ECE408/CS483/CSE408 Fall 2021

Applied Parallel Programming

Lecture 8: Tiled Convolution

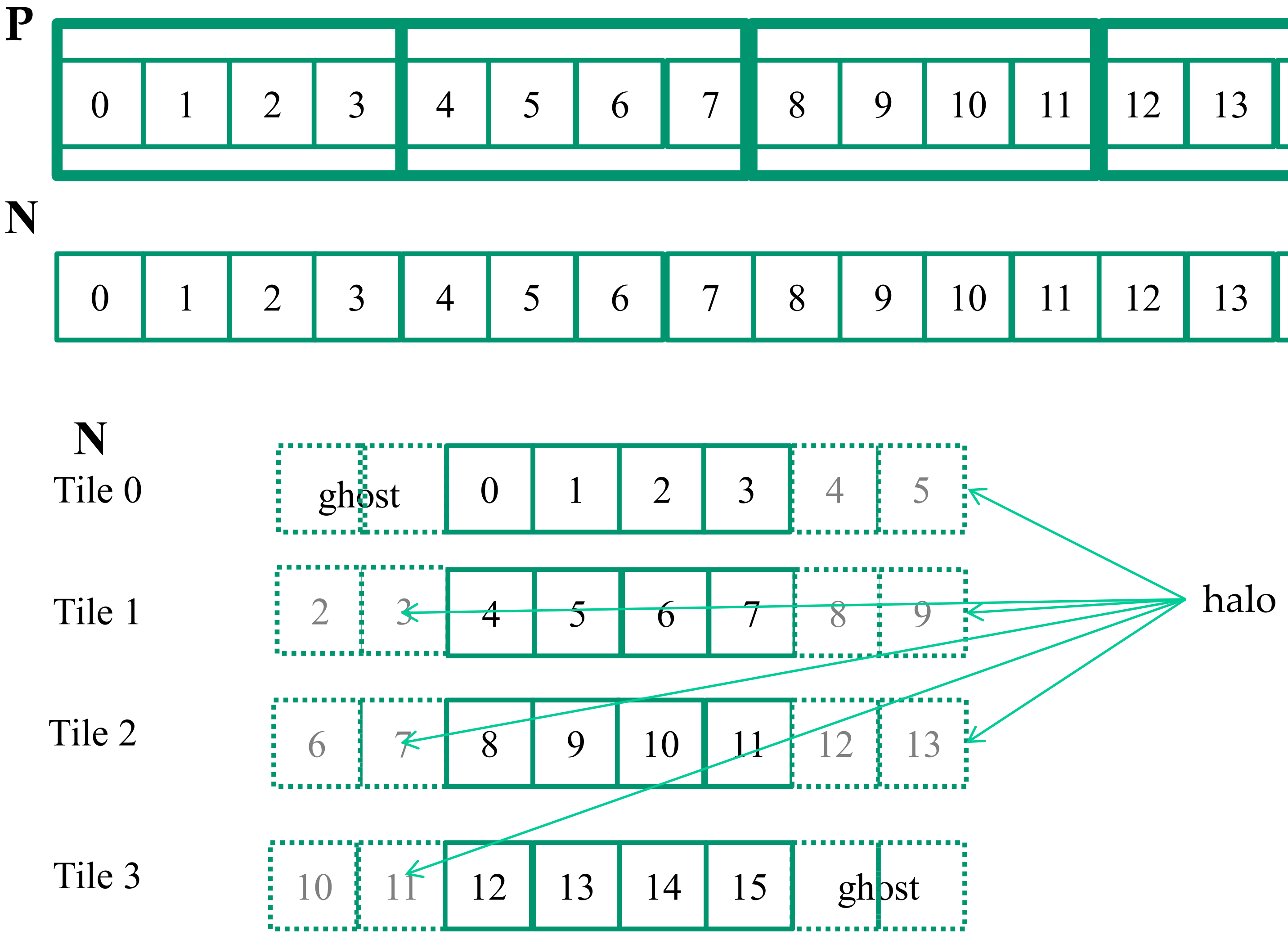
Course Reminders

- Lab 3 is due this Friday
    - Make sure to hit “Submit Attempt for Grading” when done! We still see students failing to properly submit the work :(
- Lab 4 out, it is due next week
- Midterm 1 is on Thursday, October 7th
    - On-line, everybody will be taking it at the same time
        - Thursday, Oct. 7th 8:00pm-9:20pm US Central time
        - Friday, Oct. 9th 9:00am-10:20am Beijing time

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

Tiled 1D Convolution Basic Idea



What Shall We Parallelize?

In other words,

What should one thread do?

One answer:

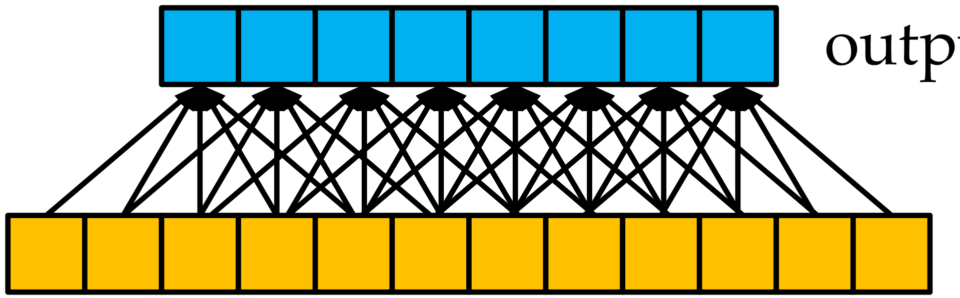
- (same as with vector sum and matrix multiply)
- **compute an output element!**

Should We Use Shared Memory?

In other words,

Can we reuse data read from global memory?

Let's look at the computation again…



Reuse reduces global memory bandwidth,
so **let's use shared memory**.

How Much Reuse is Possible?

MASK_WIDTH is 5

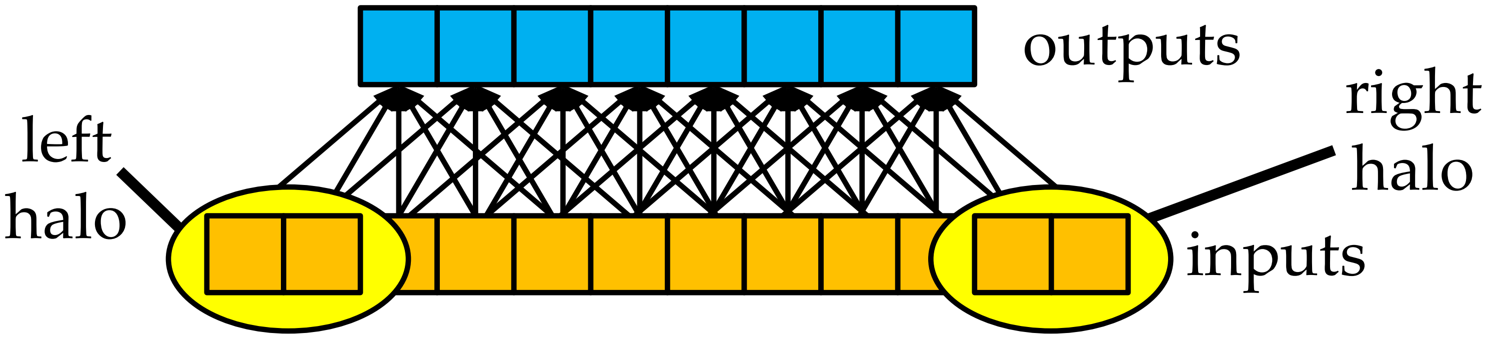
tile	2	3	4	5	6	7	8	9

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

What About the Halos?

In other words,

Do we also copy halos into shared memory?



Let's **consider both** possible answers.

Can Access Halo from Global Memory

Approach:

- threads **read halo values**
- directly **from global memory**.

Advantage:

- optimize reuse of shared memory
- (halo reuse is smaller).

Disadvantages:

- **Branch divergence**! (shared vs. global reads)
- Halo **too narrow to fill** a memory **burst**

Can Load Halo to Shared Memory

Approach:

- **load halos to shared memory**.

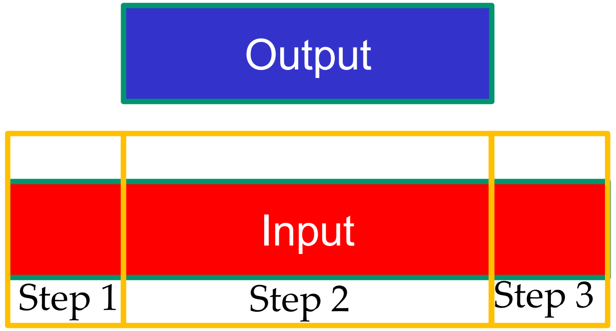
Advantages:

- **Coalesce global memory accesses**.
- **No branch divergence during computation**.

Disadvantages:

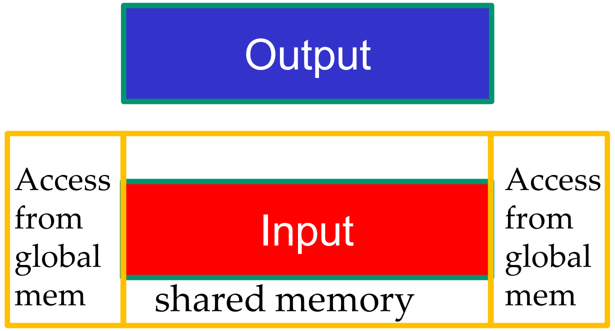
- Some threads must do >1 load, so **some branch divergence** in reading data.
- Slightly more shared memory needed.

Three Tiling Strategies



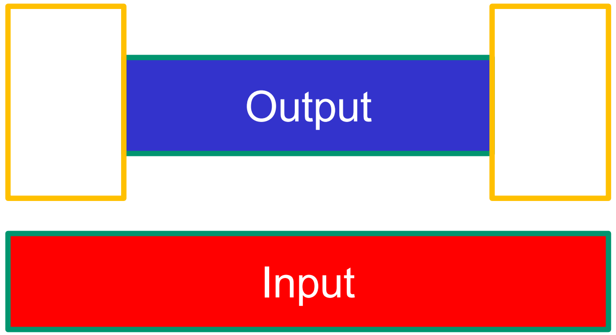
Strategy 1

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



Strategy 3

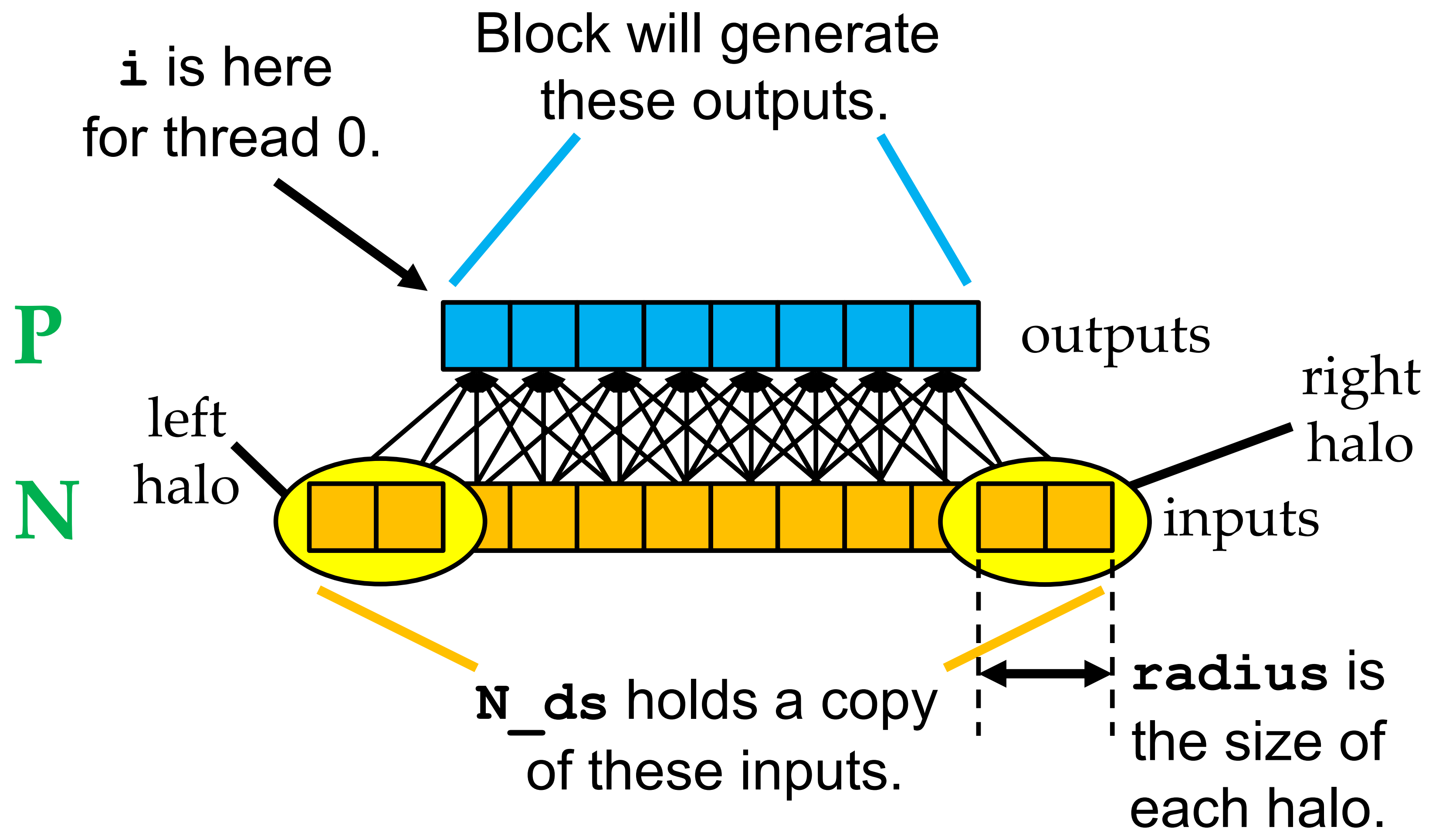
1. Block size covers **output** tile
2. Load only “core” of input tile
3. Access halo cells from global memory



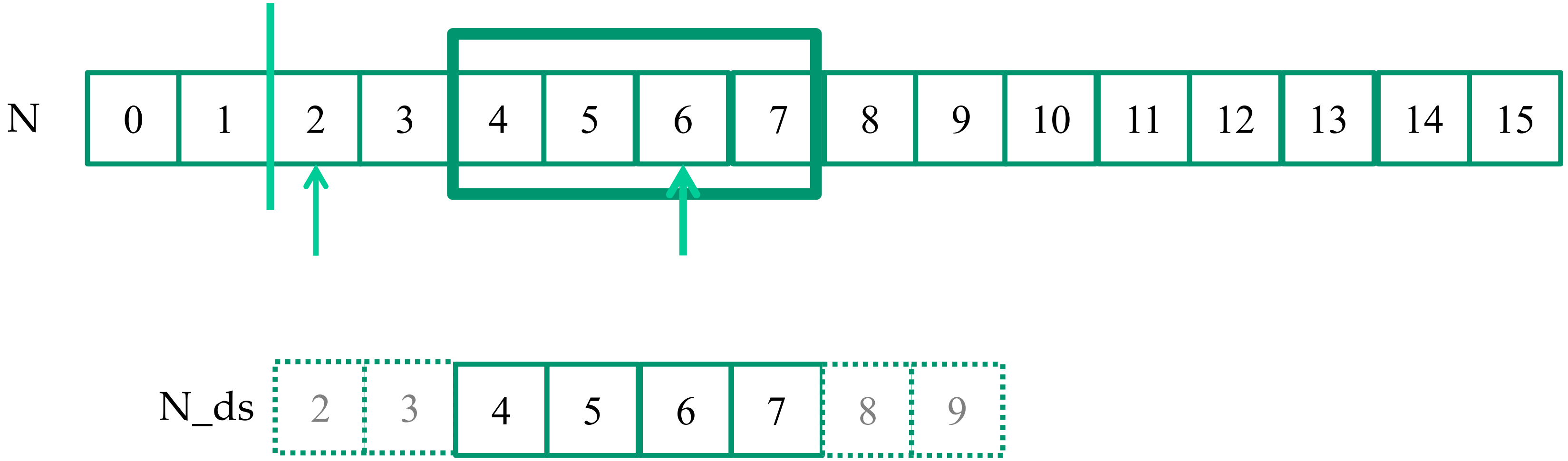
Strategy 2

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

Strategy 1: Variable Meanings for a Block

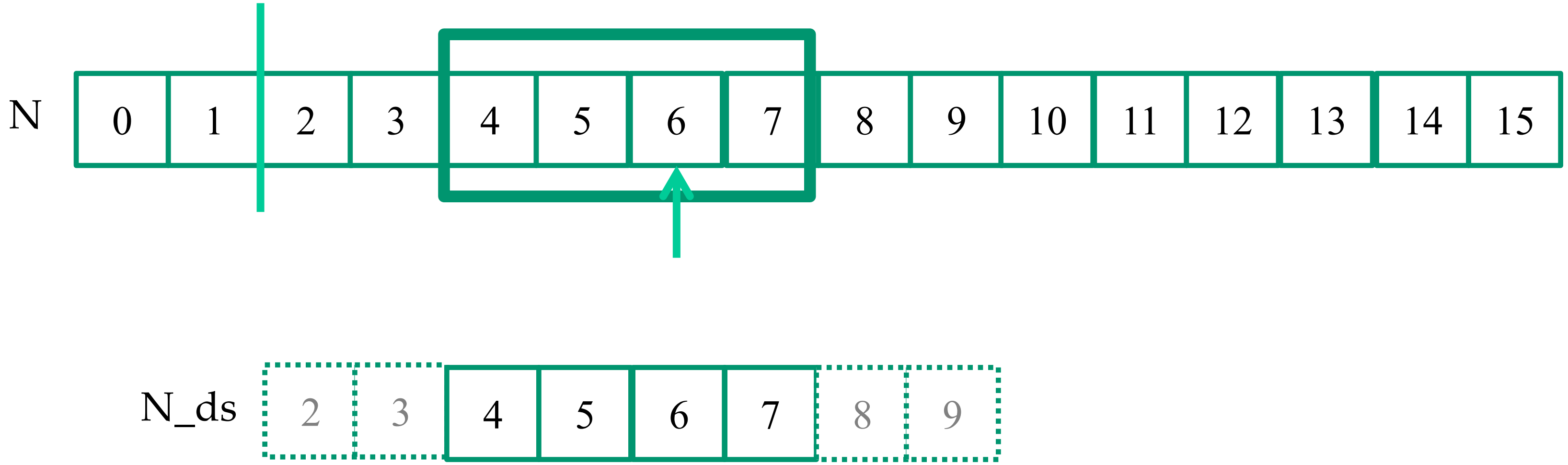


Loading the left halo



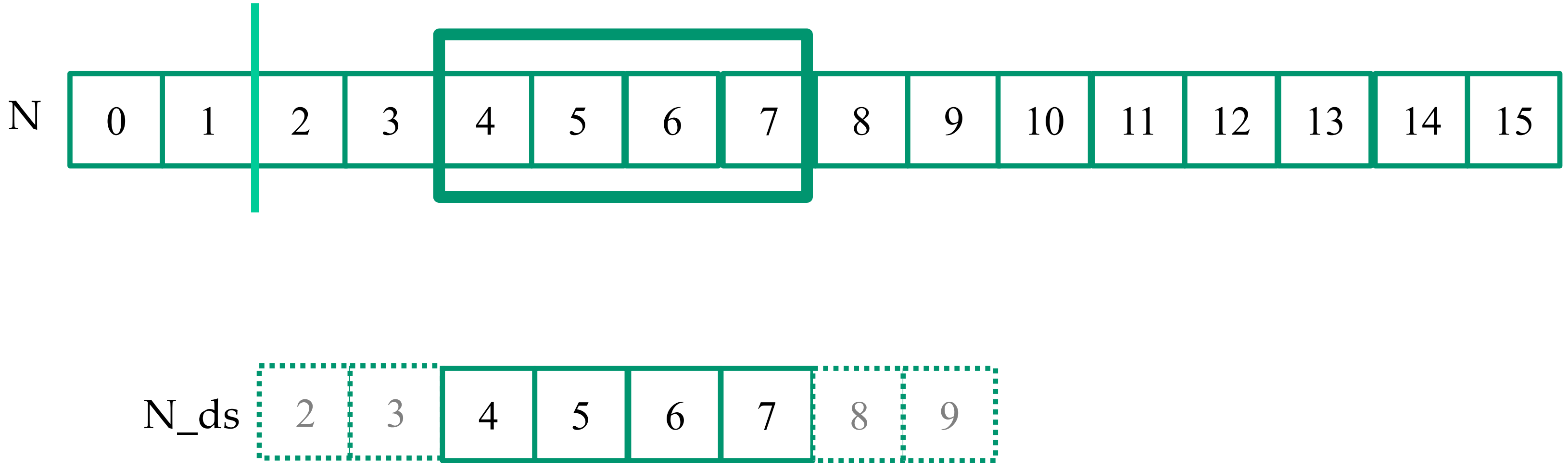

```
int radius = Mask_Width / 2;
int halo_index_left = (blockIdx.x - 1) * blockDim.x + threadIdx.x;
if (threadIdx.x >= (blockDim.x - radius)) {
   N_ds[threadIdx.x - (blockDim.x - radius)] =
      (halo_index_left < 0) ? 0 : N[halo_index_left];
}
```

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

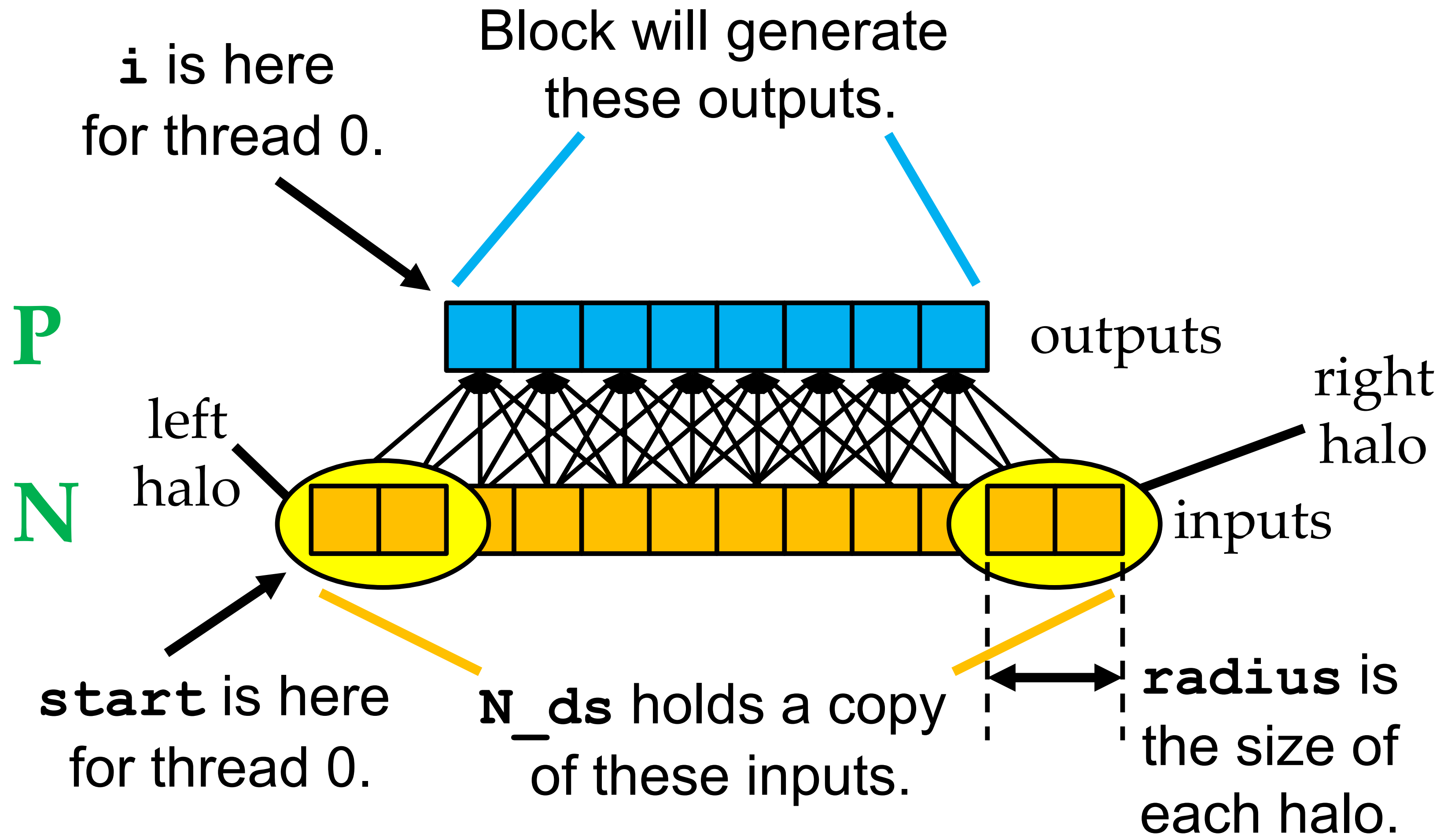



```
int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
if (threadIdx.x < radius) {
    N_ds[radius + blockDim.x + threadIdx.x] =
      (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }
```

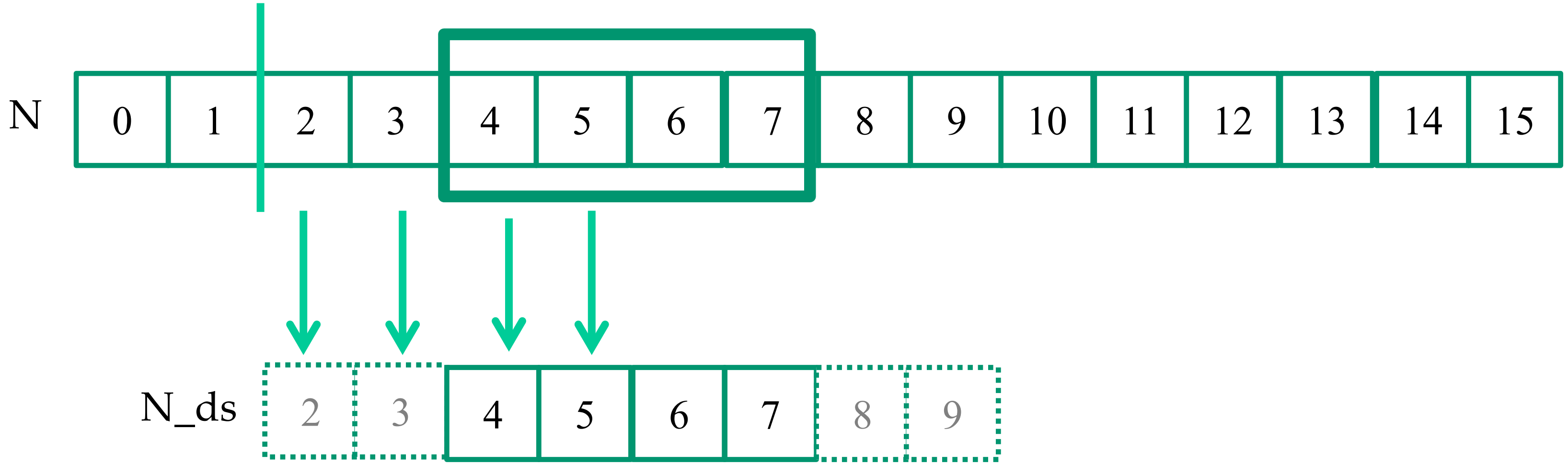
```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int Mask_Width, int Width) {

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

  int radius = Mask_Width / 2;

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

  N_ds[radius + threadIdx.x] = N[blockIdx.x * blockDim.x + threadIdx.x];  // bounds check is needed

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

Strategy 1

Alternative implementation of Strategy 1: Variable Meanings for a Block

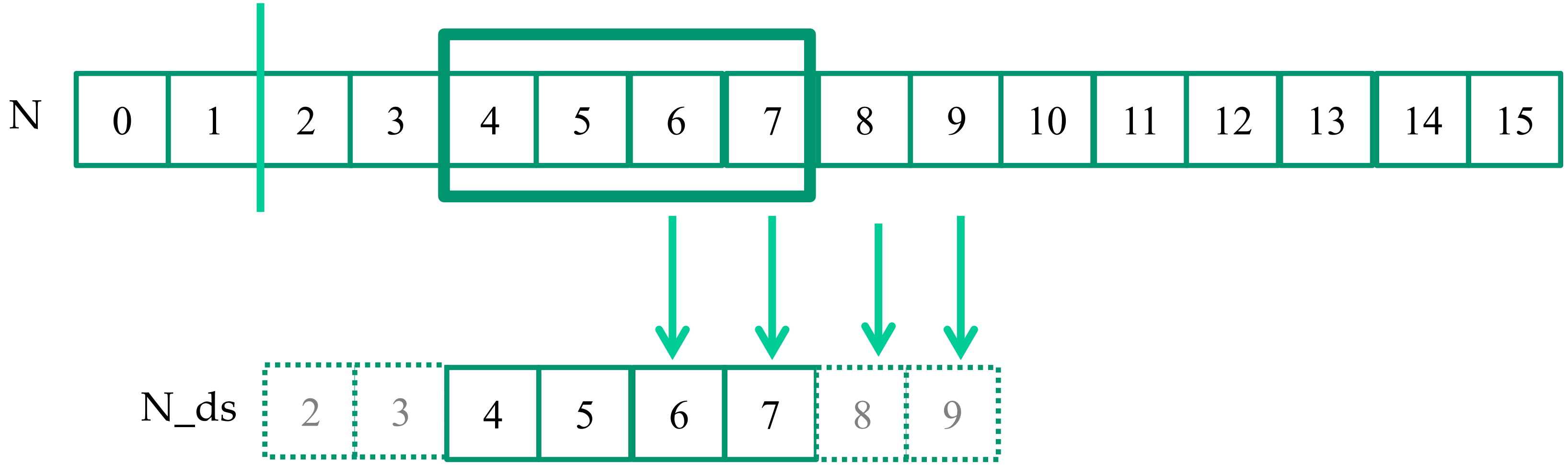


Load the Input Data – step 1




```
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 (MASK_WIDTH - 1 > threadIdx.x) {     // 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_tiled_kernel float *N, float *P, 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 (MASK_WIDTH – 1 > threadIdx.x) {      // 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_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_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

Review: What Shall We Parallelize?

In other words,

What should one thread do?

One answer:

- (same as with vector sum and matrix multiply)
- **compute an output element!**
  - **Strategy 1 & 3**

Is that our only choice? (What about Strategy 2?)

Strategy 2: Parallelize Loading of a Tile

Alternately,

- **each thread loads** one input element, and
- **some threads compute** an output.

(compared with previous approach)

Advantage:

- **No** branch **divergence for load** (high latency).
- **Avoid narrow global access** (2 × halo width).

Disadvantage:

- Branch **divergence for compute** (low latency).

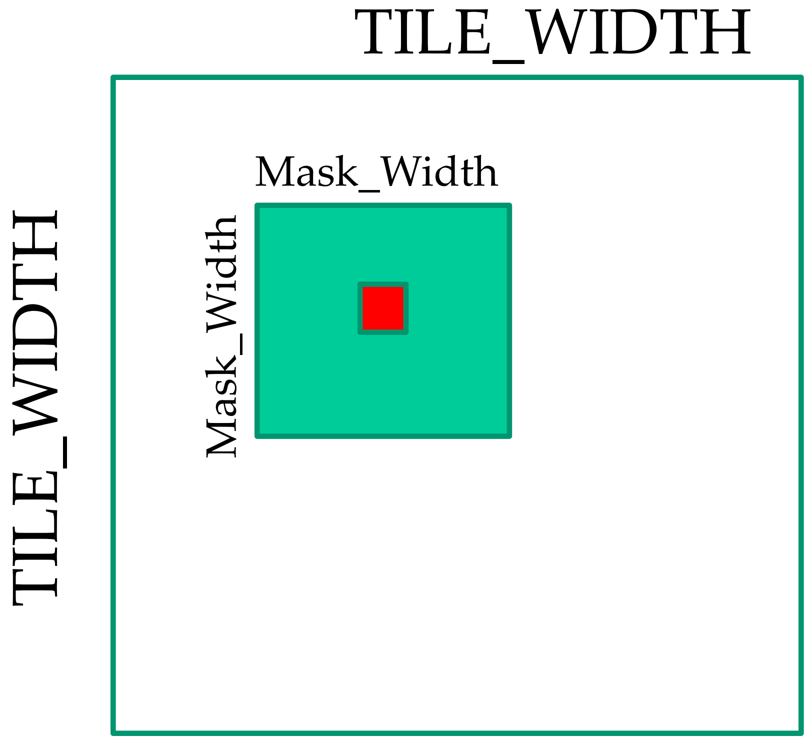
2D Example of Loading Parallelization

Let’s do an example for 2D convolution

- Thread block matches input tile size
- Each thread loads one element of input tile
- Some threads do not participate in calculating output (Strategy 2)

Parallelizing Tile Loading

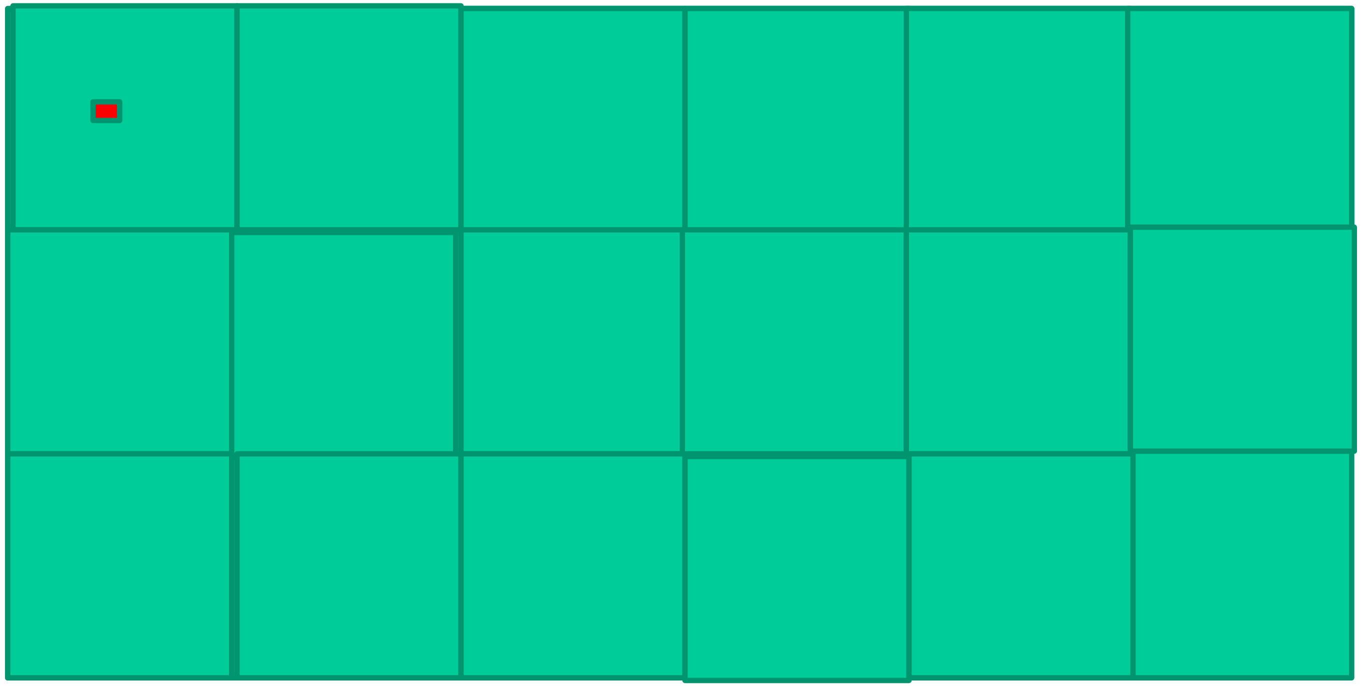
- Load a tile of N into shared memory
  - All threads participate in loading
  - A subset of threads then use each N element in shared memory



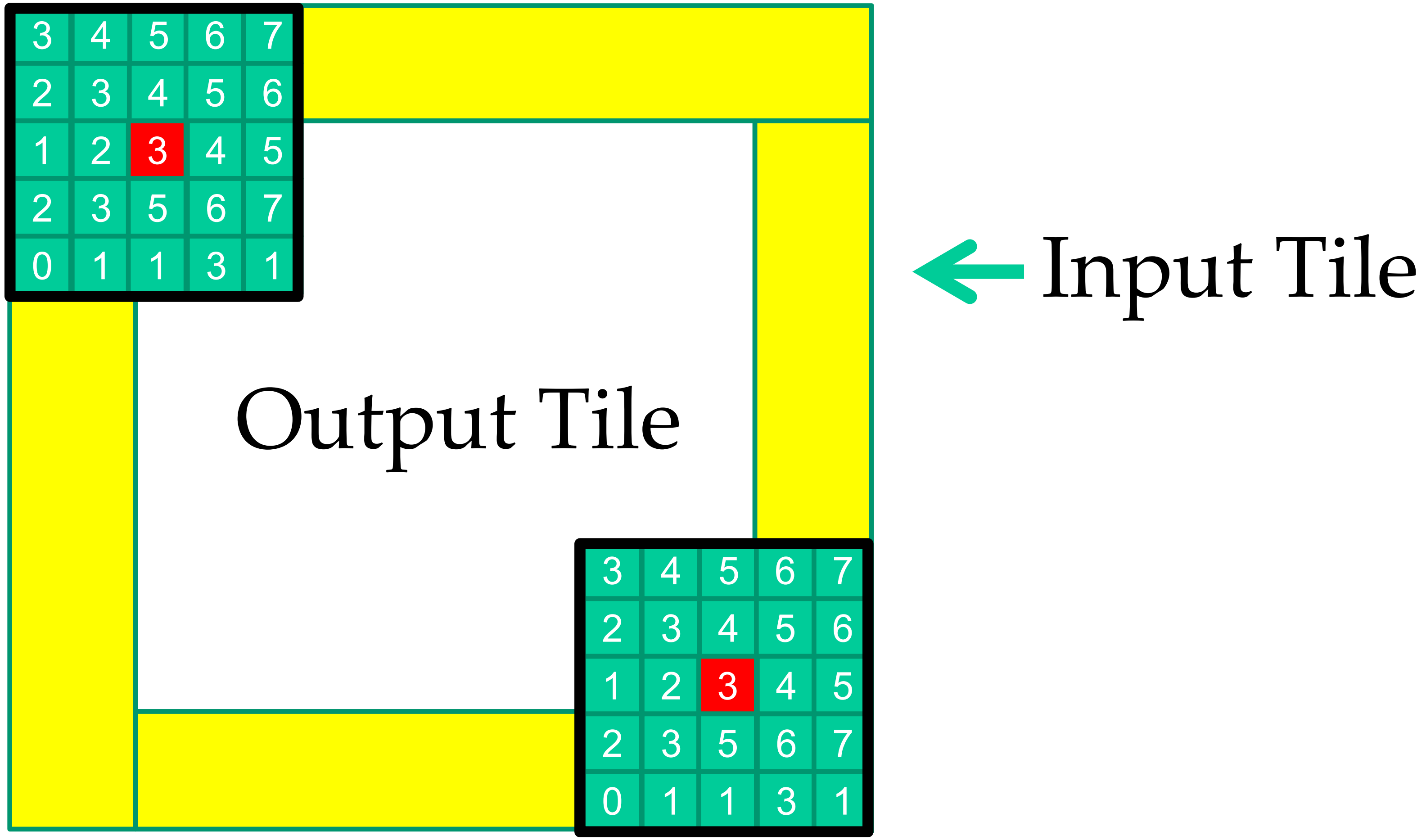
Output Tiles Still Cover the Output!

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 Block Dimensions

```
dim3 dimBlock(TILE_WIDTH + 4,TILE_WIDTH + 4, 1);
```

In general, block width (square blocks) should be

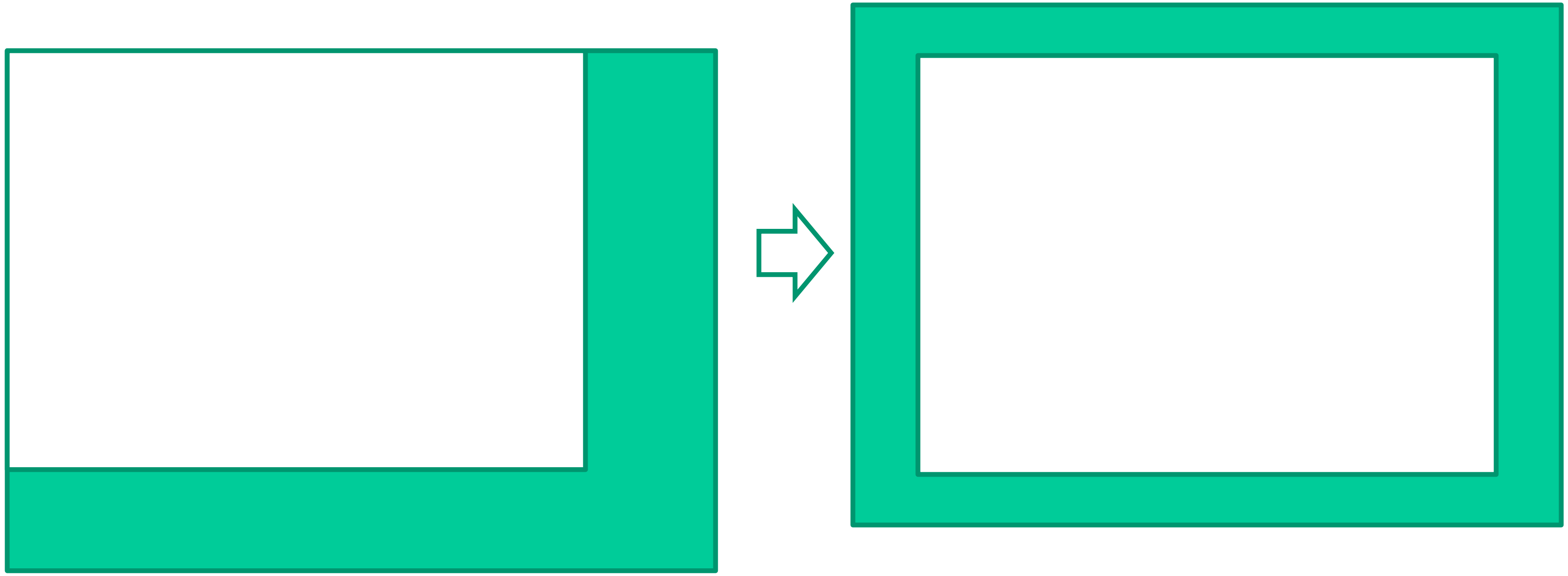
```
     TILE_WIDTH + (MASK_WIDTH-1)
```

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

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

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

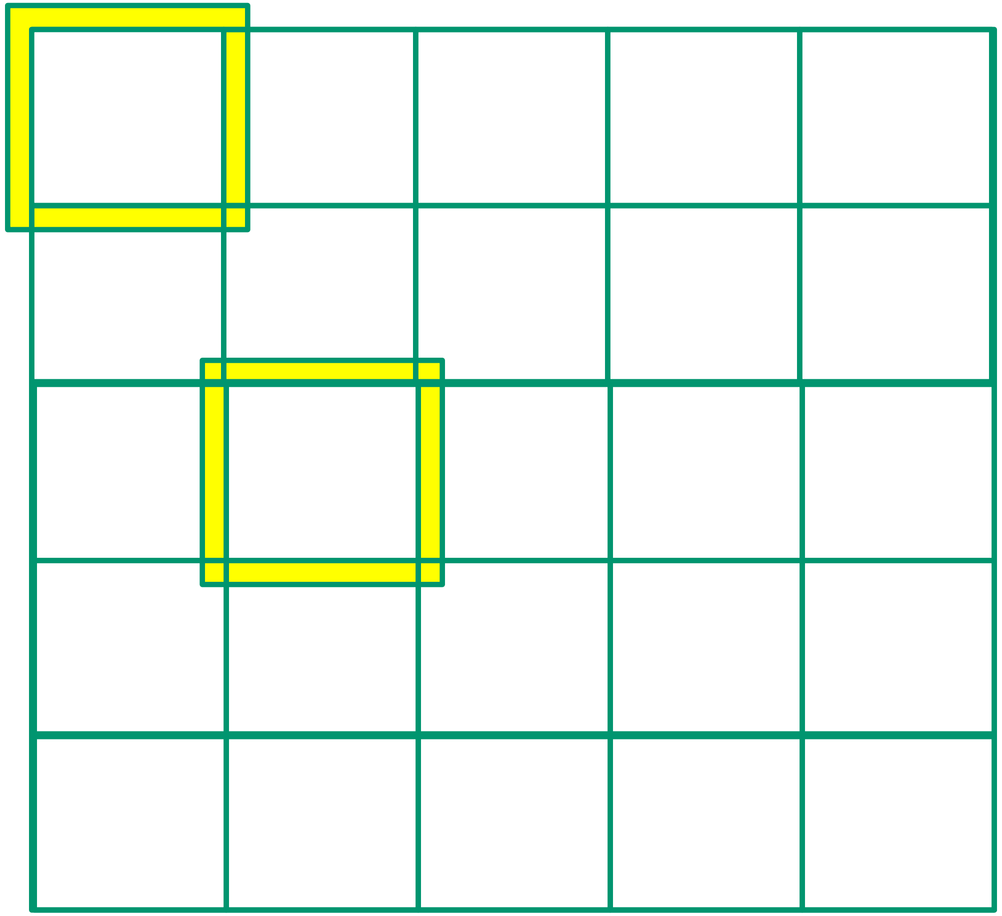
Shifting from output coordinates to input coordinates



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-2; // MASK_WIDTH / 2
int col_i = col_o-2; // (radius in
                     //  prev. code)
```

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 < 5; i++) {
    for(j = 0; j < 5; j++) {
      Pvalue += Mc[i][j] * tile[i+ty][j+tx];
    }
  }
  // if continues on next page
```

Not All Threads Write Output

```
    if(row_o < Width && col_o < Width)
      P[row_o * Width + col_o] = Pvalue;
    }
  } // end of if selecting output
    // tile threads
```

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
  - If condition and divergence during inner product computation

ANY MORE QUESTIONS? READ CHAPTER 7