Introduction:

- This project is about 2D tiled convolution for image processing. It is based on the book called **Programming Massively Parallel Processors: A Hands-on Approach 4th Edition** and the specific chapter is **Chapter 7 Convolution - An Introduction to Constant Memory and Caching.**
- Convolution is a popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision.
- Convolution is performed as a filter that transforms signals and pixels into more desirable values. For example:
  - Image blur filter that smoothes out the signal values so that one can see the big picture trend.
  - Gaussian filters are convolution filters that can be used to sharpen boundaries and edges of objects in images.

Background:

- In general, convolution is an array operation in which each output data element is a weighted sum of the corresponding input element and a collection of input elements that are centered on it.
- The weights that are used in the weighted sum calculation are defined by a filter array.
-
- Convolution can be performed on input data of different dimensionality: one-dimensional (1D) (e.g., audio), two-dimensional (2D) (e.g., photo), three- dimensional (3D) (e.g., video), and so on.
- For image processing and computer vision, input data is typically represented as 2D arrays, with pixels in an xy grid space. Image convolutions are therefore 2D convolutions. In a 2D convolution the filter f is also a 2D array.
  - We will focus on 2D convolution from now on since this is the main topic of our project.
-
- Specifically, 2D Convolution involves applying a filter to an input image to produce an output feature map.

Basic Concepts:

1. **Input Image**: A 2D array (or matrix) representing pixel values.
2. **Filter**: A smaller 2D array of weights. Common sizes are 3x3, 5x5, etc. In general, the filter does not have to be but is typically a square array.
1. **Feature Map**: The result of the convolution operation, highlighting certain features from the input image, e.g. edges.

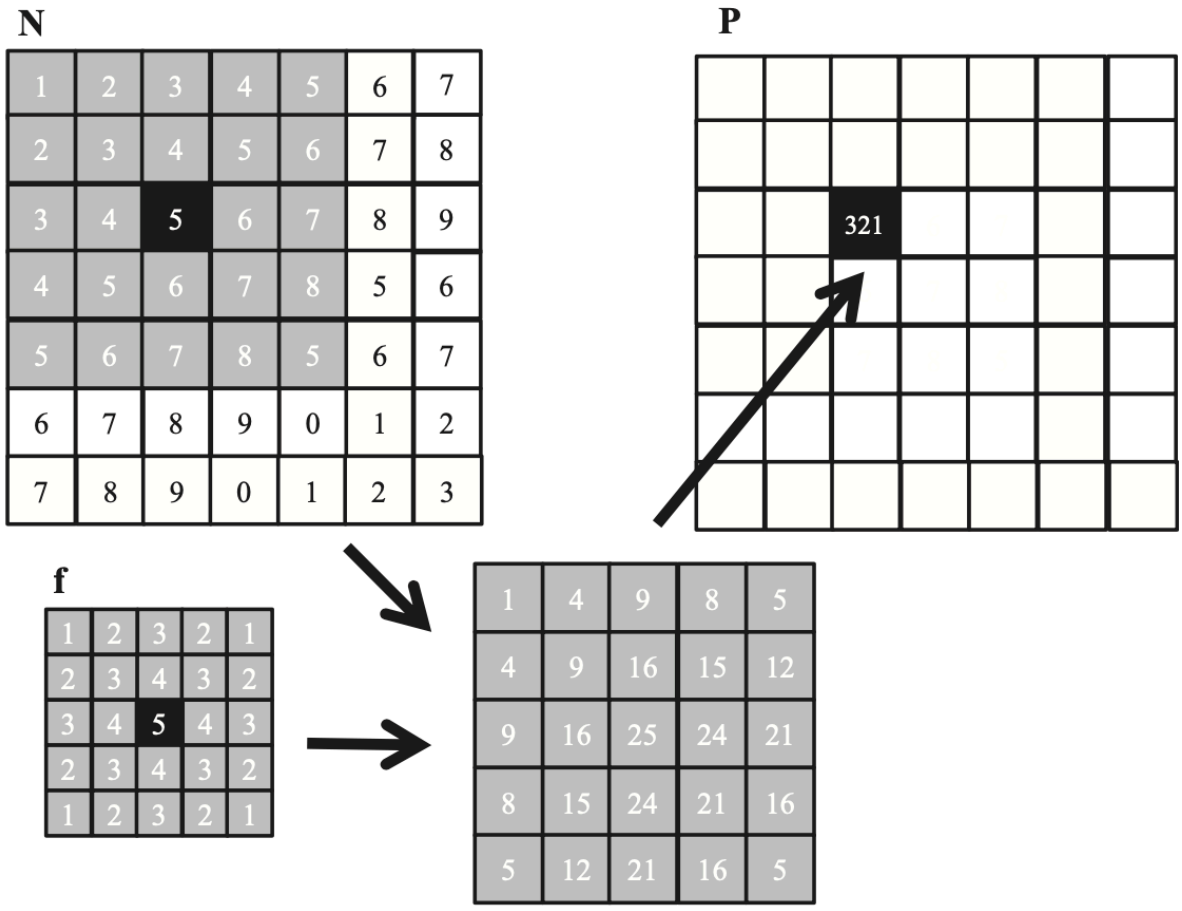
Convolution Operation:

- The convolution operation involves sliding the filter over the input image and performing element-wise multiplication and summation at each position.
- Steps:

1. Sliding the Kernel: The kernel starts at the top-left corner of the input image. It slides over the image, typically moving one pixel at a time (stride of 1), but it can move more pixels at a time if the stride is larger. We will use a stride of 1 in this project for simplicity.
2. Element-wise Multiplication and Summation: At each position, the filter is aligned with a subset of the input image pixels. Each element of the filter is multiplied by the corresponding element of the image subset, and the results are summed to get a single value. This value is placed in the corresponding position of the output feature map.

A 2D Convolution Example:

- Consider a 7x7 input image N, a 5x5 filter f, and the result 7x7 feature map P.

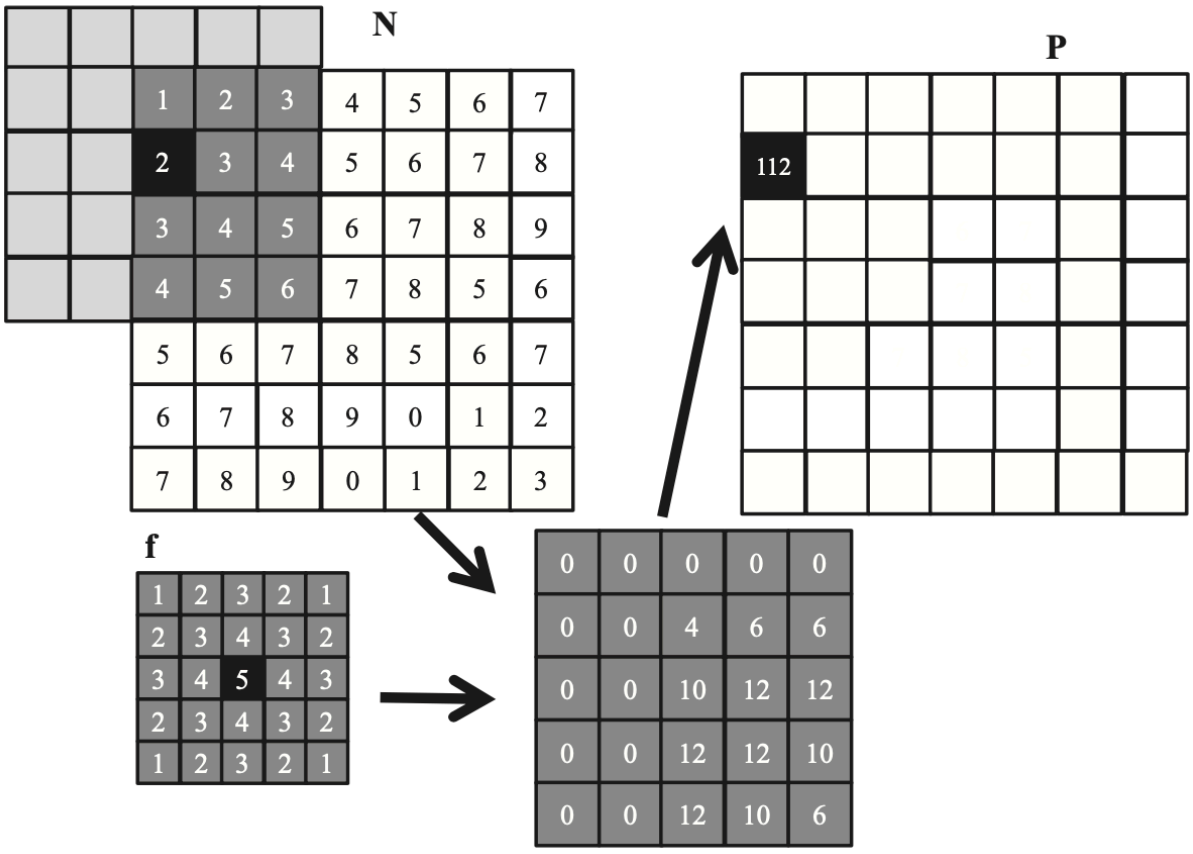


- The above diagram shows the generation of an output element (2, 2) in the feature map P. Note that the index is 0 based. We take the subarray whose center is at the corresponding location (2, 2) in the input array N. We then perform pairwise multiplication between elements of the filter array and those of the image array. For our example the result is shown as the 5 x 5 product array below N and P in the above diagram, The value of the

output element is the sum of all elements of the product array.

Boundary Conditions

- 2D convolution must also deal with boundary conditions in both the x and y dimensions. The calculation of an output feature map element may involve boundary conditions along a horizontal boundary, a vertical boundary, or both.
- The following diagram illustrates the calculation of an output feature map element (1, 0) that involves both boundaries.



- The calculation of output element (1, 0) involves two missing columns and one missing row in the subarray of input image N.
- Different applications assume different default values for these missing N elements. For our project, we assume that the default value is 0.

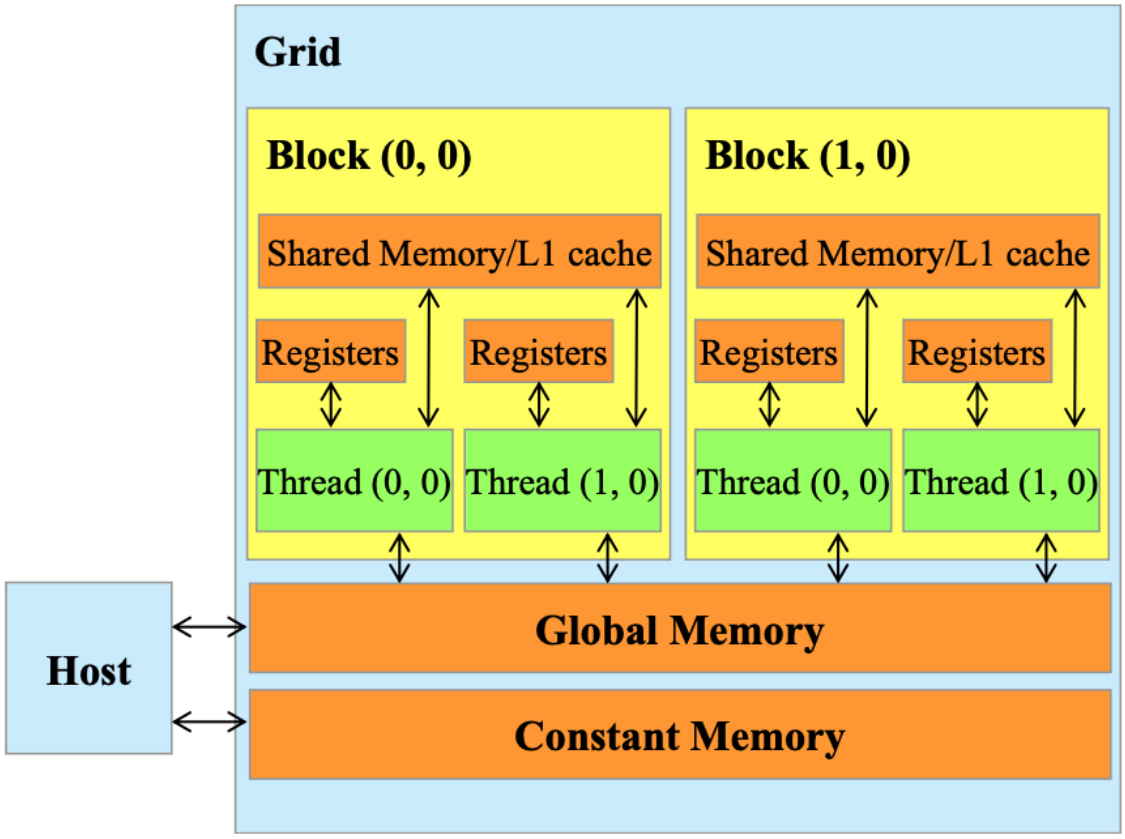
Constant Memory and Caching:

- There are three properties that make the filter an excellent candidate for constant memory and caching:
  1. The size of filter is typically small; common sizes are 3x3, 5x5, etc.
  2. The contents of filter do not change throughout the execution of the convolution

kernel.

3. All threads access the filter elements. Even better, all threads access the filter elements in the same order.

- Here is an overview of the CUDA device memory model of a GPU:

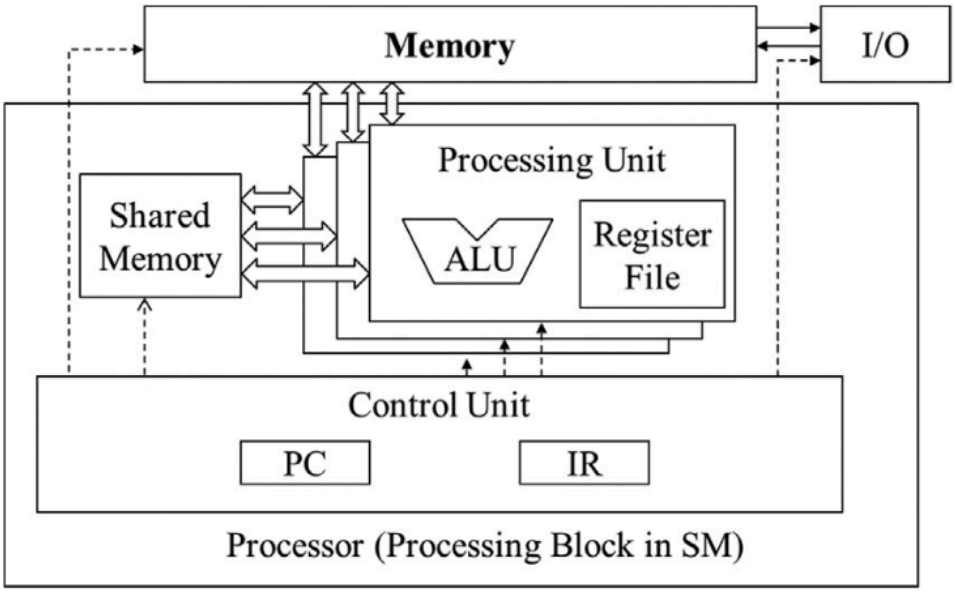


- Like global memory variables, constant memory variables are also located in DRAM. However, because the CUDA runtime knows that constant memory variables are not modified during kernel execution, it directs the hardware to aggressively cache the constant memory variables during kernel execution.

Shared Memory:

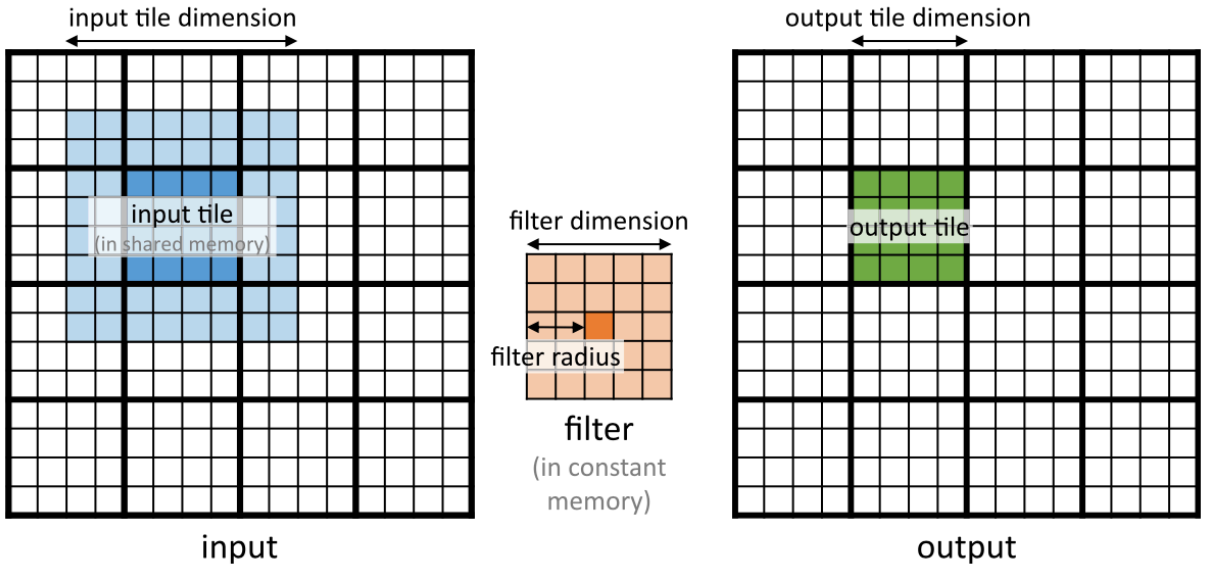
- Shared memory is designed as part of the memory space that resides on the processor chip. Because shared memory resides on-chip, it can be accessed with much lower latency and much higher throughput than the global device memory.
- Variables that reside in the shared memory are accessible by all threads in a block. That is, shared memory is designed to support efficient, high-bandwidth sharing of data among threads in a block. So shared variables are an efficient means for threads within a block to collaborate with each other.
- The scope of a shared variable is within a thread block; that is, all threads in a block see the same version of a shared variable.
- CUDA programmers often use shared variables to hold the portion of global memory data that is frequently used and reused during the execution of the kernel.

- The following diagram shows the shared memory layout in a CUDA device:



Tiled Convolution:

- We have an intrinsic tradeoff in the use of device memories in CUDA: The global device memory is large but slow, whereas the shared memory is small but fast. A common strategy is to partition the data into subsets called tiles so that each tile fits into the shared memory.
- An important criterion is that the kernel computation on these tiles can be done independently of each other.
- Since using global device memory is slow, we can address this issue by using a tiled convolution algorithm with shared memory. In this algorithm, threads collaborate to load input elements into an on-chip shared memory area for subsequent use of these elements.
- We will first establish the definitions of input and output tiles, since these definitions are important for understanding the design of the algorithm.
    - We will refer to the collection of output elements processed by each block as an output tile.
    - We define an input tile as the collection of input image N elements that are needed to calculate the feature map P elements in an output tile.
- The following diagram shows the input tile (the shaded patch on the left side) that corresponds to an output tile (the shaded patch on the right side). Note that the dimensions of the input tile need to be extended by the radius of the filter (2 in this example, while the filter dimension is 5x5) in each direction to ensure that it includes all the input elements that are needed for calculating the P elements at the edges of the output tile. This extension can make the input tiles larger than output tiles. This difference between input tile size and output tile size complicates the design of tiled convolution kernels.



- The input image N elements and output feature map P elements are stored in the Unified Memory, which can be read and written by both host and GPU.
- For the tiled convolution algorithm, all threads in a block first collaboratively load the input tile into the shared memory from Unified Memory before they calculate the elements of the output tile in Unified Memory by accessing the input elements from the shared memory.
- To recap:
  - The filter is in constant memory. It dimension is 5x5 while its radius is 2.
  - Input image N elements are in Unified Memory while input tiles are loaded into shared memory,
  - Both output feature map P elements and output tiles are stored in Unified memory.
- Our thread organization for addressing the discrepancy between the input tile size and the output tile size is the following:
  - The scheme launches thread blocks whose dimension matches that of the input tiles. This simplifies the loading of the input tiles, as each thread needs to load just one input element. However, since the input tile block dimension is larger than that of the output tile, some of the threads need to be disabled during the calculation of output elements, which can reduce the efficiency of execution resource utilization.

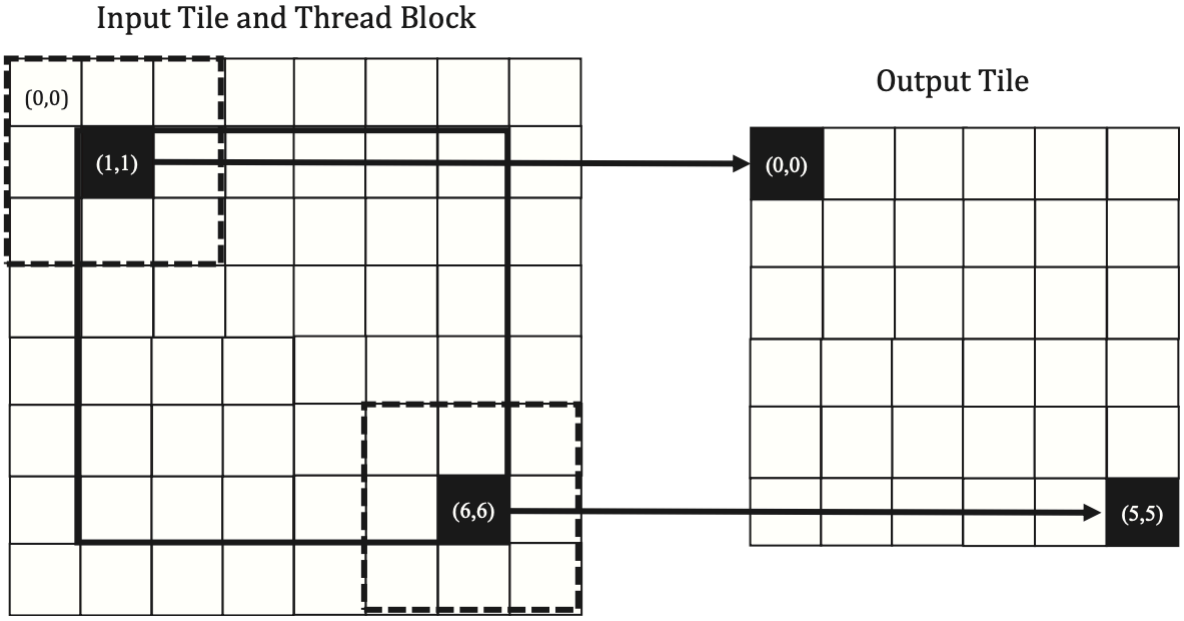
Tiled Convolution Code:

- The following diagram shows a kernel that is based on the above mentioned thread organization:

```
01  #define IN_TILE_DIM 32
02  #define OUT_TILE_DIM ((IN_TILE_DIM) - 2*(FILTER_RADIUS))
03  __constant__ float F_c[2*FILTER_RADIUS+1][2*FILTER_RADIUS+1];
04  __global__ void convolution_tiled_2D_const_mem_kernel(float *N, float *P,
05                                                        int width, int height) {
06    int col = blockIdx.x*OUT_TILE_DIM + threadIdx.x - FILTER_RADIUS;
07    int row = blockIdx.y*OUT_TILE_DIM + threadIdx.y - FILTER_RADIUS;
08    //loading input tile
09    __shared__ N_s[IN_TILE_DIM][IN_TILE_DIM];
10    if(row>=0 && row<height && col>=0 && col<width) {
11      N_s[threadIdx.y][threadIdx.x] = N[row*width + col];
12    } else {
13      N_s[threadIdx.y][threadIdx.x] = 0.0;
14    }
15    __syncthreads();
16    // Calculating output elements
17    int tileCol = threadIdx.x - FILTER_RADIUS;
18    int tileRow = threadIdx.y - FILTER_RADIUS;
19    // turning off the threads at the edges of the block
20    if (col >= 0 && col < width && row >=0 && row < height) {
21      if (tileCol>=0 && tileCol<OUT_TILE_DIM && tileRow>=0
22                    && tileRow<OUT_TILE_DIM){
23        float Pvalue = 0.0f;
24        for (int fRow = 0; fRow < 2*FILTER_RADIUS+1; fRow++) {
25          for (int fCol = 0; fCol < 2*FILTER_RADIUS+1; fCol++) {
26            Pvalue += F[fRow][fCol]*N_s[tileRow+fRow][tileCol+fCol];
27          }
28        }
29        P[row*width+col] = Pvalue;
30      }
31    }
32  }
```

- The input tile dimension is set to 32, i.e. 32x32 = 1024, which is also the size of a thread block (i.e. 32x32).
- Each thread first calculates the column index (col) and row index (row) of the input or output element that it is responsible for loading the input tile from input image N or computing the output convolution P element value (lines 06-07).
- The kernel allocates a shared memory array N_s whose size is the same as an input tile (line 09) and loads the input tile to the shared memory array (lines 10-15).
- The conditions in line 10 are used by each thread to check whether the input tile element that it is attempting to load is a boundary cell. If so, the thread does not perform a memory load. Rather, it places a zero into the shared memory.
- All threads perform a barrier synchronization (line 15) to ensure that the entire input tile is in place in the shared memory before any thread is allowed to proceed with the calculation of output P elements.
-
- Now that all the input tile elements are in the N_s array, each thread can calculate their output P element value using the N_s elements.
-
- Keep in mind that the output tile is smaller than the input tile and that the thread blocks are of the same size as the input tiles, so only a subset of the threads in each block will be used to calculate the output tile elements. There are multiple ways in which we can select

the threads for this calculation. We use a design that deactivates FILTER_RADIUS exterior layers of threads, as illustrated in the following diagram:

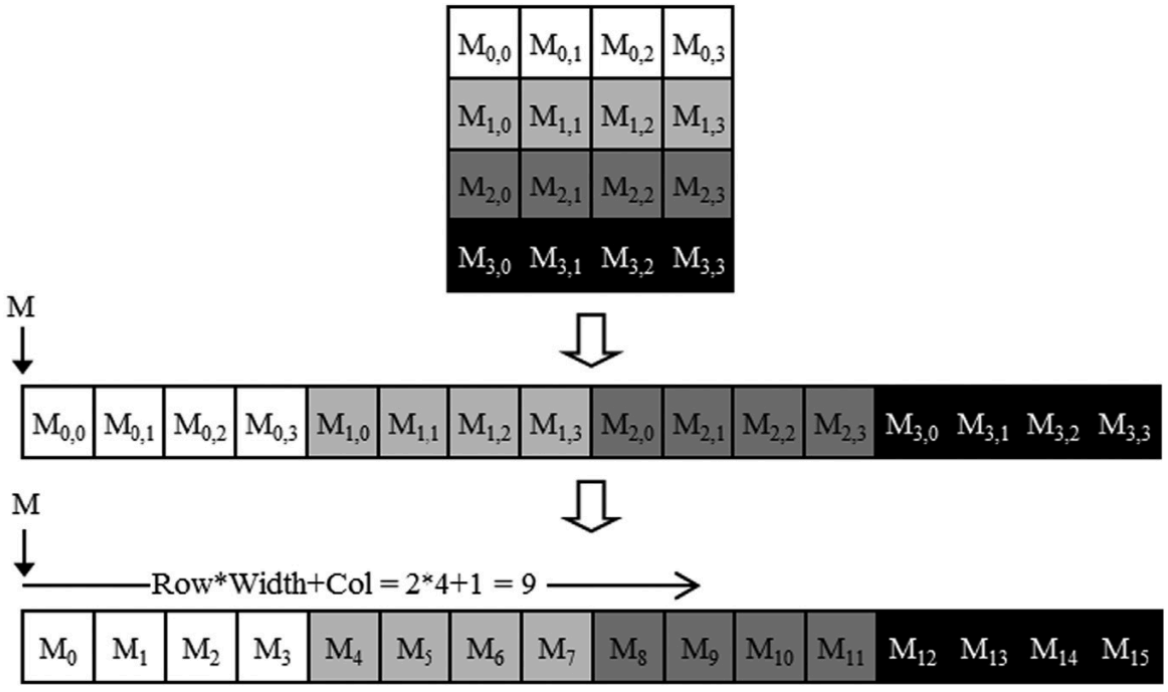


- The above diagram shows a small example of convolution using a 3x3 filter (FILTER_RADIUS=1), 8x8 input tiles, 8x8 blocks, and 6x6 output tiles.
- The left side of the diagram shows the input tile and the thread block. Since they are of the same size, they are overlaid on top of each other.
- With our design, we deactivate FILTER_RADIUS=1 exterior layer of threads.
- The heavy-line box at the center of the left side of the diagram encloses the active threads for calculating the output tile elements. In this example the threadIdx.x and threadIdx.y values of the active threads both range from 1 to 6.
- The above diagram also shows the mapping of the active threads to the output tile elements: Active thread (tx, ty) will calculate output element (tx - FILTER_RADIUS, ty - FILTER_RADIUS) using a patch of input tile elements whose upper-left corner is element (tx - FILTER_RADIUS, ty - FILTER_RADIUS) of the input tile. This is reflected in lines 17-18 of the earlier kernel code, where the column index (tileCol) and row index (tileRow) are assigned threadIdx.x - FILTER_RADIUS and threadId.y - FILTER_RADIUS, respectively.
-
- In our small example above, tileCol and tileRow of thread (1,1) receive 0 and 0, respectively. Thus thread (1, 1) calculates element (0,0) of the output tile using the 3x3 patch of input tile elements highlighted with the dashed box at the upper-left corner of the input tile.
- The fRow-fCol loop nest on lines 24-28 of the earlier kernel code iterates through the patch and generates the output P element.
- Thread (1,1) in the block will iterate through the patch whose upper-left corner is N_s[0][0], whereas thread (6,6) will iterate through the patch whose upper-left corner is N_s[5][5].

- In lines 06-07 of the earlier kernel code, blockIdx.x*OUT_TILE_DIM and blockIdx.y*OUT_TILE_DIM are the horizontal and vertical P array indices, respectively, of the beginning of the output tile assigned to the block. As we discussed earlier, threadIdx.x - r and threadIdx.y - r give the offset into the tile. Thus the row and the col variables provide the index of the output element assigned to each active thread. Each thread uses these two indices to write the final value of the output element in line 29 of the earlier kernel code.

Linearized 2D arrays:

- In the earlier kernel code line 11, input image N is accessed as N[row * width + col] instead of N[row][col]. This is due to the fact that the ANSI C standard on the basis of which CUDA C was developed requires the number of columns in N to be known at compile time for N to be accessed as a 2D array. Unfortunately, this information is not known at compile time for dynamically allocated arrays like N. Input image N is read in from an image file during run time. As a result, programmers need to explicitly linearize, or "flatten," a dynamically allocated 2D array into an equivalent 1D array in the current CUDA C. Consequently, input image N is accessed as N[row * width + col] since row-major layout scheme is used for 2D C arrays.
- Similar point can be mentioned for the output feature map P (i.e. P[row * width + col]).
- The following diagram shows the row-major layout for a 2D C array:



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 jth row and ith column of an array of Width elements in each row.

Convolutions on RGB Image:

- Each input image N actually has red (R), green (G), and blue (B) components. For simplicity, we will use the same filter to convolute with each individual component.