Parallel Computing 2020

中国科学院大学
University of Chinese Academy of Sciences

Introduction to CUDA

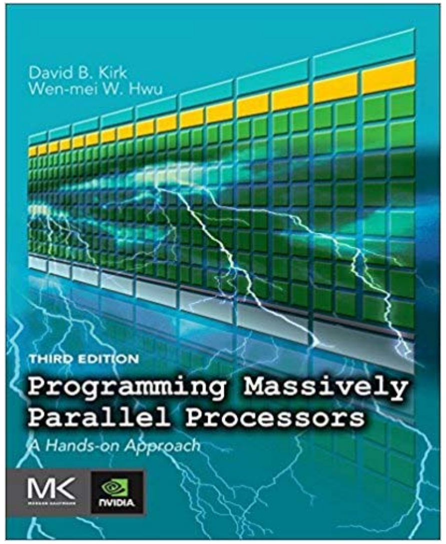
(6) Parallel Pattern: Convolution

Reference

- CUDA C Programming Guide,
  - https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

- **Programming Massively Parallel Processors,**
  - **A Hands-on Approach**
  - **Third Edition**

  - **Chapter 7**



Content

- **Convolution**
  - **An important parallel computation pattern**
- Taking advance of
  - Constant memory and caching
- Tiled 1D convolution algorithms
  - Addressing memory bandwidth issue in accessing N
  - Algorithms and analysis
- Tiled 2D convolution with halo cells
  - Algorithms and analysis

Convolution Applications

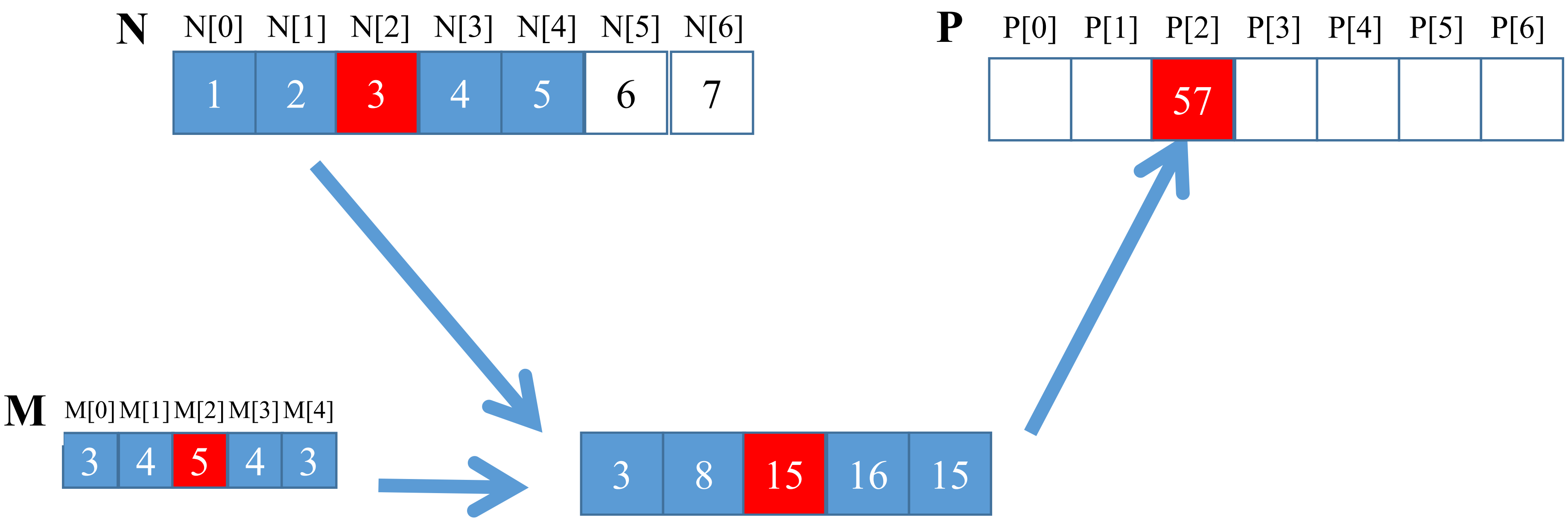
- A popular array operation that is used in various forms in signal processing, digital recording, image processing, video processing, and computer vision.
- Convolution is often performed as a **filter** that transforms signals and pixels into more desirable values.
  - Some filters smooth out the signal values so that one can see the big-picture trend.
  - Others like Gaussian filters can be used to sharpen boundaries and edges of objects in images.

Convolution Computation

- 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***
  - refer to these mask arrays as convolution **masks** to avoid confusion(with kernel in CUDA).
  - The same convolution mask is typically used for all elements of the array.

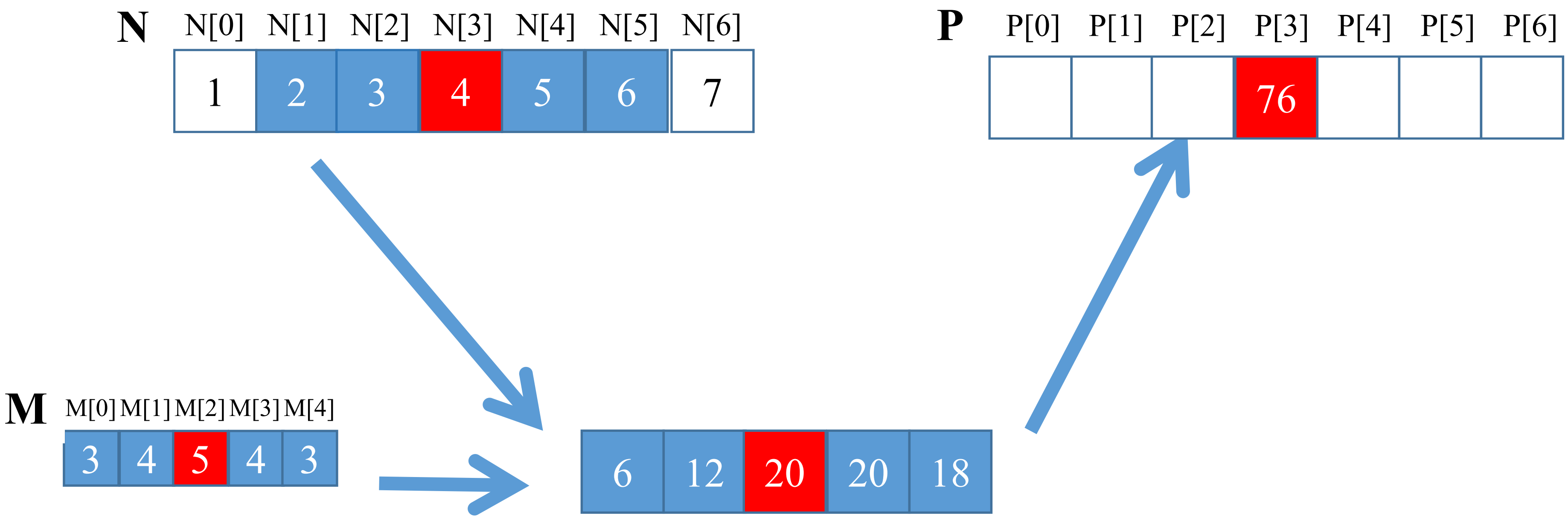
1D Convolution Example

- Commonly used for audio processing
  - Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]



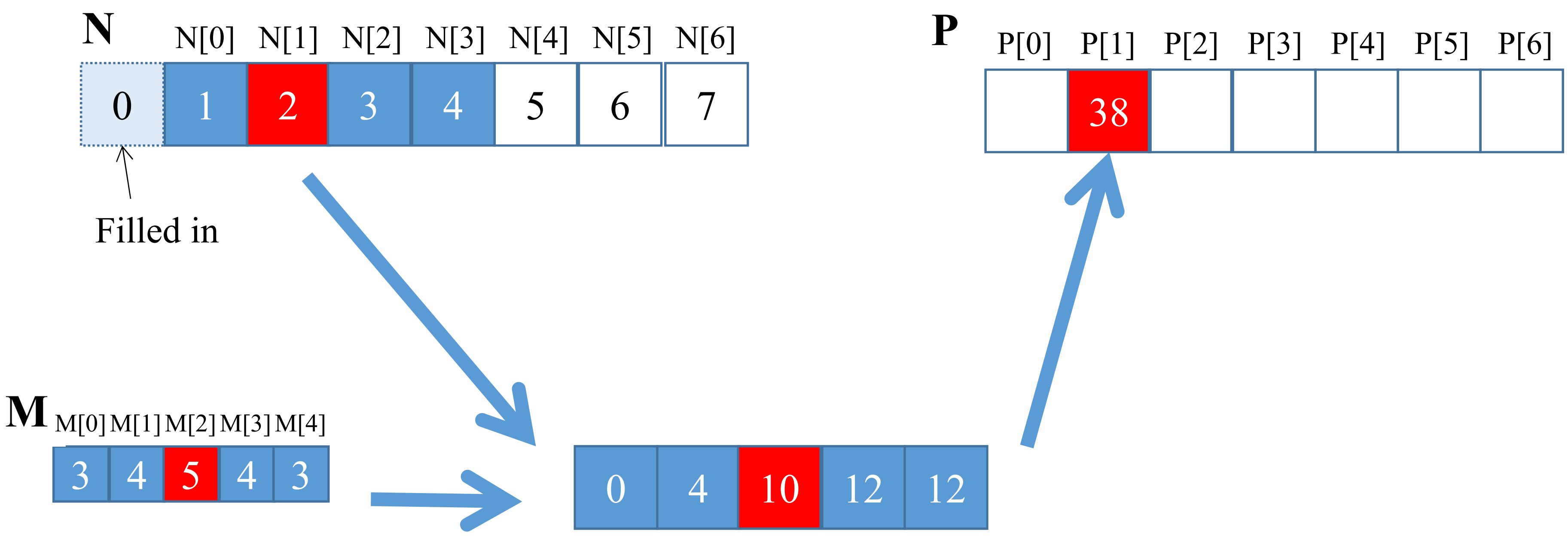
1D Convolution Example

- more on inside elements
- Calculation of P[3]



1D Convolution Boundary Condition

- 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.)



A 1D Convolution Kernel with Boundary Condition Handling

- This kernel forces all elements outside the valid data index range to 0

```
__global__ void convolution_1D_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

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

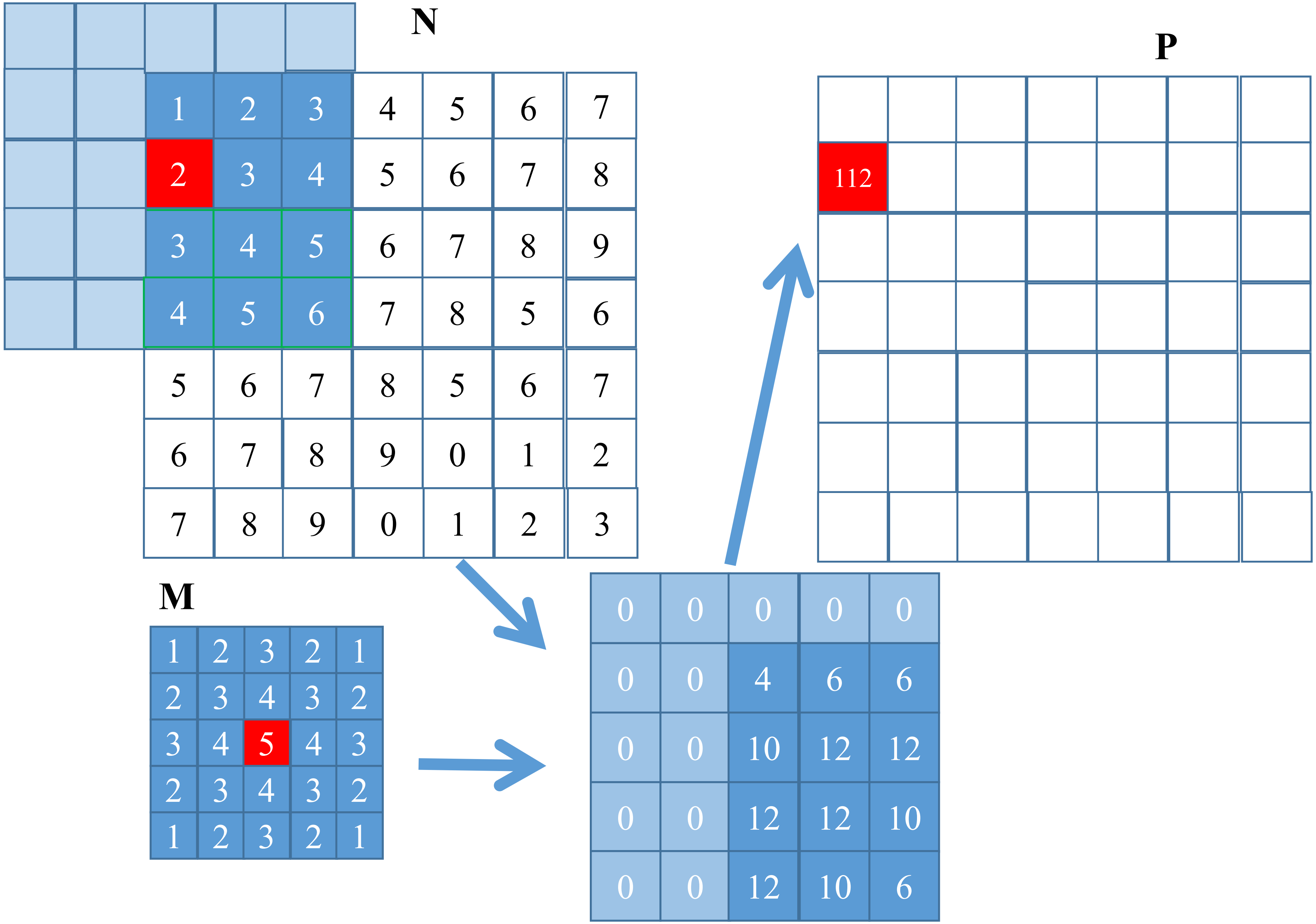
		321				

M

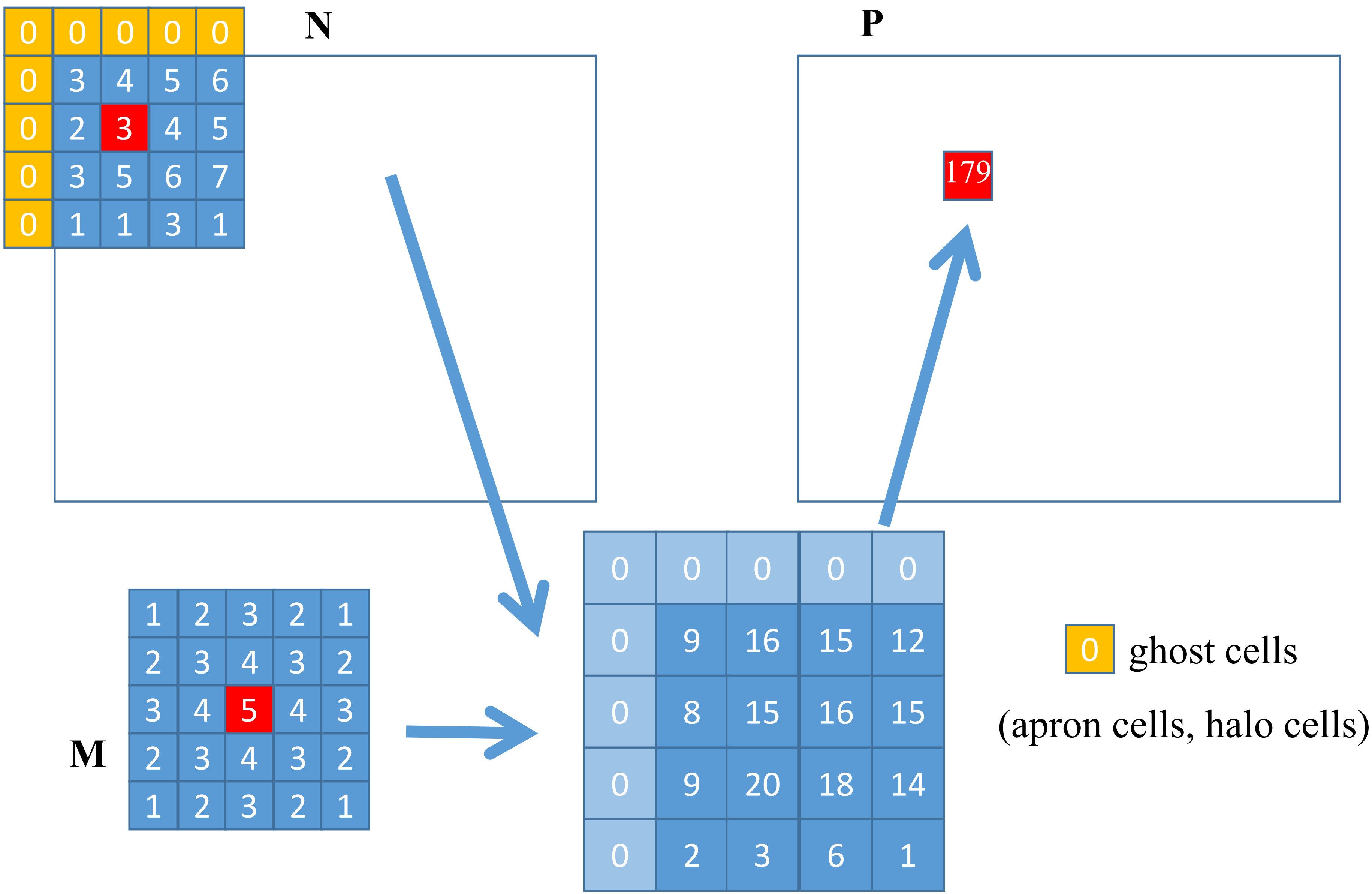
1	2	3	2	1
2	3	4	3	2
3	4	5	4	3
2	3	4	3	2
1	2	3	2	1

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

2D Convolution Boundary Condition



2D Convolution – Ghost Cells



Content

- Convolution
  - An important parallel computation pattern
- **Taking advance of**
  - **Constant memory and caching**
- Tiled 1D convolution algorithms
  - Addressing memory bandwidth issue in accessing N
  - Algorithms and analysis
- Tiled 2D convolution with halo cells
  - Algorithms and analysis

Access Pattern for M

- M is referred to as mask (a.k.a. kernel, filter, etc.)
    - Elements of M are called mask (kernel, filter) coefficients
- M features:
    - M is small in size
    - All threads need to access mask elements
    - M is not changed during kernel

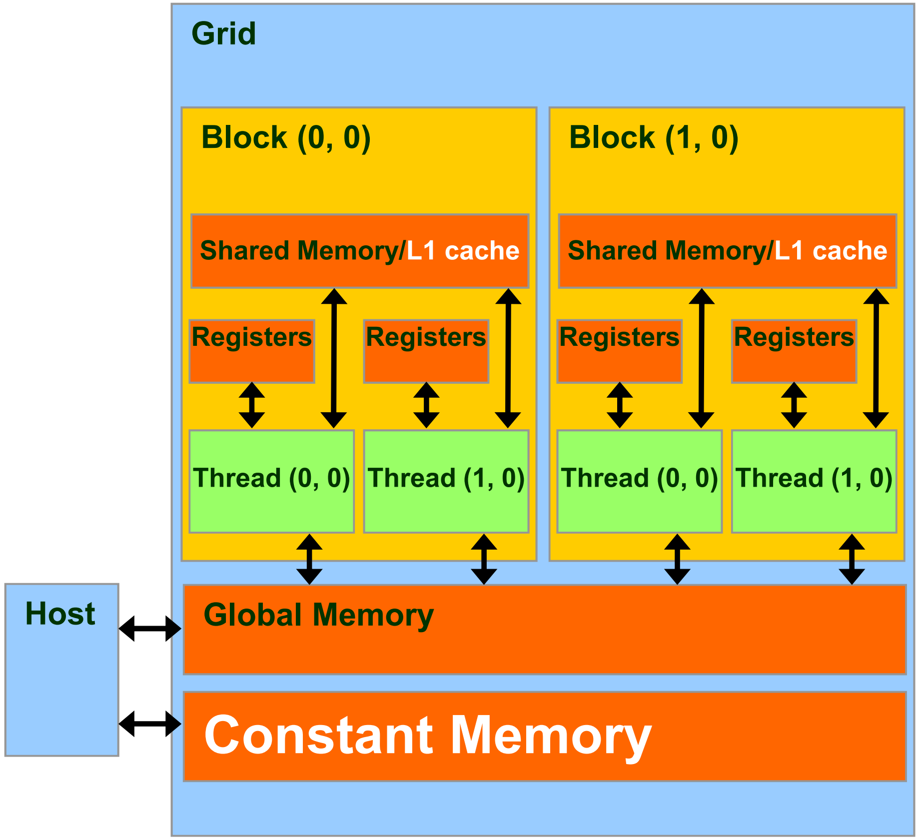
- Bonus - M elements are accessed in the same order when calculating all P elements

- M is a good candidate for **Constant Memory**

Programmer View of CUDA Memories (Review)

- Each thread can:
  - **Read/write per-thread registers (~1 cycle)**
  - Read/write per-block **shared memory (~5 cycles)**
  - Read/write per-grid **global memory (~500 cycles)**

  - **Read/only per-grid constant memory (~5 cycles with caching)**



Memory Hierarchies

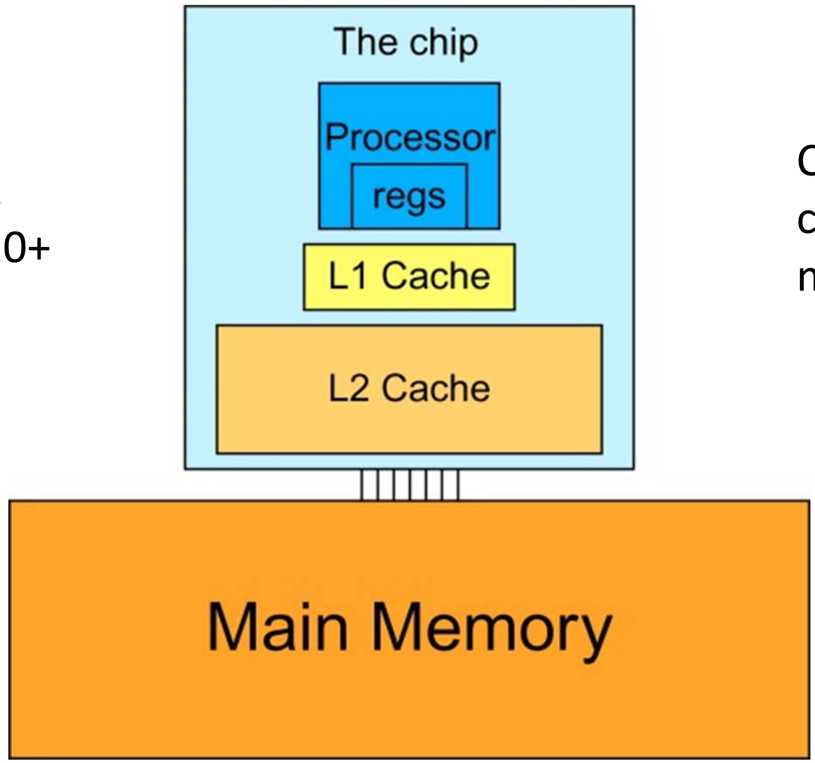
- If we had to go to global memory (DRAM) to access data all the time, the execution speed of GPUs would be limited by the global memory bandwidth

- But Constant memory variables also located in DRAM which is known as slow to access.
- How can we benefit from constant memory?
- One solution: **Caches**

Memory Hierarchies

- Simplified view of the cache hierarchy of modern processors:

L1: 16~64k X1
L2: 128k~1M X10+



Cache coherence mechanism

Cache

- A cache is an "array" of cache lines
  - A **cache line** can usually hold data from several consecutive memory addresses

- When data is requested from the global memory:
  - an entire cache line that includes the data being accessed is loaded into the cache, in an attempt to reduce global memory requests;
  - The data in the cache is a **"copy"** of the original data in global memory;

Caches - Cont’d

Some definitions:

- ***Spatial locality***: when the data elements stored in consecutive memory locations are access consecutively
- ***Temporal locality***: when the same data element is access multiple times in short period of time

- Both spatial locality and temporal locality improve the performance of caches

Scratchpad vs. Cache

- Scratchpad (shared memory in CUDA) is another type of temporary storage used to relieve main memory contention.
  - In terms of distance from the processor, scratchpad is similar to L1 cache.

- Unlike cache, scratchpad does not necessarily hold a copy of data that is also in main memory
  - Scratchpad requires explicit data transfer instructions into locations in the scratchpad , whereas cache doesn’t

Constant Cache in GPUs

- Modification to cached data needs to be (eventually) reflected back to the original data in global memory
  - Requires logic to track the modified status, etc.

- Constant cache is a special cache for constant data:
  - Data declared in the constant memory will not be modified during kernel execution.
  - Constant cache can be accessed with higher throughput than L1 cache for some common patterns

How to Use Constant Memory

- Host code allocates, initializes variables the same way as any other variables that need to be copied to the device

- Use **cudaMemcpyToSymbol(dest, src, size)** to copy the variable into the device memory

- This copy function tells the device that the variable will not be modified by the kernel and can be safely cached.

A 1D Convolution Kernel using constant memory

- This kernel forces all elements outside the valid data index range to 0

```
__global__ void convolution_1D_basic_kernel( float *N, 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;

}
```

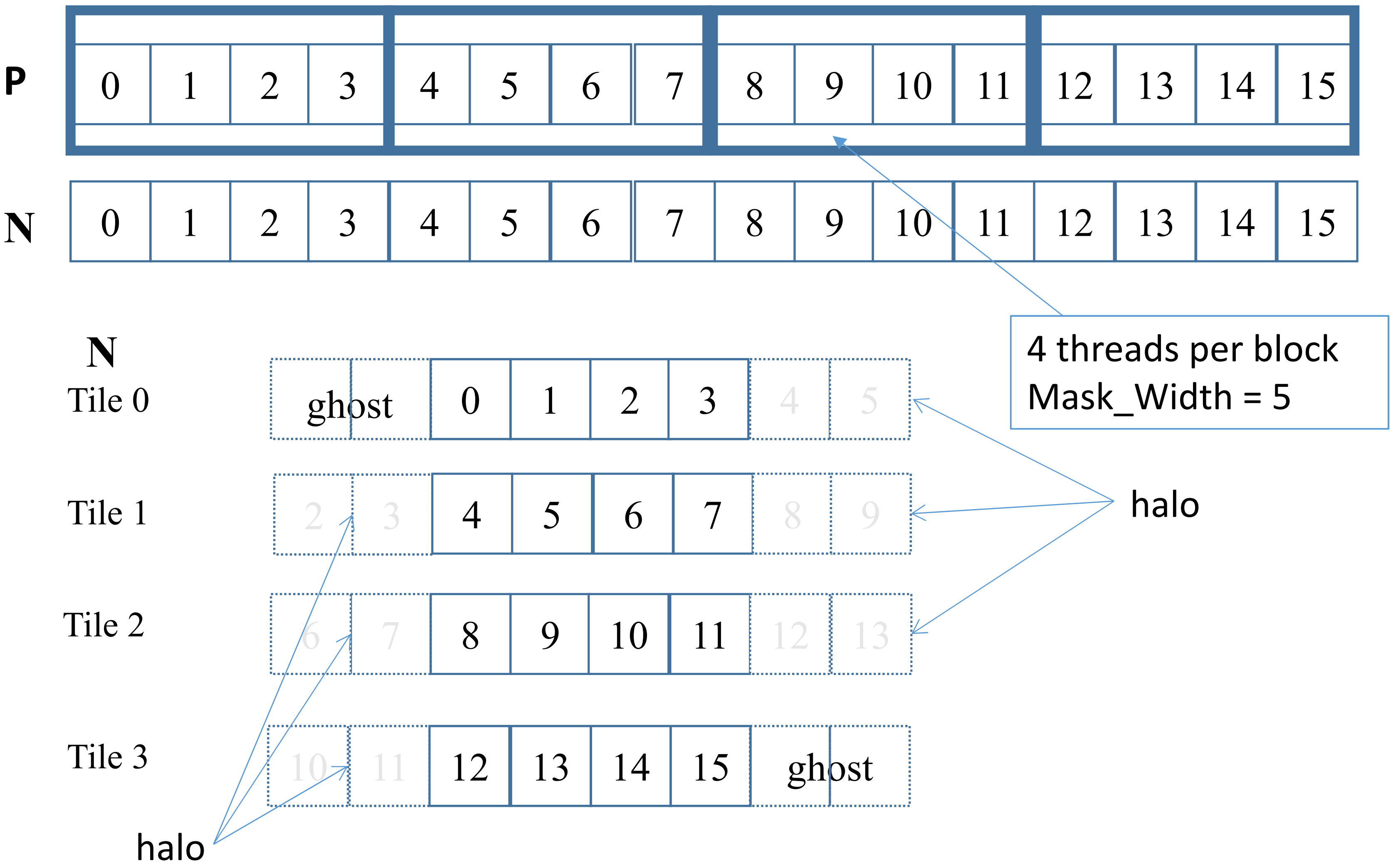
Host Code

```
 // global variable, outside any kernel/function
   __constant__ float Mc[MASK_WIDTH][MASK_WIDTH];
…
 // 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);
```

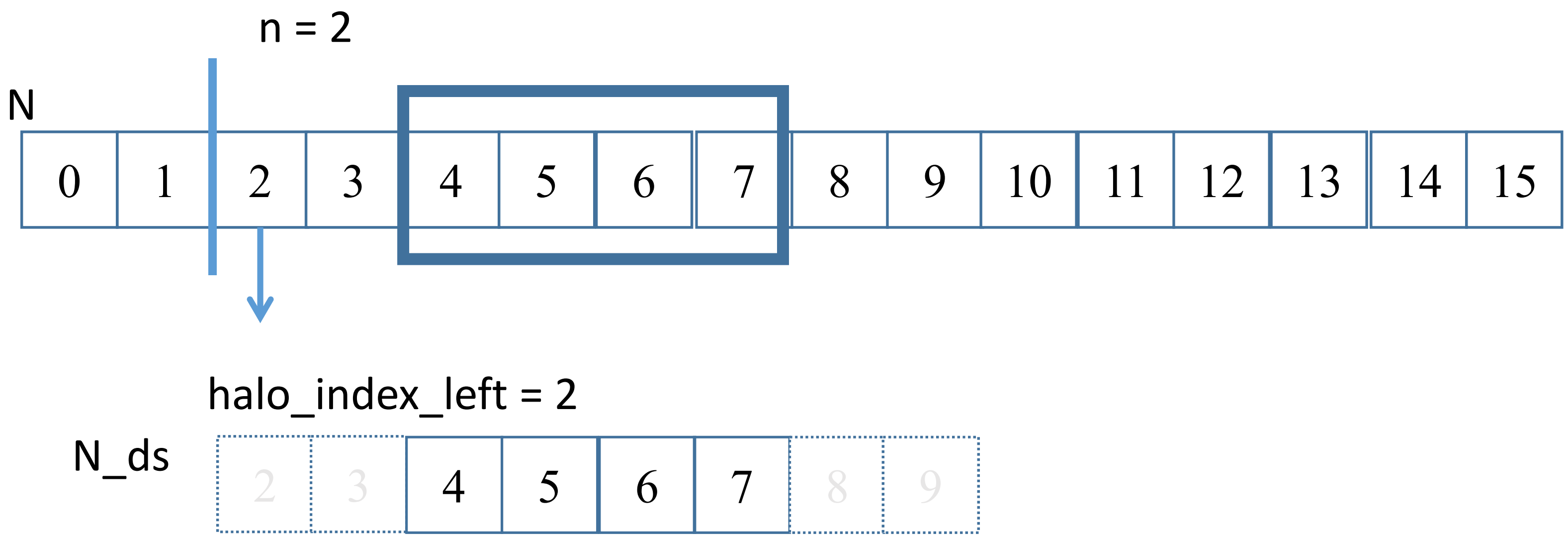
Content

- Convolution
  - An important parallel computation pattern
- Taking advance of
  - Constant memory and caching
- **Tiled 1D convolution algorithms**
  - Addressing memory bandwidth issue in accessing N
  - Algorithms and analysis
- Tiled 2D convolution with halo cells
  - Algorithms and analysis

Tiled 1D Convolution Basic Idea



Loading the left halo

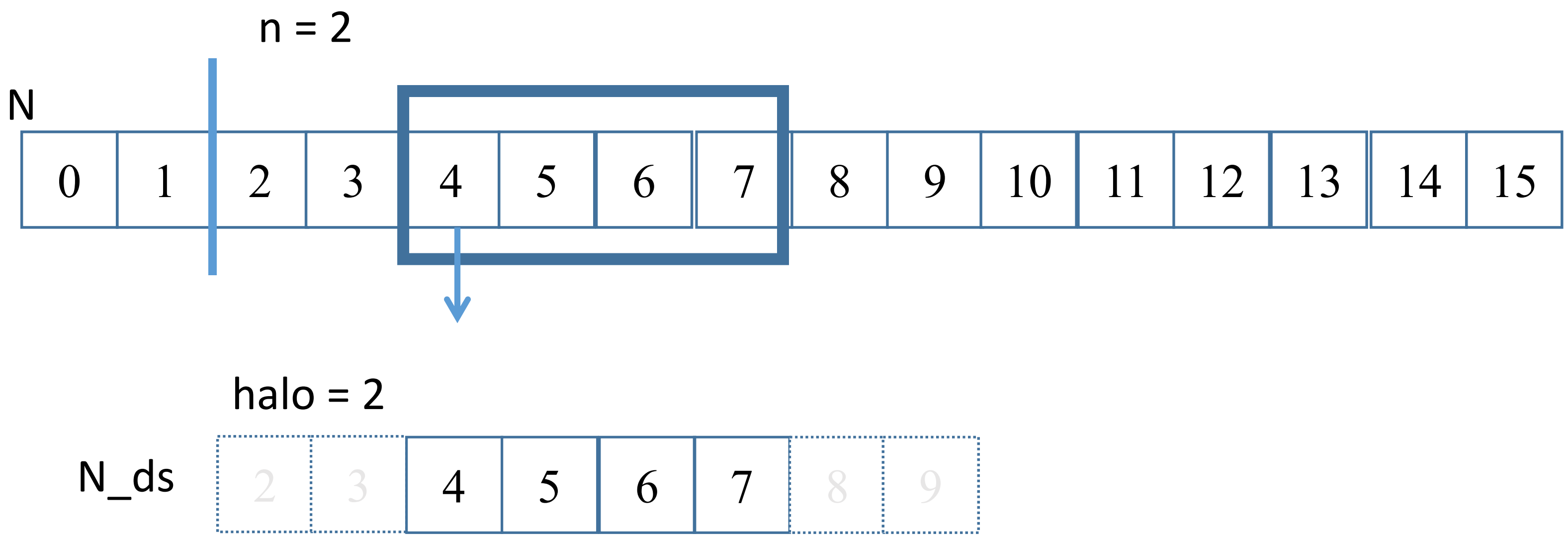


Previous tile

```
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];
  }
```

Last n

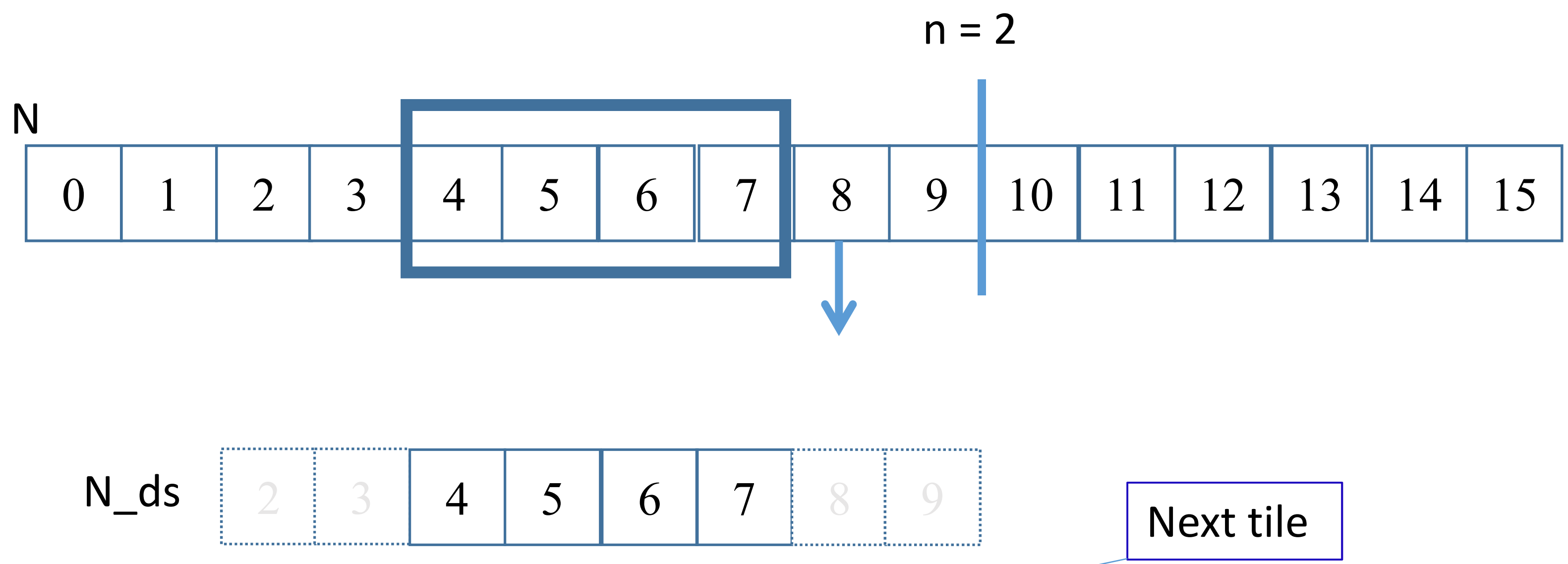
Loading the internal elements




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

Shift n, number of left halo

Loading the right halo




```
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];
  }
```

```
__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;
}
```

Shared Memory Data Reuse

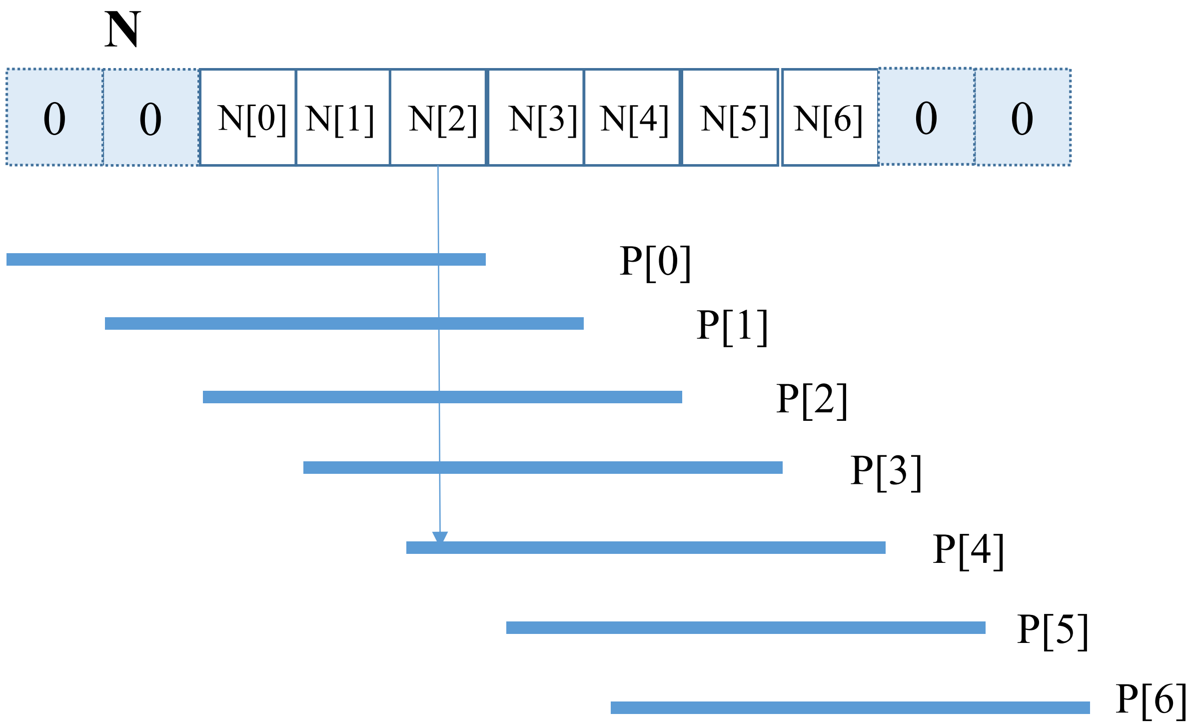
N_ds

2	3	4	5	6	7	8	9

Mask_Width is 5

- 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)

Shared Memory Data Reuse



A Small 1D Example

TILE_SIZE = 8, Mask_Width=5

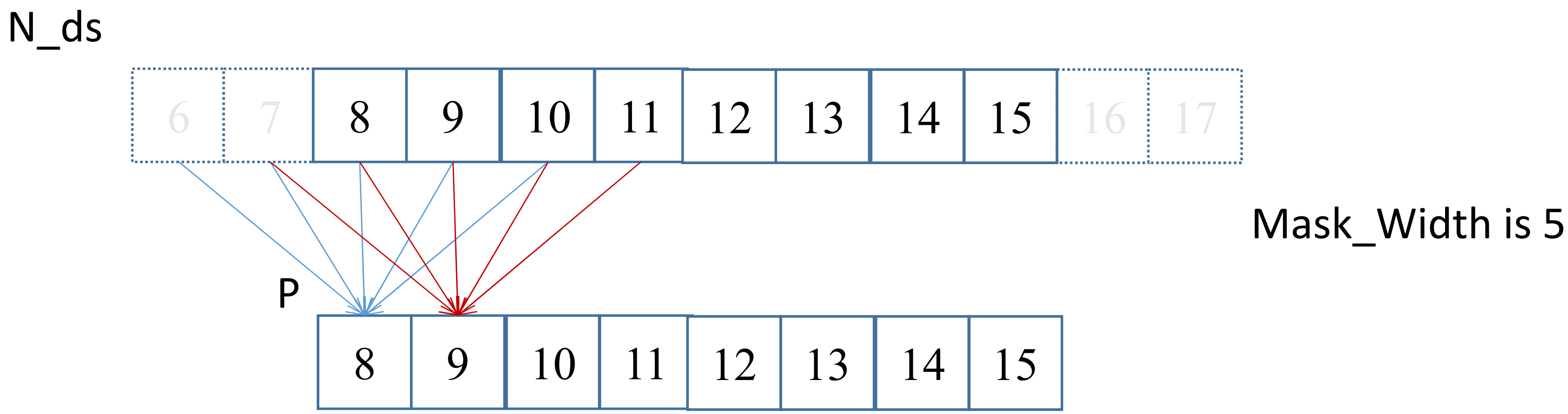
N_ds	6	7	8	9	10	11	12	13	14	15	16	17

Mask_Width is 5

P	8	9	10	11	12	13	14	15

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

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 N elements are used for the output tile.

A simple way to calculate tiling benefit

- For internal tiles:
  - (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
- For a boundary tile:
  - (8+(5-1)/2)=10 elements loaded
  - Total accesses is 40-3= 37

In General for 1D, internal tiles

- The total number of global memory accesses to the (TILE_SIZE+Mask_Width-1) N elements replaced by shared memory accesses is

1 + 2 + … + Mask_Width-1+ Mask_Width * (TILE_SIZE -Mask_Width+1) + Mask_Width-1+… + 2 + 1

= ((Mask_Width-1) *Mask_Width)/2+ Mask_Width*(TILE_SIZE-Mask_Width+1) + ((Mask_Width-1) *Mask_Width)/2

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

= **Mask_Width*(TILE_SIZE)**

In General, for 1D convolution kernel

- For internal thread blocks, the ratio of memory accesses between the basic and the tiled 1D kernel:

```
(blockDim.x*(2n+1)) / (blockDim.x+2n)
```

- whereas the ratio for boundary blocks is:

```
(blockDim.x*(2n+1) - n(n+1)/2) / (blockDim.x+n)
```

- For most situations, blockDim.x is much larger than n.

```
(blockDim.x*(2n+1)/ blockDim.x
= 2n+1 = Mask_Width
```

Bandwidth Reduction for 1D

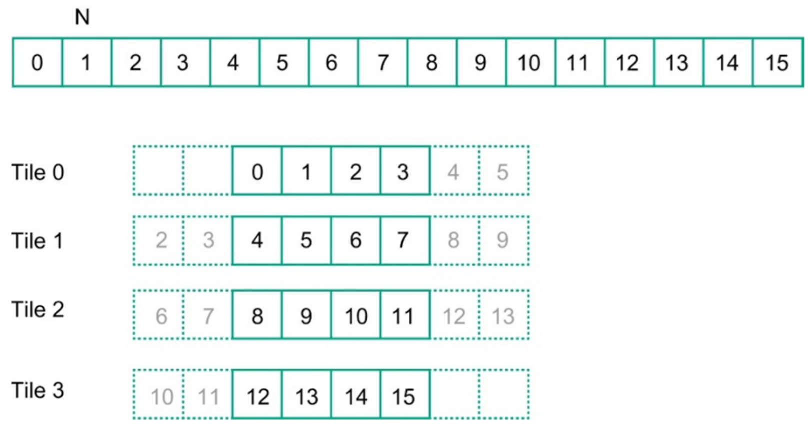
- The reduction is

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

Simpler Tiled 1D convolution

- Using general caching:
  - Modern GPU like Fermi has L1 and L2 caches:
  - L1 is private to each SM
  - L2 is shared among all SMs



- Blocks can take advantages of the fact that their halo cells maybe available in L2 cache.
- Simpler version that only load the internal elements of each tile into the shared memory.

```
__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;

}
```

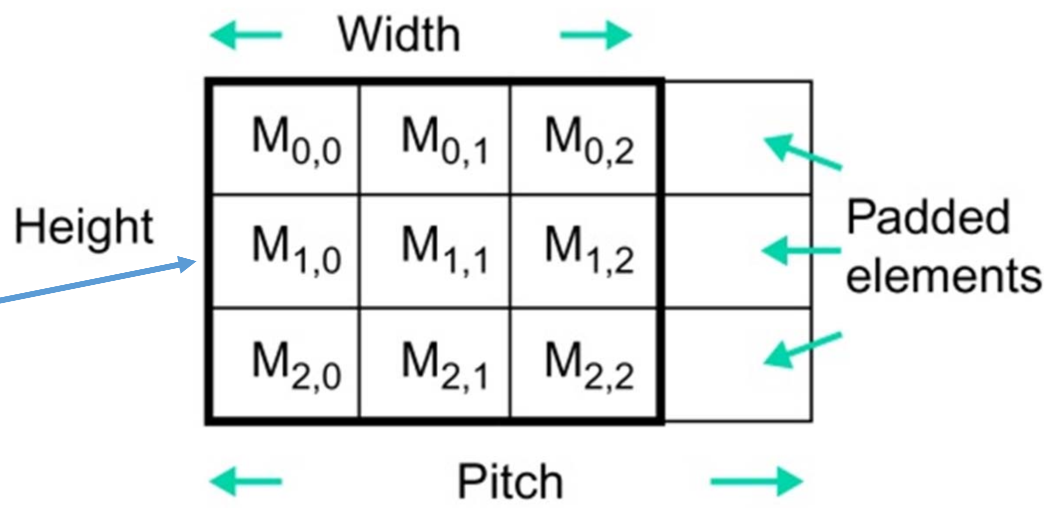
Content

- Convolution
  - An important parallel computation pattern
- Taking advance of
  - Constant memory and caching
- Tiled 1D convolution algorithms
  - Addressing memory bandwidth issue in accessing N
  - Algorithms and analysis
- **Tiled 2D convolution with halo cells**
  - **Algorithms and analysis**

Padded format

- Images come in all size and shapes, and stored in row-major layout when reading from files to memory.

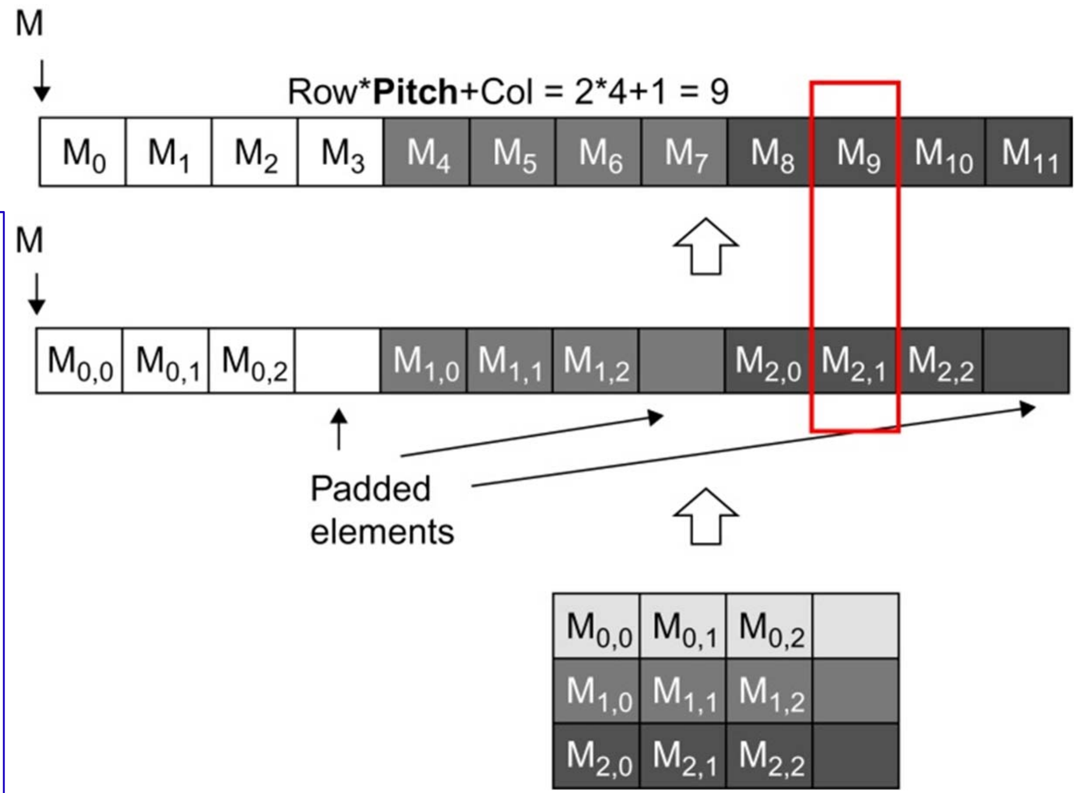
Take 4 pixel as a burst, elements in row 1 need two DRAM bursts to load.



- If the width of the image in terms of bytes is not a multiple of DRAM burst size, poor utilization of DRAM band-width.

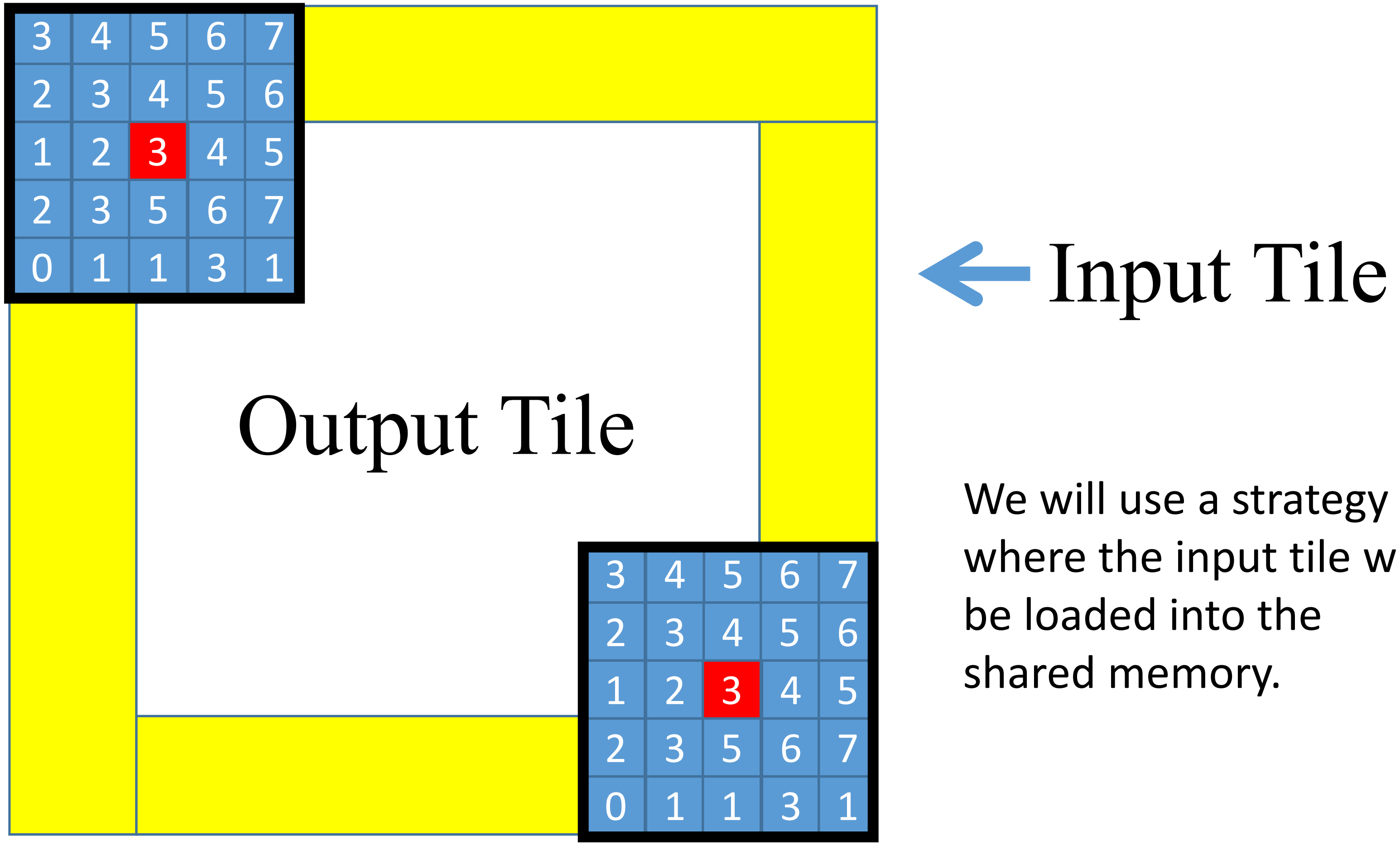
Padded format

```
//Image Matrix
declaration
typedef struct {
        int width;
        int height;
        int pitch;
        int channels;
        float* data;
} *wbImage_t;
```



Linearized 1D index = row * pitch + column

Input tiles need to be larger than output tiles.

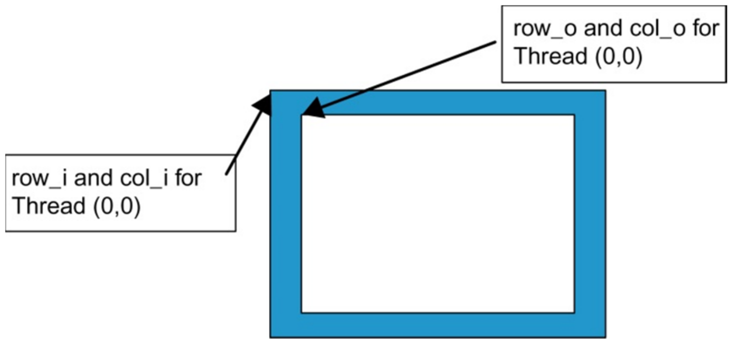


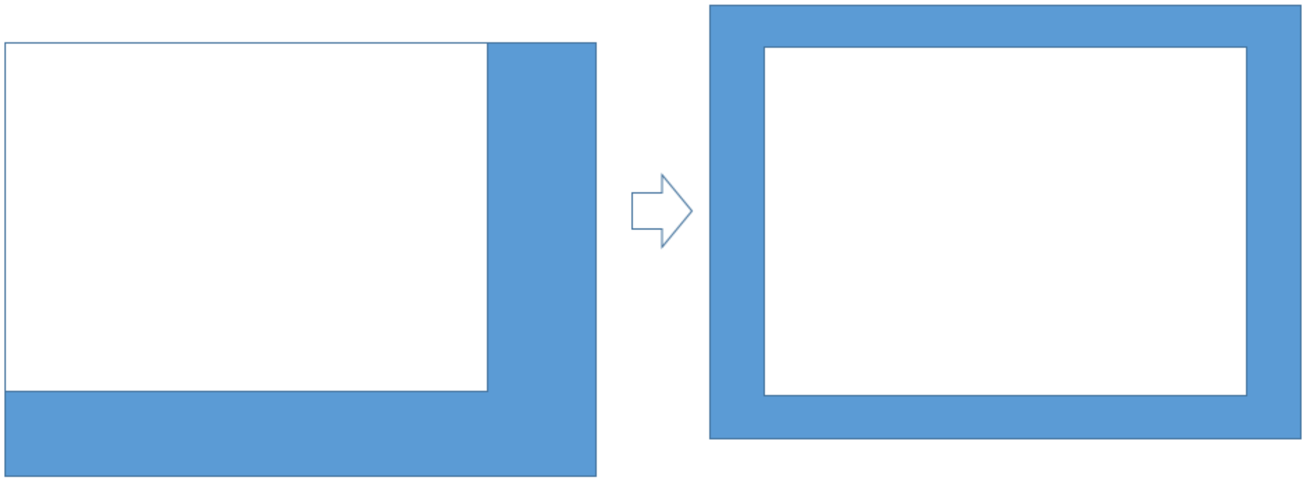
We will use a strategy where the input tile will be loaded into the shared memory.

Input v.s. Output tile

```
int tx = threadIdx.x;
int ty = threadIdx.y;
int row_o = blockIdx.y * O_TILE_SIZE + ty;
int col_o = blockIdx.x * O_TILE_SIZE + tx;

int row_i = row_o - Mask_Width / 2;
int col_i = col_o - Mask_Width / 2;
```





Input v.s. Output tile

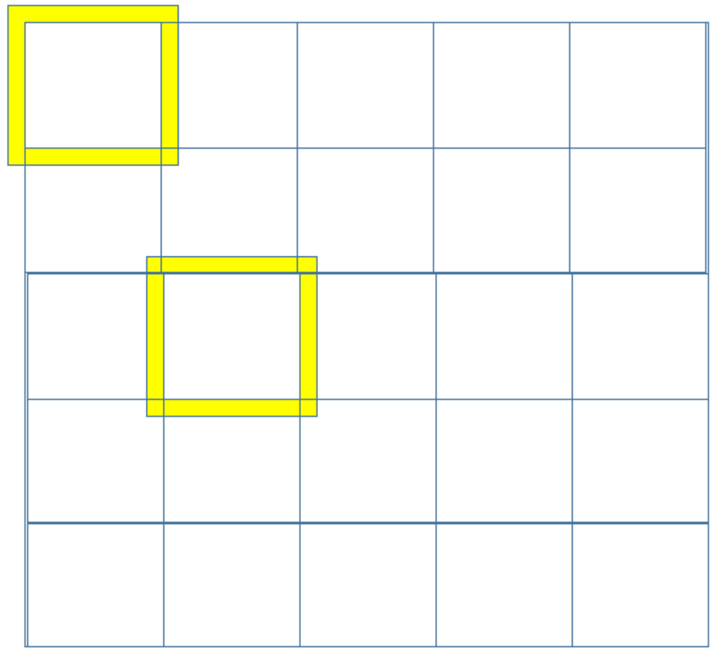
- Use a thread block that matches input tile
  - Each thread loads one element of the input tile
  - Some threads do not participate in calculating output

- There will be if statements and control divergence to make sure read in valid elements.

Load Input Tile

```
__shared__ float N_ds[TILE_SIZE +
MAX_MASK_WIDTH-1] [TILE_SIZE +
MAX_MASK_WIDTH-1]

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

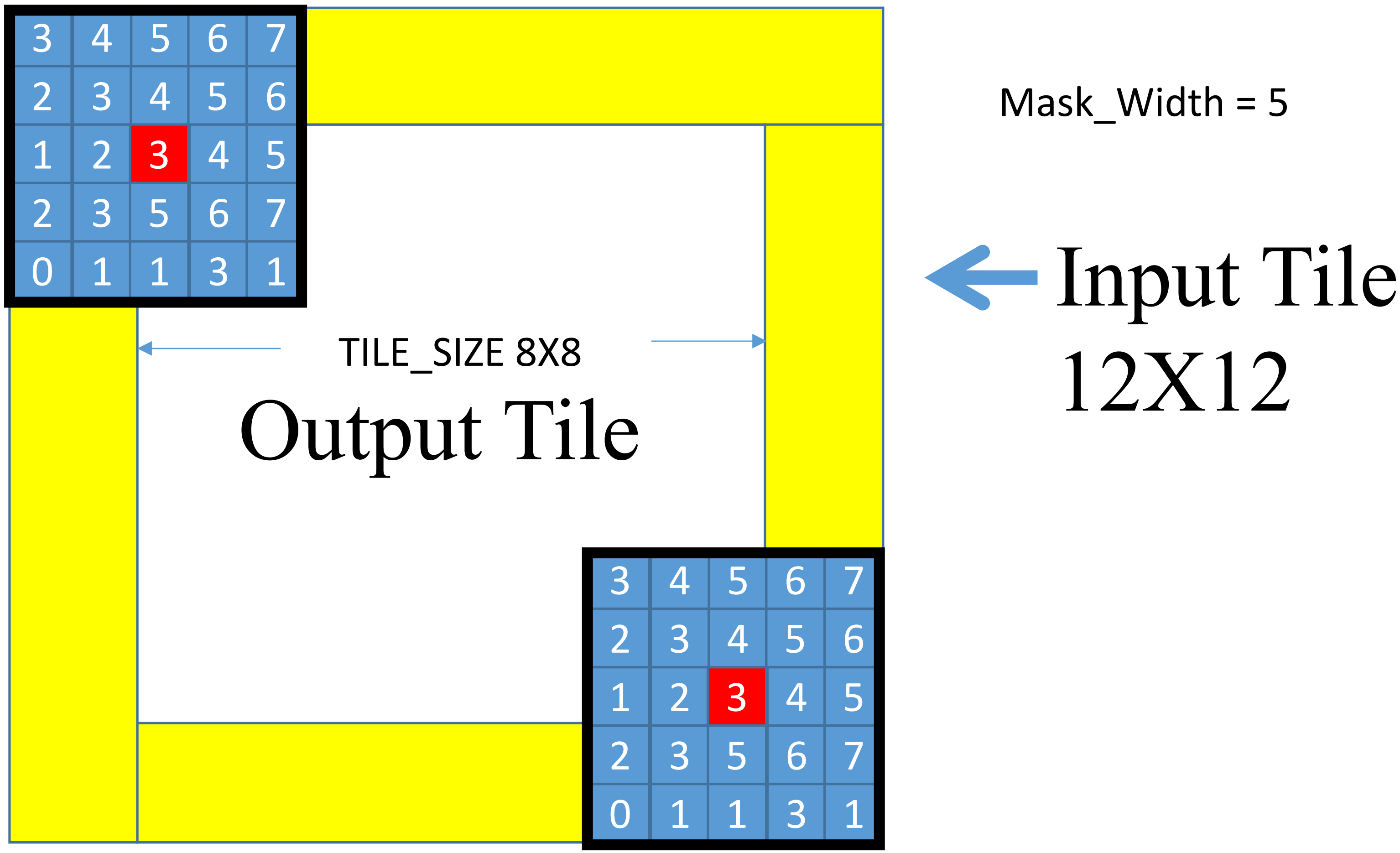


Threads that loads halos outside N should return 0.0

Some threads do not participate in calculating output.

```
float output = 0;
if(ty < O_TILE_SIZE && tx < O_TILE_SIZE)
{
    for(i = 0; i < MASK_WIDTH; i++) {
      for(j = 0; j < MASK_WIDTH; j++) {
        output += M[i][j] * N_ds[i+ty][j+tx];
       }
    }
    if(row_o < height && col_o < width)
       data[row_o * width + col_o] = output;
}
```

Analysis 2D Tile convolution



A Simple Analysis

- for a small 8X8 output tile example (Mask_Width =5)
  - 12X12=144 N elements need to be loaded into shared memory
  - The calculation of each P element needs to access 25 N elements
  - 8X8X25 = 1,600 global memory accesses are converted into shared memory accesses

- A reduction of 1,600/144 = 11X

In General

- Tiled :
  - $(O\_TILE\_SIZE+Mask\_Width-1)^2$ N elements need to be loaded into shared memory
- Basic:
  - The calculation of each P element needs to access $Mask\_Width^2$ N elements
  - $O\_TILE\_SIZE^2 * Mask\_Width^2$ global memory accesses are converted into shared memory accesses

- The reduction is

$$\boxed{O\_TILE\_SIZE^2 * Mask\_Width^2 / (O\_TILE\_SIZE+Mask\_Width-1)^2}$$

Bandwidth Reduction for 2D

- The reduction is

$$O\_TILE\_SIZE^2 * Mask\_Width^2 / (O\_TILE\_SIZE+Mask\_Width-1)^2$$

TILE_SIZE	8	16	32	64
Reduction Mask_Width = 5	11.1	16	19.7	22.1
Reduction Mask_Width = 9	20.3	36	51.8	64

9x9 = 81

Input size: 64+8 = 72;
Tile size: 72x72 = 5184 or 20,736 bytes,
larger than shared memory available.

Summary

- Taking advance of
  - Constant memory and caching
- Tiled convolution algorithms
  - Addressing memory bandwidth issue in accessing N