Heterogeneous Parallel Programming COMP4901D

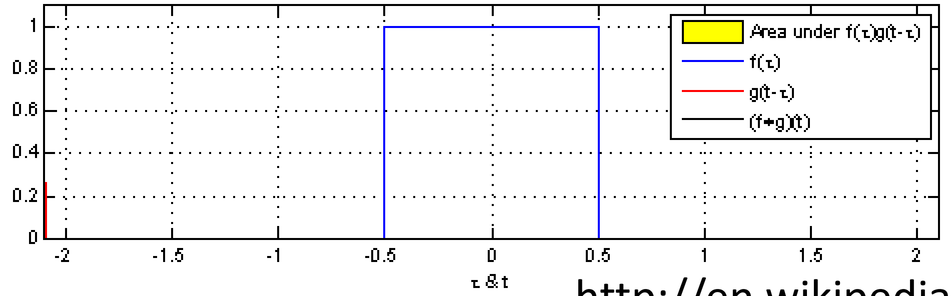
CUDA Example: Convolution

Overview

- Convolution
  - Math operation
  - Baseline CUDA implementation
  - Tiling and reuse in convolution

Convolution: an Math Operation

- Given two functions, produce a third function
  - The output function is a modified version of one of the two input functions
  - The output function is on the area overlap of the two functions
  - The output function is given on the amount one of the input functions translates



http://en.wikipedia.org/wiki/Convolution

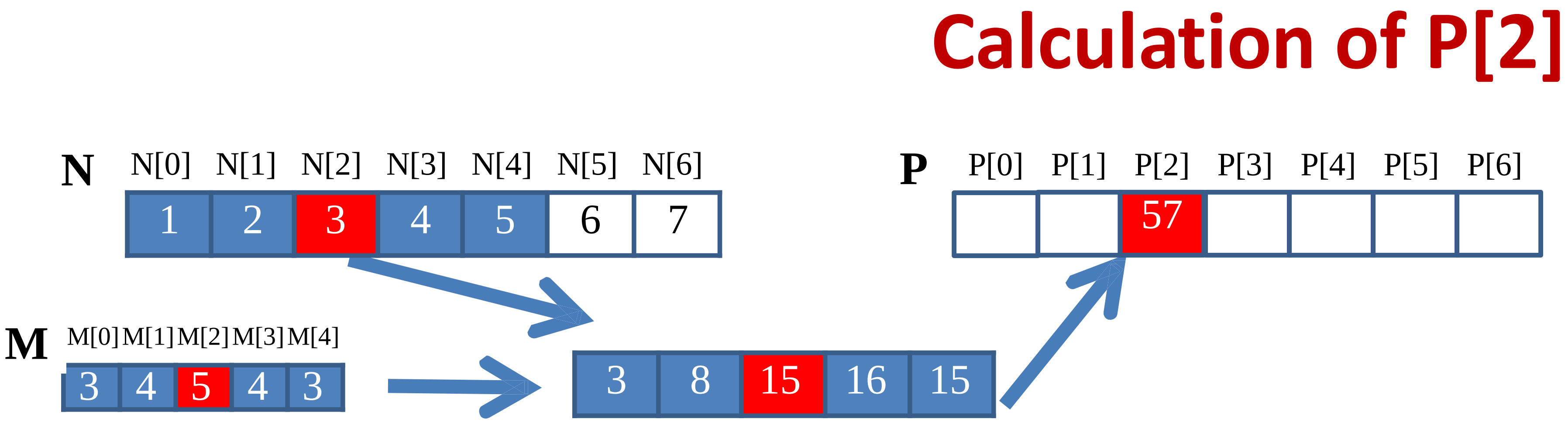
Applications of Convolution

- Signal, image, and video processing
  - Filtering out noises to show the big picture
  - Sharpening boundaries/edges
- Science and engineering computation
  - Fluid dynamics simulation
  - partial differential equations

Computation of Convolution

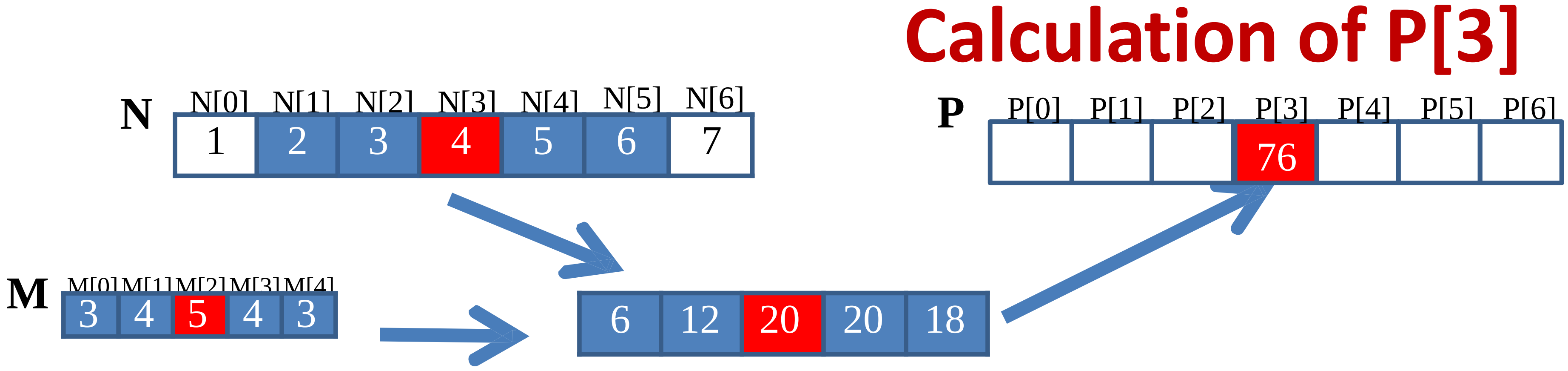
- An array operation where each output data element is a weighted sum of a collection of neighboring input elements
- The weights used in the weighted sum calculation are defined by an input mask array, commonly referred to as the **convolution kernel**
  - We will refer to these mask arrays as **convolution masks** to avoid confusion with CUDA kernel programs.
  - The same convolution mask is typically used for all elements of the array.

1D Convolution Example

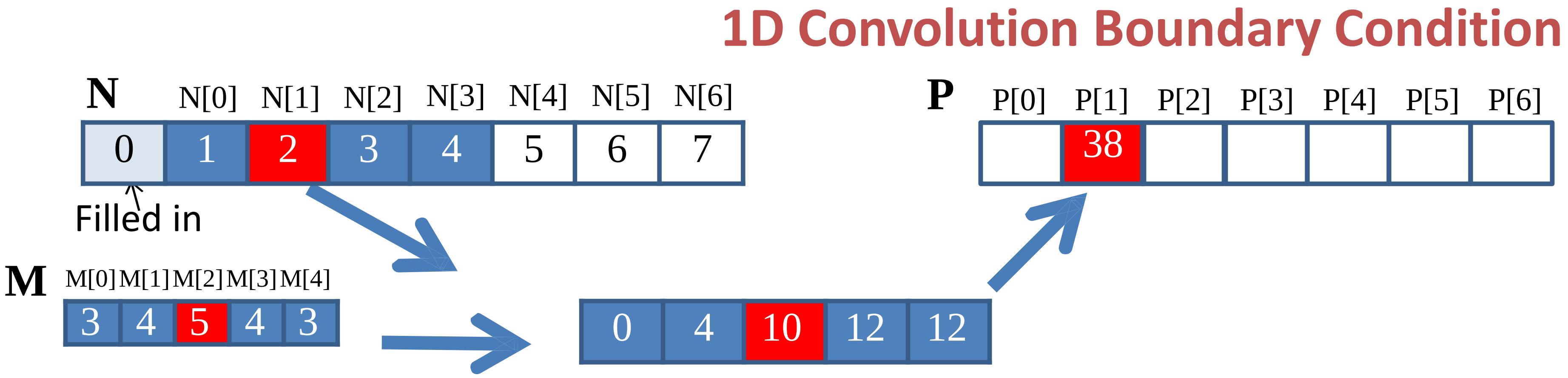


- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)

1D Convolution Example (Cont.)



1D Convolution Example (Cont.)



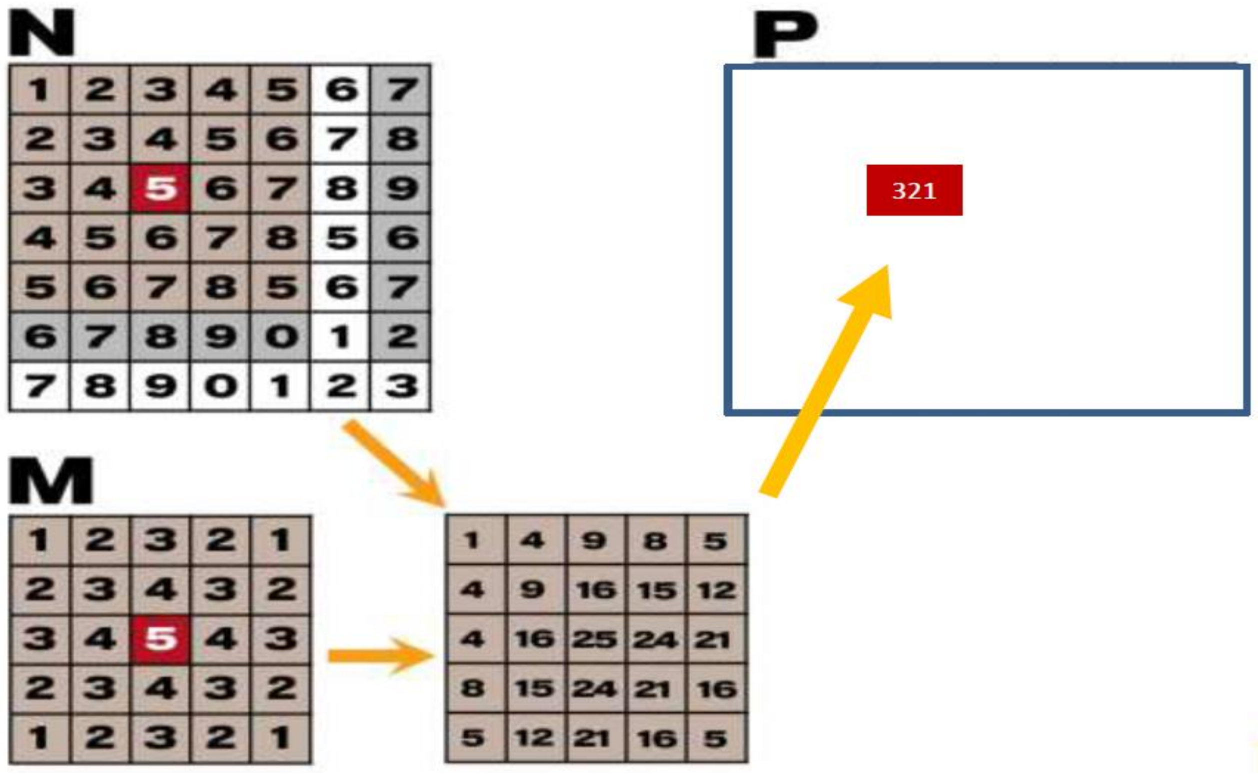
1D Convolution Boundary Condition
N
N[0] N[1] N[2] N[3] N[4] N[5] N[6]
0 1 2 3 4 5 6 7
Filled in
M
M[0] M[1] M[2] M[3] M[4]
3 4 5 4 3
0 4 10 12 12
P
P[0] P[1] P[2] P[3] P[4] P[5] P[6]
38

- Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with "ghost" elements.
- Different policies (0, replicates of boundary values, etc.) may be used to fill in the boundaries.

1D Convolution Example in CUDA

```
__global__ void convolution_lD_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;
}
```

2D Convolution Example



N

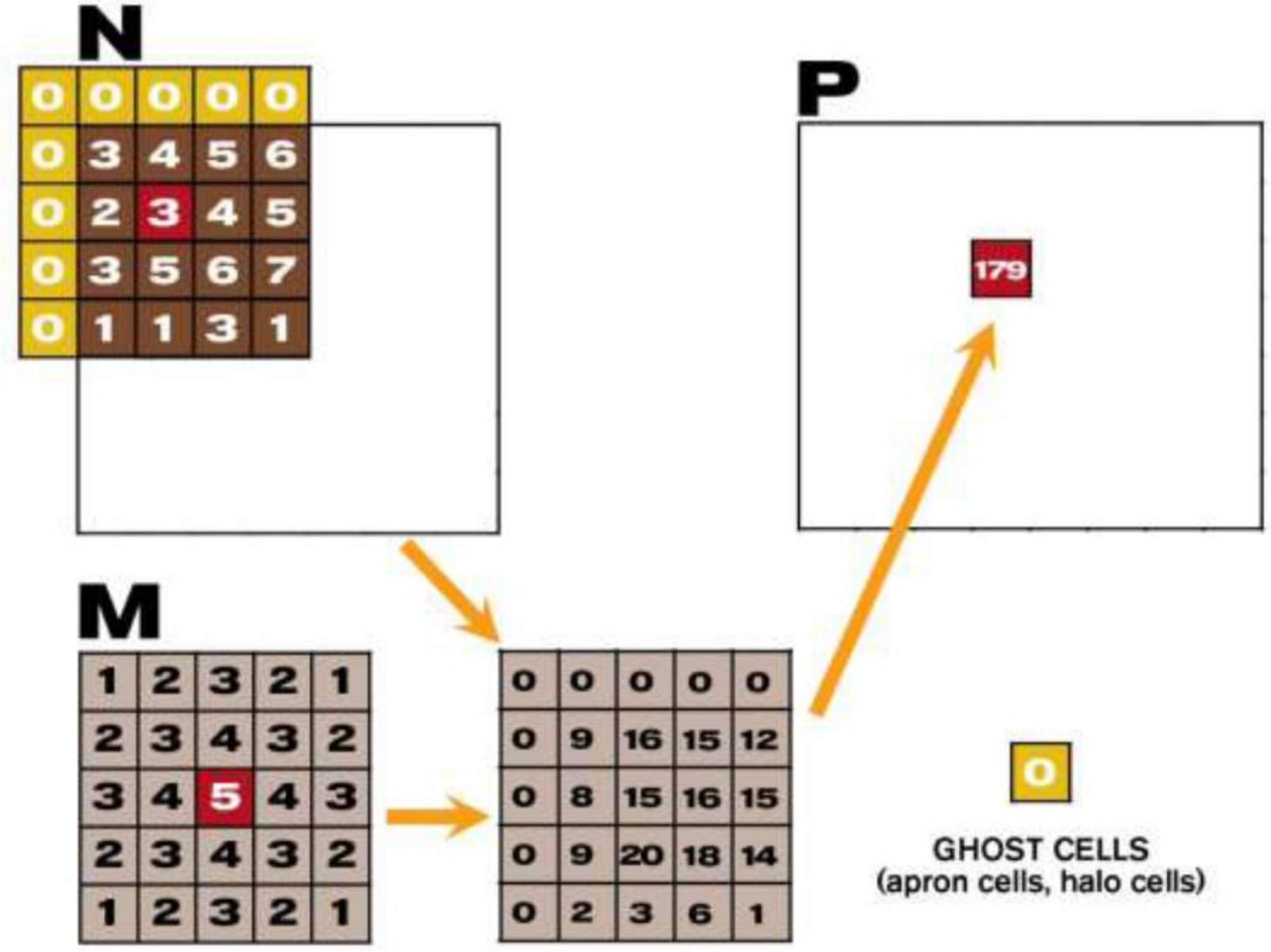
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	5	6
5	6	7	8	5	6	7
6	7	8	9	0	1	2
7	8	9	0	1	2	3

M

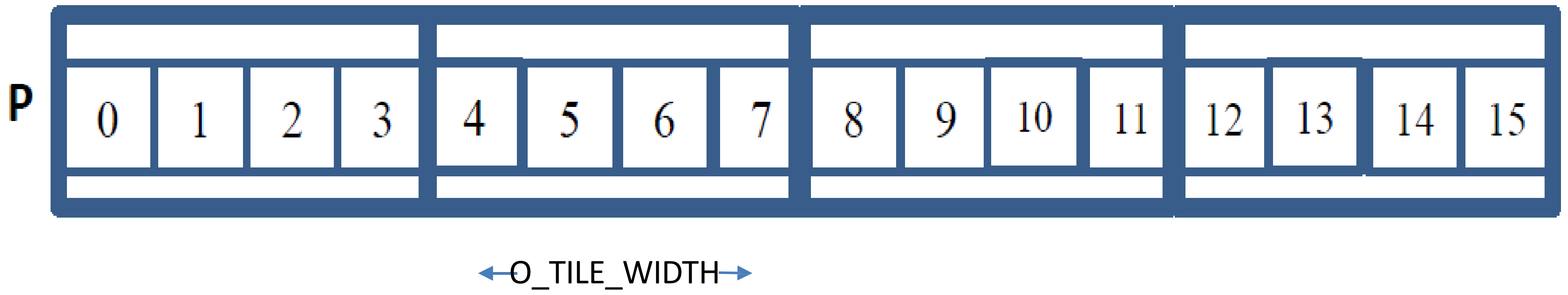
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

1	4	9	8	5
4	9	16	15	12
4	16	25	24	21
8	15	24	21	16
5	12	21	16	5

2D Convolution Example: Boundaries



Tiling Convolution: Output Elements



Each thread block calculates an output tile

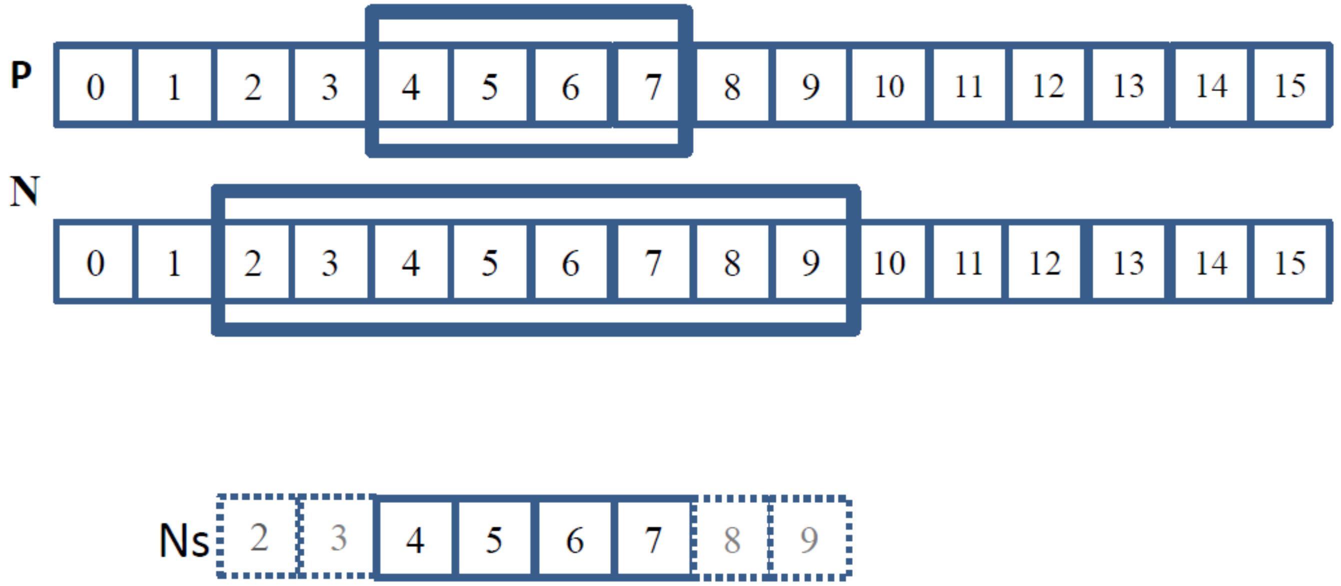
Each output tile width is O_TILE_WIDTH

For each thread,

index_o = blockIdx.x*O_TILE_WIDTH + threadIdx.x

O_TILE_WIDTH is 4 in this example

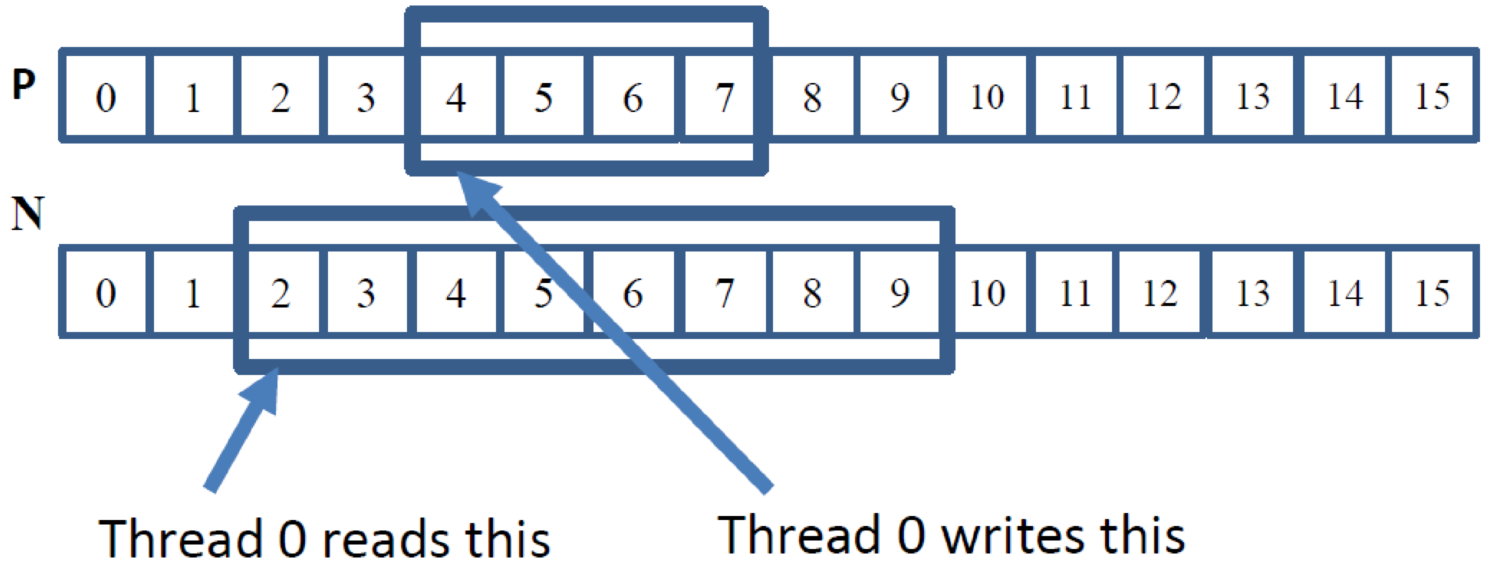
Tiling Convolution: Input Elements



Each input tile has all values needed to calculate the corresponding output tile.

Size each thread block to cover input tiles
blockDim.x is 8 in this example

Thread to Input Data Mapping



For each thread,
index_i = index_o – n

where n is Mask_Width /2. n is 2 in this example

All Threads Participate in Loading Input

```
float output = 0.0f;

if((index_i >= 0)  && (index_i < Width) ) {
  Ns[tx] = N[index_i];
}
else{
  Ns[tx] = 0.0f;
}
```

Only Some Threads Calculate Output

```
if (threadIdx.x < O_TILE_WIDTH){
   output = 0.0f;
   for(j = 0; j < Mask_Width; j++) {
       output += M[j] * Ns[j+threadIdx.x];
   }
   P[index_o] = output;
}
```

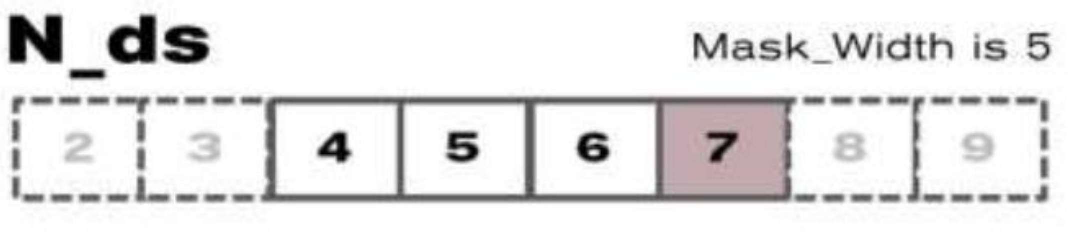
Only Threads 0 through O_TILE_WIDTH-1 participate in calculation of output.

Setting Block Size

- #define O_TILE_WIDTH 1020
- #define BLOCK_WIDTH (O_TILE_WIDTH + 4)
- dim3 dimBlock(BLOCK_WIDTH,1, 1);
- dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
- The Mask_Width is 5 in this example
- In general, block width should be

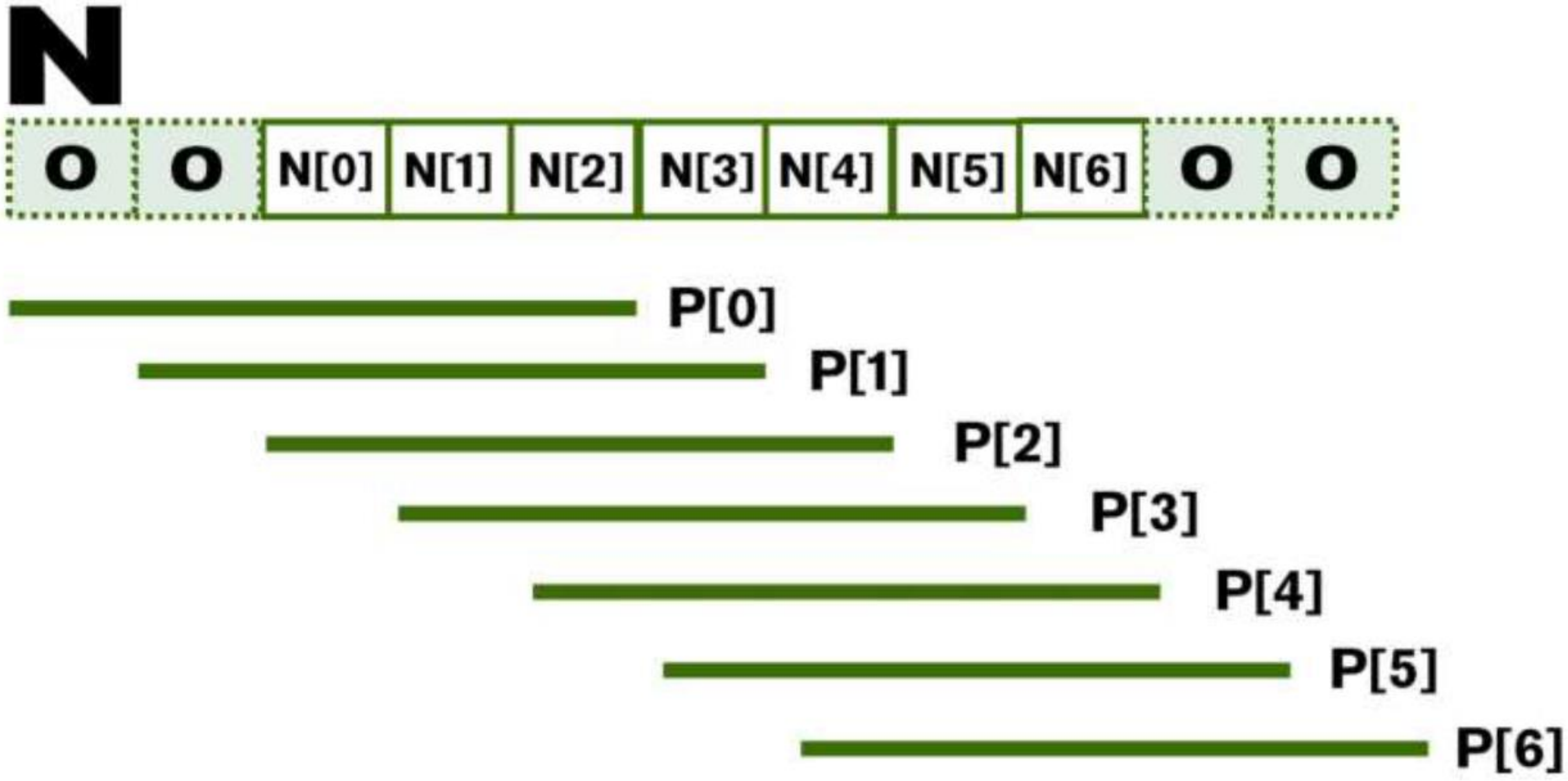
output tile width + (mask width-1)

Shared Memory Data Reuse

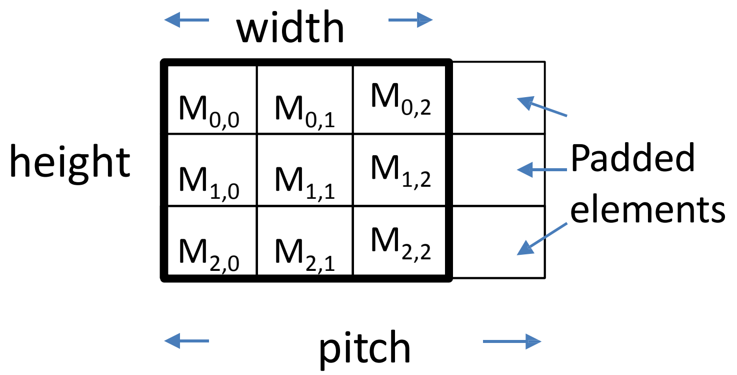


Element 2 is used by thread 4 (1X)
Element 3 is used by threads 4, 5 (2X)
Element 4 is used by threads 4, 5, 6 (3X)
Element 5 is used by threads 4, 5, 6, 7 (4X)
Element 6 is used by threads 4, 5, 6, 7 (4X)
Element 7 is used by threads 5, 6, 7 (3X)
Element 8 is used by threads 6, 7 (2X)
Element 9 is used by thread 7 (1X)

Ghost Cells



Example 2D Image Matrix with Padding



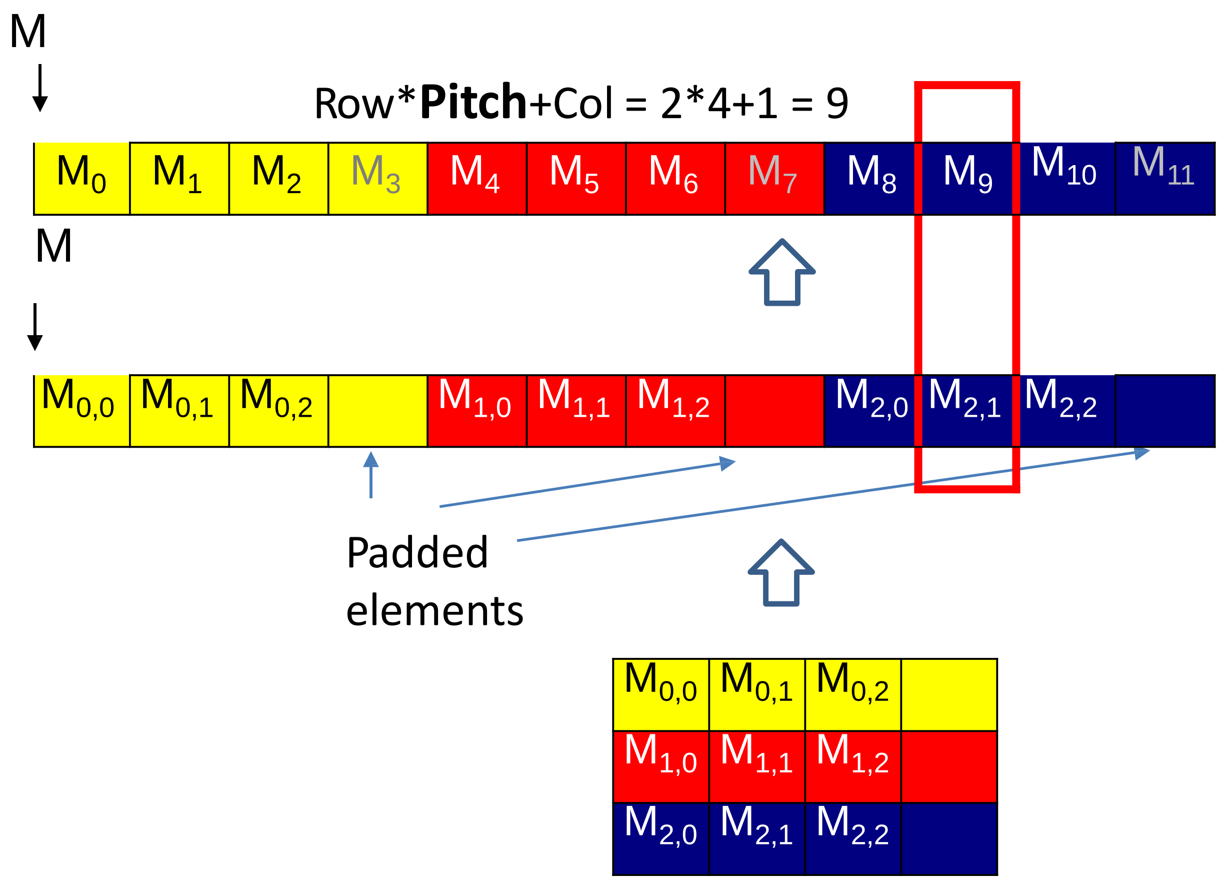
- Example: a 3X3 matrix padded into a 3X4 matrix

Height is 3
Width is 3
Channels is 1
Pitch is 4

- It is sometimes desirable to pad each row of a 2D matrix to multiples of DRAM bursts

  - So each row starts at the DRAM burst boundary
  - Effectively adding columns
  - This is usually done automatically by matrix allocation function
  - Pitch can be different for different hardware

Row-Major Layout with Pitch


M
Row*Pitch+Col = 2*4+1 = 9
M_0 M_1 M_2 M_3 M_4 M_5 M_6 M_7 M_8 M_9 M_{10} M_{11}
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Padded elements
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$

Using Constant Memory for Mask

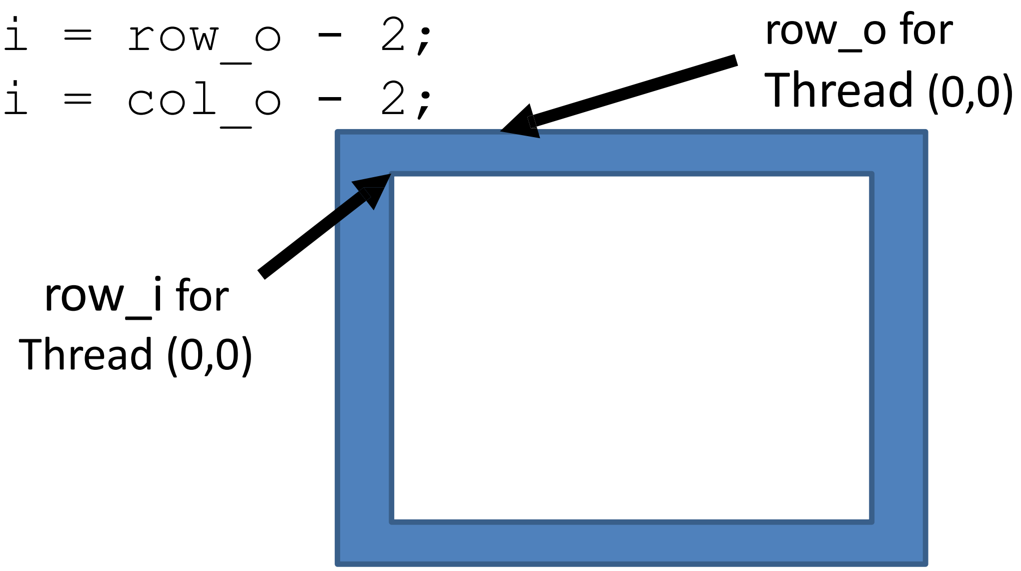
- Mask is used by all threads but not modified in the convolution kernel
  - All threads in a warp access the same locations at each point in time
- CUDA devices provide constant memory whose contents are aggressively cached
  - Cached values are broadcast to all threads in a warp
  - Effectively magnifies memory bandwidth without consuming shared memory

- Use of **const __restrict__** qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

```
__global__ void convolution_2D_kernel(float *P, float *N,
  height, width, channels,
  const float * __restrict__ M) {
```

Shifting from Output Coordinates to Input Coordinate

```
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 - 2;
int col_i = col_o - 2;
```



Taking Care of Boundaries

```
    if((row_i >=0) && (row_i < height) &&
       (col_i >= 0) && (col_i < width) ) {
      Ns[ty][tx] = data[row_i*width + col_i];
} else{
  Ns[ty][tx]      = 0.0f;
}
```

Use of width here is OK since pitch is set to width in this example.

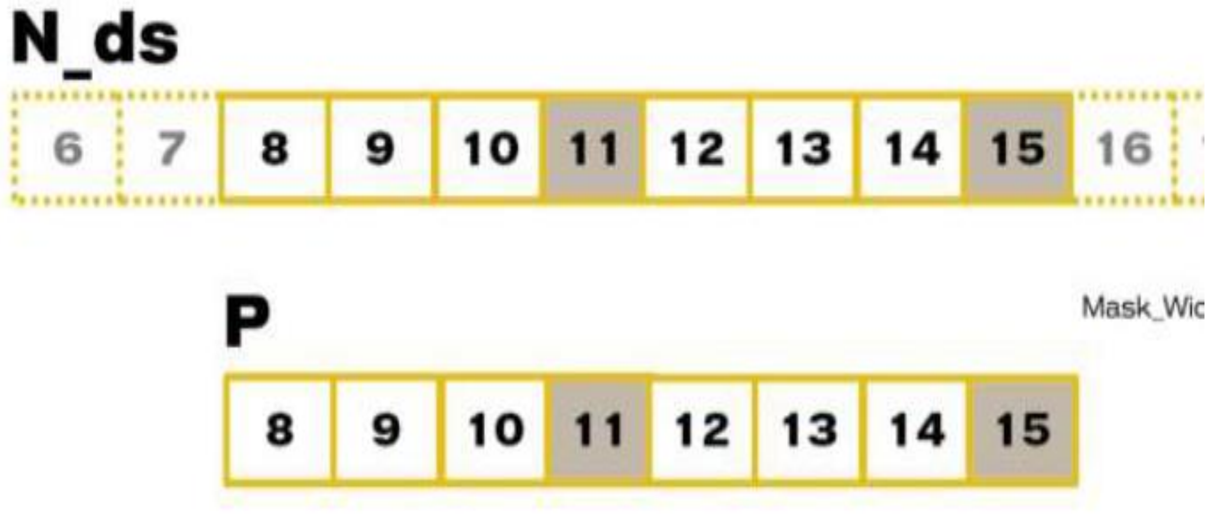
Threads within the Output Tile Calculate

```
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] * Ns[i+ty][j+tx];
        }
    }
}
```

Threads within the Matrix Boundary Output

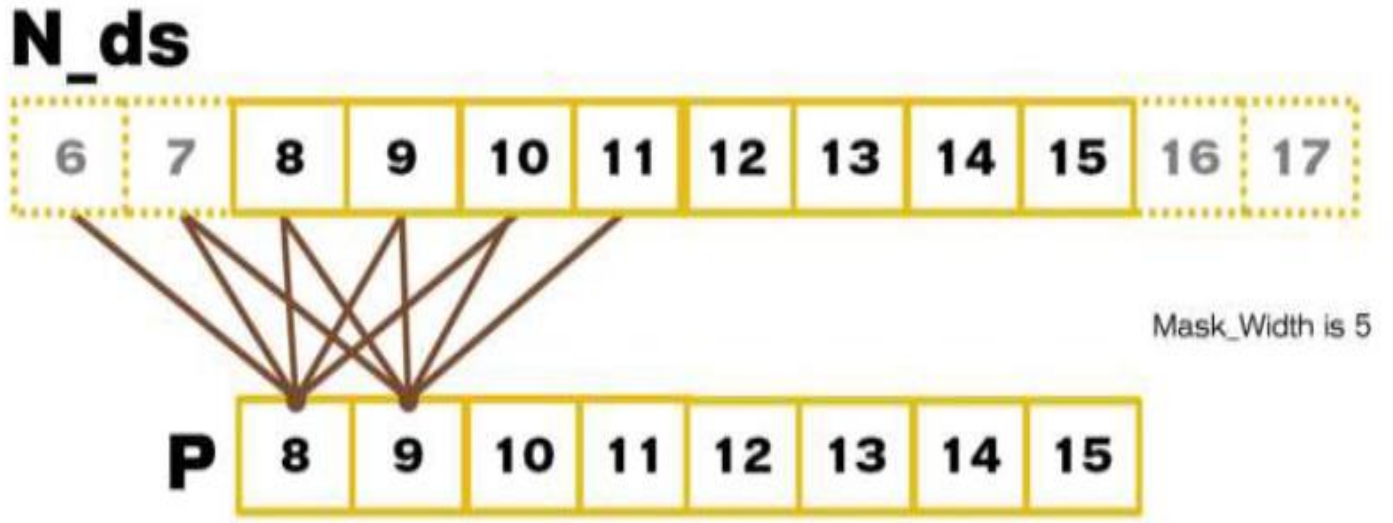
```
if(row_o < height && col_o < width)
  data[row_o*width + col_o] = output;
```

An Eight-Element Convolution Tile



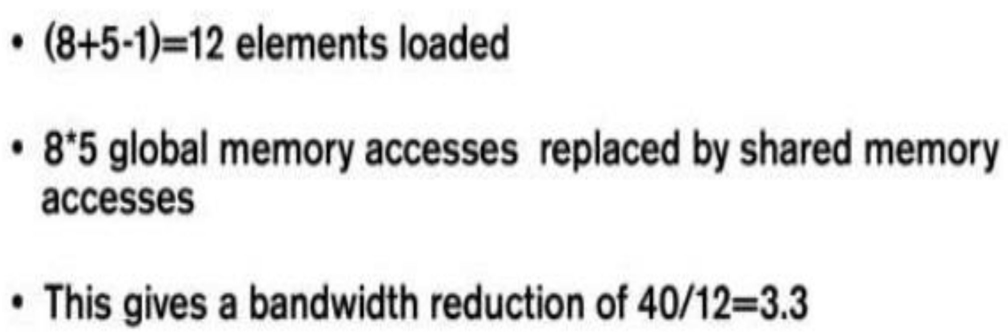
For Mask_Width = 5, we load 8+5-1 = 12 elements (12 memory l

Each Output Element Uses 5 Input Elements



P[8] uses N[6], N[7], N[8], N[9], N[10]
P[9] uses N[7], N[8], N[9], N[10], N[11]
P[10] uses N[8], N[9], N[10], N[11], N[12]
...
P[14] uses N[12], N[13], N[14], N[15],N[16]
P[15] uses N[13], N[14], N[15], N[16], N[17]

A Simple Way of Calculating Tiling Benefit



- (8+5-1)=12 elements loaded
- 8*5 global memory accesses replaced by shared memory accesses
- This gives a bandwidth reduction of 40/12=3.3

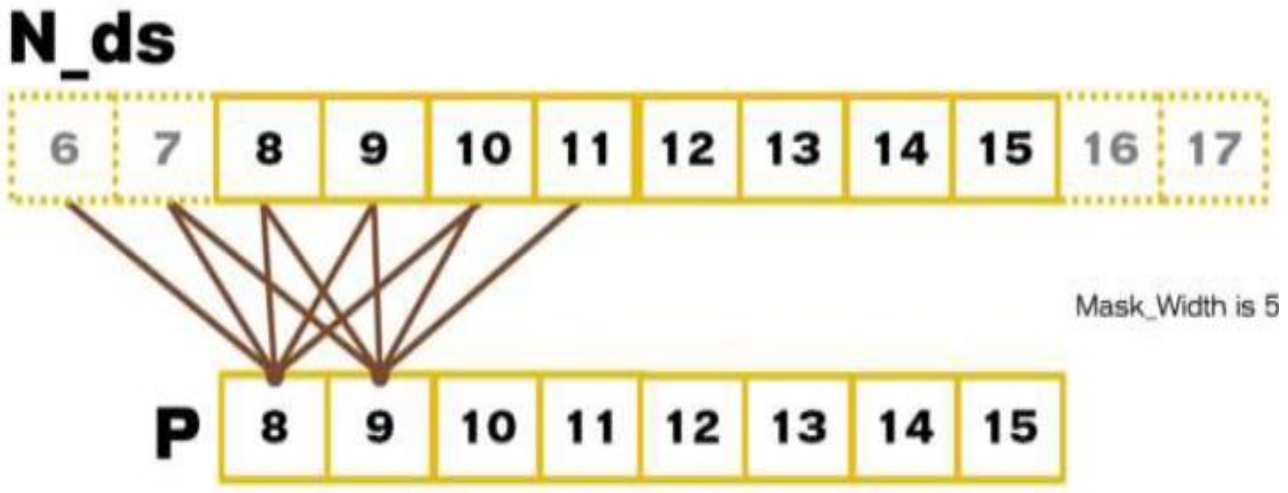
Memory Access Reduction of 1D Tiled Convolution

- O_TILE_WIDTH+MASK_WIDTH -1 elements loaded for each input tile
- O_TILE_WIDTH*MASK_WIDTH global memory accesses replaced by shared memory accesses
- This gives a reduction factor of

(O_TILE_WIDTH*MASK_WIDTH)/(O_TILE_WIDTH+MASK_WIDTH-1)

This ignores ghost elements in edge tiles.

Another Way of Looking at Reuse



N[6] is used by P[8] (1X)
N[7] is used by P[8], P[9] (2X)
N[8] is used by P[8], P[9], P[10] (3X)
N[9] is used by P[8], P[9], P[10], P[11] (4X)
N[10] is used by P[8], P[9], P[10], P[11], P[12] (5X)
... (5X)
N[14] is uses by P[12], P[13], P[14], P[15] (4X)
N[15] is used by P[13], P[14], P[15] (3X)

Another Way of Calculating Memory Access Reduction

The total number of global memory accesses (to the (8+5-1)=12 N elements) replaced by shared memory accesses is:

$$1 + 2 + 3 + 4 + 5 * (8-5+1) + 4 + 3 + 2 + 1$$
$$= 10 + 20 + 10$$
$$= 40$$

So the reduction is:

$$40/12 = 3.3$$

Another Way of Getting the Same Formula

The total number of global memory accesses to the input tile can be calculated as

1 + 2+…+ MASK_WIDTH-1 + MASK_WIDTH*(O_TILE_WIDTH-MASK_WIDTH+1) + MASK_WIDTH-1 + …+ 2 + 1
= MASK_WIDTH * (MASK_WIDTH-1) + MASK_WIDTH * (O_TILE_WIDTH-MASK_WIDTH+1)
= MASK_WIDTH * O_TILE_WIDTH

For a total of O_TILE_WIDTH + MASK_WIDTH -1 input tile elements

Memory Access Reduction: 2D Tiled Convolution

- (O_TILE_WIDTH+MASK_WIDTH-1)^2 input elements need to be loaded into shared memory
- The calculation of each output element needs to access MASK_WIDTH^2 input elements
- O_TILE_WIDTH^2 * MASK_WIDTH^2 global memory accesses are converted into shared memory accesses
- The reduction ratio is

O_TILE_WIDTH^2 * MASK_WIDTH^2 / (O_TILE_WIDTH+MASK_WIDTH-1)^2

Summary

- Convolution is widely used in computation-intensive applications.
- Parallel convolution can be done in CUDA.
- Tiling with shared memory improves memory performance of parallel convolution.