主要的区别是，在kernel执行期间，线程不能更改常量内存变量。此外，常量内存的大小非常小，目前为64KB。

为了使用常量内存，主机代码需要以不同于全局内存变量的方式分配和复制常量内存变量。为了在常量内存中声明M数组，主机代码将其声明为全局变量，如下所示:

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

这是一个全局变量声明，应该在源文件中任何函数的外部。关键字剩余的__constant__(两边各有两个下划线)告诉编译器数组M应该被放置到设备常量内存中。

假设主机代码已经用Mask_Width个元素在主机内存中分配并初始化了M_h数组作为模板。M_h的内容可以按照如下方式转移到device常量内存中的M中:

```
cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

请注意，这是一种特殊的内存复制功能，该功能通知CUDA运行时系统，在kernel执行期间不会更改要复制到常量内存中的数据。 通常，cudaMemcpyToSymble（ ）函数的用法如下：

```
cudaMemcpyToSymbol(dest, src, size)
```

其中dest是指向常量内存中目标位置的指针，src是指向host内存中源数据的指针，而size是要复制的字节数。

kernel函数以全局变量的形式访问常量内存变量。因此，它们的指针不需要作为参数传递给kernel。我们可以修改我们的kernel去使用常量内存，如图7.8所示。请注意，kernel看起来几乎与图7.6中的kernel相同。唯一的区别是M不再通过作为参数传入的指针来访问。它现在作为一个由host代码声明的全局变量访问。请记住，所有C语言全局变量的作用域规则都适用于这里。如果host代码和kernel代码在不同的文件中，内核代码文件必须包含相关的外部声明信息，以确保M的声明对kernel是可见的。

```
__global__ void convolution_1D_ba sic_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
    if (N start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
    }
  }
  P[i] = Pvalue;

}
```

FIGURE 7.8

A 1D convolution kernel using constant memory for M.

与全局内存变量一样，常量内存变量也位于DRAM中。但是，因为CUDA运行时知道在kernel执行过程中不会修改，所以它会指示硬件在kernel执行过程中主动缓存常量内存变量。为了理解经常常量内存的好处，我们首先需要更多地了解现代处理器内存和缓存层次结构。

正如我们在第5章中讨论的，DRAM的长延迟和有限的带宽已经成为几乎所有现代处理器的主要瓶颈。为了缓解内存瓶颈的影响，现代处理器通常采用芯片上的缓存存储器，或缓存，以减少需要从主存储器(DRAM)访问的变量的次数，如图7.9所示。

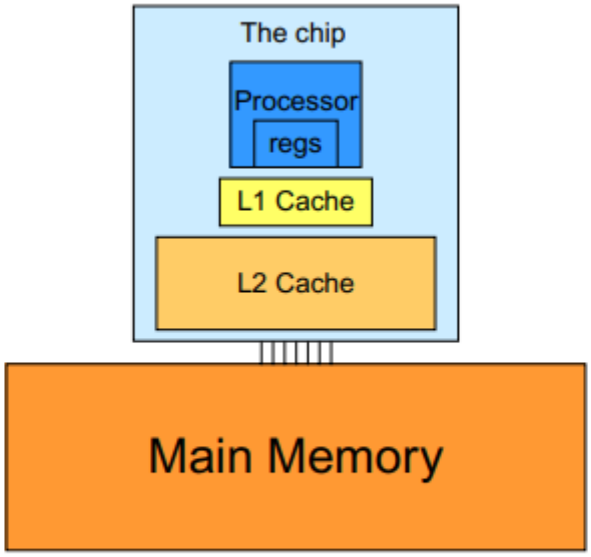

FIGURE 7.9

A simplified view of the cache hierarchy of modern processors.

不像CUDA共享内存，或者普通内存，缓存对程序是“透明的”。也就是说，为了使用CUDA共享内存，程序需要将变量声明为__shared__，并显式地将全局内存变量移动到共享内存变量中。另一方面，当使用缓存时，程序只是访问原始变量。处理器硬件将自动保留一些最近或经常使用的变量在缓存中，并记住它们的原始DRAM地址。当稍后使用保留的变量之一时，硬件将从它们的地址检测到缓存中有该变量的副本可用。然后，将从缓存中提供变量的值，从而消除了访问DRAM的需要。

在内存大小和内存速度之间需要权衡。结果，现代处理器通常采用多层缓存。这些缓存级别的编号约定反映了到处理器的距离。最低的一级，L1或一级，是直接连接到处理器核心的缓存。它的运行速度在延迟和带宽方面都非常接近处理器。但是，L1缓存的大小比较小，通常在16KB到64KB之间。L2缓存更大，从128KB到1MB，但是访问它需要几十个周期。它们通常在CUDA设备中的多个处理器内核或SM之间共享。在今天的一些高端处理器中，甚至有L3缓存，其大小可以达到几个MB。

在大规模并行处理器中使用缓存的一个主要设计问题是缓存一致性，当一个或多个处理器内核修改缓存数据时就会出现这种情况。因为L1缓存通常只直接连接到一个处理器kernel上，所以其他处理器kernel不容易观察到它内容的变化。如果修改后的变量在运行在不同处理器上的线程之间共享，就会产生问题。需要一种缓存一致性机制来确保其他处理器核心缓存的内容得到更新。在大规模并行处理器中提供缓存一致性是困难和昂贵的。然而，它们的存在通常简化了并行软件开发。因此，现代CPU通常支持处理器kernel之间的高速缓存一致性。尽管现代GPU提供了两个级别的缓存，但它们通常没有缓存一致性，以最大限度地利用可用硬件资源来增加处理器的算术吞吐量。

在大规模并行处理器中使用缓存时，常量内存变量扮演了一个有趣的角色。由于它们在kernel执行期间不会改变，所以在kernel执行期间不存在缓存一致性问题。因此，硬件可以主动地将常量变量值缓存到L1缓存中。此外，这些处理器中缓存的设计通常经过优化，以便将一个值发送给大量线程。因此，当所有的线程访问相同的常量内存变量时(如M的情况)，缓存可以提供巨大的带宽来满足线程的数据需求。此外，由于M的大小通常较小，我们可以假设所有M元素都可以有效地从缓存中访问。因此，我们可以简单地假设没有DRAM带宽花费在M的访问上。通过使用常量内存和缓存，我们有效地将浮点运算与内存访问的比率提高了一倍，达到2。

事实证明，对N输入数组元素的访问也可以从较新的GPU的缓存中受益。我们将在第7.5节中回到这一点。

7.4考虑超边界元素的一维卷积分块算法

现在我们将用分块卷积算法来解决访问N的数组元素时的内存带宽问题。回想一下，在分块算法中，线程协作将输入元素加载到芯片上存储器中，然后访问芯片上存储器，以便后续更快地使用这些元素。为了简单起见，我们将继续假设每个线程计算一个输出P元素。一个区块中最多有1024个线程，可以处理最多1024个数据元素。我们将每个区块处理的输出元素集合称为输出块。图7.10显示了一个16个元素的一维卷积的示例，使用四个线程区块，每个区块有四个线程。在本例中，有四个输出块。第一个输出块覆盖N[0]到N[3]，第二个输出块覆盖N[4]到N[7]，第三个输出快覆盖N[8]到N[11]，第四个输出块覆盖N[12]到N[15]。注意，保持示例较小，我们每个块只使用4个线程。实际上，对于当前生成的硬件，每个块至少应该有32个线程。从这一点开始，我们将假设M元素在常量内存中。

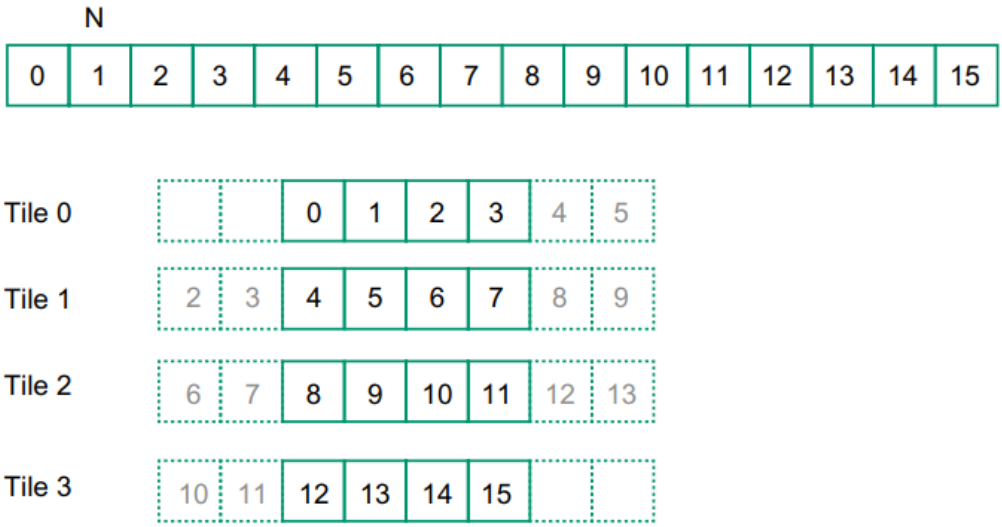

FIGURE 7.10

A 1D tiled convolution example.

我们将讨论减少全局内存访问总数的两种输入数据分块策略。第一个是最直观的，涉及到将计算线程区块的所有输出元素所需的所有输入数据元素加载到共享内存中。要加载的输入元素的数量取决于卷积模板的大小。为了简单起见，我们继续假设卷积模板大小是一个奇数，等于2*n+1。也就是说，每个输出元素P[i]是对应输入元素N[i]处的输入元素、左边的N个输入元素(N[i−N], …N[i−1])和右边的N个输入元素(N[i+1], …N[i+ N])的加权和。图7.10给出了一个Mask_Width=5和n=2的例子。

区块0中的线程计算从P[0]到P[3]的输出元素。它们都需要输入元素n[0]到n[5]。注意，计算还需要N[0]左边的两个边界外元素。在图7.6的块0的左端显示了两个虚线的空元素。假定这些虚影元素的默认值为0。Tile 3在输入数组N的右端也有类似的情况。在我们的讨论中，我们将像Tile 0和Tile 3这样的块称为边界块，因为它们包含了输入数组N边界上或边界外的元素。

第1块中的线程计算从P[4]到P[7]的输出元素。它们共同需要输入元素N[2]到N[9]，如图7.7所示。注意，元素N[2]和N[3]被两个块所计算，因此两次被加载到共享内存中，一次加载到块0的共享内存中，一次加载到块1的共享内存中。由于区块共享内存的内容仅对区块内的线程可见，因此需要将这些元素加载到各自的共享内存中，以便所有相关的线程访问它们。包含在多个块中并由多个区块加载的元素通常被称为光晕单元（Halo Cells）或裙单元（Skirt Cells），因为它们“挂”在由单个块单独使用的部分的一侧（since they “hang” from the side of the part that is used solely by a single block.）。我们将称输入块中只被自己块所单独使用的元素为内部元素。Tile 1和Tile 2通常被称为内部块，因为它们在输入数组N的边界之外没有包含任何边界外元素。

现在我们将展示将输入块加载到共享内存中的kernel代码。我们首先声明一个共享内存数组N_ds来保存每个块需要的的N元素。共享内存数组的大小必须足够大，以容纳输入块的左裙单元格、中间单元格和右裙单元格。我们假设Mask_Width是一个奇数。假设常量MAX_MASK_WIDTH指定Mask_Width的最大可能值。共享内存数组的最大可能大小是TILE_SIZE + MAX_MASK_WIDTH - 1，它在kernel中声明如下:

```
__shared__ float N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
```

然后，我们加载左裙单元，其中包括前一个块的最后一个n = Mask_Width/2中心元素。例如，在图7.6中（7.10），第1块的左裙单元格由第0块的最后2个中心元素组成。在C中，假设Mask_Width是一个奇数，表达式Mask_Width/2将得到一个与(mask_wth -1)/2相同的整数值。我们将使用块的最后(Mask_Width/2)个线程来加载左裙元素。这可以通过以下两个语句来实现:

```
int halo_index_left = (blockIdx.x - 1)*blockDim.x +
threadIdx.x;
if (threadIdx.x >= blockDim.x - n) {
  N_ds[threadIdx.x - (blockDim.x - n)] =
    (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

在第一个语句中，我们使用表达式(blockIdx.x-1)*blockDim.x+threadIdx.x将线程索引映射到前一个块的元素索引。然后，我们使用if语句中的条件挑选最后n个线程来加载所需的左裙元素。例如，在图7.6中（7.10），blockDim.x = 4, n = 2；只有线程2和线程3将会通过if半段。线程0和线程1由于条件判断没有通过，将不会加载任何内容。

对于所有的线程，我们还需要检查它们的裙单元格是否实际上是边界外单元格。这可以通过测试计算的halo_index_left值是否为负数来检查。如果是负数，裙单元格实际上是边界外单元格，因为它们的N索引是负的超出了N索引的有效范围。在这种情况下，条件判断将给这个元素值赋为0。否则，条件语句将使用halo_ index_left将适当的N元素加载到共享内存中。共享内存索引计算是以左裙单元格为0索引开始的数组。例如，在图7.6中，blockDim.x - n = 2。因此，对于块1中的线程2将把最左裙元素加载到N_ds[0]中，而线程3将把下一个左裙元素加载到N_ds[1]中。但是，对于块0的线程2和线程3都将值0加载到N_ds[0]和N_ds[1]中。

下一步是加载输入块的中心单元格。这是通过threadIdx.x和blockIdx.x计算出适当的N索引，如下面的语句所示。读者应该熟悉使用的N索引表达式:

```
N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x +
threadIdx.x];
```

因为N_ds数组的前n个元素已经包含了左裙单元格，所以需要将中心元素加载到N_ds的下一部分中。把n加threadIdx.x来作为每个线程的索引，以便将其加载的中心元素写入N_ds。

我们现在加载右侧裙元素，这与加载左侧裙元素非常相似。我们首先通过blockIdx.x和threadIdx.x计算下一个输出块的对应位置元素索引。通过(blockIdx.x+1)*blockDim.x算出正确右裙元素的N索引。在本例中，我们加载下一个块的开始的n个元素。

```
int halo_index_right = (blockIdx.x + 1)*blockDim.x +
threadIdx.x;
if (threadIdx.x < n) {
   N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
}
```

现在所有输入图块元素都在N_ds中，每个线程可以使用N_ds元素计算其输出P元素值。 每个线程将使用N_ds的不同部分。 线程0将使用N_ds [0]至N_ds [Mask_Width-1]； 线程1将使用N_ds [1]到N [Mask_Width]。通常，每个线程将使用N_ds [threadIdx.x]到N [threadIdx.x + Mask_Width-1]。这在以下for循环中实现，以计算分配给线程的P元素：

```
float Pvalue = 0;
for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
}
P[i] = Pvalue;
```

但是，一定不要忘记使用__syncthreads ( ) 进行线程同步，以确保在任何人开始从共享内存中使用它们之前，同一块中的所有线程都已经加载完它们分配的N元素。

注意，用于加载左裙和右裙单元格的条件语句将0值放置到第一个和最后一个线程区块的适当N_ds元素中，因此乘法和累加的代码比基本算法简单。

分块算法的1D卷积核比基本核长得多，也复杂得多。为了减少N元素的DRAM访问次数，我们引入了额外的代码复杂性。目标是提高计算与内存访问比率，使实现的性能不受或更少受DRAM带宽的限制。我们将通过比较图7.8和7.11中每个线程区块对kernel执行的DRAM访问次数来评估。

```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width,
  int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];

  int n = Mask_Width/2;

  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }

  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }

  __syncthreads();

  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;

}
```

FIGURE 7.11

A tiled 1D convolution kernel using constant memory for M.

图7.8中有两种情况。对于不处理边界外单元格的线程区块，每个线程访问的N元素的数量为Mask_Width。因此，每个线程区块访问的N元素的总数为blockDim.x * Mask_Width或blockDim.x * (2 n + 1)。例如，如果Mask_Width等于5，并且每个块包含1024个线程，那么每个块总共访问5120个N元素。

对于第一个和最后一个区块，处理边界外单元格的线程不会对边界外单元格执行内存访问。这减少了内存访问的次数。我们可以通过枚举每个使用边界外单元的线程，来计算减少的内存访问数。图7.12中的一个小例子说明了这一点。最左边的边界外单元格被一个线程所使用。第二个左边界外单元被两个线程所使用。总的来说,边界外单元的数量是n，使用这些边界外元素的线程的数量从左到右是1,2,…n。这是一个简单的求和，边界外单元总共减少的访问总数为n (n + 1) / 2。对于我们的简单示例，Mask_Width等

于5,n等于2，由于边界外单元格避免的访问次数是2*3/2 = 3。对右边界外单元的类似分析也得出了同样的结果。很明显，边界外单元格对于卷积模型大小较小的大的线程区块的影响是不显著的。

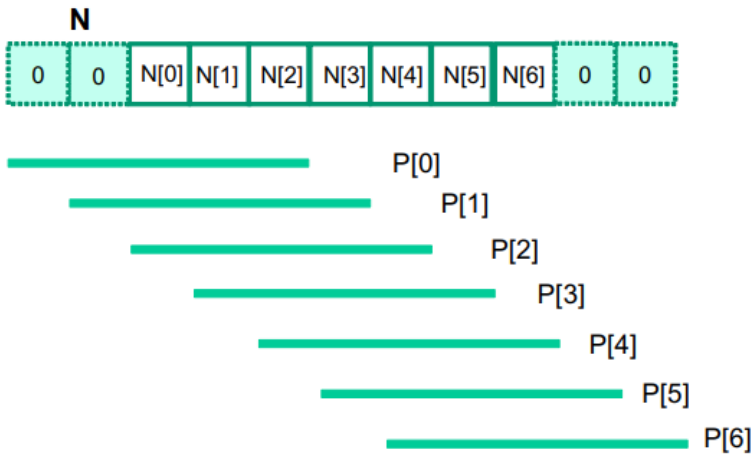

FIGURE 7.12

A small example of accessing N elements and ghost cells.

现在，我们通过图7.11中的分块算法kernel来计算N元素的内存访问总数。 所有的内存访问的代码都已变成了将N元素加载到共享内存中的代码。 在分块算法kernel中，每个N元素仅由一个线程加载。 但是，对于不处理边界外单元的块，还将加载2n个裙单元，从左侧开始n个，从右侧开始n个。 因此，得到内部线程区块加载blockDim.x + 2n个元素，边界线程区块加载blockDim + n个元素。

对于内部线程区块，基本算法和分块算法在一维卷积kernel的内存访问比率为：

```
(blockDim.x*(2n+1)) / (blockDim.x+2n)
```

边界区块的比例是：

```
(blockDim.x*(2n+1) - n(n+1)/2) / (blockDim.x+n)
```

在大多数情况下，blockDim.x远大于n。通过消除较小项n（n + 1）/ 2和n可以近似得出这两个比率：

```
(blockDim.x*(2n+1)/ blockDim.x = 2n+1 = Mask_Width
```

这应该是一个非常直观的结果。 在原始算法中，每个N元素由大约Mask_Width个线程冗余加载。 例如，在图7.12中，N [2]由5个相乘加和得到，需要分别加载P [2]，P [3]，P [4]，P [5]和P [6]。即，存储器访问减少量与卷积模型尺寸大致成比例。

然而，在实践中，较小的项的影响可能是重大的，不能忽视。例如，如果blockDim.x为128，n为5，内部块的比率为:

```
(128*11 - 10) / (128 + 10) = 1398 / 138 = 10.13
```

相反，近似比率为11。应该清楚，随着blockDim.x变小，比率也变小。 例如，如果blockDim为32且n为5，则内部块的比率变为：

```
(32*11 - 10) / (32+10) = 8.14
```

使用较小的块时，读者应始终小心。 它们可能导致内存访问的减少明显少于预期。 实际上，由于芯片上存储器数量不足，通常使用较小的块，特别是对于2D和3D卷积，其中所需的芯片上存储器的数量随块的尺寸增长而快速增长。

7.5更简单的一维分块卷积——总缓存

在图7.11中，代码的大部分复杂性都与加载左右裙单元格以及内部元素到共享内存有关。更近期的gpu,如费米架构，提供一般的L1和L2缓存，其中L1是每个SM私有的，而L2是在所有SM之间共享的。这使得区块有机会在L2缓存中访问它们的裙单元格。

回忆一下，一个区块的裙单元格也是邻近区块的内部细胞。例如，在图7.10中，块1的裙单元格N[2]和N[3]也是块0的内部元素。在块1需要使用这些裙单元格的时候，由于块0的访问，它们已经在L2缓存中，这是很有可能的。因此，对这些裙单元的内存访问可以自然地从L2缓存访问，而不会导致额外的DRAM流量。也就是说，我们可以将访问留给原始N元素中的这些裙单元格，而不是将它们加载到N_ds中。现在我们提出一种更简单的分块1D卷积算法，它只将每个块的内部元素加载到共享内存中。

在更简单的分块kernel中，共享内存N_ds数组只需要保存输出块的内部元素。因此，它是用TILE_SIZE声明的，而不是TILE_SIZE+Mask_Width-1。

```
__shared__ float N_ds[TILE_SIZE];
```

仅需一行代码即可完成块的加载，非常简单：

```
N_ds[threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];
```

在使用N_ds中的元素之前，我们仍然需要同步。然而，计算P元素的循环变得更加复杂。它需要添加条件来检查裙单元和边界外单元的使用情况。边界外单元格的处理与图7.6中的条件语句相同。乘法累加语句变得更加复杂，如图7.13所示。

```
__syncthreads();

int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);
float Pvalue = 0;
for (int j = 0; j < Mask_Width; j++) {
  int N_index = N_start_point + j;
  if (N_index >= 0  && N_index < Width) {
    if ((N_index >= This_tile_start_point)
      && (N_index < Next_tile_start_point)) {
      Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
    } else {
      Pvalue += N[N_index] * M[j];
    }
  }
}
P[i] = Pvalue;
```

FIGURE 7.13

Using general caching for halo cells.

变量This_tile_start_point和Next_tile_start_point保存当前块处理的内部N元素的起始位置索引，以及下一个块处理的内部N元素的起始位置索引。例如，在图7.10中，Block 1的This_ tile_start_point的值是4,Next_tile_start_point的值是8。

新的if语句通过This_tile_start_point和Next_tile_start_point来判断当前对N元素的访问是否属于块内部。如果元素在块中，也就是说，它是当前块的内部元素，那么可以从共享内存中的N_ds数组访问它。否则，它将从N数组中访问，N数组很可能在L2缓存中。使用通用缓存的完整分块算法的kernel如图7.14所示。

```
__global__ void convolution_1D_tiled_caching_kernel(float *N, float *P, int
  Mask_Width,int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE];

  N_ds[threadIdx.x] = N[i];

  __syncthreads();

  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j++) {
     int N_index = N_start_point + j;
     if (N_index >= 0  && N_index < Width) {
       if ((N_index >= This_tile_start_point)
         && (N_index < Next_tile_start_point)) {
         Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
       } else {
         Pvalue += N[N_index] * M[j];
       }
     }
  }
  P[i] = Pvalue;

}
```

FIGURE 7.14

A simpler tiled 1D convolution kernel using constant memory and general caching.

7.6带有边界外单元的分块2D卷积

既然我们已经学会了如何计算一维分块并行卷积，我们可以很容易地将我们的知识扩展到二维。为了更有趣一点，我们将使用一个基于图像库和应用程序中经常遇到的2D图像格式类型的示例。

正如我们在第3章中看到的，现实世界中的图像是用2D矩阵表示的，大小和形状各异。当将这些图像从文件读入内存时，图像处理库通常以行为主的布局存储这些图像。如果图像宽度（字节数）不是DRAM加速单元大小的倍数，那么第1行及以上的起始点可能与DRAM急速单元边界不一致。正如我们在第5章看到的，当我们试图访问其中一行中的数据时，这种不对齐会导致DRAM带宽的低利用率。因此，当将图像从文件读入内存时，图像库通常也会将图像转换为填充格式，如图7.15所示。

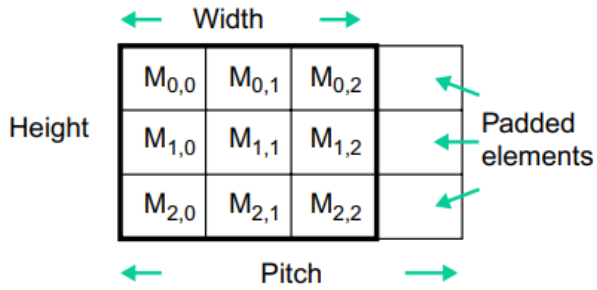

FIGURE 7.15

A padded image format and the concept of pitch.

在图7.15中，我们假设原始图像为3x3。我们进一步假设每个DRAM加速单元包含4个像素。没有填充，第1行的M1,0将是在一个DRAM计算单元中的，而M1,1和M1,2将在下一个DRAM加速单元。访问第1行将需要两次DRAM加速，浪费一半的内存带宽。为了解决这个问题，库在每行的末尾填充一个元素。通过填充元素，每一行占据整个DRAM加速大小。当我们访问第1行或第2行时，现在可以在一个DRAM加

速中访问整个行。一般来说，图像要大得多;每一行可以包含多个DRAM加速。填充元素被添加之后，每一行结束在DRAM计算单元的边界。

通过填充，图像矩阵被填充元素放大。然而，在计算过程中，例如图像模糊，不应该处理填充元素。因此，库数据结构将指明图像的原始宽度和高度，如图7.15所示。但是，库还必须向用户提供有关填充元素的信息，以便用户代码能够正确地找到所有行的实际起始位置。该信息记做填充矩阵的间距。

图7.16显示了如何在以行为主的布局的填充图像矩阵中访问图像像素元素。下面的布局是线性化的顺序。注意，填充的元素位于每行的末尾。顶部布局显示了填充矩阵中像素元素的线性化1D指标。如前所示，第0行初始的三个元素M0、1、M0、2、M0、3变成了线性化一维数组中的M0、M1、M2。注意，M3, M7，和M11填充元素是“假的”线性化元素。第1行原始元素M1、1、M1、2、M1、3的线性化1D指数为M4、M5、M6。也就是说，如图7.16顶部所示，为了计算像素元素的线性化1D指数，我们将在表达式中使用pitch而不是width:

```
Linearized 1D index = row * pitch + column
```

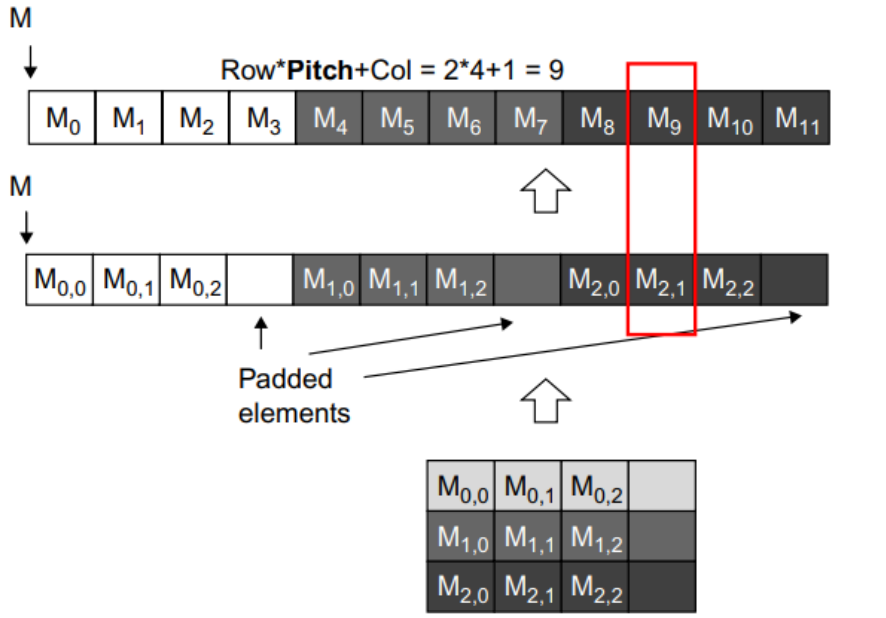

FIGURE 7.16

Row-major layout of a 2D image matrix with padded elements.

但是，当我们遍历一行时，我们将使用width作为循环边界，以确保在计算中只使用原始元素。

图7.17显示了我们将在kernel代码示例中使用的图像类型。注意: channel表示像素中通道的数量:RGB彩色图像有3个通道，灰度图像有1个通道，正如我们在第2章看到的，我们假设这些属性的值将在我们调用2D卷积kernel时被用作参数。

```
// Image Matrix Structure declaration
//
typedef struct {
   int width;
   int height;
   int pitch;
   int channels;
   float* data;
} * wbImage_t;
```

FIGURE 7.17

The C type structure definition of the image pixel element.

我们现在准备设计一个分块的2D卷积kernel。一般来说，我们会发现2D卷积kernel的设计是7.5节中给出的1D卷积kernel的简单扩展。我们首先需要设计每个线程区块要处理的输入和输出块，如图所示。7.18。请注意，输入块必须包括边界外单元格，并且在每个方向上超出其相应的输出块一定数量的边界外单元。图7.19显示了kernel的第一部分:

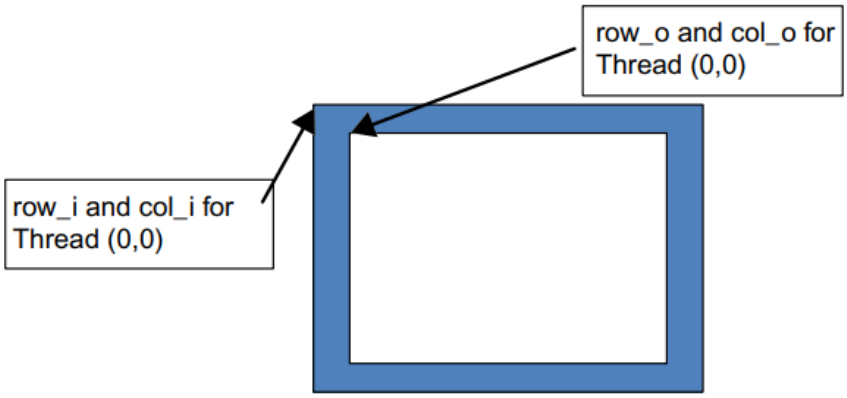

FIGURE 7.18

Starting element indices of the input tile versus output tile.

```
__global__ void convolution_2D_tiled_kernel(float *P, float *N, int height, int width,
                           int pitch, int channels, int Mask_Width,
                           const float __restrict__ *M)
{

  int tx = threadIdx.x;
  int ty = threadIdx.y;
  int row_o = blockIdx.y*O_TILE_WIDTH + ty;
  int col_o = blockIdx.x*O_TILE_WIDTH + tx;

  int row_i = row_o - Mask_Width/2;
  int col_i = col_o - Mask_Width/2;
```

FIGURE 7.19

Part 1 of a 2D convolution kernel.

kernel的每个线程首先计算其输出元素的y和x索引，用col_o和row_o变量表示。thread0、0(负责左上角的输出元素)的索引值如图7.18所示。然后，每个线程通过从row_o和col_o中减去(Mask_Width/2)并将结果分配给row_i和col_i来计算要加载到共享内存中的输入元素的y和x索引，如图7.18所示。请注意，由thread0,0加载的输入块元素也显示在图7.18中。为了简化图7.14中kernel上的分块算法代码，我们将把每个线程区块配置为与输入块相同的大小。在这个设计中，我们可以简单地让每个线程加载一个输入N元素。我们将在计算输出时关闭一些线程，因为每个块中的线程数量比数据元素数量要多。

现在我们已经准备好将输入块加载到共享内存中(图7.20)。所有线程都参与这个过程，但是每个线程都需要检查其输入块元素的y和x索引是否在输入的有效范围内。如果不是，它试图加载的输入元素实际上是一个边界外元素，应该在共享内存中加载0.0值。这些线程属于计算靠近图像边缘的线程区块。注意，当我们从像素的y和x索引计算线性化的1D索引时，我们使用了间距值（pitch）。还请注意，此代码仅适用于通道数为1的情况。通常，我们应该使用for循环来根据当前通道的数量加载所有像素通道值。

```
__shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1]
           [TILE_SIZE+MAX_MASK_HEIGHT-1];
if((row_i >= 0) && (row_i < height) &&
   (col_i >= 0) && (col_i < width)) {
  N_ds[ty][tx] = data[row_i * pitch + col_i];
} else{
  N_ds[ty][tx] = 0.0f;
}
```

FIGURE 7.20

Part 2 of a 2D convolution kernel.

kernel的最后一部分，如图7.21所示，使用共享内存中的输入元素计算输出值。请记住，线程区块中的线程数比输出块的像素数还要多。if语句确保只有索引都小于O_TILE_WIDTH的线程才应该参与输出像素的计算。双重嵌套的for循环遍历卷积模板数组，并对卷积模板元素值和输入像素值执行乘法和累加操作。由于共享内存N_ds中的输入块包含所有边界外元素，索引表达式N_ds[i+ty][j+tx]给出了需要与M[i][j]相乘的N_ds元素。读者应该注意到，这是图7.11中对应for循环的索引表达式的简单扩展。最后，输出元素在有效范围内的所有线程将它们的结果值写入各自的输出元素中。

```
float output = 0.0f;
if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH){
   for(i = 0; i < MASK_WIDTH; i++) {
     for(j = 0; j < MASK_WIDTH; j++) {
       output += M[i][j] * N_ds[i+ty][j+tx];
     }
   }

   if(row_o < height && col_o < width){
      data[row_o*width + col_o] = output;
   }
 }
```

FIGURE 7.21

Part 3 of a 2D convolution kernel.

为了评估二维分块kernel相对于普通kernel的好处，我们可以参考一维卷积的分析。在普通kernel中，线程区块中的每个线程都将对图像数组执行(Mask_Width)^2次访问。因此，每个线程区块总共对图像数组执行(Mask_Width)^2*(O_TILE_WIDTH)^2次访问。

在分块算法的kernel中，线程区块中的所有线程共同加载一个输入块。因此，一个线程区块访问图像数组的总次数为(O_TILE_WIDTH+Mask_Width-1)^2。即普通算法的二维卷积kernel与分块算法的二维卷积kernel之间的图像数组访问比率为：

$$(Mask\_Width)^{2*}(O\_TILE\_WIDTH)^2 / (O\_TILE\_WIDTH+Mask\_Width-1)^2$$

比例越大，分块算法与普通算法相比在减少内存访问次数方面越有效。

图7.22显示了当我们改变输出块大小O_TILE_WIDTH时图像阵列访问减少率的趋势。当O_TILE_WIDTH变得非常大时，卷积模板的大小与块大小相比可以忽略不计。因此，加载的每个输入元素将使用大约（Mask_Width^2次。对于Mask_Width值5，当O_TILE_SIZE变得比5大得多时，我们预计该比率将接近25。例如，对于O_TILE_SIZE=64，比率是22.1。这明显高于O_TILE_WIDTH=8的11.1比率。重要的要点是，我们必须有一个足够大的O_TILE_WIDTH，以便分块算法kernel能够发挥其优势。较大的O_TILE_WIDTH的代价是保存输入块所需的共享内存量也要很大。

TiILE_WIDTH	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

FIGURE 7.22

Image array access reduction ratio for different tile sizes.

对于较大的Mask_Width，例如图7.22底部一行中的9，理想的比率应该是9^2 =81。但是，即使使用较大的O_TILE_WIDTH，比如64，比率也只有64。注意，O_TILE_WIDTH=64和Mask_Width=9会使得输入块大小为72^2 =5184个元素或20,736字节单精度浮点数。这比当前一代gpu的每个SM中可用共享内存的数量还要多。由求解差分方程的有限差分方法派生的模具计算通常需要Mask_Width为9或更高才能实现数值稳定性，此类模板计算可受益于下一代GPU中更大量的共享内存。

7.7总结

在这一章中，我们研究了一种重要的并行计算模式，卷积。卷积被广泛应用于计算机视觉和视频处理等领域，但它也代表了一种通用模式，构成了许多并行算法的基础。例如，可以把偏微分方程求解器中的模板算法看作是卷积的一种特殊情况。另一个例子是，也可以将网格点力的计算或电位值的计算看作是卷积的一种特殊情况。

我们提出了一个基本的并行卷积算法，其实现将受到访问输入N和卷积模板M元素的DRAM带宽的限制。然后，我们引入了常量内存，并对kernel和host代码进行了简单修改，以利用常量缓存并消除卷积模板元素的几乎所有DRAM访问。我们进一步介绍了一个分块并行卷积算法，通过引入更多的控制流异化和编程复杂性来减少DRAM带宽消耗。最后，我们提出了一个更简单的分块并行卷积算法，它利用了L2缓存。

虽然我们只展示了一维卷积的kernel示例，但这些技术直接适用于2D和3D卷积。一般来说，N和M数组的索引计算由于维数较高而十分复杂。此外，每个线程将有更多的循环嵌套，因为在加载块和/或计算输出值时需要遍历多个维度。我们鼓励读者作为作业来完成这些高维kernel的代码。