ECE408/CS483/CSE408 Fall 2022

Applied Parallel Programming

Lecture 8: Tiled Convolution

Course Reminders

- Lab updates
  - Lab 3 is due this Friday at 8pm CT
  - Lab 4 will be posted on Friday

- Midterm 1 is on Tuesday, October 11th
  - On-line, everybody will be taking it at the same time
    - Will be scheduled for the evening to overlap with ZJUI

Creating Technology that has Impact

ageia

2004-2008
Acquired by Nvidia

Notable Alum:
Chris Lamb

PERSONIFY

2011-2018
Acquired by Foxconn

Founding Team:
Quang Nguyen
Dennis Lin
Simon Venshtain

DigitalGemini
Data to Drive the Future

2019-2022
Acquired by Uhnder

Founding Team:
Henry Haase
Yuan Zhang
Yumai Sun
Ke Sun

ROKMETRO

2020-

Founding Team:
John Paul
Stephen Hurwitt
Siqi Zhang
Ryan Oberlander

Trova

2020-

Founding Team:
Yucheng Liang
Chak Ho Chan
Shuchen Zhang

A
Alchemy

3

Objective

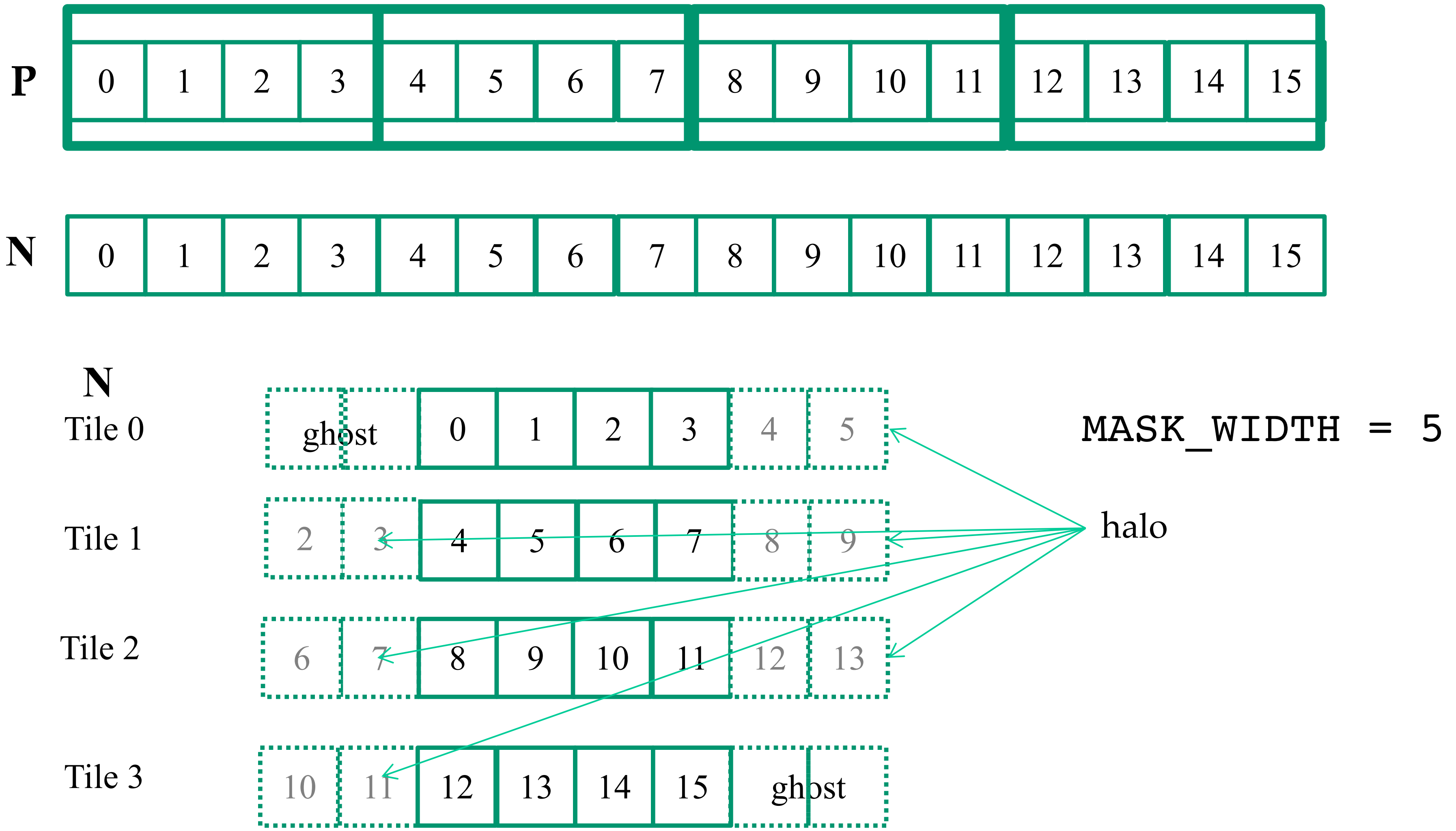
- To learn about tiled convolution algorithms
  - Some intricate aspects of tiling algorithms
  - Output tiles versus input tiles
  - Three different styles of input tile loading
  - To prepare for Lab 4

Are we memory limited?

For the 1D case, every output element requires 2*`MASK_WIDTH` loads (of M and N each) and 2*`MASK_WIDTH` floating point operations. **Memory limited.**

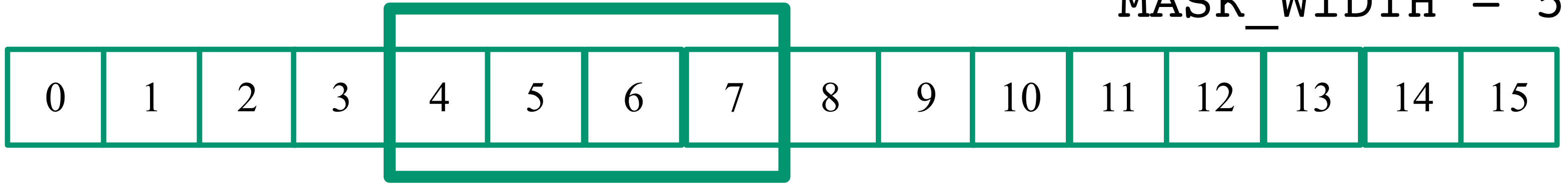
For the 2D case, every output element requires 2*`MASK_WIDTH`^{2} loads and 2*`MASK_WIDTH`^{2} floating point operations. **Memory limited.**

Tiled 1D Convolution Basic Idea



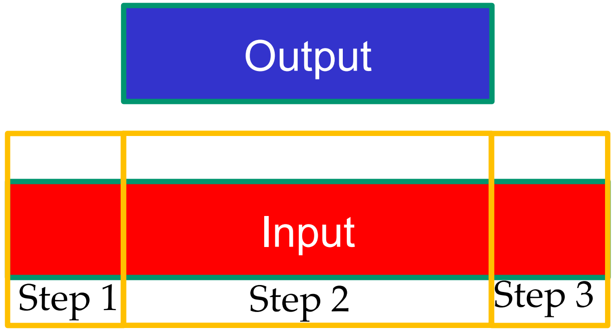
How Much Reuse is there? Consider Tile 1

`MASK_WIDTH = 5`



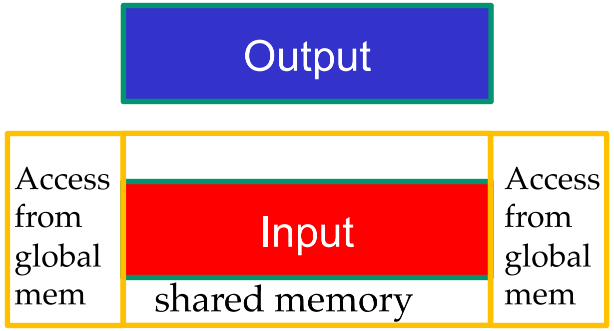
- Element 2 is used to calculate output 4 (1×)
- Element 3 is used to calculate outputs 4, 5 (2×)
- Element 4 is used to calculate outputs 4, 5, 6 (3×)
- Element 5 is used to calculate outputs 4, 5, 6, 7 (4×)
- Element 6 is used to calculate outputs 4, 5, 6, 7 (4×)
- Element 7 is used to calculate outputs 5, 6, 7 (3×)
- Element 8 is used to calculate outputs 6, 7 (2×)
- Element 9 is used to calculate outputs 7 (1×)

Three Tiling Strategies



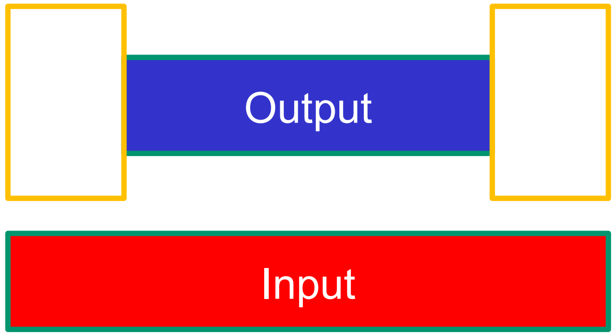
Strategy 1

1. Block size covers **output** tile
2. Use multiple steps to load input tile



Strategy 3

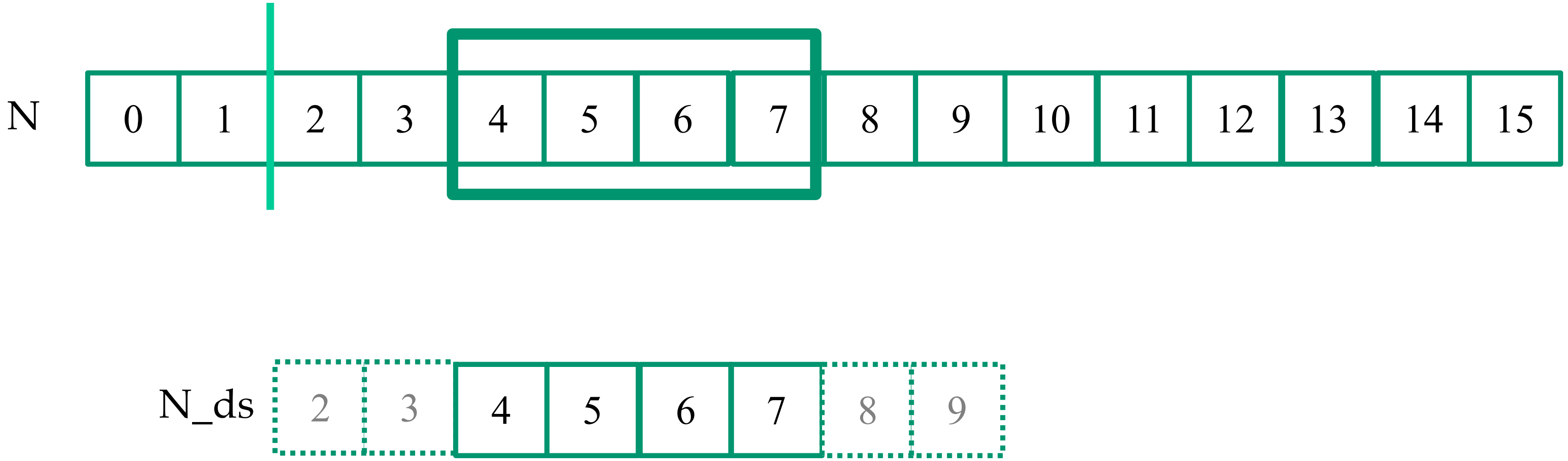
1. Block size covers **output** tile
2. Load only “core” of input tile
3. Access halo cells from global memory



Strategy 2

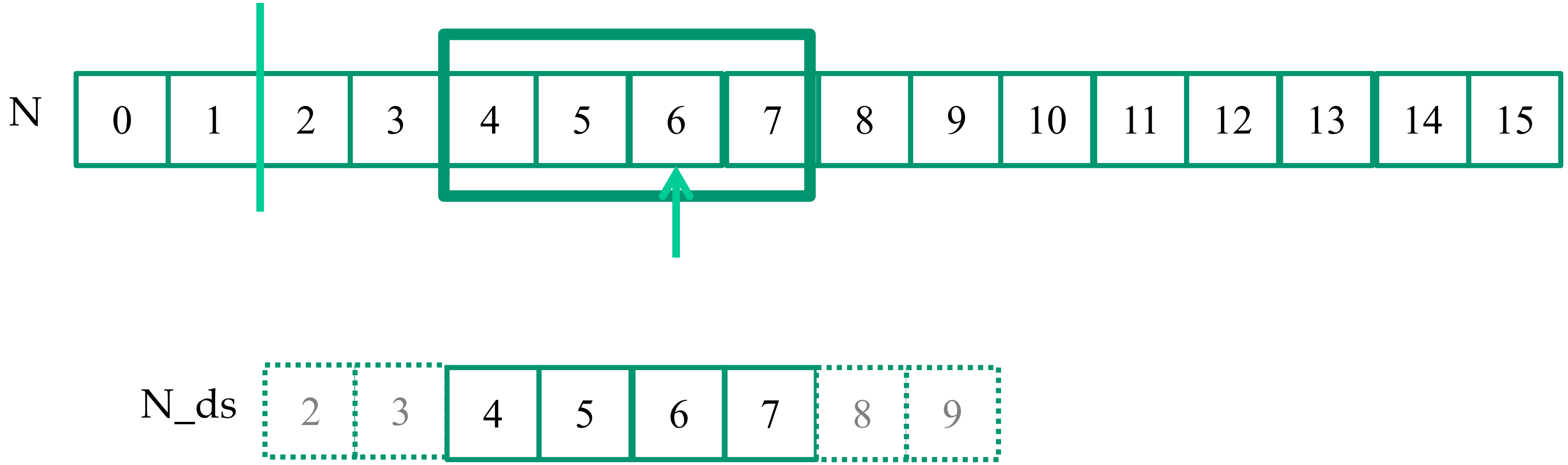
1. Block size covers **input** tile
2. Load input tile in one step
3. Turn off some threads when calculating output

Loading the left halo



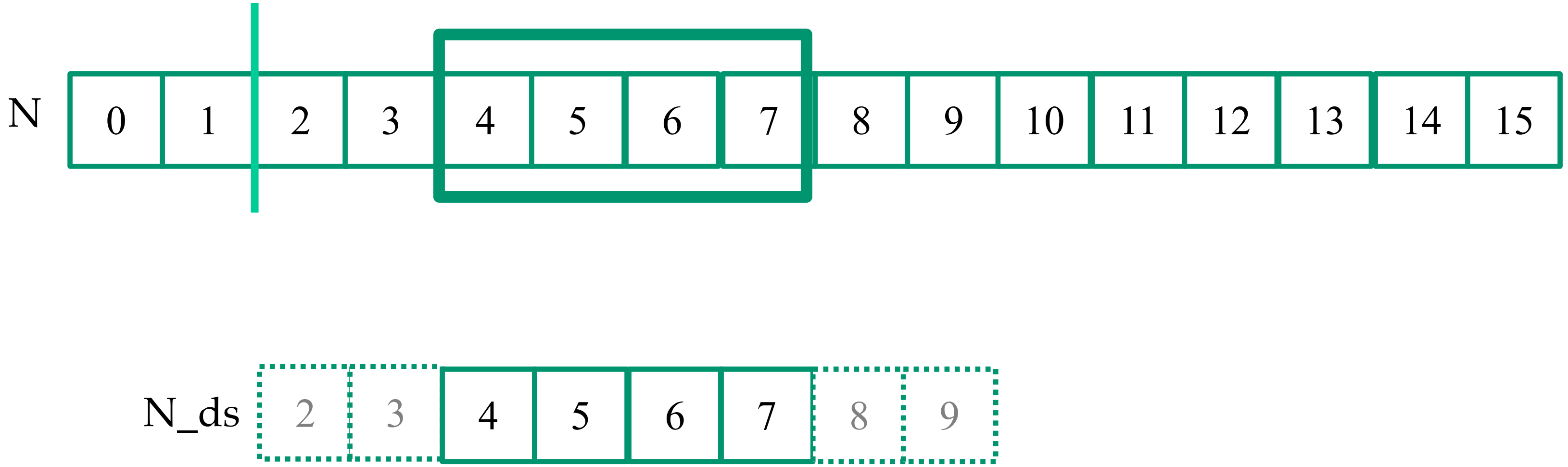

```
__shared__  float  N_ds[TILE_SIZE + MASK_WIDTH - 1];
int radius = Mask_Width / 2;
int halo_index_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;

if (threadIdx.x >= (blockDim.x - radius)) {
   if (halo_index_left < 0)
      N_ds[threadIdx.x - (blockDim.x - radius)] = 0;
   else
      N_ds[threadIdx.x - (blockDim.x - radius)] = N[halo_index_left];
}
```

Loading the internal elements




```
if ((blockIdx.x * blockDim.x + threadIdx.x) < Width)
  N_ds[radius + threadIdx.x] = N[blockIdx.x * blockDim.x + threadIdx.x];
else
  N_ds[radius + threadIdx.x] = 0.0f;
```

Loading the right halo

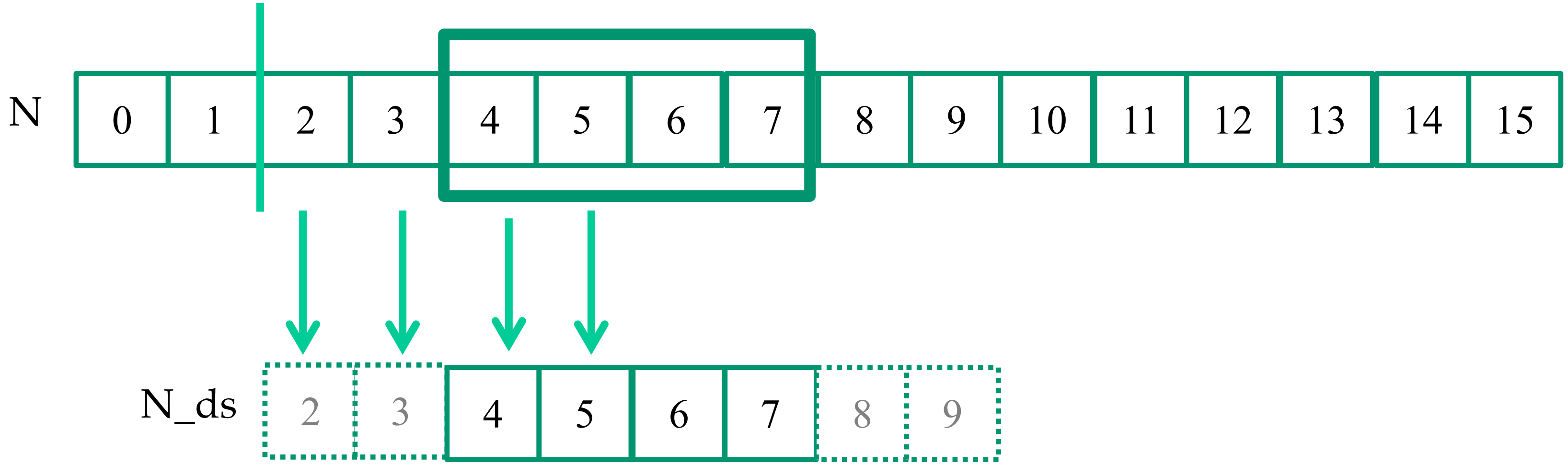



```
int halo_index_right = (blockIdx.x + 1) * blockDim.x + threadIdx.x;

if (threadIdx.x < radius) {
    if (halo_index_right >= Width)
      N_ds[radius + blockDim.x + threadIdx.x] = 0;
    else
      N_ds[radius + blockDim.x + threadIdx.x] = N[halo_index_right];
}
```

```
__global__ void convolution_1D(float *N, float *P, float *M, int Width) {

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

  int radius = Mask_Width / 2;

  int halo_index_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
  if (threadIdx.x >= (blockDim.x - radius)) {
     if (halo_index_left < 0)
        N_ds[threadIdx.x - (blockDim.x - radius)] = 0.0f;
     else
        N_ds[threadIdx.x - (blockDim.x - radius)] = N[halo_index_left];
  }

  if ((blockIdx.x * blockDim.x + threadIdx.x) < Width)
     N_ds[radius + threadIdx.x] = N[blockIdx.x * blockDim.x + threadIdx.x];
  else
     N_ds[radius + threadIdx.x] = 0.0f;

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

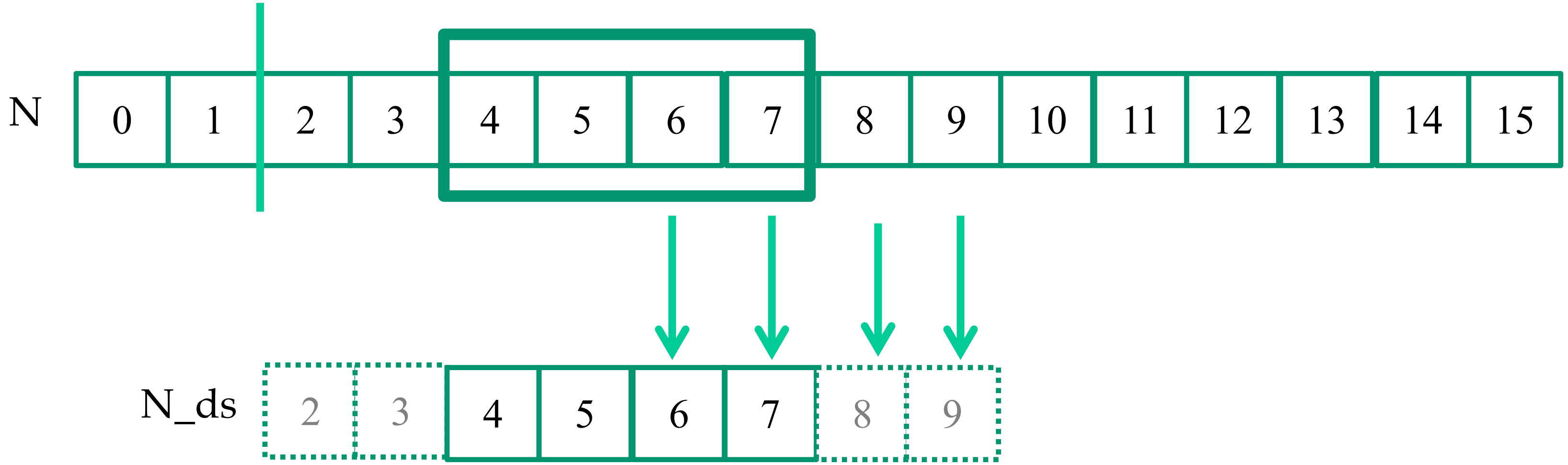
Strategy 1

Load the Input Data – step 1




```
int start = i - radius;
if (0 <= start && Width > start) {        // all threads
  N_ds[threadIdx.x] = N[start];
} else {
  N_ds[threadIdx.x] = 0.0f;
}
```

Load the Input Data – step 2




```
if (threadIdx.x < (MASK_WIDTH - 1)) {    // some threads
  start += TILE_SIZE;
  if (Width > start) {
    N_ds[threadIdx.x + TILE_SIZE] = N[start];
  } else {
    N_ds[threadIdx.x + TILE_SIZE] = 0.0f;
  }
}
```

```
__global__ void convolution_1D(float *N, float *P, float *M, int Width) {

  int I = blockIdx.x * blockDim.x + threadIdx.x;
  __shared__ float N_ds[TILE_SIZE + MASK_WIDTH - 1];
  int radius = MASK_WIDTH / 2;
  int start = i - radius;

  if (0 <= start && Width > start) {        // all threads
    N_ds[threadIdx.x] = N[start];
  else
    N_ds[threadIdx.x] = 0.0f;

  if (threadIdx.x < (MASK_WIDTH - 1)) {    // some threads
    start += TILE_SIZE;
    if (Width > start) {
      N_ds[threadIdx.x + TILE_SIZE] = N[start];
    else
      N_ds[threadIdx.x + TILE_SIZE] = 0.0f;
  }

  __syncthreads();

  float Pvalue = 0.0f;
  for (int j = 0; MASK_WIDTH > j; j++) {
    Pvalue += N_ds[threadIdx.x + j] * Mc[j];
  }
  P[i] = Pvalue;
}
```

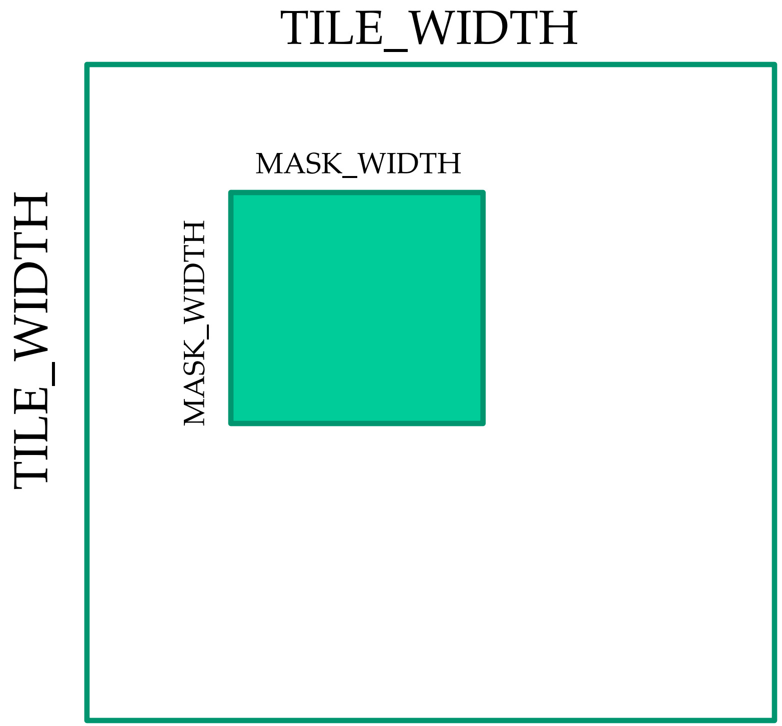
Alt. Strategy 1

```
__global__ void convolution_1D(float *N, float *P, float *M, int Width) {

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

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

  __syncthreads();

  int radius = MASK_WIDTH / 2;
  int This_tile_start_point = blockIdx.x * blockDim.x;
  int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
  int N_start_point = i - radius;
  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-radius+j] * M[j];
       else
         Pvalue += N[N_index] * M[j];
     }
  }
  P[i] = Pvalue;
}
```

Strategy 3

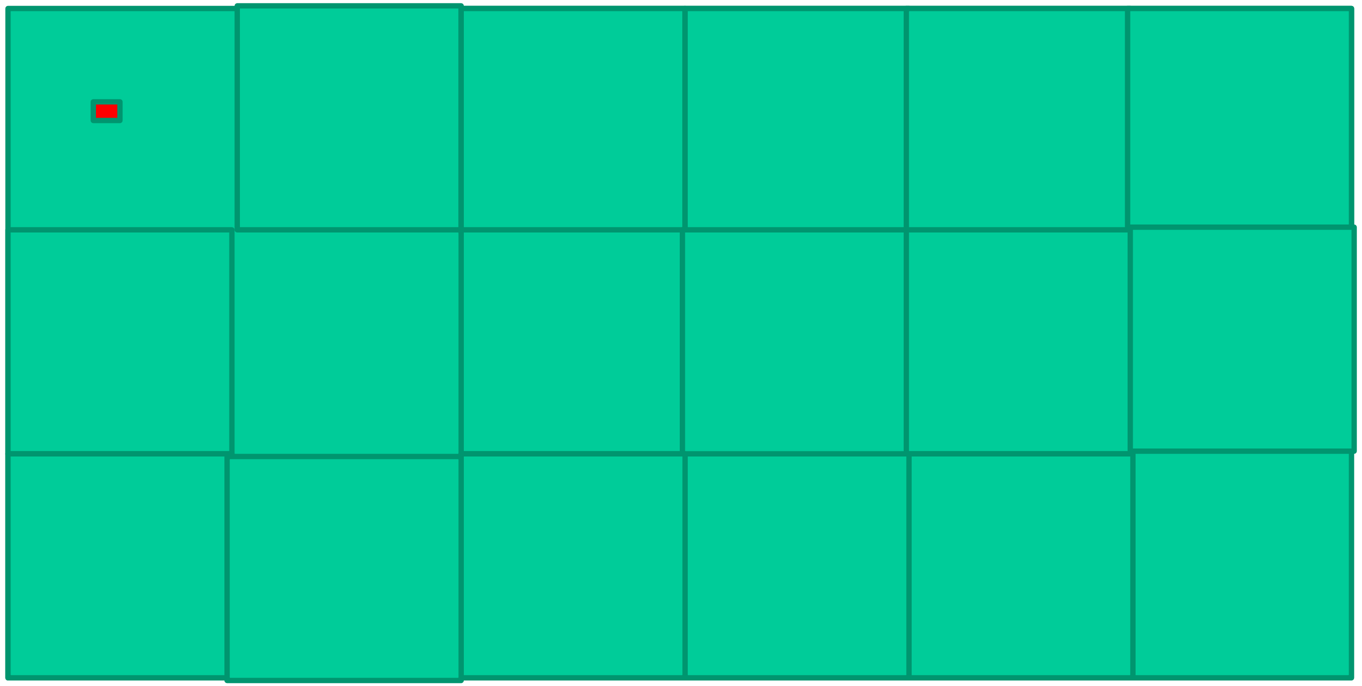
Strategy 2 in 2D

- All threads participate in loading of N into Shared Memory
- A **subset of threads** then calculate P

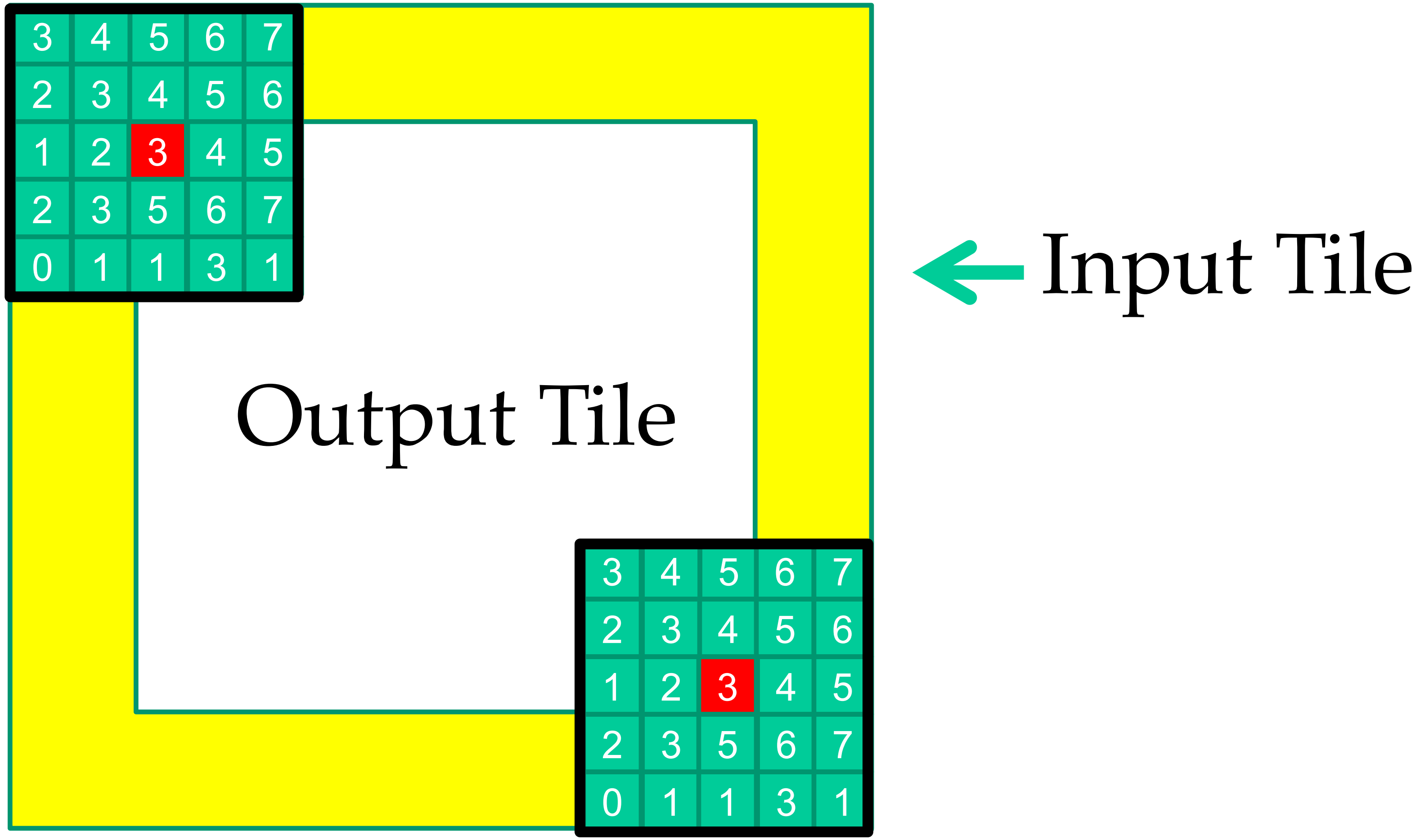


Strategy 2 for 2D Convolution Thread Index and Output Index

```
row_o = blockIdx.y * TILE_WIDTH + threadIdx.y;
col_o = blockIdx.x * TILE_WIDTH + threadIdx.x;
```



Input tiles need to be larger than output tiles.



Setting Grid/Block Dimensions

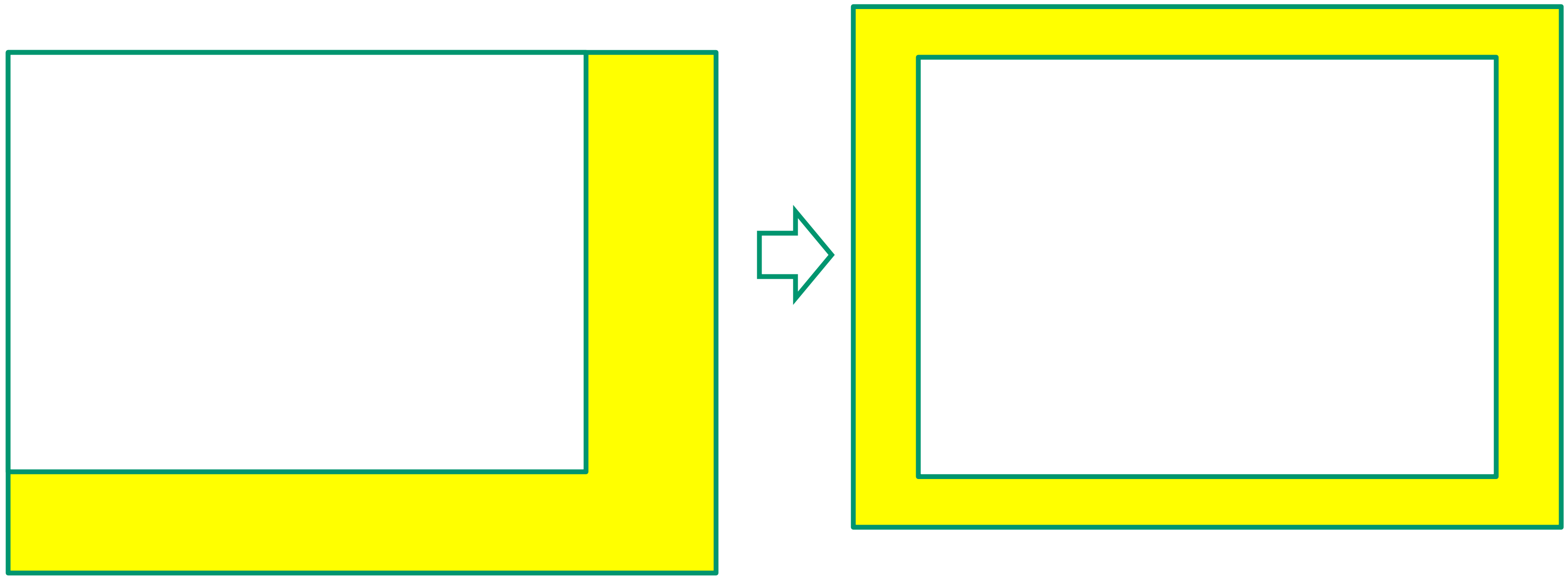
```
dim3 dimGrid(ceil(P.width/(1.0*TILE_WIDTH)),
             ceil(P.height/(1.0*TILE_WIDTH)), 1)
```

There need to be enough blocks to generate all P elements.

```
dim3 dimBlock(TILE_WIDTH + MASK_WIDTH - 1,
              TILE_WIDTH + MASK_WIDTH - 1, 1);
```

There need to be enough threads to load entire tile of input.

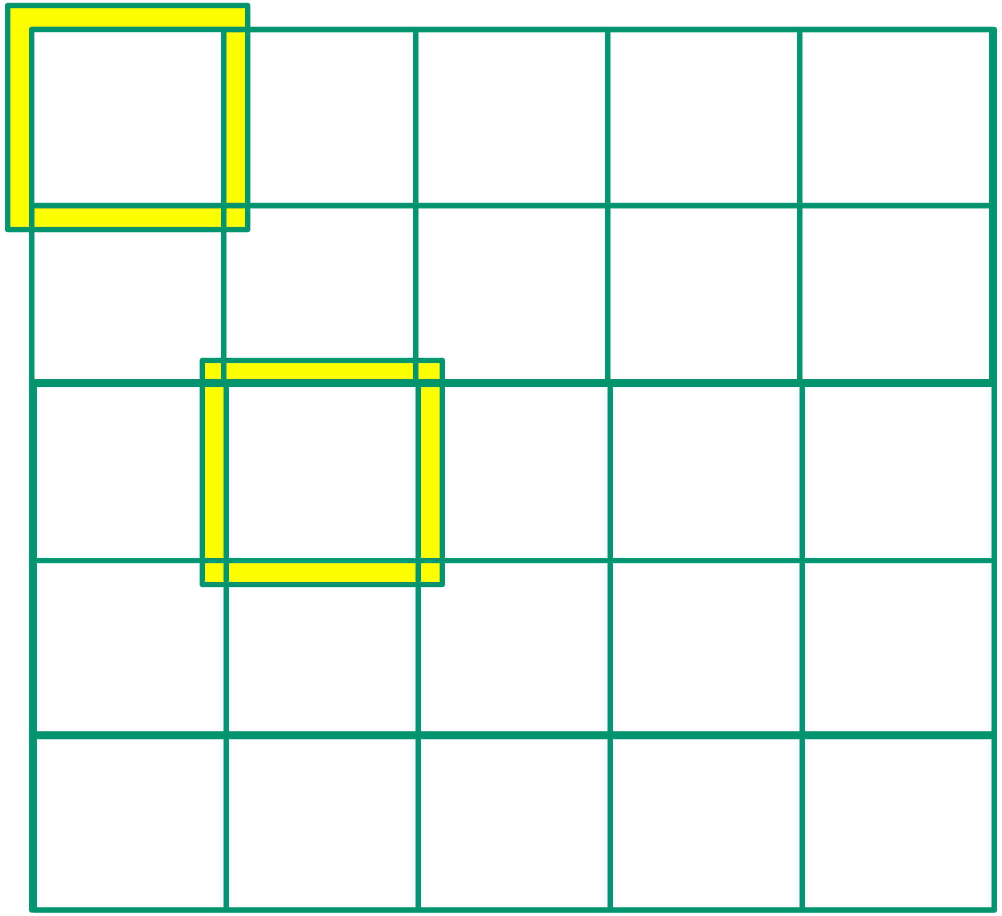
Shifting from output coordinates to input coordinates



Shifting from output coordinates to input coordinates

```
int tx = threadIdx.x;
int ty = threadIdx.y;

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

int row_i = row_o - (MASK_WIDTH / 2);
int col_i = col_o - (MASK_WIDTH / 2);
```

Threads that loads halos outside N should return 0.0



Taking Care of Boundaries

```
float Pvalue = 0.0f;
if((row_i >= 0) && (row_i < Width) &&
    (col_i >= 0)  && (col_i < Width))
  tile[ty][tx] = N[row_i*Width + col_i];
else
  tile[ty][tx] = 0.0f;

__syncthreads (); // wait for tile
```

Not All Threads Calculate Output

```
if(ty < TILE_WIDTH && tx <TILE_WIDTH){
  for(i = 0; i < MASK_WIDTH; i++)
    for(j = 0; j < MASK_WIDTH; j++)
      Pvalue += Mc[i][j] * tile[i+ty][j+tx];

  if(row_o < Width && col_o < Width)
     P[row_o * Width + col_o] = Pvalue;
  }
}
```

Alternatively

- You can extend the1D strategy 3 tiled convolution into a 2D strategy 3 tiled convolution.
  – Each input tile matches its corresponding output tile
  – All halo elements will be loaded from global memory
  – But… control divergence within inner product computation

ANY MORE QUESTIONS?
READ CHAPTER 7