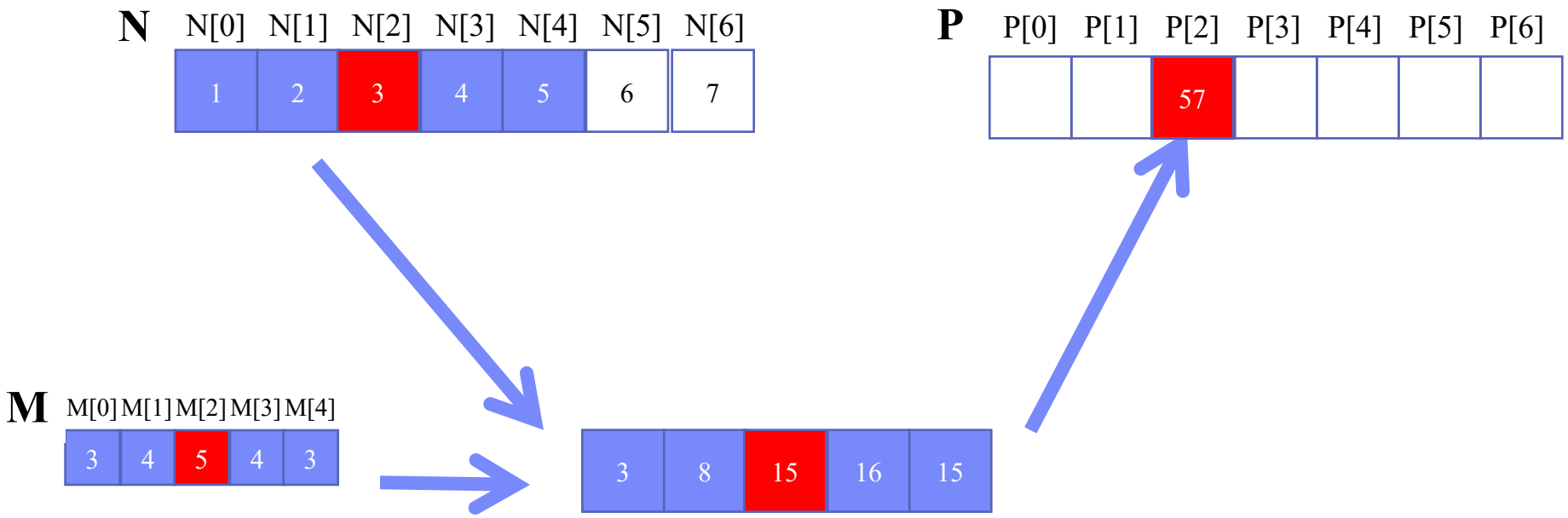
# Parallel Patterns: Convolution 2 (Tiled Convolution)

Prof. Seokin Hong

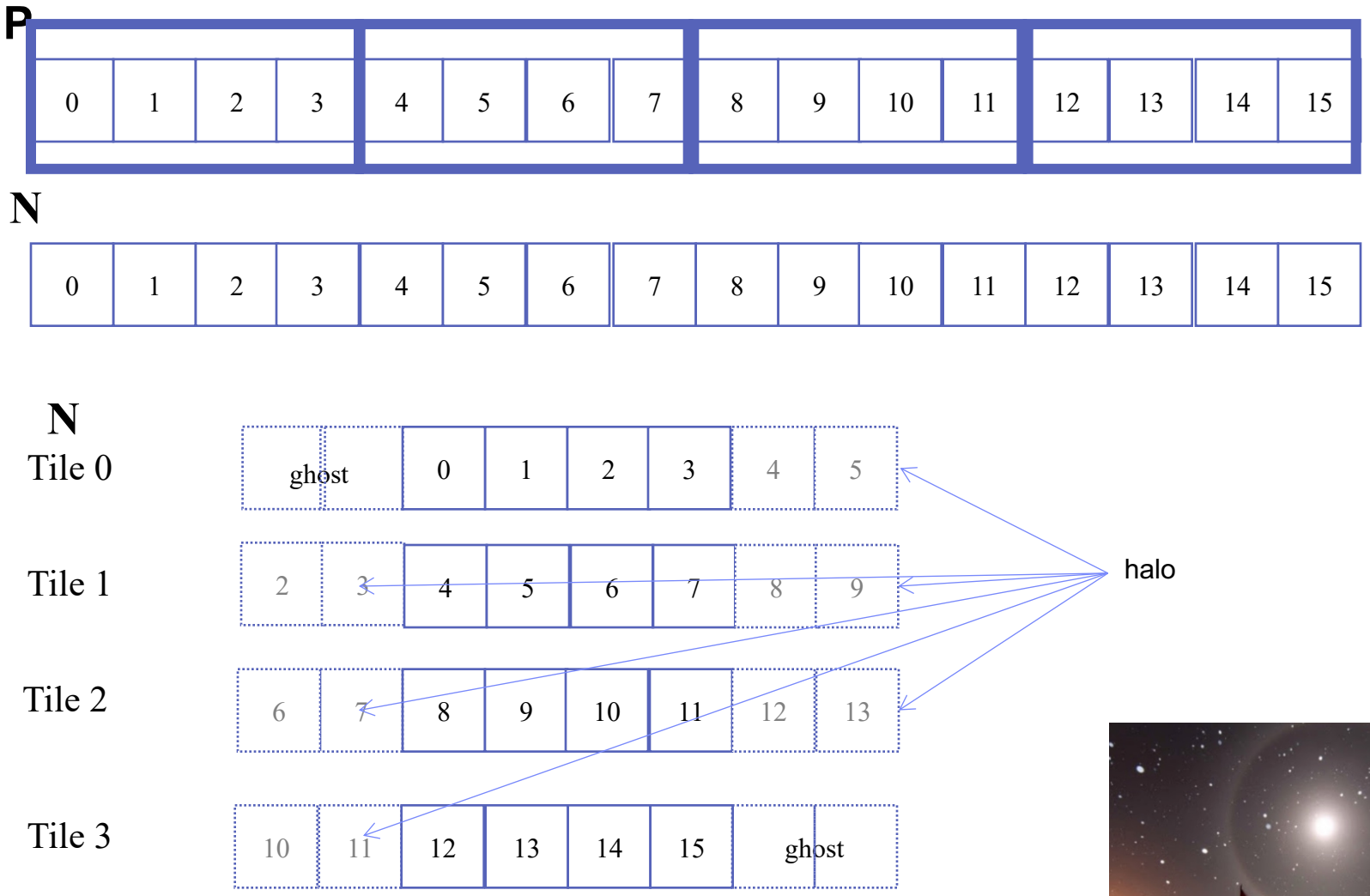
# Review: 1D Convolution

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

- Calculation of P[2]



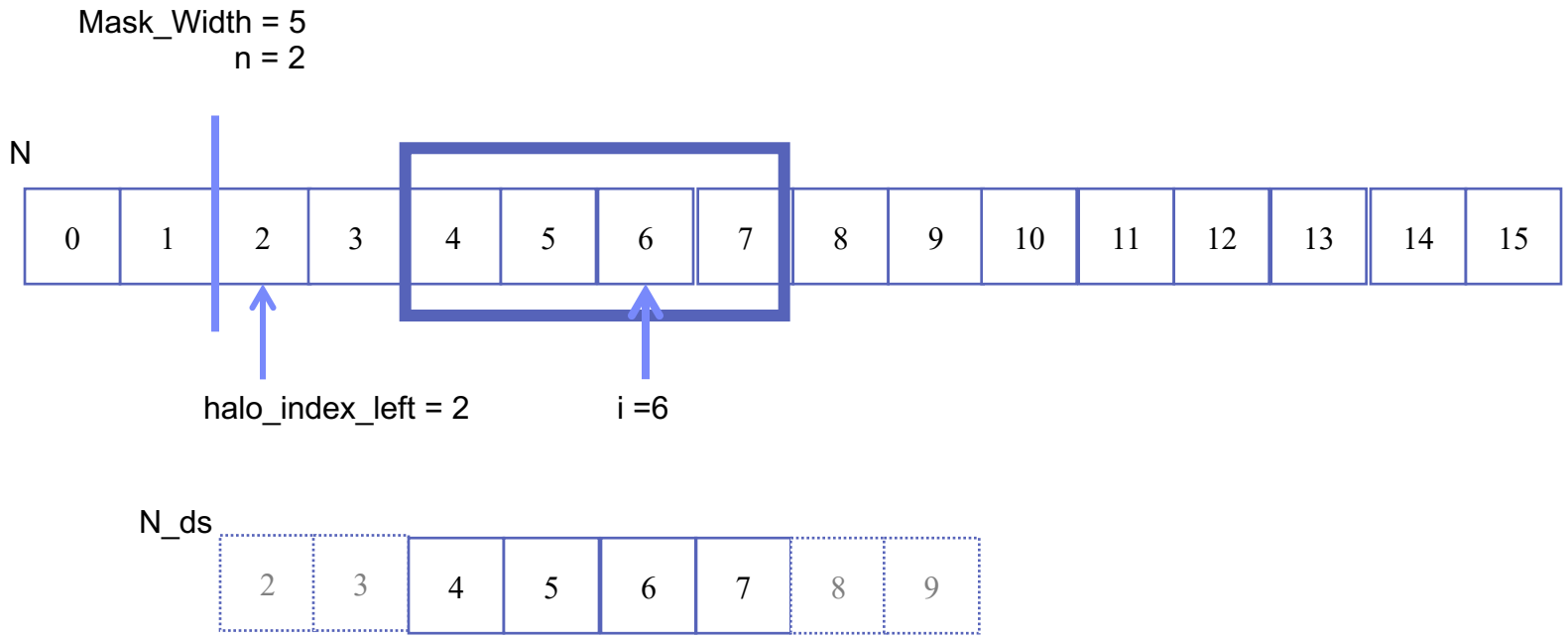
# Tiled 1D Convolution Basic Idea





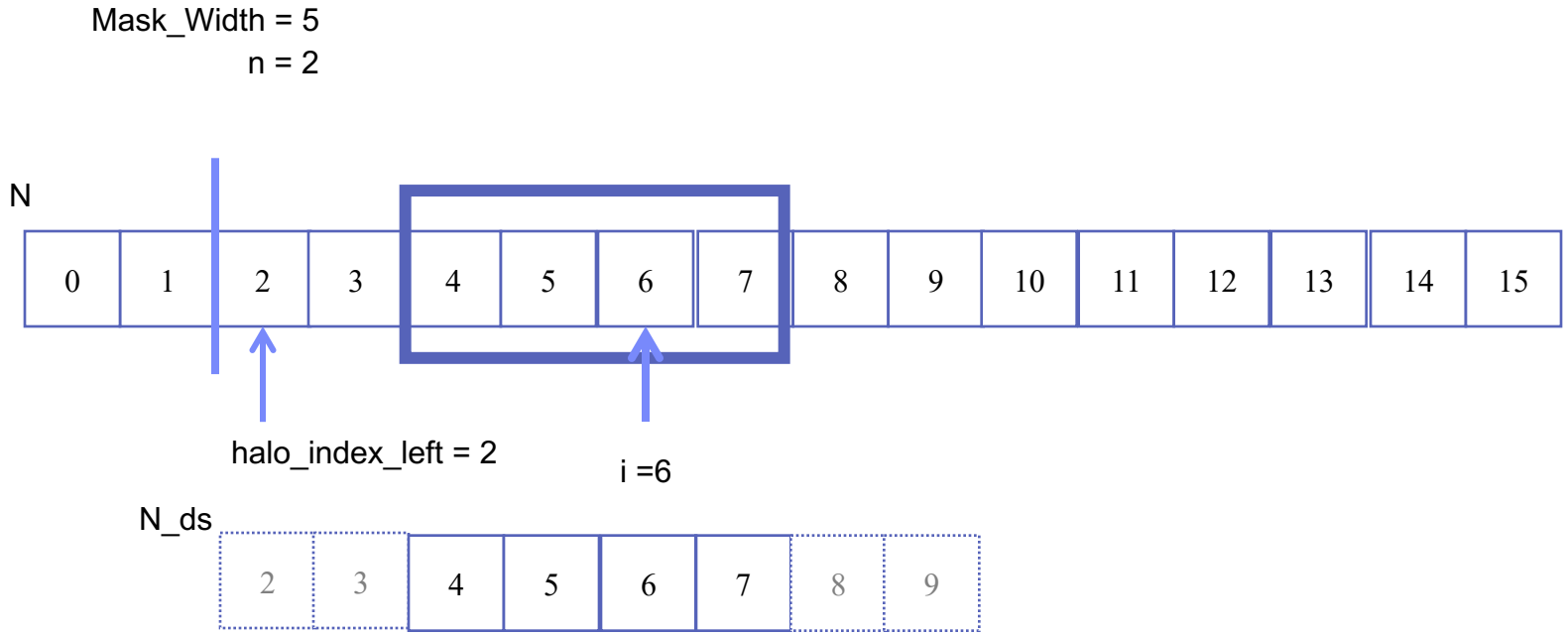

Halo: 헤일로

# Loading the left halo to the shared memory



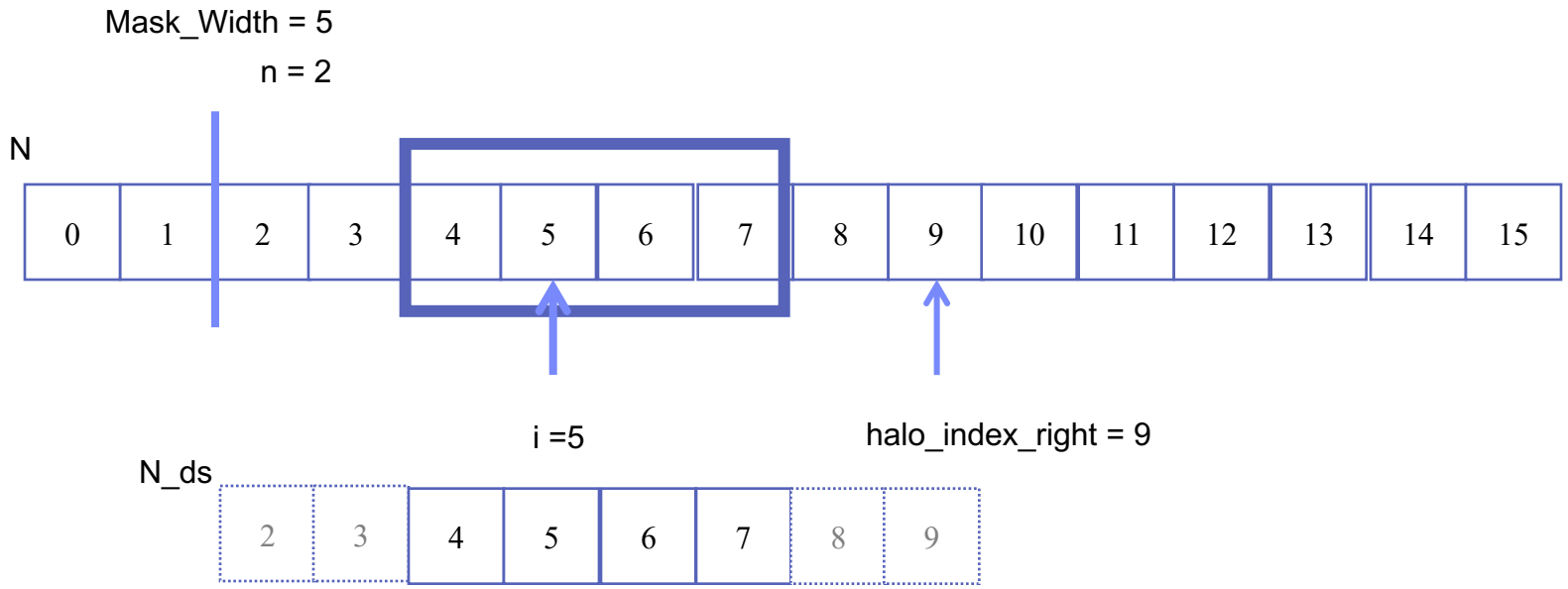

```
int n = Mask_Width/2;
int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
```

# Loading the internal elements to the shared memory




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

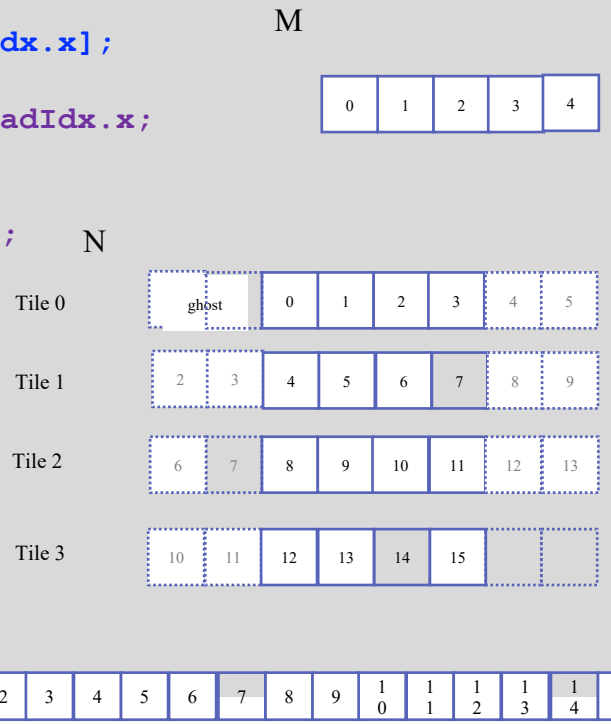
# Loading the right halo to the shared memory




```
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];
  }
```

# Tiled 1D Convolution: CUDA Kernel

```
__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;

}
```

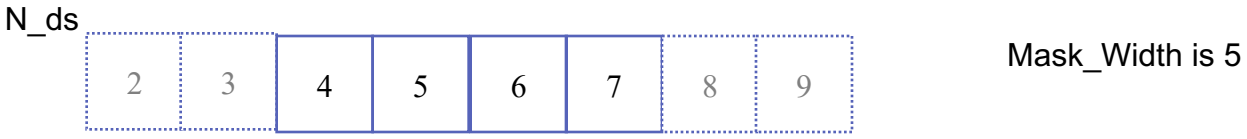


# Review: Host Code using constant memory

```
  // global variable, outside any kernel/function
  __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
main()
{
  // allocate N, P, initialize N elements, copy N to Nd
  Matrix  M;
  M  = AllocateMatrix(MASK_WIDTH, MASK_WIDTH, 1);
  // initialize M elements
 ….
  cudaMemcpyToSymbol(Mc, M.elements,
            MASK_WIDTH*MASK_WIDTH*sizeof(float));
  ConvolutionKernel<<<dimGrid, dimBlock>>>(Nd, Pd);
 ….
}
```

# Data Reuse in the Shared Memory

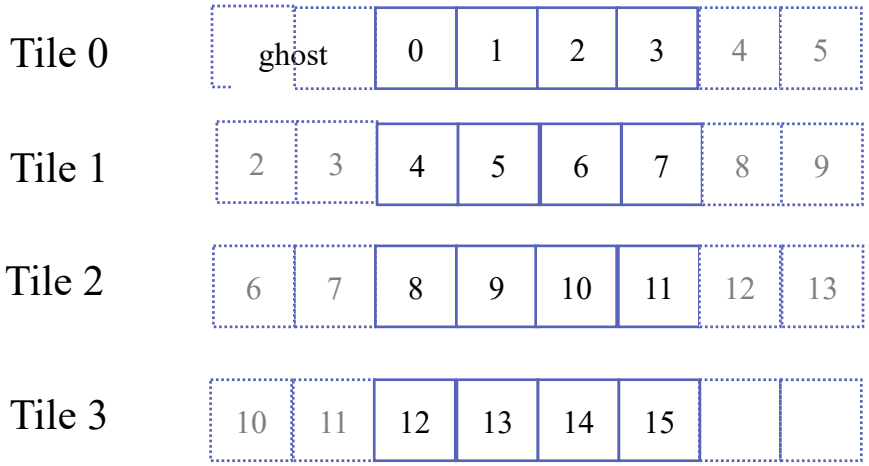
- Element 2 is used by thread 4 (1X)
- Element 3 is used by threads 4, 5 (2X)
- Element 4 is used by threads 4, 5, 6 (3X)
- Element 5 is used by threads 4, 5, 6, 7 (4X)
- Element 6 is used by threads 4, 5, 6, 7 (4X)
- Element 7 is used by threads 5, 6, 7 (3X)
- Element 8 is used by threads 6, 7 (2X)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

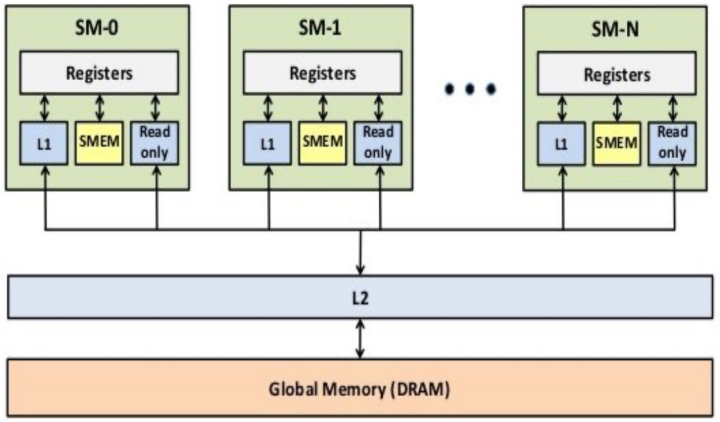


# Tiled 1D convolution with general caching

- Loading the left and right halo cells into the shared memory makes the CUDA kernel complex. ☹

- Recent GPUs provide general L1 and L2 caches
  - L1 is private to each SM
  - L2 is **shared** among all SMs
- Halo cells of a block are also interal cells of a neighboring block
- Halo cells may be available in the L2 cache
  - → might not need to load the halo cells to the shared memory



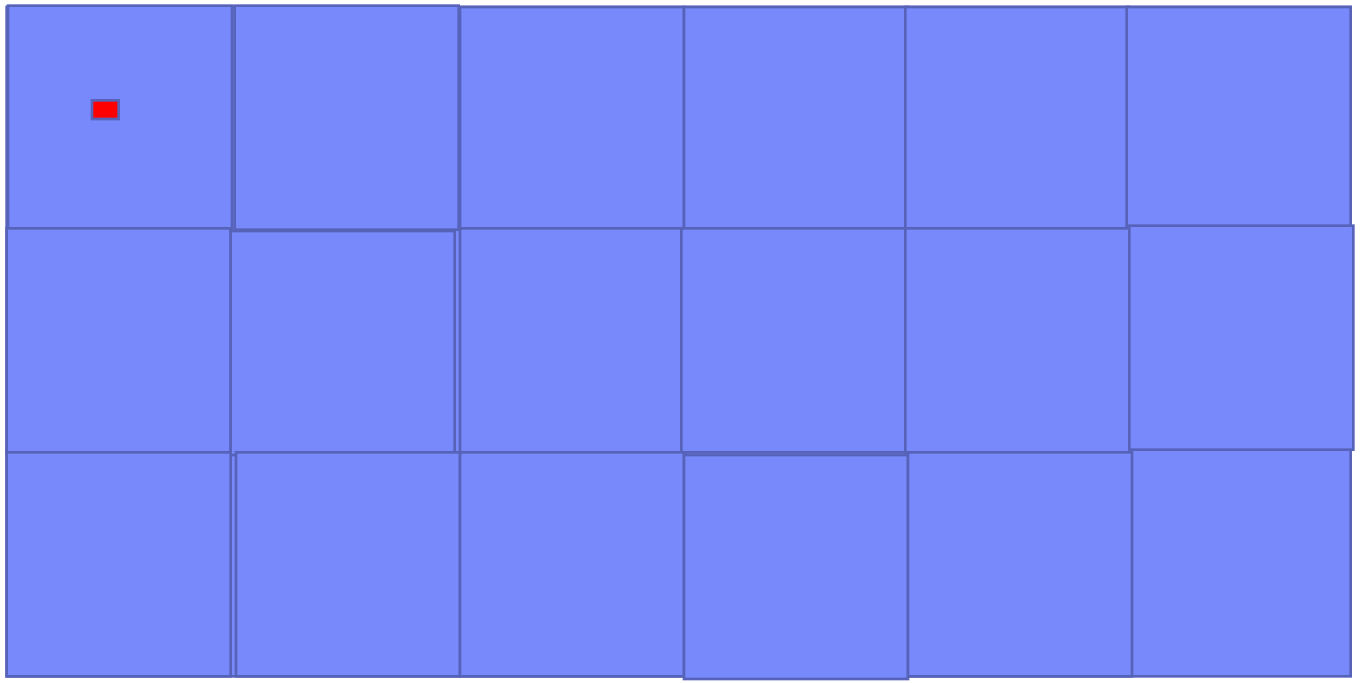



# Tiled 1D convolution with general caching

```
__global__ void convolution_1D_tiled_cache_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;
}
```

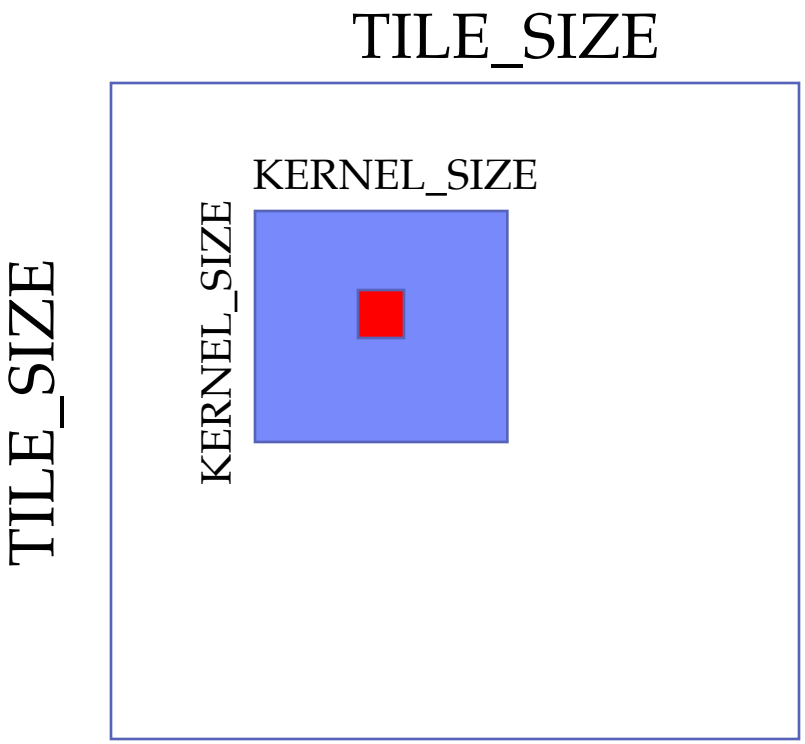
# Tiled 2D convolution

- Use a thread block to calculate a tile of P
  - Thread Block size determined by the TILE_SIZE



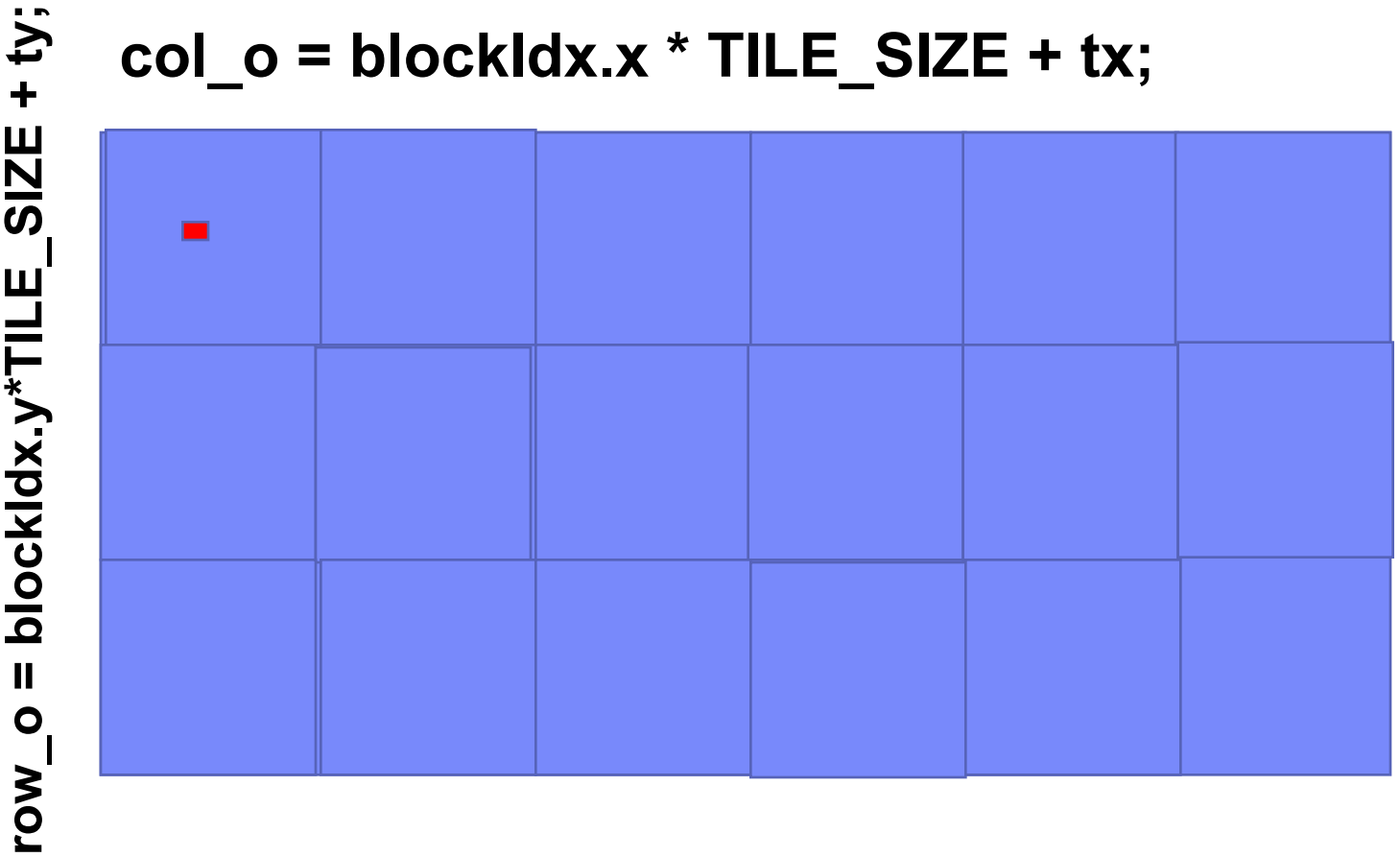
# High-Level Tiling Strategy

- Load a tile of N into shared memory (SM)
  - All threads participate in loading
  - A subset of threads use each N element in SM



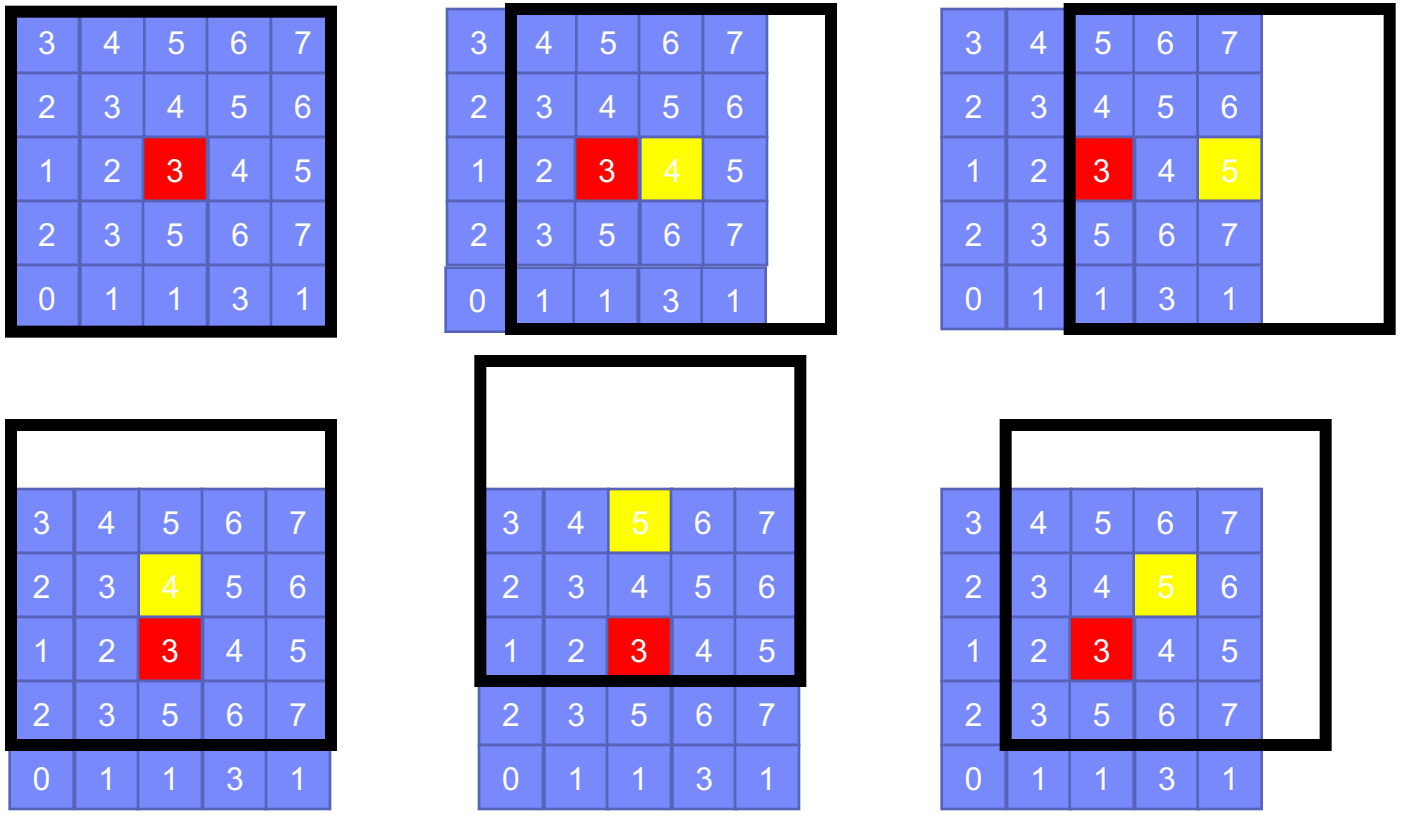
# Output Tiling and Thread Index (P)

- Use a thread block to calculate a tile of P
  - Each output tile is of TILE_SIZE for both x and y



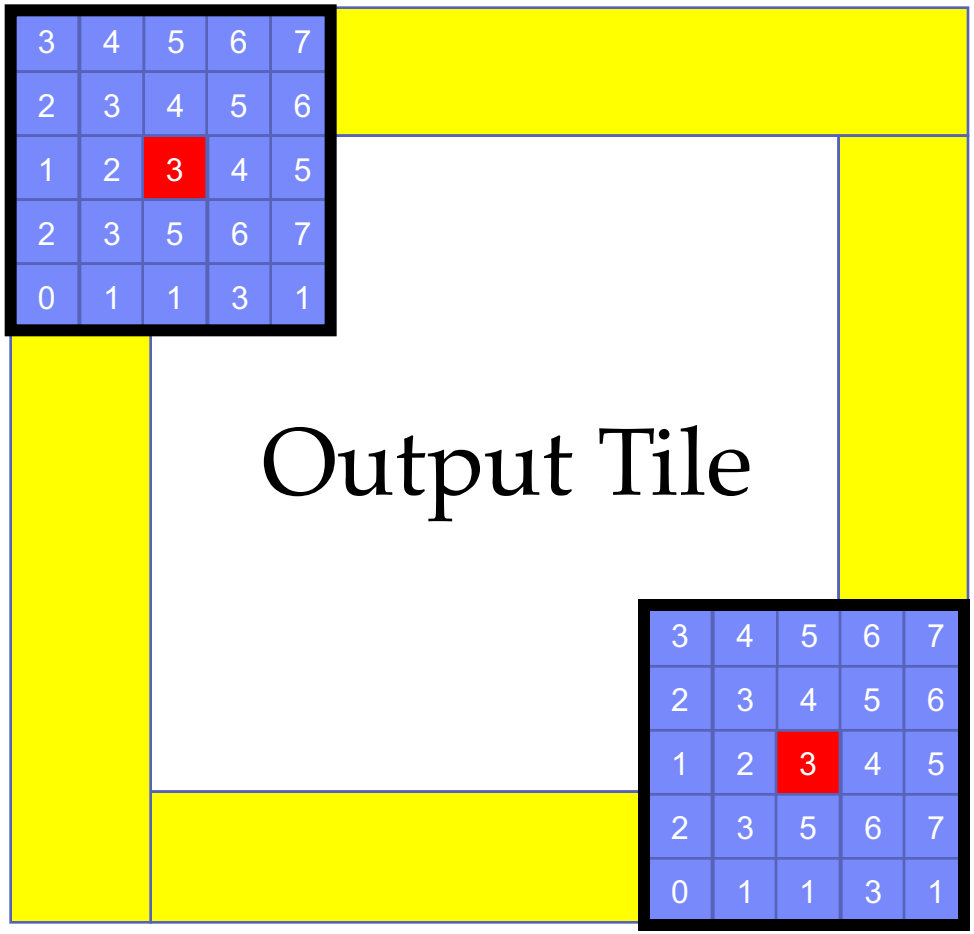
# Tiling N

- Each N element is used in calculating up to KERNEL_SIZE * KERNEL_SIZE P elements (all elements in the tile)




**Load input matrix to the shared memory!!**

# Input tiles need to be larger than output tiles.




Input Tile

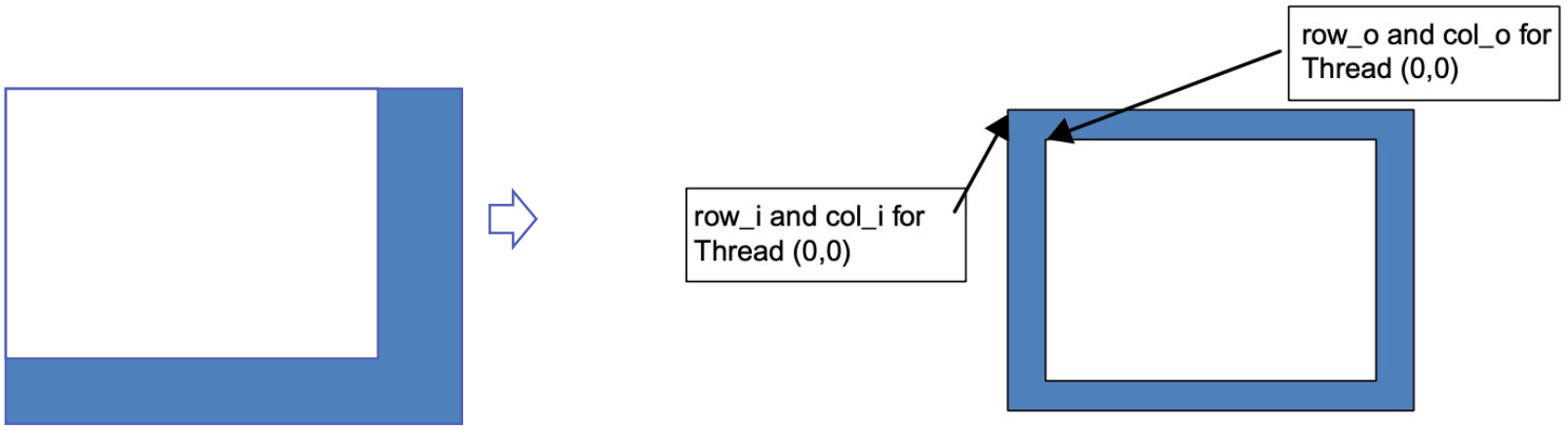
We will use a strategy where the input tile will be loaded into the shared memory.

# Dealing with Mismatch

- Use a thread block that matches input tile
  - Each thread loads one element of the input tile
  - Some threads do not participate in calculating output
    - → There will be if statements and control divergence

# Shifting from output coordinates to input coordinates

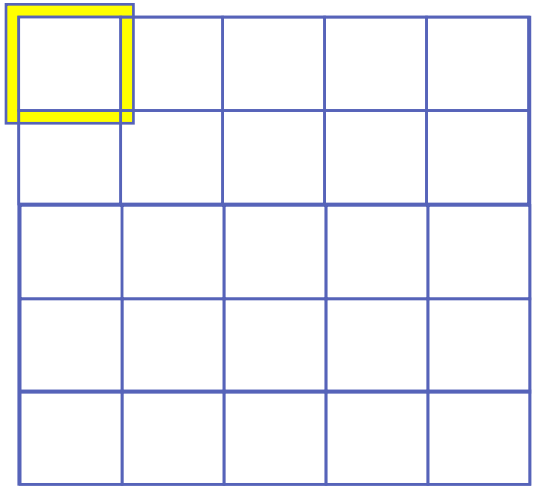
```
__global__ void convolution(Matrix N, Matrix P) {
    __shared__ float N_ds[BLOCK_SIZE][BLOCK_SIZE];
    int ty= threadIdx.y;
    int tx= threadIdx.x;

    int row_o = blockIdx.y*TILE_SIZE+ ty;
    int col_o = blockIdx.x*TILE_SIZE+ tx;
    int row_i = row_o – KERNEL_SIZE/2;
    int col_i = col_o – KERNEL_SIZE/2;
    float output = 0.0f;
```



# Loading input matrix to the shared memory

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

__syncthreads();
```

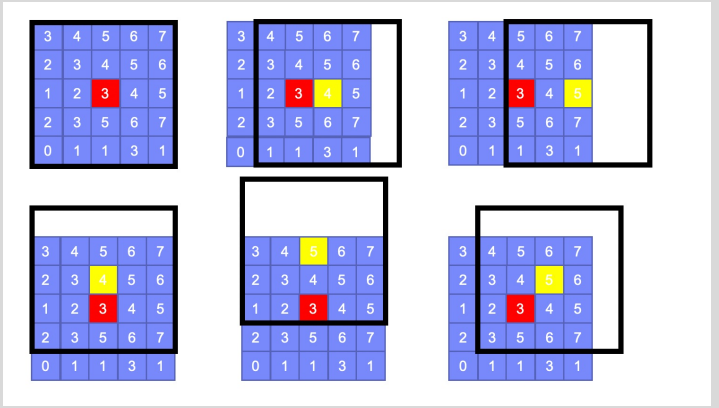
- Threads that loads ghost should return 0.0



# Do convolution!

- Some threads do not participate in calculating output.

```
if(ty < TILE_SIZE && tx < TILE_SIZE){
    for(i = 0; i < KERNEL_SIZE; i++) {
      for(j = 0; j < KERNEL_SIZE; j++) {
        output += Mc[i][j] * Ns[i+ty][j+tx];
       }
    }

   if(row_o < P.height && col_o < P.width)
      P.elements[row_o * P.width + col_o] = output;
  }
}
```



# Setting Block Size

- In general, block size should be

  TILE_SIZE + (KERNEL_SIZE -1)

- **If kernel size = 5,**

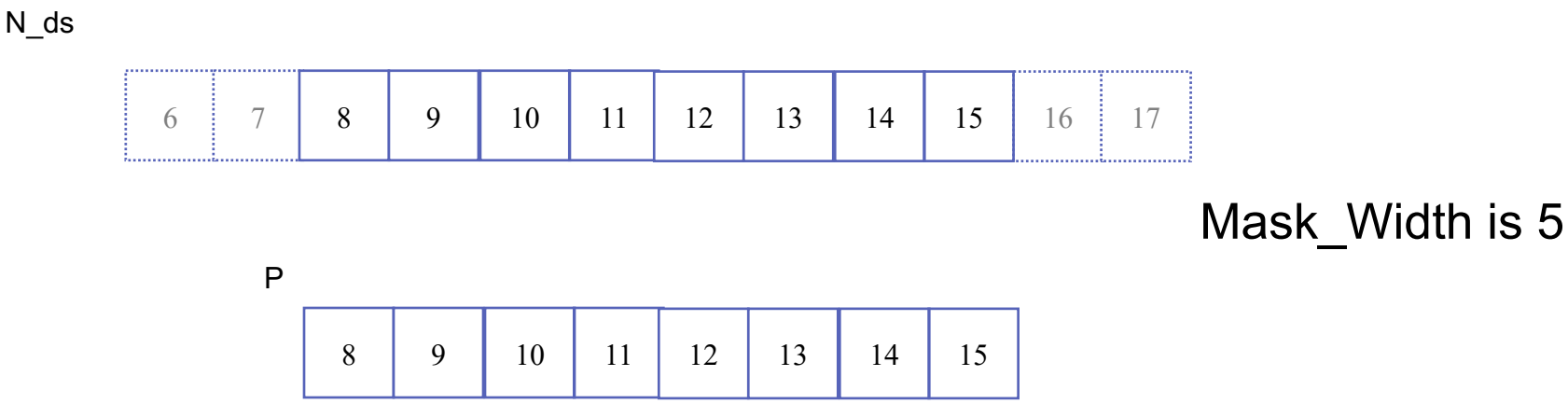
**#define BLOCK_SIZE (TILE_SIZE + 4)**

```
dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
dim3 dimGrid(ceil(N.width/TILE_SIZE), ceil(N.height/TILE_SIZE), 1);

convolution<<<dimGrid, dimBlock>>>(N_d, P_d);
```

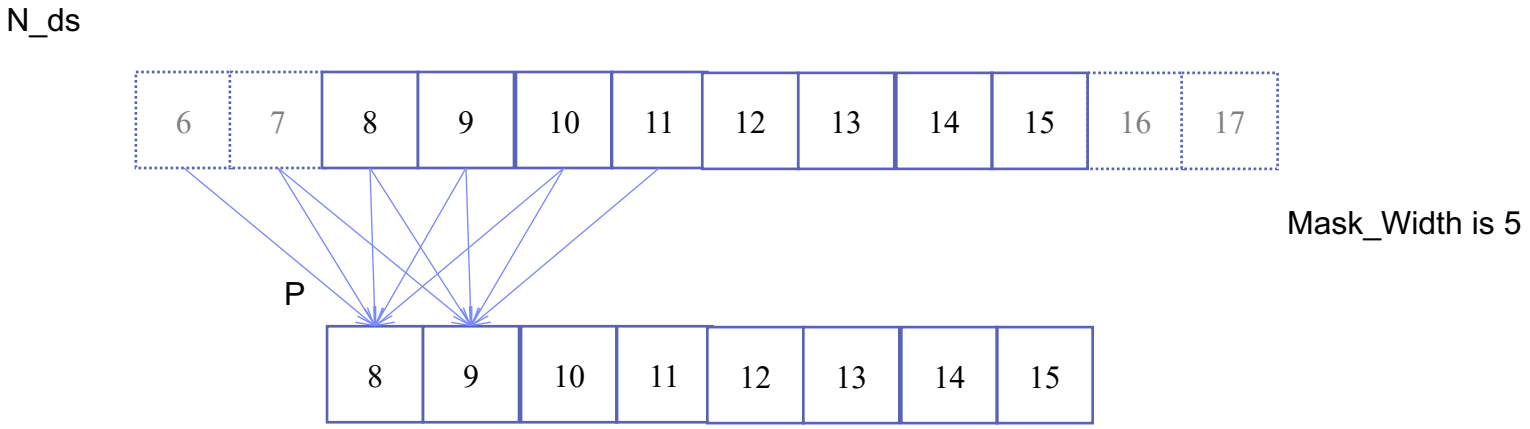
# Tiled Convolution Analysis

# A Small 1D Example: TILE_SIZE = 8, Mask_Width=5

- output and input tiles for block 1
- For Mask_Width = 5, each block loads 8+5-1 = 12 elements (12 memory loads)



# A Small 1D Example: TILE_SIZE = 8, Mask_Width=5




- Each output P element uses 5 N elements (in N_ds)
  - 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 Total of 8 * 5 elements are used for the output tile.

# A simple way to calculate 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

- **In General, for 1D**
  - TILE_SIZE + Mask_Width -1 elements loaded
  - TILE_SIZE * Mask_Width global memory accesses replaced by shared memory access
  - This gives a reduction of bandwidth by

    (TILE_SIZE *Mask_Width)/(TILE_SIZE+Mask_Width-1)

# Bandwidth Reduction for 1D

- **The reduction is**
  - Mask_Width * (TILE_SIZE)/(TILE_SIZE+Mask_Width-1)

| TILE_SIZE | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|
| Reduction Mask_Width = 5 | 4.0 | 4.4 | 4.7 | 4.9 | 4.9 |
| Reduction Mask_Width = 9 | 6.0 | 7.2 | 8.0 | 8.5 | 8.7 |