GPU并行设计实验报告

牛远卓

2022/06/23

（1） 代码的功能是什么

在数学上，卷积衡量两个函数之间的重叠量[1]。
它可以被认为是一种混合操作，整合了一个数据集与另一个数据集的点对点的乘法。
一个数据集与另一个数据集的整合。

$$r(i) = (s * k)(i) = \int s(i-n)k(n)dn$$

在离散的条件下，这可以写成：

$$r(i) = (s * k)(i) = \sum s(i-n)k(n)\,.$$

卷积可以通过增加第二维的索引扩展到二维。

$$r(i) = (s * k)(i,j) = \sum_n \sum_m s(i-n, j-m)k(n,m)$$

在图像处理的背景下，卷积过滤器只是过滤器权重与每个输出像素周围的窗口的标量乘积。
在每个输出像素周围的窗口中输入像素的权重的标量乘积。
这种标量乘积是一种并行操作，非常适合在高度并行的硬件（如GPU）上进行计算。
硬件上进行计算，如GPU。以下是图2的演示。

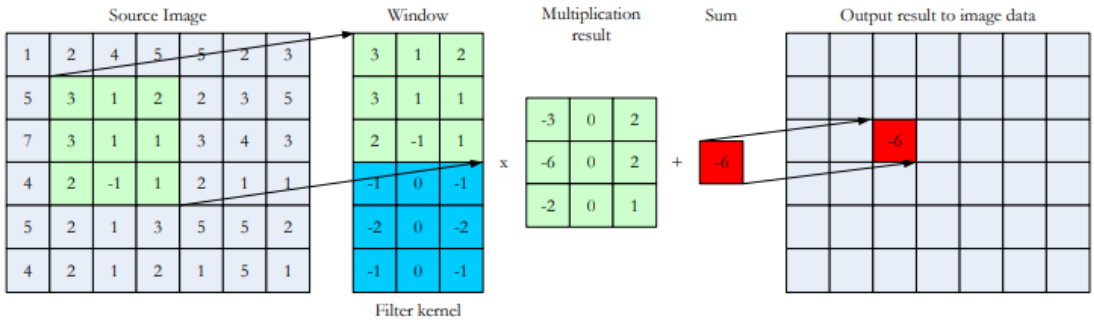


Figure 2: The basic convolution method.

```
  // Compute and store results
  cg::sync(cta);
#pragma unroll

  for (int i = COLUMNS_HALO_STEPS;
       i < COLUMNS_HALO_STEPS + COLUMNS_RESULT_STEPS; i++) {
    float sum = 0;
#pragma unroll

    for (int j = -KERNEL_RADIUS; j <= KERNEL_RADIUS; j++) {
      sum += c_Kernel[KERNEL_RADIUS - j] *
             s_Data[threadIdx.x][threadIdx.y + i * COLUMNS_BLOCKDIM_Y + j];
    }

    d_Dst[i * COLUMNS_BLOCKDIM_Y * pitch] = sum;
  }
}
```

（2）实现的细节

可分离的滤波器

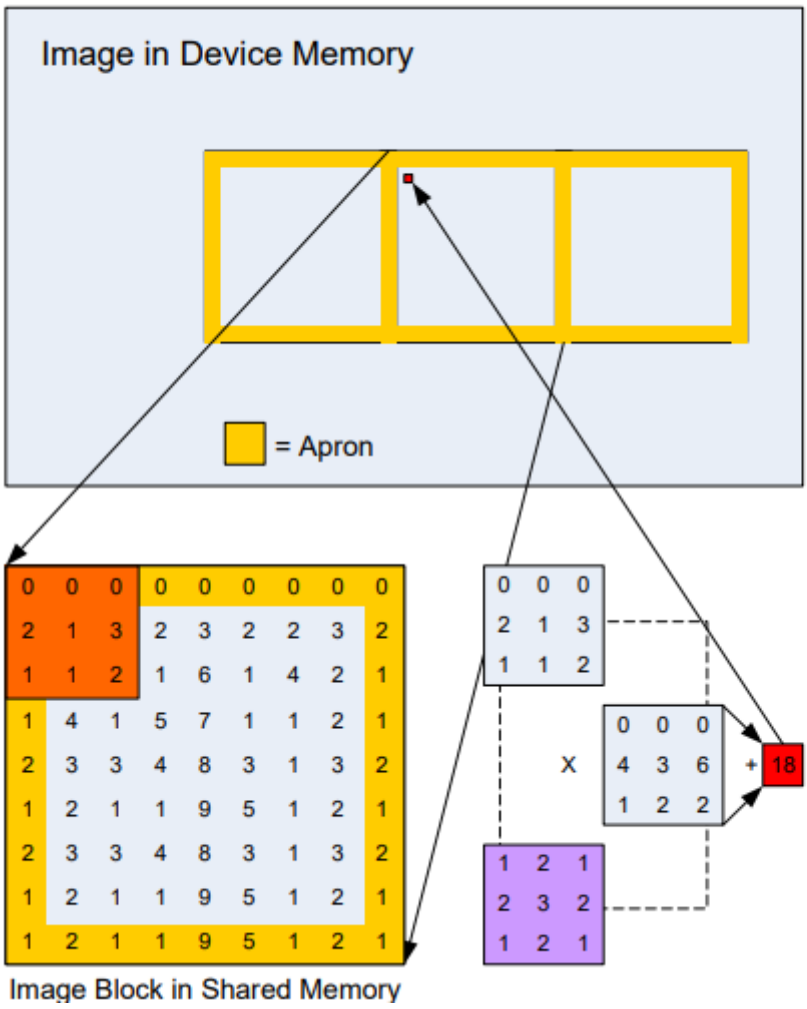
一般来说，一个二维卷积滤波器需要对每个输出像素进行n*m的乘法运算。
每个输出像素，其中n和m是滤波器内核的宽度和高度。可分离的
滤波器是一种特殊类型的滤波器，可以表示为两个一维滤波器的组合，一个是图像上的行，一个是列。
一个可分离的滤波器可以被分为两个连续的一维卷积操作。因此每个输出像素只需要n+m的乘法运算。
举例来说。
图2中的滤波器是一个可分离的Sobel[2]边缘检测滤波器。

将 $\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ 应用给图像等价于 $\begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$ 与 $\begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$ 的连续应用。

可分离的滤波器的好处是在实施中提供更多的灵活性。此外，还可以减少算术的复杂性和每个数据点的计算的带宽使用。
每个数据点的算术复杂性和带宽使用。
NVIDIA CUDA SDK中的convolutionSeparable代码样本使用了一个可分离的高斯[3]模糊滤波器。

共享内存和padding

该算法本身有些复杂。对于任何合理的过滤器内核大小，共享内存阵列边缘的共享内存阵列边缘的像素将取决于不在共享内存中的像素。
在一个线程块中的图像块周围，为了过滤图像块，需要一个宽度为内核半径的padding。因此，每个线程块必须把要过滤的像素和padding像素加载到共享内存中。这在下图中显示所示。注意：一个块的padding与相邻块重叠。图像边缘的块的padding在图像边缘的块的padding延伸到图像之外--这些像素可以是夹在图像边缘的像素的颜色上，或者将它们设置为零。



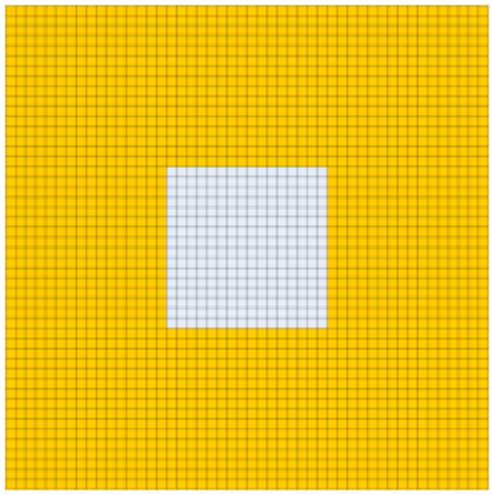

```
// Load left halo
#pragma unroll

  for (int i = 0; i < ROWS_HALO_STEPS; i++) {
    s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] =
        (baseX >= -i * ROWS_BLOCKDIM_X) ? d_Src[i * ROWS_BLOCKDIM_X] : 0;
  }

// Load right halo
#pragma unroll

  for (int i = ROWS_HALO_STEPS + ROWS_RESULT_STEPS;
       i < ROWS_HALO_STEPS + ROWS_RESULT_STEPS + ROWS_HALO_STEPS; i++) {
    s_Data[threadIdx.y][threadIdx.x + i * ROWS_BLOCKDIM_X] =
        (imageW - baseX > i * ROWS_BLOCKDIM_X) ? d_Src[i * ROWS_BLOCKDIM_X] : 0;
  }
```

上图是代码中处理padding。

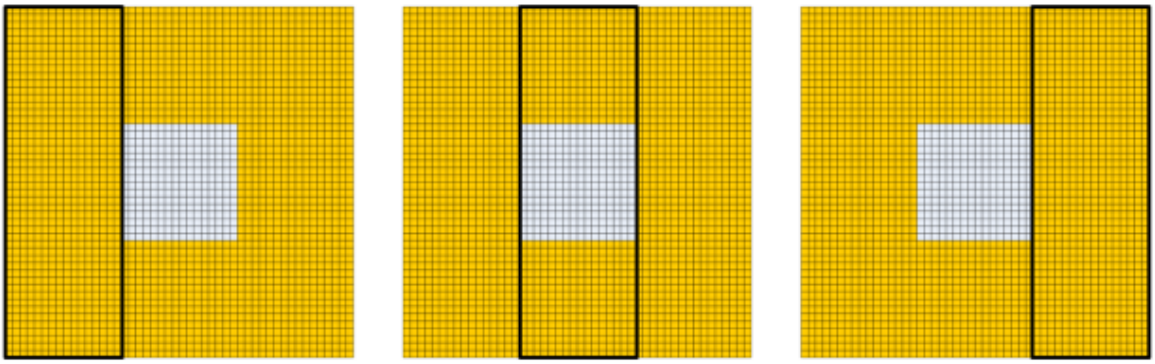
避免闲置线程

如果每个装入共享内存的像素使用一个线程，那么线程装入padding像素的线程在滤波器计算过程中会被闲置。随着过滤器的半径
增加，空闲线程的比例也会增加。这就浪费了很多可用的并行性，在有限的共享内存的情况下，大半径内核的浪费会相当大。
作为一个例子，考虑一个16x16的图像块和一个半径为16的内核。这只能允许每个多处理器有一个活动

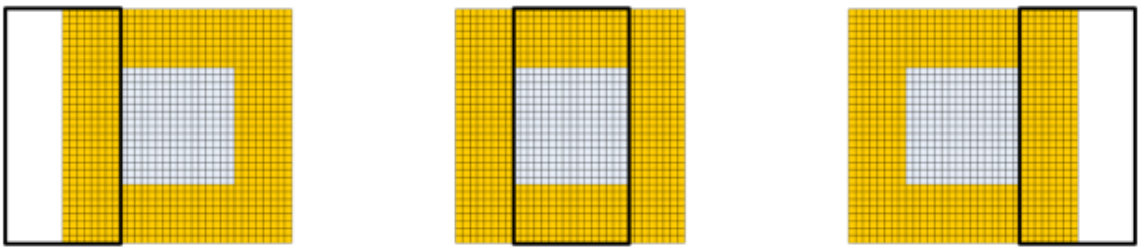
块。假设每个像素有4个字节，一个块将使用9216字节。这超过了每个多处理器可用的16KB共享内存的一半的一半以上。在这种情况下，只有1/9的线程在加载阶段后处于活动状态。正如下图中所示。



本实验可以通过减少每个区块的线程总数来减少闲置线程的数量并使用每个线程将多个像素加载到共享内存。例如，如果本实验
使用一个垂直列的线程，其宽度与本实验正在处理的图像块相同。
本实验在共享内存中有一个48x48像素的区域，但只有16x48个线程。本实验将数据读成三列，如下图示。



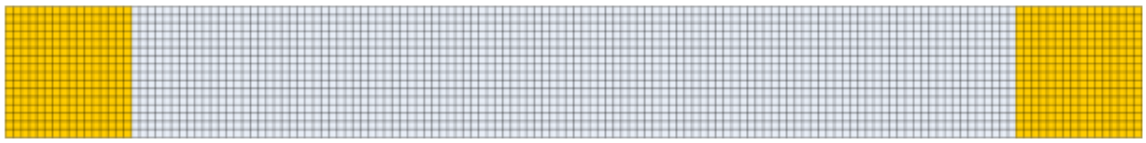
在过滤器的计算过程中，三分之一的线程是活跃的，而不是九分之一。因为负载阶段的带宽有限，所以性能不应该受到影响。这可以
通过将线程块减少到16x16个线程，并将图像块划分为9个方块的像素，可以更进一步块分为9个像素方块。这确保了所有线程在计算阶段都处于活动状态。请注意，一个块中的线程数必须是warp尺寸的倍数。块中的线程数必须是经线大小的倍数（在G80 GPU上为32个线程），以获得最佳效率。如果padding的宽度不如块的宽度，本实验就会发现在加载阶段有一些线程是不活动的。如下图所示。幸运的是，设备内存负载将被凝聚在一起，只要每个半warp的第一个适当地对齐。即使一些线程有条件地跳过它们的负载，就不会出现上述问题。
（关于设备内存访问的更多信息，请参见《CUDA编程指南》）。



这些变化可能不会持续改善性能，因为额外的复杂性可能超过了其优势。在下一节中，本实验将利用可分离的
过滤器的优势来进一步减少浪费或闲置的线程。

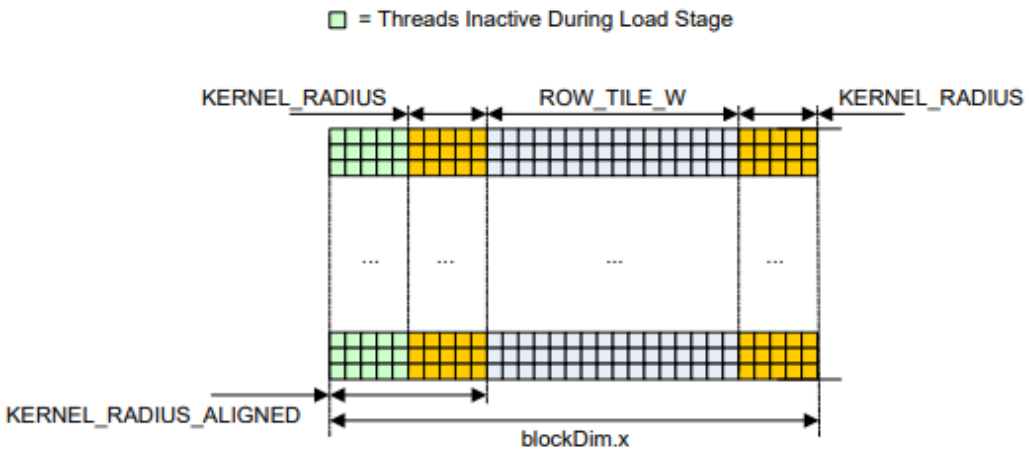
可分离的过滤器提高了效率

除了通过平铺来减少空闲线程的数量外，本实验还可以通过将处理过程分成两部分来减少不必要的数据负载。在一个可分离的滤波器中，对两个维度各执行一次处理。在上一节的最后一项技术中，一个48x48的区域包括一个16像素的padding。每个像素在
在图像外部的padding宽度区域内的每个像素将被加载9次，因为相邻块之间有重叠。
如果本实验把计算分成水平（行）和垂直（列）两部分。在每个通道之间写到全局内存，每个像素最多会被加载六次。对于一个小块（16x16）来说，这并没有什么好处。真正的好处是，它不再需要加载顶部和底部的padding区域（用于水平通道）的像素。这使得在每个线程块中可以加载更多的像素进行处理。！！！本实验是受限于线程块大小而不是共享内存大小。为了达到更高的效率，每个线程必须处理一个以上的像素。本实验可以增加一个线程块处理的图像的宽度。如下图所示。这导致了性能的显著提高。



优化内存的凝聚力

DRAM的带宽要比主机CPU存储器的带宽高得多。然而，为了实现高内存吞吐量，GPU试图将多个线程的访问凝聚成一个单一的内存事务。如果一个warp中的所有线程（32个线程）同时读取连续的字，那么对32个字的单一读取可以以最佳速度执行。读取32个随机地址，那么只能实现总DRAM带宽的一小部分，性能将大大降低。
32个线程的基础读/写地址也必须满足半线程的对齐的要求，以便进行凝聚。如果读取的是四字节的值，那么warp的基本地址必须是64。如果带有apron的数据集没有以这种方式对齐，那么本实验本实验就必须修复它，使其对齐。在行过滤器中，使用的方法是在处理的前缘有额外的线程，以使线程Idx.x == 0始终读取正确对齐的地址，从而满足全局内存的对齐。这可能看起来这似乎是对线程的一种浪费，但当一个线程块所处理的数据块足够大时，它就显得不那么重要了。因为它降低了padding像素与输出像素的比例。



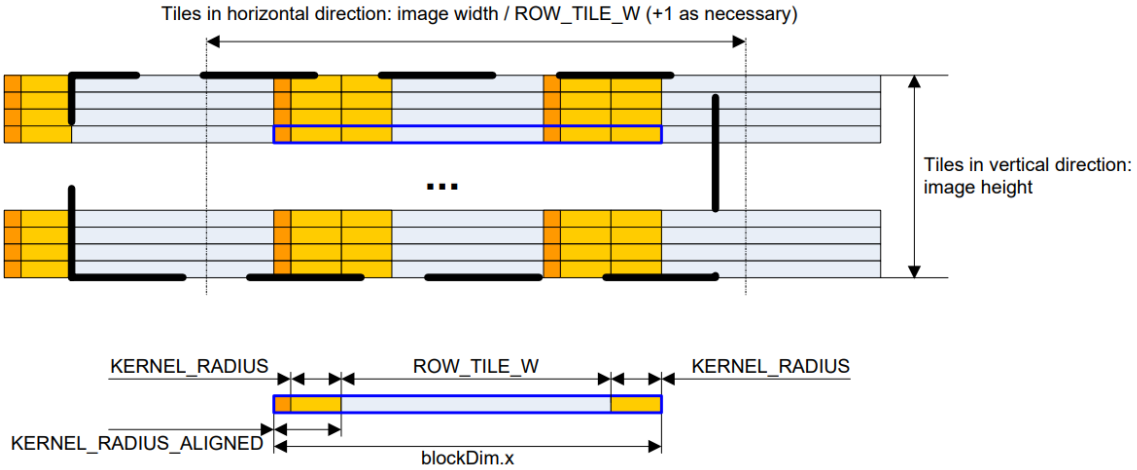
在列卷积过程中，只要图像宽度是16的倍数，padding就不会影响凝聚对齐。

每个图像卷积通道在行和列通道中都被分离成两个在相应的CUDA内核中分为两个子阶段。第一阶段是将数据从全局内存加载到共享内存中，第二阶段进行过滤并将结果写回全局内存。本实验不能忘记，当行或列处理被图像边界夹住的情况，并将夹住的共享内存的用正确的值来初始化钳制的共享内存索引。不在输入图像边界内的索引通常是初始化时，要么是零，要么是与被钳制的图像坐标对应的值。在 这个

例子中，本实验选择了前者。

在这两个阶段之间，有一个__syncthreads( ) 调用，以确保所有线程在任何处理开始之前已经写入共享内存。这是必要的，因为
线程依赖于其他线程加载的数据。

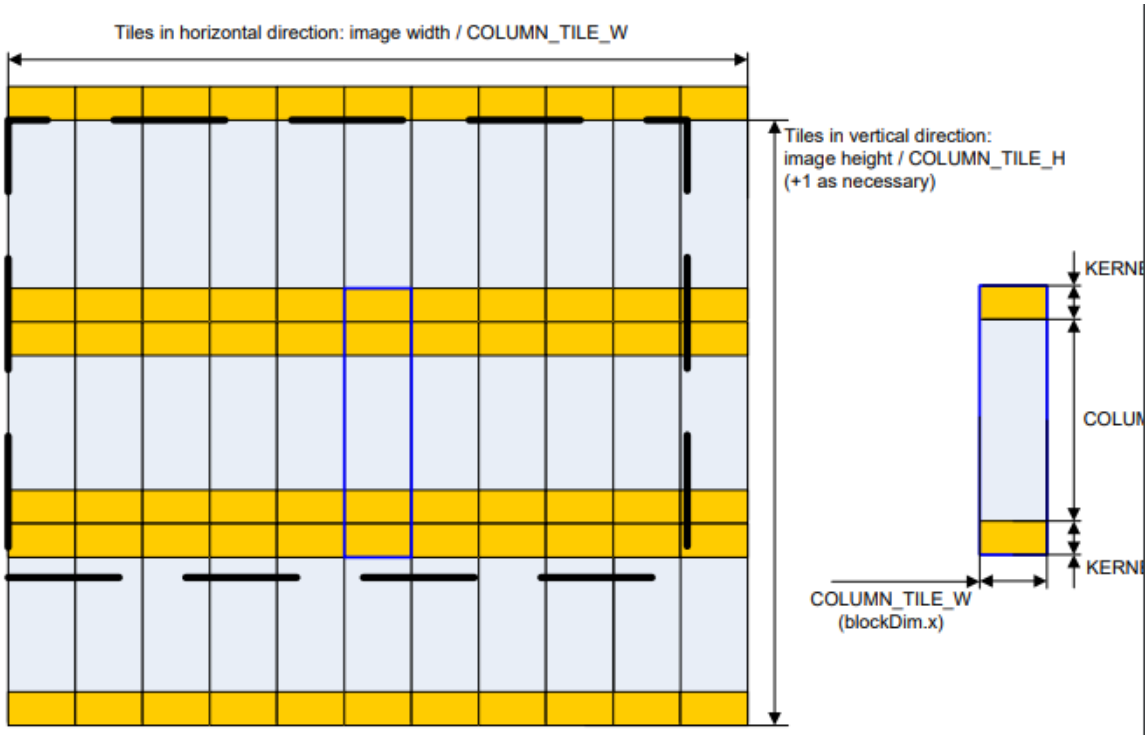
行过滤器



在加载和处理阶段，每个活动线程加载/输出一个像素。在计算阶段，每个线程在两倍于滤波器半径的宽度上循环，再加上1，将每个像素乘以存储在常量存储器中的相应的滤波系数。
在一个半warp中的每个线程都访问相同的恒定地址，因此不会因为恒定内存库冲突而受到惩罚。此外，连续的线程总是访问连续的共享内存地址，因此也不会发生共享内存库冲突。

```
__shared__ float
    s_Data[ROWS_BLOCKDIM_Y][(ROWS_RESULT_STEPS + 2 * ROWS_HALO_STEPS) *
                            ROWS_BLOCKDIM_X];

// Offset to the left halo edge
const int baseX =
    (blockIdx.x * ROWS_RESULT_STEPS - ROWS_HALO_STEPS) * ROWS_BLOCKDIM_X +
    threadIdx.x;
const int baseY = blockIdx.y * ROWS_BLOCKDIM_Y + threadIdx.y;

d_Src += baseY * pitch + baseX;
d_Dst += baseY * pitch + baseX;
```

列过滤器



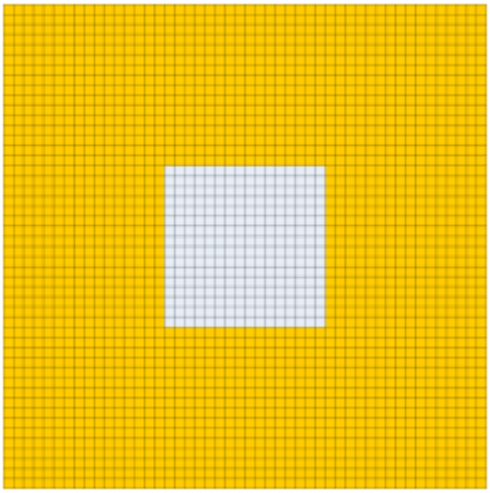
列滤波通道的操作与行滤波通道很相似。主要区别是，线程ID在整个过滤区域内增加，而不是沿着它。正如在行过滤
中，一个半warp中的线程总是访问不同的共享内存库，但是下一个/上一个地址的计算涉及到增量/减量的问题。
COLUMN_TILE_W，而不是简单的1。"凝聚对齐 "的线程，因为本实验假设瓦片宽度是凝聚读取大小的倍数。为了减少padding与输出的比例，本实验希望图像片尽可能的高，所以为了有合理的共享内存的利用率，本实验尽可能地选择薄的图像片：16列。

```
// Handle to thread block group
cg::thread_block cta = cg::this_thread_block();
__shared__ float s_Data[COLUMNS_BLOCKDIM_X][(COLUMNS_RESULT_STEPS +
                                             2 * COLUMNS_HALO_STEPS) *
                                                COLUMNS_BLOCKDIM_Y +
                                            1];

// Offset to the upper halo edge
const int baseX = blockIdx.x * COLUMNS_BLOCKDIM_X + threadIdx.x;
const int baseY = (blockIdx.y * COLUMNS_RESULT_STEPS - COLUMNS_HALO_STEPS) *
                      COLUMNS_BLOCKDIM_Y +
                  threadIdx.y;
d_Src += baseY * pitch + baseX;
d_Dst += baseY * pitch + baseX;
```

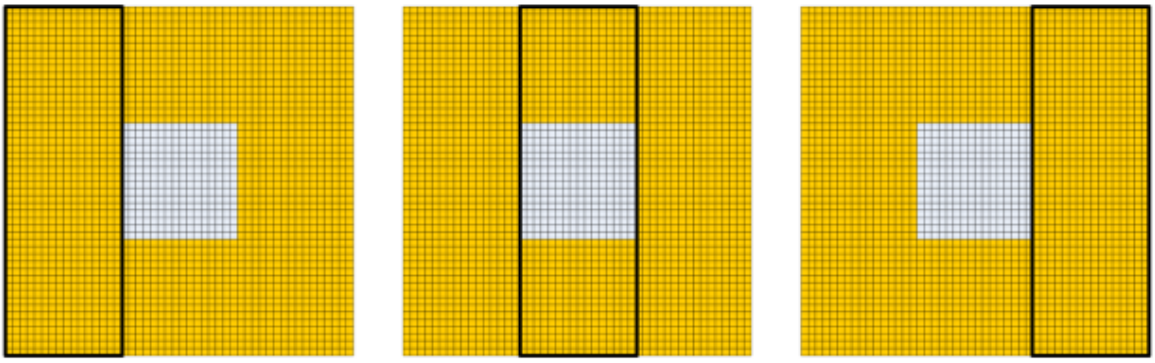
（3）涉及的gpu编程技术

避免闲置线程

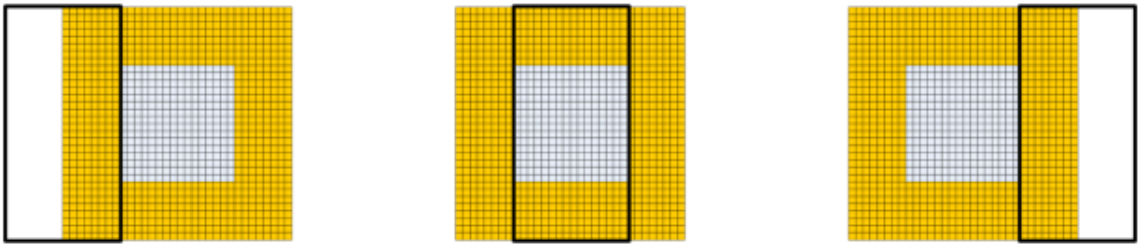
如果每个装入共享内存的像素使用一个线程，那么线程装入padding像素的线程在滤波器计算过程中会被闲置。随着过滤器的半径
增加，空闲线程的比例也会增加。这就浪费了很多可用的并行性，在有限的共享内存的情况下，大半径内核的浪费会相当大。
作为一个例子，考虑一个16x16的图像块和一个半径为16的内核。这只能允许每个多处理器有一个活动块。假设每个像素有4个字节，一个块将使用9216字节。这超过了每个多处理器可用的16KB共享内存的一半的一半以上。在这种情况下，只有1/9的线程在加载阶段后处于活动状态。正如下图中所示。



本实验可以通过减少每个区块的线程总数来减少闲置线程的数量并使用每个线程将多个像素加载到共享内存。例如，如果本实验
使用一个垂直列的线程，其宽度与本实验正在处理的图像块相同。
本实验在共享内存中有一个48x48像素的区域，但只有16x48个线程。本实验将数据读成三列，如下图示。



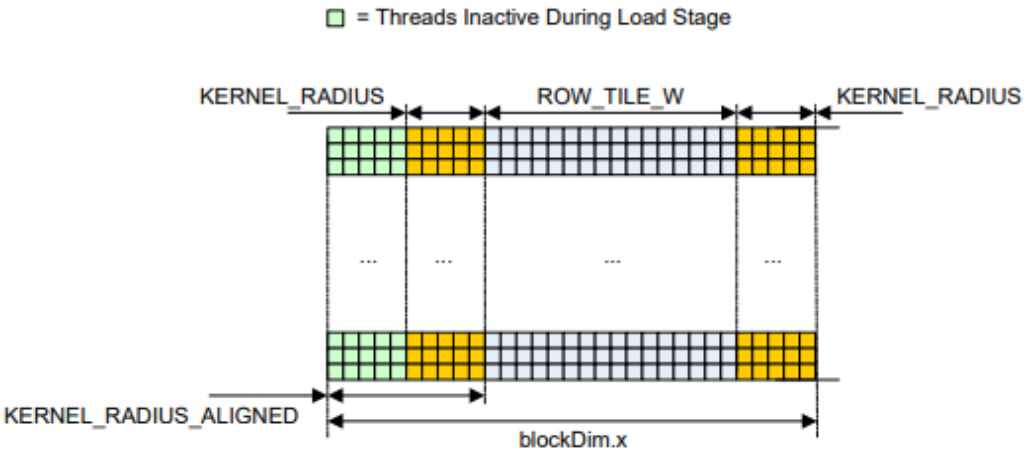
在过滤器的计算过程中，三分之一的线程是活跃的，而不是九分之一。因为负载阶段的带宽有限，所以性能不应该受到影响。这可以
通过将线程块减少到16x16个线程，并将图像块划分为9个方块的像素，可以更进一步块分为9个像素方块。这确保了所有线程在计算阶段都处于活动状态。请注意，一个块中的线程数必须是warp尺寸的倍数。块中的线程数必须是经线大小的倍数（在G80 GPU上为32个线程），以获得最佳效率。如果padding的宽度不如块的宽度，本实验就会发现在加载阶段有一些线程是不活动的。如下图所示。幸运的是，设备内存负载将被凝聚在一起，只要每个半warp的第一个适当地对齐。即使一些线程有条件地跳过它们的负载，就不会出现上述问题。
（关于设备内存访问的更多信息，请参见《CUDA编程指南》）。



这些变化可能不会持续改善性能，因为额外的复杂性可能超过了其优势。在下一节中，本实验将利用可分离的
过滤器的优势来进一步减少浪费或闲置的线程。

优化内存的凝聚力

DRAM的带宽要比主机CPU存储器的带宽高得多。然而，为了实现高内存吞吐量，GPU试图将多个线程的访问凝聚成一个单一的内存事务。如果一个warp中的所有线程（32个线程）同时读取连续的字，那么对32个字的单一读取可以以最佳速度执行。读取32个随机地址，那么只能实现总DRAM带宽的一小部分，性能将大大降低。
32个线程的基础读/写地址也必须满足半线程的对齐的要求，以便进行凝聚。如果读取的是四字节的值，那么warp的基本地址必须是64。如果带有apron的数据集没有以这种方式对齐，那么本实验本实验就必须修复它，使其对齐。在行过滤器中，使用的方法是在处理的前缘有额外的线程，以使线程Idx.x == 0始终读取正确对齐的地址，从而满足全局内存的对齐。这可能看起来这似乎是对线程的一种浪费，但当一个线程块所处理的数据块足够大时，它就显得不那么重要了。因为它降低了padding像素与输出像素的比例。



在列卷积过程中，只要图像宽度是16的倍数，padding就不会影响凝聚对齐。

每个图像卷积通道在行和列通道中都被分离成两个在相应的CUDA内核中分为两个子阶段。第一阶段是将数据从全局内存加载到共享内存中，第二阶段进行过滤并将结果写回全局内存。本实验不能忘记，当行或列处理被图像边界夹住的情况，并将夹住的共享内存的用正确的值来初始化钳制的共享内存索引。不在输入图像边界内的索引通常是初始化时，要么是零，要么是与被钳制的图像坐标对应的值。在 这个例子中，本实验选择了前者。

在这两个阶段之间，有一个__syncthreads( ) 调用，以确保所有线程在任何处理开始之前已经写入共享内存。这是必要的，因为
线程依赖于其他线程加载的数据。

我自己写了个按照定义写的带有s共享内存的版本，与源码做性能对比。结果是源码提升了一个数量级。

图像大小	原定义(shared)	源码
100	343ms	54ms
1000	3127ms	469ms
10000	32339ms	8710ms
100000	345235ms	76891ms

（4）自己的感受

要优化，可以从一下几点思考。

首先，所进行的运算有没有别的代替的方法。这主要要运用到数学上的知识，找到那个最适合在计算机上运行的方法。在本次实验中对应了将一个卷积核替换成两个向量的连续。

其次，根据gpu特性设计正确的块大小与线程数以提高gpu利用率。这在本实验中对应着将有定义出发的（一个线程对应一个output的像素，一个块对应一个卷积和的大小）改成一个线程处理多个像素，并且一个Block大于一个卷积核的大小。

当然，还要考虑到要处理的数据量的情况。要抓住问题的主要矛盾。这在本实验中体现为本实验是受限于线程块大小而不是共享内存大小。为了达到更高的效率，每个线程必须处理一个以上的像素。本实验可以增加一个线程块处理的图像的宽度。这导致了性能的显著提高。