ECE408 / CS483 / CSE408
Summer 2024

Applied Parallel Programming

Lecture 9: Tiled Convolution

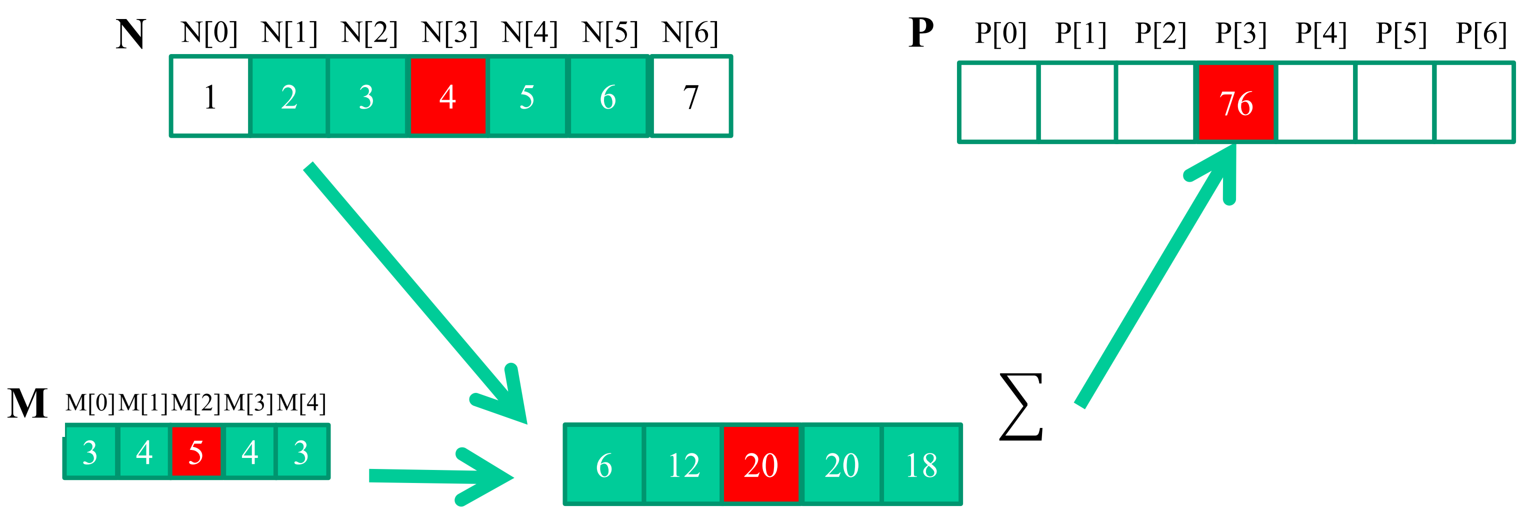
What Will You Learn Today?

- tiled convolution algorithms
  - some intricate aspects of tiling algorithms
  - output tiles versus input tiles
  - three different styles of input tile loading

- prepare for MP-4: tiled 3D convolution

Recall the 1D Convolution Operation

- Calculation of P[3]



And the 2D Convolution Operation

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

P

		321				

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

$\sum$

Are We Memory-Limited?

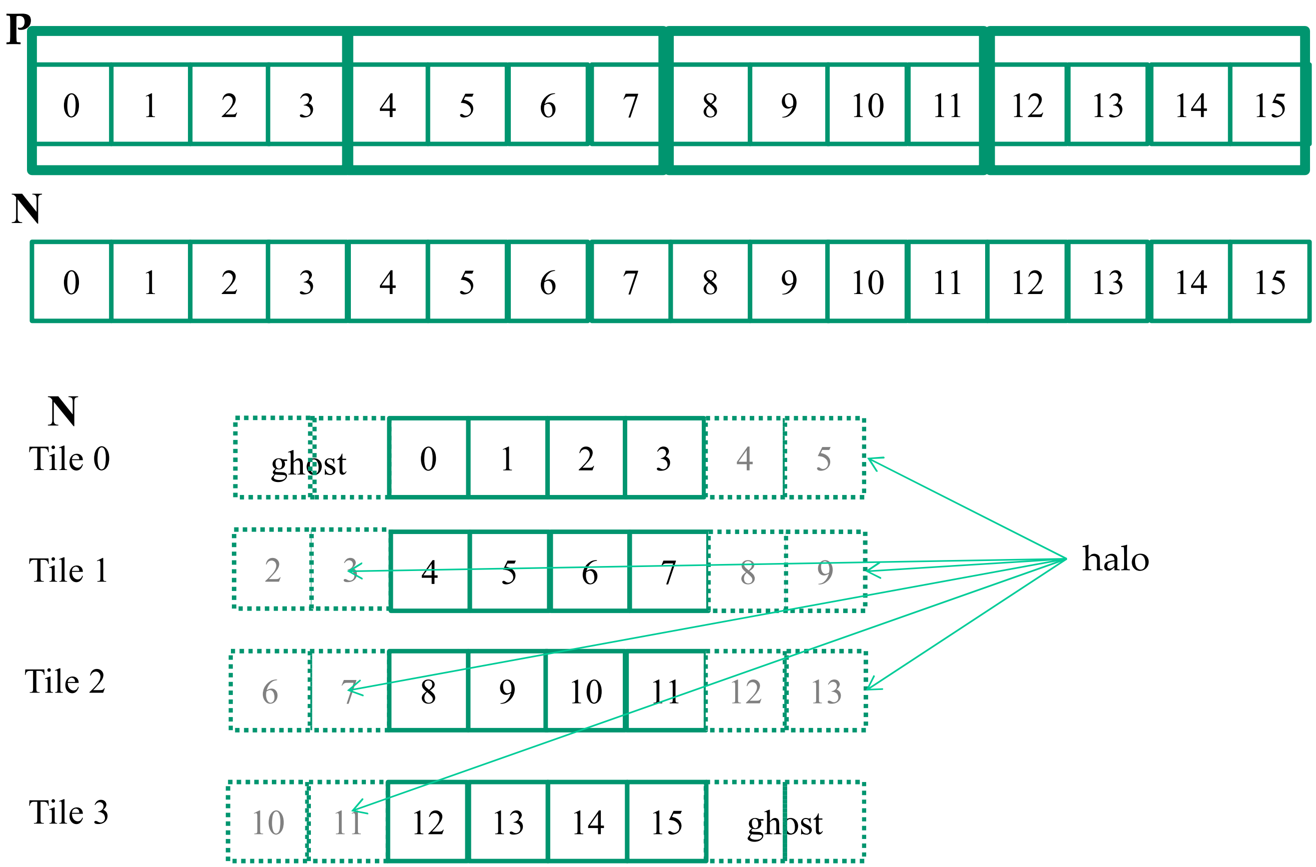
For the 1D case, every output element requires

- 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.
- **Also memory-limited.**

Tiled 1D Convolution Basic Idea



What Shall We Parallelize?

In other words,

What should one thread do?

One answer:

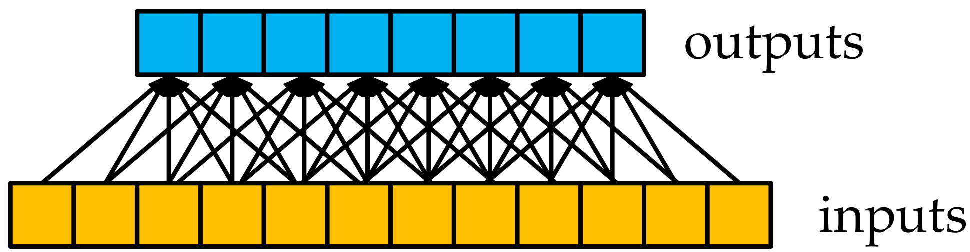
- (same as with vector sum and matrix multiply)
- **compute an output element!**

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

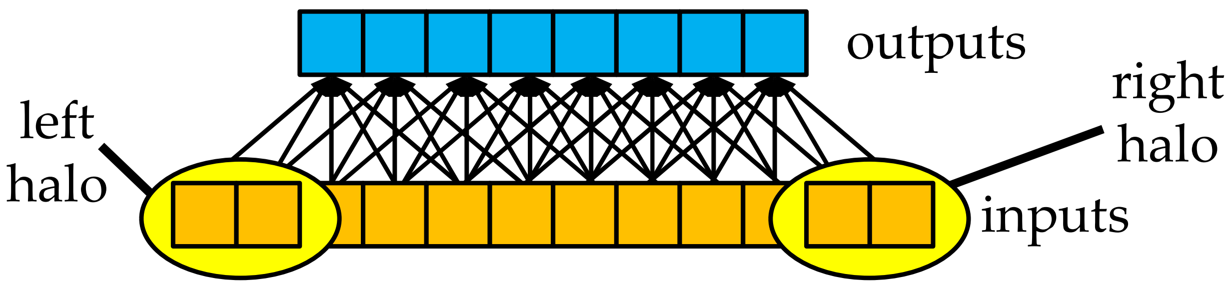
tile	2	3	4	5	6	7	8	9

- 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 Mem.

One answer: no,

- 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**

A really bad idea on early GPUs, but later GPUs offer larger last-level caches (shared by all SMs), making performance competitive.

```
__global__
void convolution_1D_tiled_cache_kernel(float *N, float *P, int Width)
{
  __shared__ float  tile[TILE_WIDTH];

  int This_tile_start_point = blockIdx.x * blockDim.x;
  int i = This_tile_start_point + threadIdx.x;

  tile[threadIdx.x] = N[i];  // boundary checking is missing here

  __syncthreads();

  int radius = MASK_WIDTH / 2;
  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)) {
       int tile_index = N_index - This_tile_start_point;
       if ((tile_index >= 0) && (tile_index < blockDim.x))
         Pvalue += tile[tile_index] * Mc[j];
       else
         Pvalue += N[N_index] * Mc[j];
     }
  }
  P[i] = Pvalue;
}
```

1D convolution with halos read from global memory

Can Load Halo to Shared Mem.

Better answer: yes,
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.

Let's write the code!

Allocate and Initialize Variables

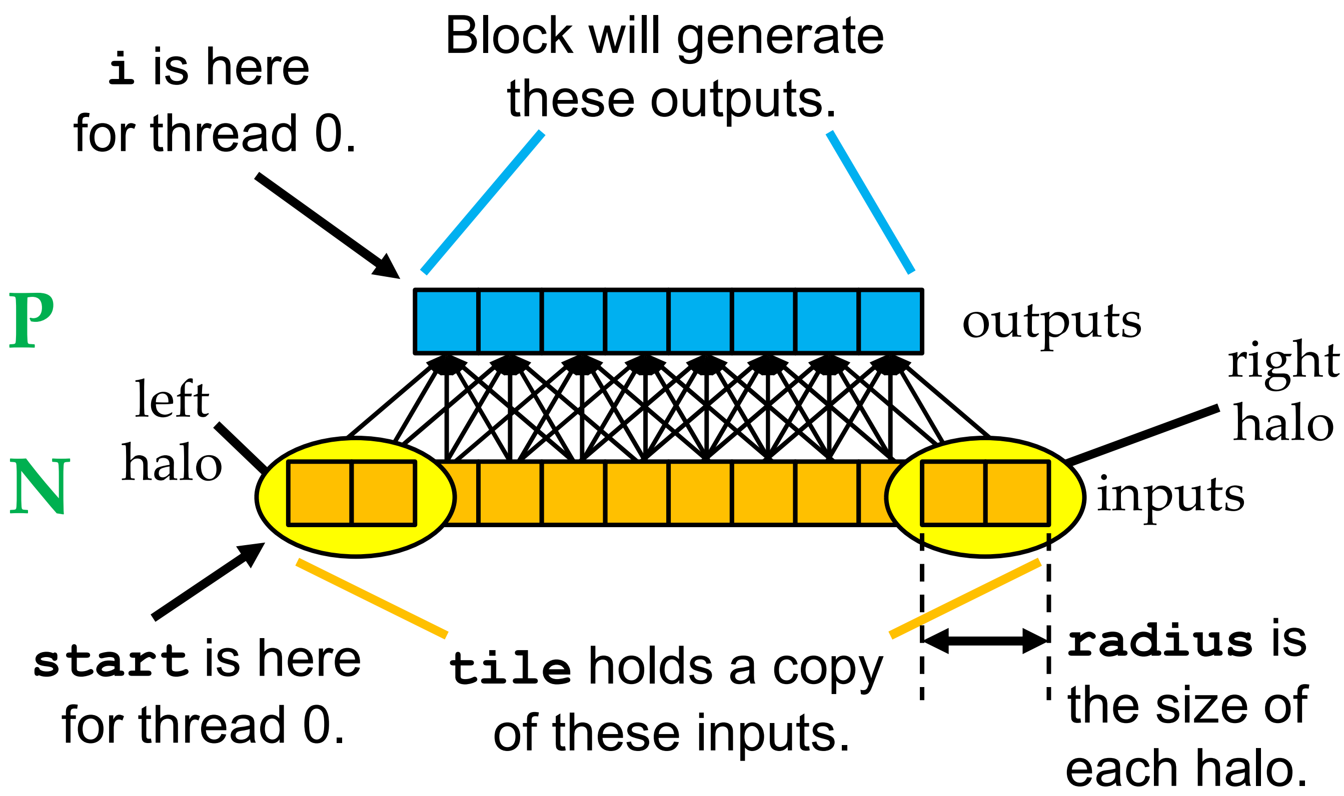
```
__global__ void convolution_1D_tiled_kernel
    (float *N, float *P, int Width)
{
  // shared tile with space for both halos
  __shared__ float tile[TILE_SIZE + MASK_WIDTH - 1];

  int radius = MASK_WIDTH / 2;  // a useful constant

  // this thread's index into output P
  int i      = blockIdx.x * blockDim.x + threadIdx.x;

  // this thread's starting element of N
  int start  = i - radius;
```

Variable Meanings for a Block



Load the Input Data

```
if (0 <= start && Width > start) {  // all threads
  tile[threadIdx.x] = N[start];
} else {
  tile[threadIdx.x] = 0.0f;
}
if (MASK_WIDTH – 1 > threadIdx.x) { // some threads
  start += TILE_SIZE;
  if (Width > start) {
    tile[threadIdx.x + TILE_SIZE] = N[start];
  } else {
    tile[threadIdx.x + TILE_SIZE] = 0.0f;
  }
}
__syncthreads(); // OUTSIDE of if's
```

And Compute an Output Element

```
if (i < Width) {  // only threads computing outputs

  float Pvalue = 0;       // running sum

  // compute output element
  for (int j = 0; MASK_WIDTH > j; j++) {
    Pvalue += tile[threadIdx.x + j] * Mc[j];
  }

  // write to P
  P[i] = Pvalue;
}
```

```
__global__
void convolution_1D_tiled_cache_kernel(float *N, float *P, int Width) {
  __shared__ float tile[TILE_SIZE + MASK_WIDTH - 1];

  int radius = MASK_WIDTH / 2;
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int start = i - radius;

  if (0 <= start && Width > start) {       // all threads
    tile[threadIdx.x] = N[start];
  } else {
    tile[threadIdx.x] = 0.0f;
  }
  if (MASK_WIDTH - 1 > threadIdx.x) {      // some threads
    start += TILE_SIZE;
    if (Width > start) {
      tile[threadIdx.x + TILE_SIZE] = N[start];
    else
      tile[threadIdx.x + TILE_SIZE] = 0.0f;
  }
  __syncthreads();
  if (i < Width) {
    float Pvalue = 0.0f;
    for (int j = 0; MASK_WIDTH > j; j++) {
      Pvalue += tile[threadIdx.x + j] * Mc[j];
    }
    P[i] = Pvalue;
  }
}
```

1D convolution with halos read into shared memory (output parallelism)

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!**

Is that our only choice?

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).

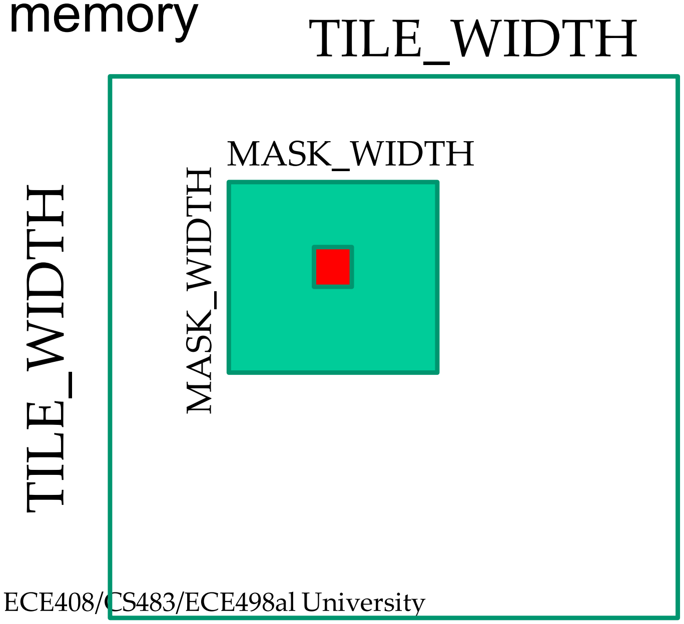
2D Example of Loading Parallelization

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,

Parallelizing Tile Loading

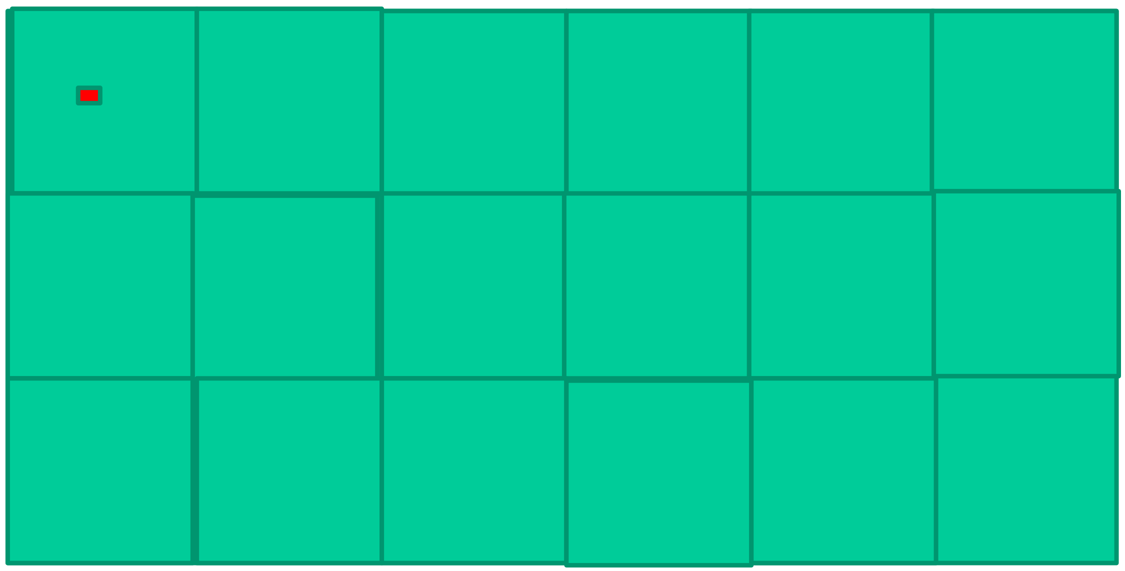
- Load a tile of N into shared memory
  - All threads participate in loading
  - A subset of threads then use each N element in shared memory



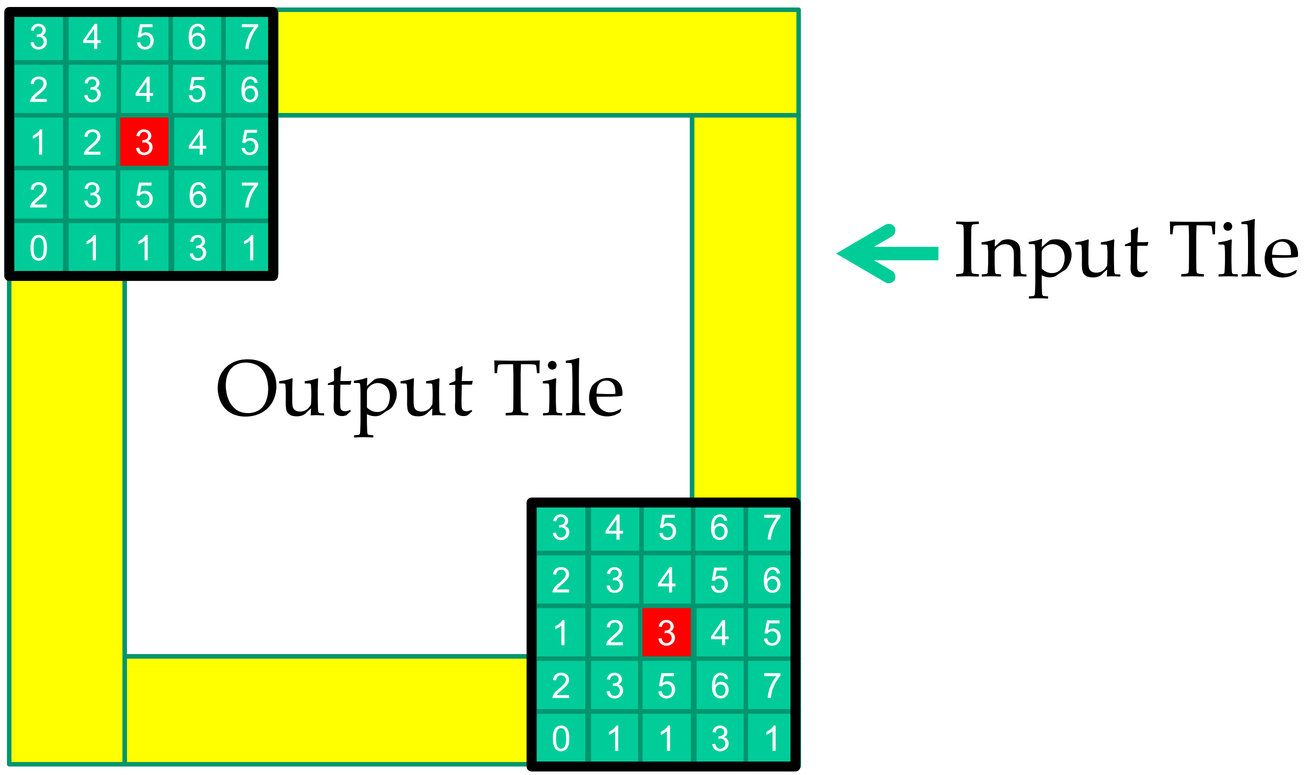
Output Tiles Still Cover the Output!

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 Width

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

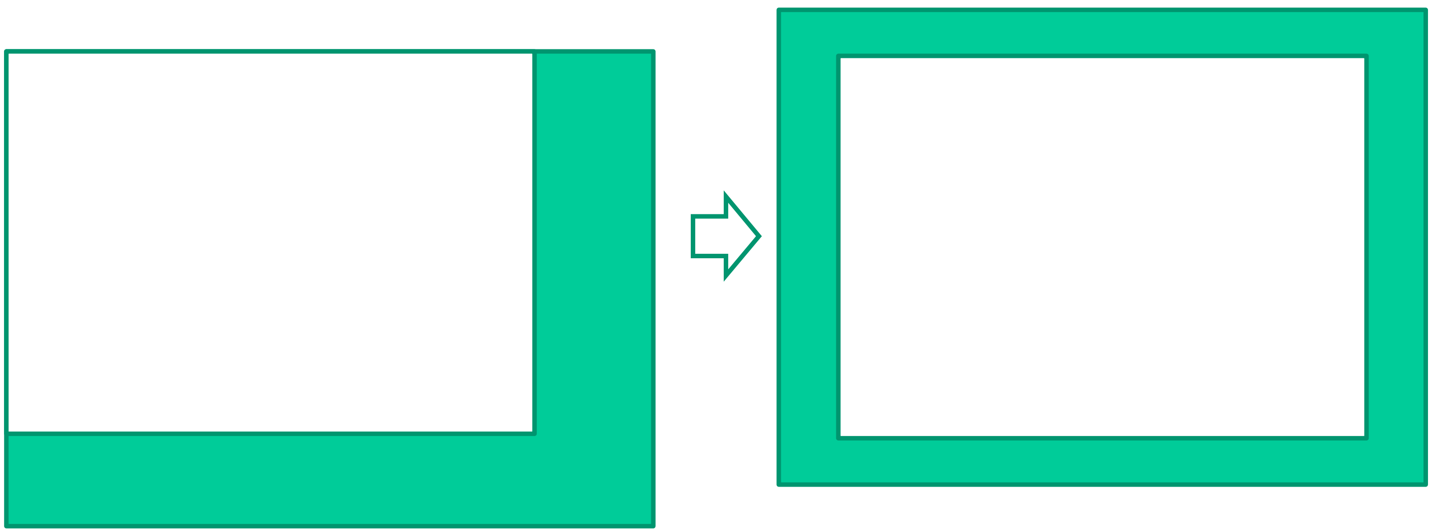
In general, block width should be

TILE_WIDTH + (MASK_WIDTH - 1)

Dim3 dimGrid(ceil(Width/(1.0*TILE_WIDTH)),
ceil(Width/(1.0*TILE_WIDTH)), 1)

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

Shifting from output coordinates to input coordinates

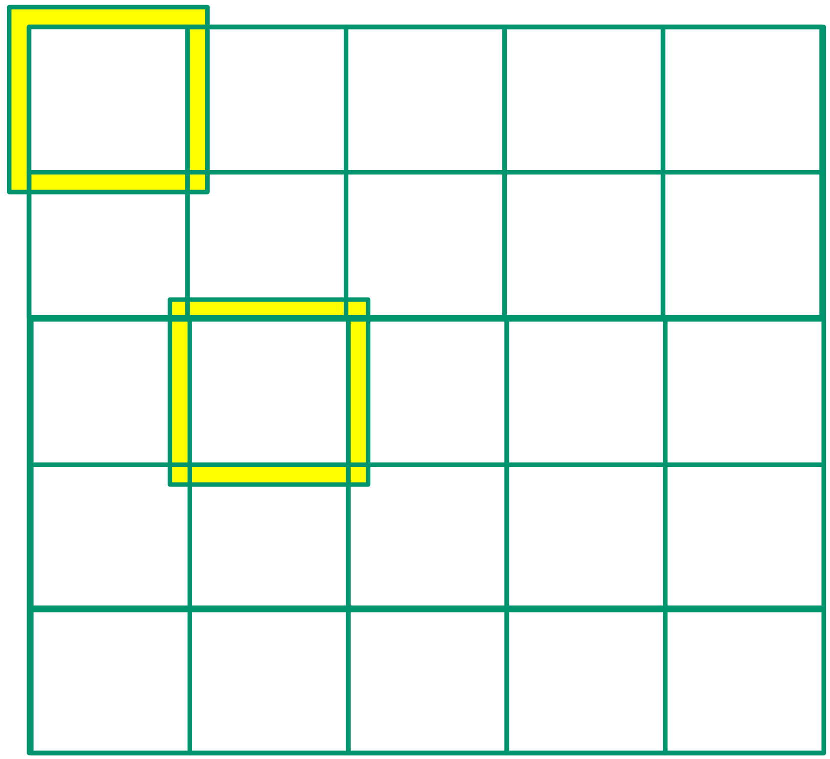


Shifting from output coordinates to input coordinates

```
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;
}
__sync_threads (); // 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
```

Which Strategy Will You Choose?

- We recommend the input parallelization strategy for MP4.
- Alternatively, you may instead choose to parallelize based on either of the two output parallelization strategies
  - reading halo values from global memory, or
  - Keeping the halos in shared memory and issuing more than one load from some threads.

YOU MUST NOTE YOUR STRATEGY
AT THE START OF YOUR `lab4.cu` FILE!

QUESTIONS?

READ CHAPTER 7!

Problem Solving

Let's think a little more deeply about the tradeoffs with the parallelization strategies. Assuming that…

- we use input parallelism
- with **16×16** thread blocks (**256** threads/block)
- And a **5×5** mask (**MASK_WIDTH** of **5**),

Q: how many threads compute output elements?

A: TILE_WIDTH is 16 – 5 + 1 = 12, so 12×12 = 144 threads.

Problem Solving

Assuming that…

- **we use input parallelism**
- **with 16×16 thread blocks (256 threads/block)**
- **And a 5×5 mask (MASK_WIDTH of 5),**
- **TILE_WIDTH is 12, so 144 threads compute output elements.**

Q: How many of the 8 warps are active during computation?

Here was our condition for computation:

```
if(ty < TILE_WIDTH && tx <TILE_WIDTH){
```

A: Not as bad as it might seem: warps 6 and 7 have ty >= 12 (TILE_WIDTH is 12), so they do nothing during computation.

Problem Solving

Assuming input parallelism, 16×16 thread blocks, a 5×5 mask, TILE_WIDTH of 12, and 144 threads computing output elements, warps 6 and 7 do nothing during computation.

Here was our condition for computation:

```
if(ty < TILE_WIDTH && tx <TILE_WIDTH){
```

Q: What happens with warps 0 through 5?

Warps 0 through 5 have 24 active threads,

- so **wasting some computation resources**,
- but since we **operate out of shared memory**,
- having **less parallelism** during computation is **probably ok**.

(We could use a more complex mapping to reduce to 5 warps.)

Problem Solving

For larger masks, input parallelism may be less attractive:

- with the same **16×16** thread blocks (**256** threads/block) and
- a **9×9** mask (**MASK_WIDTH** of **9**),
- **TILE_WIDTH** is **16 – 9 + 1 = 8**,
- so output computation requires **8x8 = 64** threads, or **2** warps,
- but **4** warps are active (with **16** active threads each).

Even if we remap to use **2** warps,

- the other **6** warps still count against warp resources,
- so each SM uses only **¼ of possible warps for computation**.

Problem Solving

Using output parallelism with a large mask

- improves computation utilization
- at the expense of loading more data.

For the same **16×16** thread blocks (**256** threads/block)

- and **9×9** mask (**MASK_WIDTH** of **9**),
- output parallelism gives an **input tile of 16 + 9 - 1 = 24**, which
- requires **up to three loads from global memory per thread**.
- But **all threads** are **active during computation** of outputs.

Problem Solving

We might also **compute multiple elements per thread**.

A **24×24** input tile (floats) requires only **2.25 kB**.

Using the same **16×16** thread blocks,

- we might instead **compute 4 elements per thread**,
- using a **40×40 input tile** (floats),
- which **requires 6.25 kB**,
- allowing **10 thread blocks on an SM** with 64kB of shared memory
- (limit on threads will be more restrictive).

Problem Solving

Which approach is best?

As you can see, the **answer** often

- **depends on** both the **parameters of** the **problem and**
- the **resources available** on the GPU
- (SMs, shared memory, caches, threads and thread blocks per SM, and so forth).