中国科学院大学计算机学院专业选修课（硕博通用课程）

# GPU架构与编程

## 第三课：CUDA编程（二）

赵地
中科院计算所
2024年秋季学期

## 讲授内容

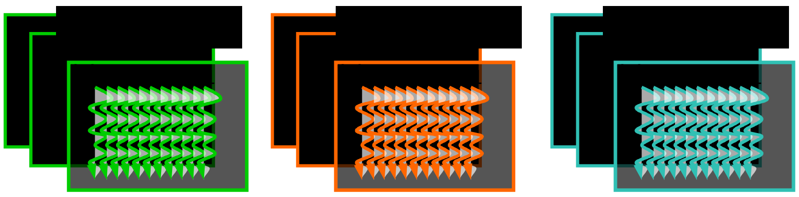
- Thread Execution Efficiency
- Memory Access Performance
- Parallel Computation Patterns (Stencil)

## 讲授内容：Thread Execution Efficiency

① Warps and SIMD Hardware

② Performance Impact of Control Divergence

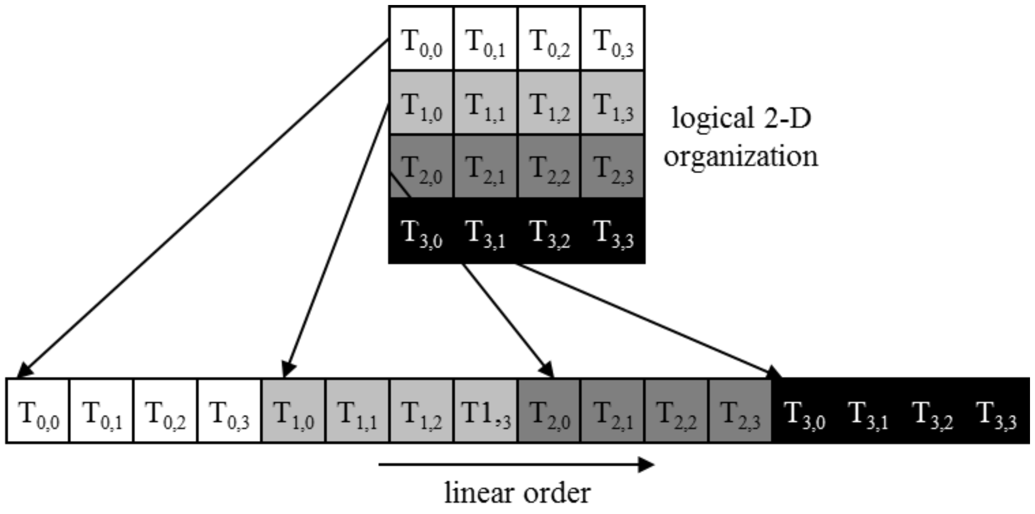
## Warps as Scheduling Units



– Each block is divided into 32-thread warps
  – An implementation technique, not part of the CUDA programming model
  – Warps are scheduling units in SM
  – Threads in a warp execute in Single Instruction Multiple Data (SIMD) manner
  – The number of threads in a warp may vary in future generations

## Warps in Multi-dimensional Thread Blocks

- The thread blocks are first linearized into 1D in row major order
  - In x-dimension first, y-dimension next, and z-dimension last $T_{row,col} = T_{y,x}$



$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$
$T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T_{1,3}$
$T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$
$T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
logical 2-D organization
$T_{0,0}$ $T_{0,1}$ $T_{0,2}$ $T_{0,3}$ $T_{1,0}$ $T_{1,1}$ $T_{1,2}$ $T1_{,3}$ $T_{2,0}$ $T_{2,1}$ $T_{2,2}$ $T_{2,3}$ $T_{3,0}$ $T_{3,1}$ $T_{3,2}$ $T_{3,3}$
linear order

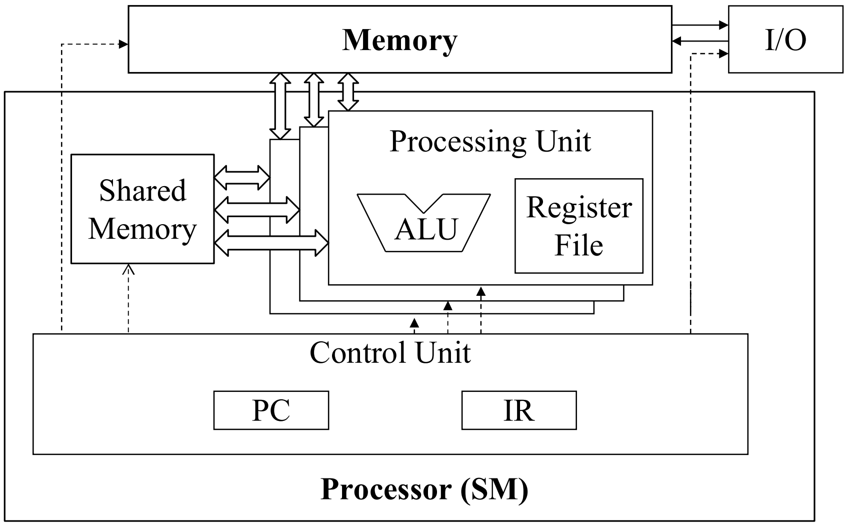
Figure 6.1: Placing 2D threads into linear order

## Blocks are partitioned after linearization

- Linearized thread blocks are partitioned
  - Thread indices within a warp are consecutive and increasing
  - Warp 0 starts with Thread 0
- Partitioning scheme is consistent across devices
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
- DO NOT rely on any ordering within or between warps
  - If there are any dependencies between threads, you must __syncthreads() to get correct results (more later).

## SMs are SIMD Processors

– Control unit for instruction fetch, decode, and control is shared among multiple processing units
  – Control overhead is minimized (Module 1)



## SIMD Execution Among Threads in a Warp

– All threads in a warp must execute the same instruction at any point in time

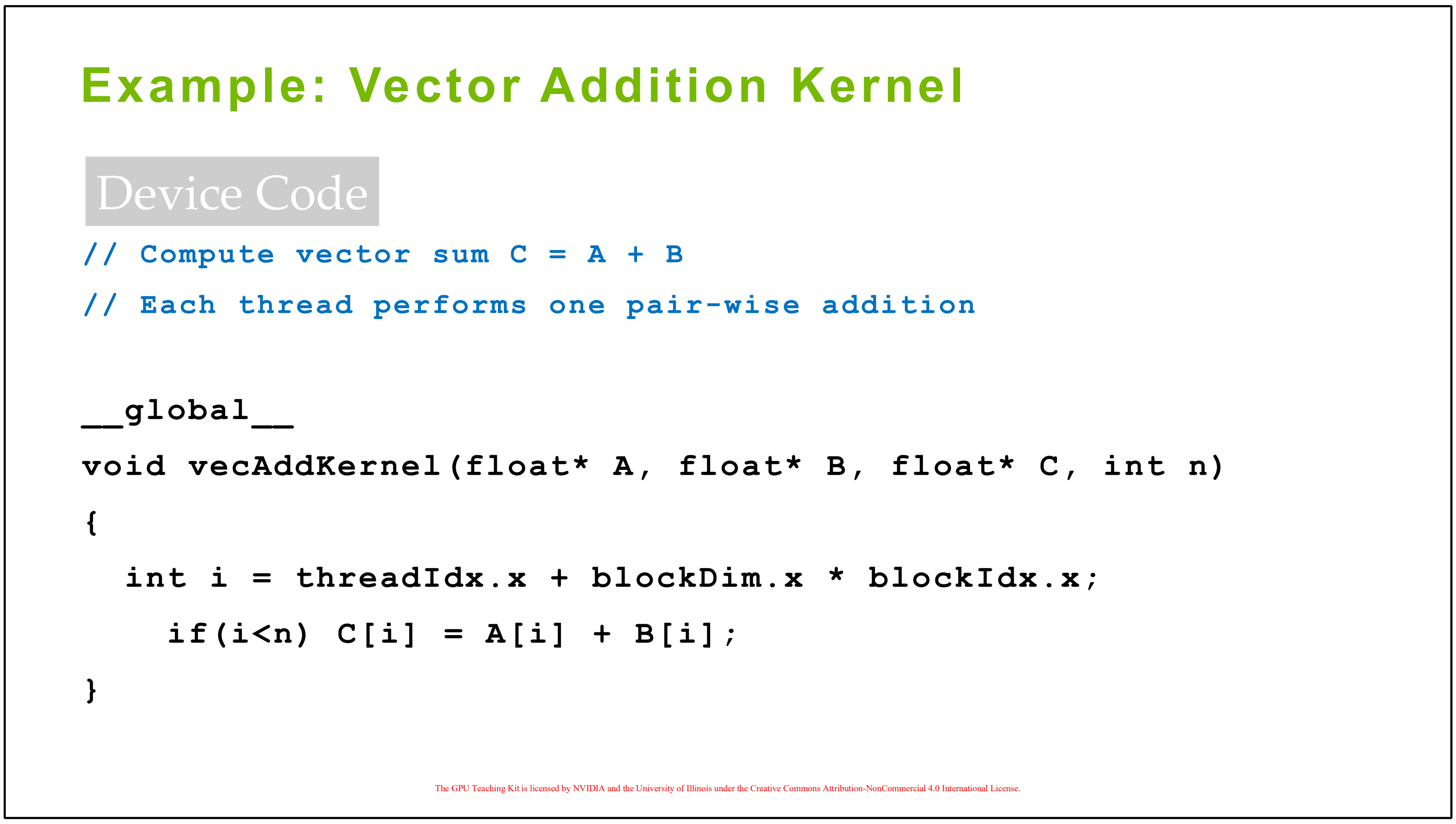
– This works efficiently if all threads follow the same control flow path
  – All if-then-else statements make the same decision
  – All loops iterate the same number of times

## Control Divergence

- Control divergence occurs when threads in a warp take different control flow paths by making different control decisions
  - Some take the then-path and others take the else-path of an if-statement
  - Some threads take different number of loop iterations than others
- The execution of threads taking different paths are serialized in current GPUs
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more
  - During the execution of each path, all threads taking that path will be executed in parallel
  - The number of different paths can be large when considering nested control flow statements

## Control Divergence Examples

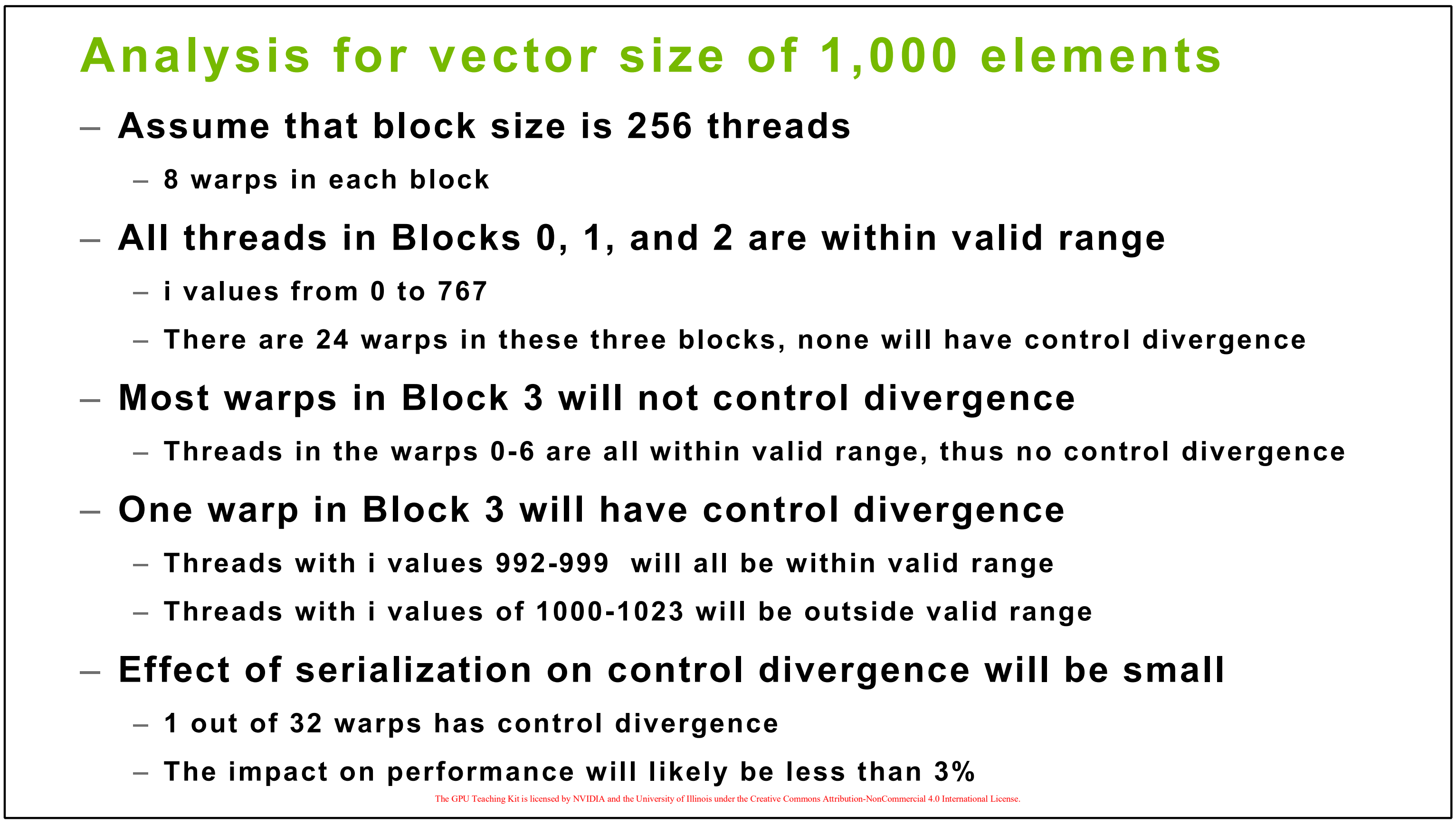
- Divergence can arise when branch or loop condition is a function of thread indices
- Example kernel statement with divergence:
  - if (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Decision granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
- Example without divergence:
  - If (blockIdx.x > 2) { }
  - Decision granularity is a multiple of blocks size; all threads in any given warp follow the same path



## Example: Vector Addition Kernel

Device Code

```
// Compute vector sum C = A + B
// Each thread performs one pair-wise addition

__global__
void vecAddKernel(float* A, float* B, float* C, int n)
{
  int i = threadIdx.x + blockDim.x * blockIdx.x;
    if(i<n) C[i] = A[i] + B[i];
}
```



## Analysis for vector size of 1,000 elements

- Assume that block size is 256 threads
  - 8 warps in each block
- All threads in Blocks 0, 1, and 2 are within valid range
  - i values from 0 to 767
  - There are 24 warps in these three blocks, none will have control divergence
- Most warps in Block 3 will not control divergence
  - Threads in the warps 0-6 are all within valid range, thus no control divergence
- One warp in Block 3 will have control divergence
  - Threads with i values 992-999 will all be within valid range
  - Threads with i values of 1000-1023 will be outside valid range
- Effect of serialization on control divergence will be small
  - 1 out of 32 warps has control divergence
  - The impact on performance will likely be less than 3%

## 讲授内容：Thread Execution Efficiency

① Warps and SIMD Hardware

② Performance Impact of Control Divergence

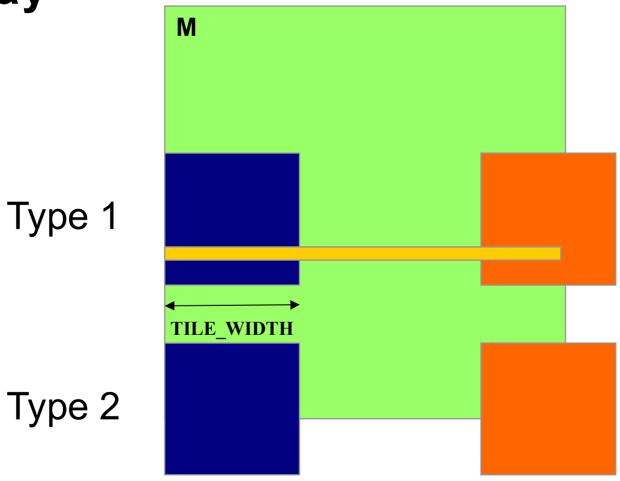
## Performance Impact of Control Divergence

- Boundary condition checks are vital for complete functionality and robustness of parallel code
  - The tiled matrix multiplication kernel has many boundary condition checks
  - The concern is that these checks may cause significant performance degradation
  - For example, see the tile loading code below:

```
if(Row < Width && t * TILE_WIDTH+tx < Width) {
      ds_M[ty][tx] = M[Row * Width + p * TILE_WIDTH + tx];
   } else {
    ds_M[ty][tx] = 0.0;
   }
   if (p*TILE_WIDTH+ty < Width && Col < Width) {
      ds_N[ty][tx] = N[(p*TILE_WIDTH + ty) * Width + Col];
   } else {
      ds_N[ty][tx] = 0.0;
   }
```

## Two types of blocks in loading M Tiles

- Type 1. Blocks whose tiles are all within valid range until the last phase.
- Type 2. Blocks whose tiles are partially outside the valid range all the way



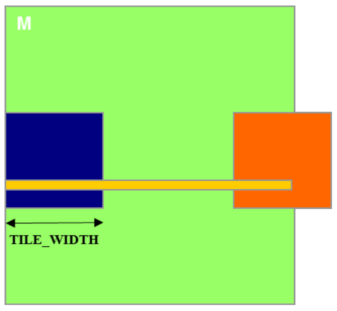
## Analysis of Control Divergence Impact

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each thread will go through 7 phases (ceiling of 100/16)
- There are 49 thread blocks (7 in each dimension)

## Control Divergence in Loading M Tiles

- Assume 16x16 tiles and thread blocks
- Each thread block has 8 warps (256/32)
- Assume square matrices of 100x100
- Each warp will go through 7 phases (ceiling of 100/16)
- There are 42 (6*7) Type 1 blocks, with a total of 336 (8*42) warps
- They all have 7 phases, so there are 2,352 (336*7) warp-phases
- The warps have control divergence only in their last phase
- 336 warp-phases have control divergence

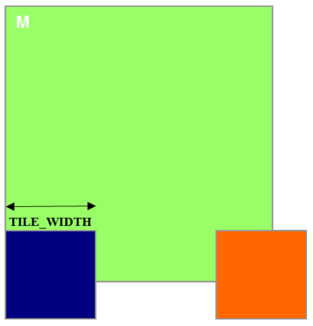
Type 1



## Control Divergence in Loading M Tiles (Type 2)

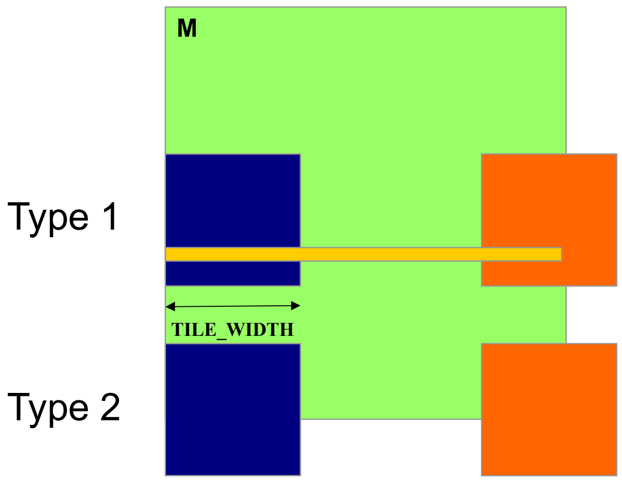
- Type 2: the 7 block assigned to load the bottom tiles, with a total of 56 (8*7) warps
- They all have 7 phases, so there are 392 (56*7) warp-phases
- The first 2 warps in each Type 2 block will stay within the valid range until the last phase
- The 6 remaining warps stay outside the valid range
- So, only 14 (2*7) warp-phases have control divergence

Type 2



## Overall Impact of Control Divergence

- Type 1 Blocks: 336 out of 2,352 warp-phases have control divergence
- Type 2 Blocks: 14 out of 392 warp-phases have control divergence
- The performance impact is expected to be less than 12% (350/2,944 or (336+14)/(2352+14))



## Additional Comments

- The calculation of impact of control divergence in loading N tiles is somewhat different and is left as an exercise
- The estimated performance impact is data dependent.
  - For larger matrices, the impact will be significantly smaller
- In general, the impact of control divergence for boundary condition checking for large input data sets should be insignificant
  - One should not hesitate to use boundary checks to ensure full functionality
- The fact that a kernel is full of control flow constructs does not mean that there will be heavy occurrence of control divergence
- We will cover some algorithm patterns that naturally incur control divergence (such as parallel reduction) in the Parallel Algorithm Patterns modules

## 讲授内容

- ➢Thread Execution Efficiency
- ➢Memory Access Performance
- ➢Parallel Computation Patterns (Stencil)

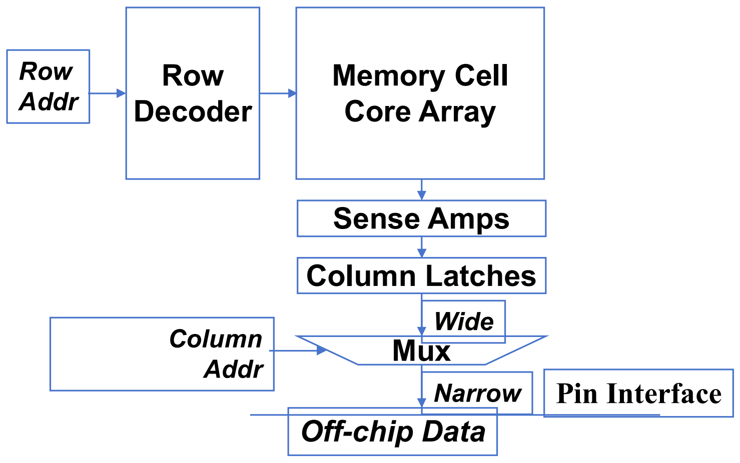
## 讲授内容： Memory Access Performance

① **DRAM Bandwidth**

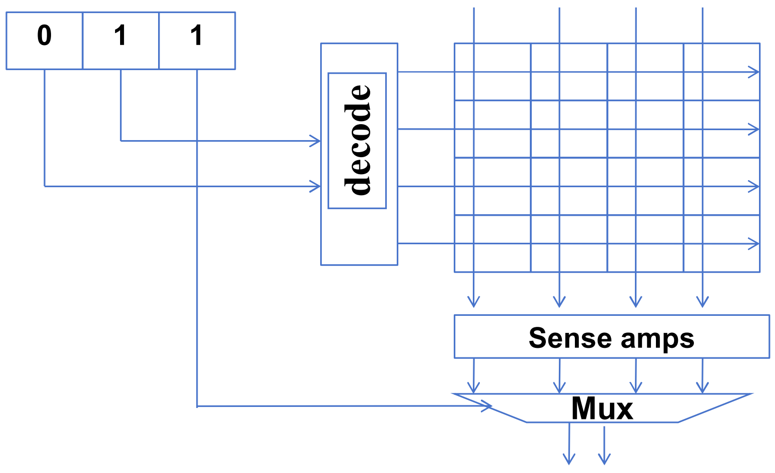
② **Memory Coalescing in CUDA**

## DRAM Core Array Organization

- Each DRAM core array has about 16M bits
- Each bit is stored in a tiny capacitor made of one transistor

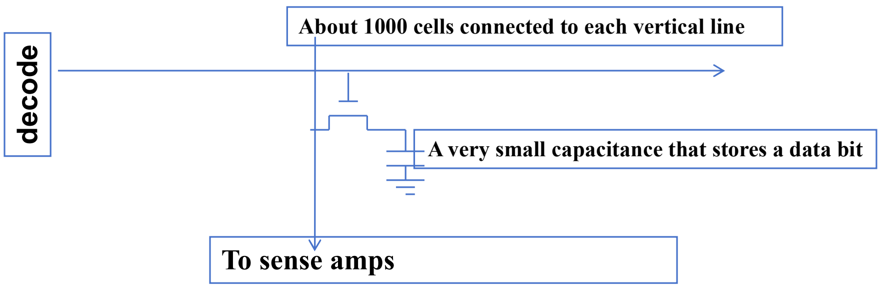


## A very small (8x2-bit) DRAM Core Array



## DRAM Core Arrays are Slow

- Reading from a cell in the core array is a very slow process
  - DDR: Core speed = ½ interface speed
  - DDR2/GDDR3: Core speed = ¼ interface speed
  - DDR3/GDDR4: Core speed = ⅛ interface speed
  - … likely to be worse in the future

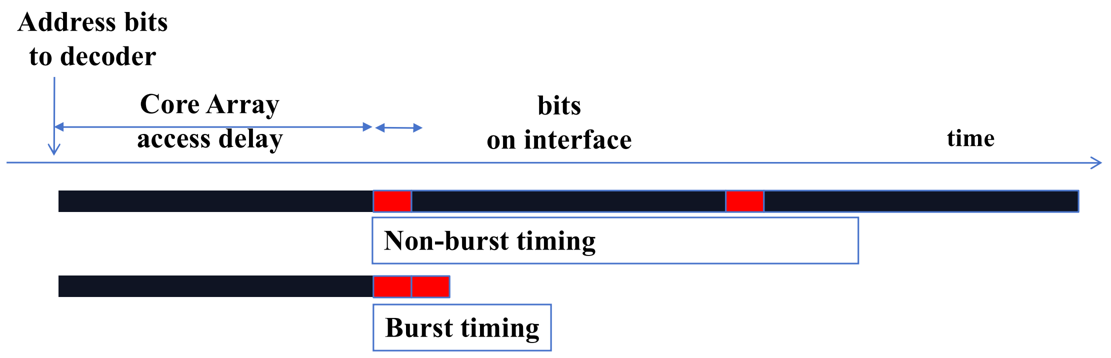


decode
About 1000 cells connected to each vertical line
A very small capacitance that stores a data bit
To sense amps

## DRAM Bursting

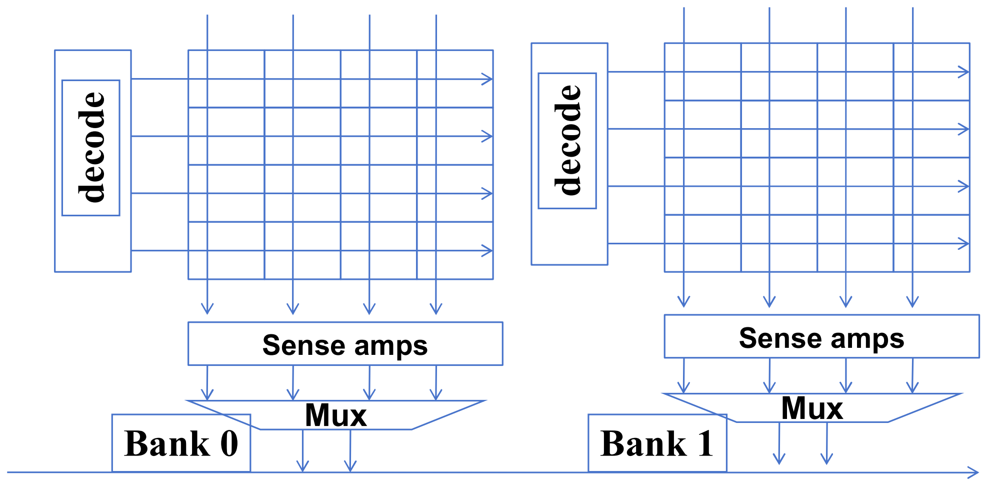
- For DDR{2,3} SDRAM cores clocked at 1/N speed of the interface:
  - Load (N × interface width) of DRAM bits from the same row at once to an internal buffer, then transfer in N steps at interface speed
  - DDR3/GDDR4: buffer width = 8X interface width

## DRAM Bursting Timing Example

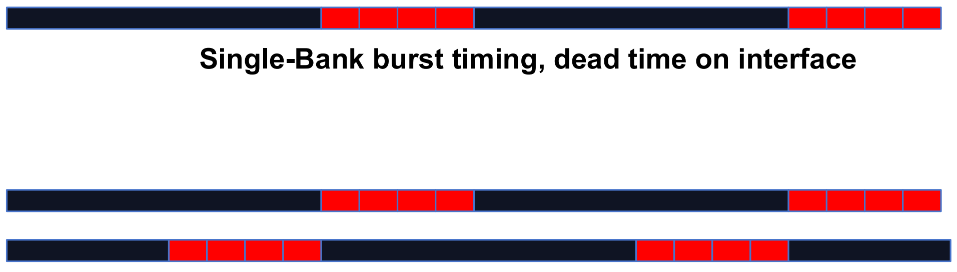


**Modern DRAM systems are designed to always be accessed in burst mode. Burst bytes are transferred to the processor but discarded when accesses are not to sequential locations.**

## Multiple DRAM Banks



## DRAM Bursting with Banking



Single-Bank burst timing, dead time on interface

Multi-Bank burst timing, reduced dead time

## GPU off-chip memory subsystem

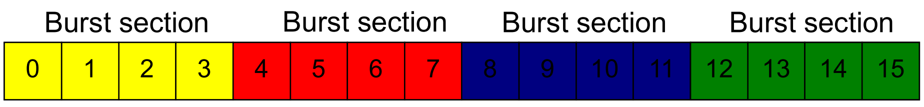
- NVIDIA RTX6000 GPU:
  - Peak global memory bandwidth = 672GB/s
- Global memory (GDDR6) interface @ 7GHz
  - 14 Gbps pin speed
  - For GDDR6 32-bit interface, we can sustain only about 56 GB/s
  - We need a lot more bandwidth (672 GB/s) – thus 12 memory channels

## 讲授内容：Memory Access Performance

① **DRAM Bandwidth**

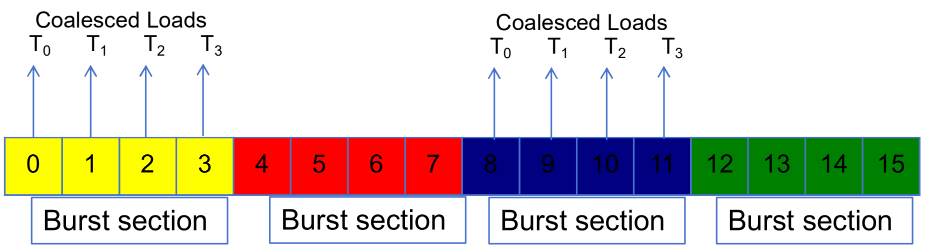
② **Memory Coalescing in CUDA**

## DRAM Burst – A System View



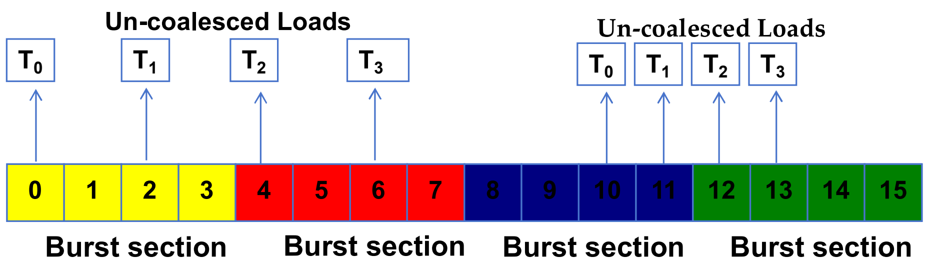
- **Each address space is partitioned into burst sections**
  - **Whenever a location is accessed, all other locations in the same section are also delivered to the processor**
- **Basic example: a 16-byte address space, 4-byte burst sections**
  - **In practice, we have at least 4GB address space, burst section sizes of 128-bytes or more**

## Memory Coalescing



- When all threads of a warp execute a load instruction, if all accessed locations fall into the same burst section, only one DRAM request will be made and the access is fully coalesced.

## Un-coalesced Accesses



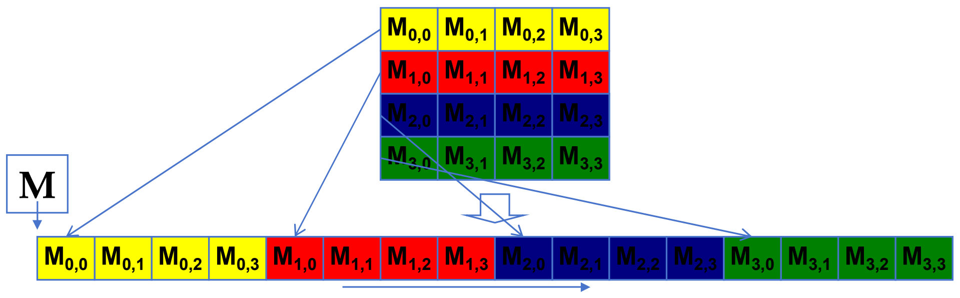
- When the accessed locations spread across burst section boundaries:
  - Coalescing fails
  - Multiple DRAM requests are made
  - The access is not fully coalesced.
- Some of the bytes accessed and transferred are not used by the threads

## How to judge if an access is coalesced?

– Accesses in a warp are to consecutive locations if the index in an array access is in the form of

```
A[(expression with terms
independent of threadIdx.x) +
threadIdx.x]
```

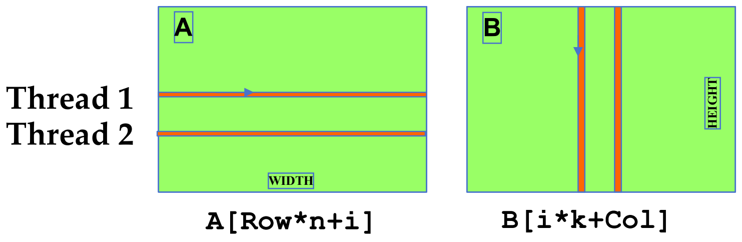
## A 2D C Array in Linear Memory Space



M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{0,3}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{1,3}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$ $M_{2,3}$
$M_{3,0}$ $M_{3,1}$ $M_{3,2}$ $M_{3,3}$

linearized order in increasing address

## Two Access Patterns of Basic Matrix Multiplication

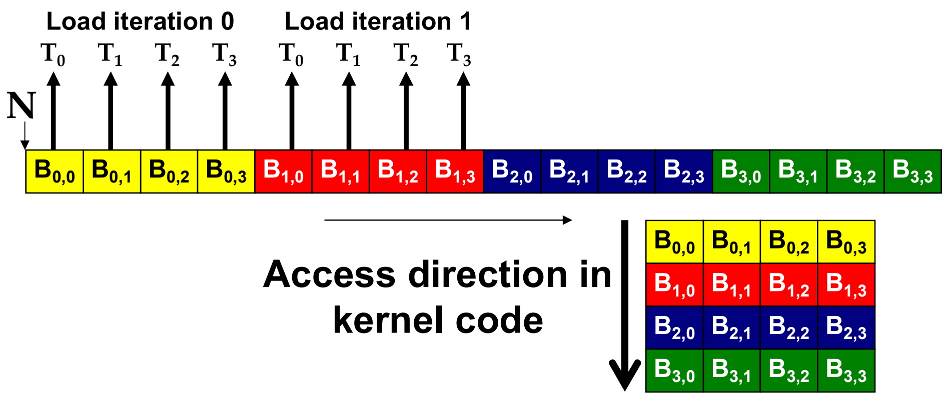


***i*** **is the loop counter in the inner product loop of the kernel code**

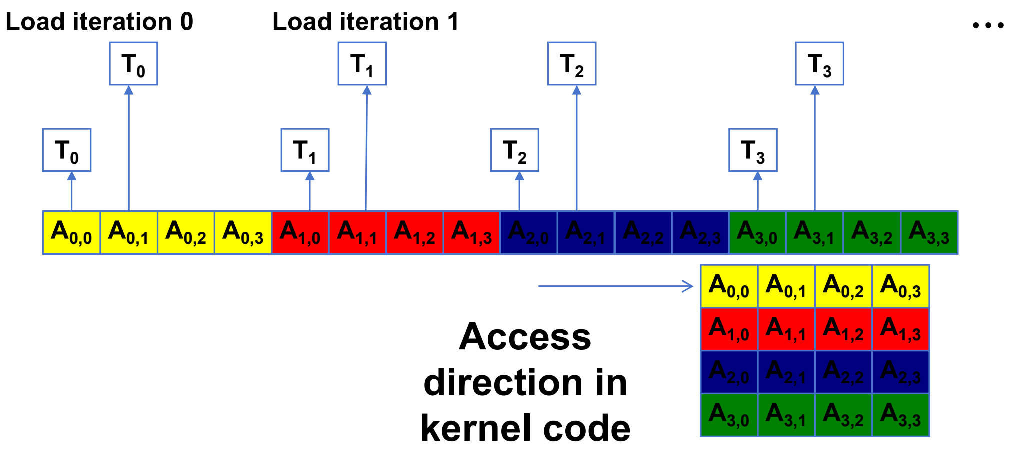
**A is m × n, B is n × k**

```
Col = blockIdx.x*blockDim.x + threadIdx.x
```

## B accesses are coalesced


Load iteration 0
Load iteration 1
$T_0$ $T_1$ $T_2$ $T_3$
$T_0$ $T_1$ $T_2$ $T_3$
N
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$ $B_{0,3}$ $B_{1,0}$ $B_{1,1}$ $B_{1,2}$ $B_{1,3}$ $B_{2,0}$ $B_{2,1}$ $B_{2,2}$ $B_{2,3}$ $B_{3,0}$ $B_{3,1}$ $B_{3,2}$ $B_{3,3}$
Access direction in kernel code

## A Accesses are Not Coalesced



Load iteration 0
Load iteration 1
…
$T_0$ $T_0$ $T_1$ $T_1$ $T_2$ $T_2$ $T_3$ $T_3$
$A_{0,0}$ $A_{0,1}$ $A_{0,2}$ $A_{0,3}$ $A_{1,0}$ $A_{1,1}$ $A_{1,2}$ $A_{1,3}$ $A_{2,0}$ $A_{2,1}$ $A_{2,2}$ $A_{2,3}$ $A_{3,0}$ $A_{3,1}$ $A_{3,2}$ $A_{3,3}$
Access direction in kernel code

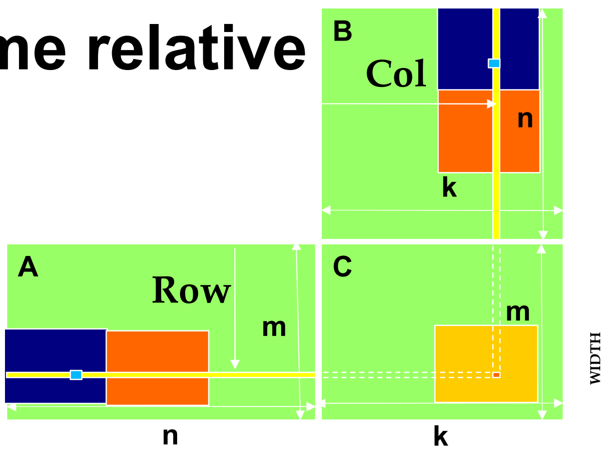
## Loading an Input Tile

**Have each thread load an A element and a B element at the same relative position as its C element.**

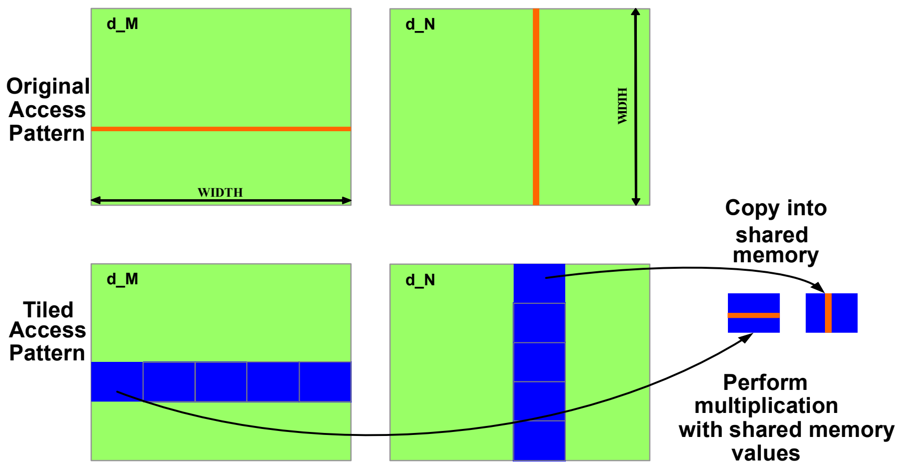
```
int tx = threadIdx.x
int ty = threadIdx.y
```

**Accessing tile 0 2D indexing:**

```
    A[Row][tx]
    B[ty][Col]
```



## Corner Turning



## 讲授内容

- ➢Thread Execution Efficiency
- ➢Memory Access Performance
- ➢Parallel Computation Patterns (Stencil)

## 讲授内容：Parallel Computation Patterns (Stencil)

① Convolution

② Tiled Convolution

③ Tile Boundary Conditions

④ Analyzing Data Reuse in Tiled Convolution

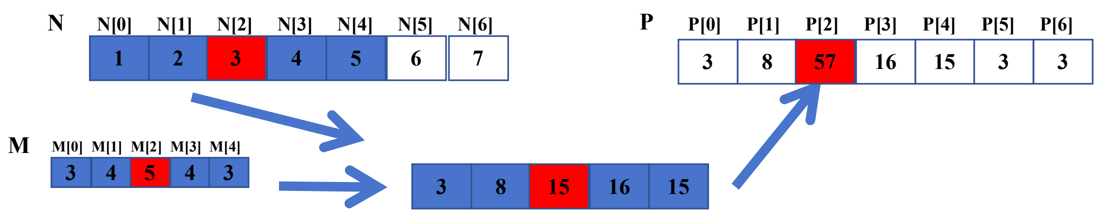
## Convolution as a Filter

– Often performed as a filter that transforms signal or pixel values 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 – a computational definition

- 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*
  - We will refer to these mask arrays as convolution masks to avoid confusion.
  - The value pattern of the mask array elements defines the type of filtering done
  - Our image blur example in Module 3 is a special case where all mask elements are of the same value and hard coded into the source code.

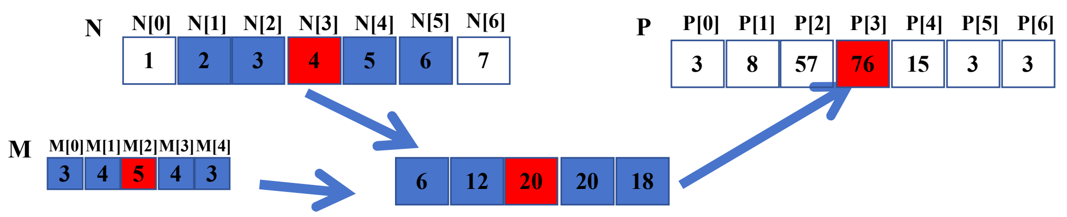
## 1D Convolution Example



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

```
P[2] = N[0]*M[0] + N[1]*M[1] + N[2]*M[2] + N[3]*M[3] +
N[4]*M[4]
```

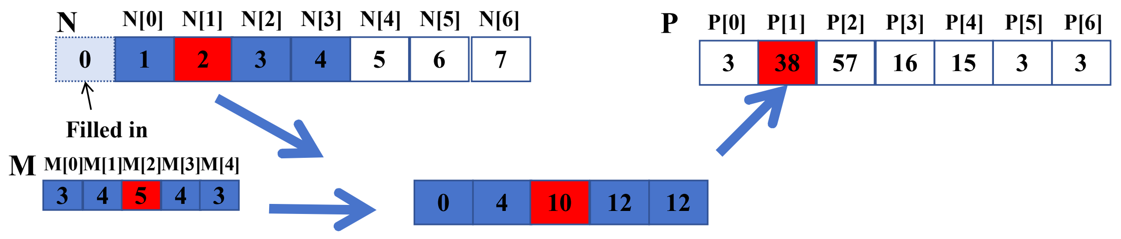
## Calculation of P[3]



– The figure shows calculation of P[3]

```
P[3] = N[1]*M[0] + N[2]*M[1] + N[3]*M[2] + N[4]*M[3] +
N[6]*M[4]
```

## Convolution Boundary Condition



– Calculation of output elements near the boundaries (beginning and end) of the 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 input range to 0**

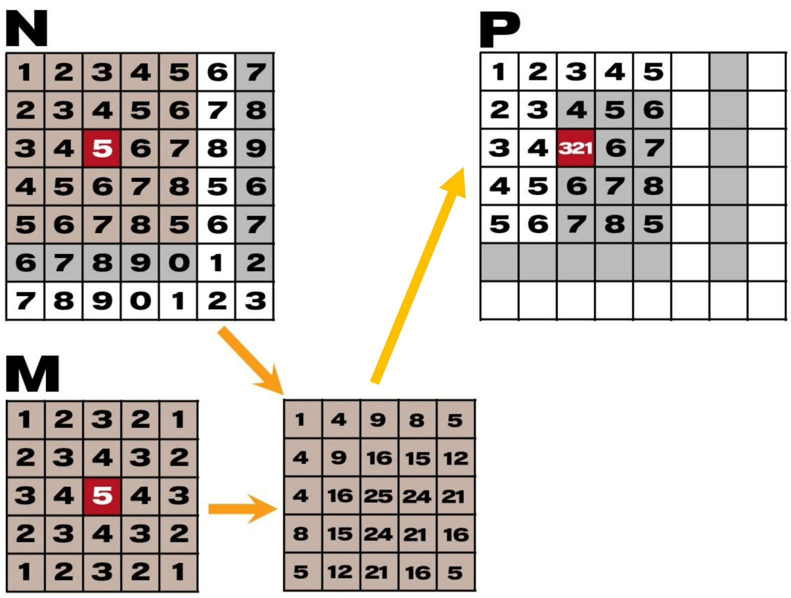
```
__global__ void convolution_1D_basic_kernel(float *N, float *M,
    float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    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;
}
```

## A 1D Convolution Kernel with Boundary Condition Handling

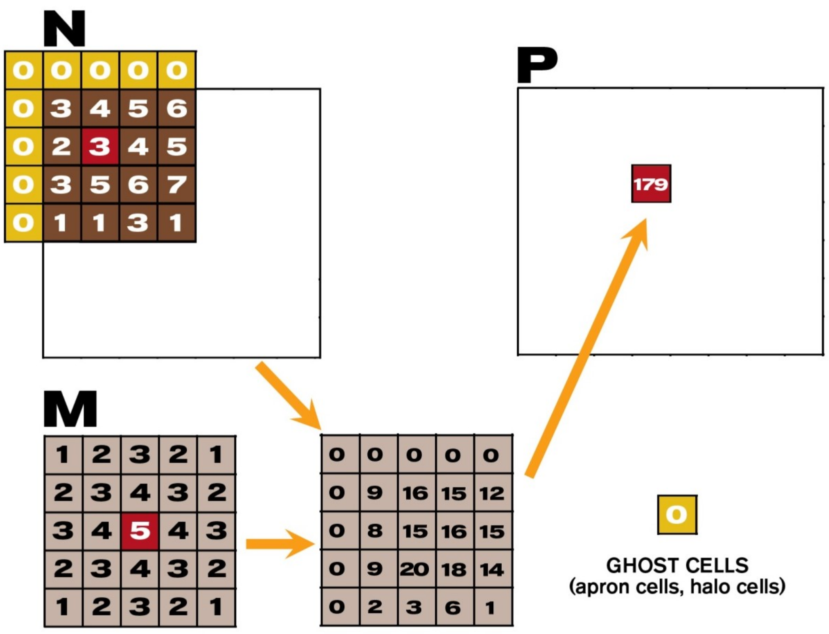
– **This kernel forces all elements outside the valid input range to 0**

```
__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);
    if (i < Width) {
       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;
     }
}
```

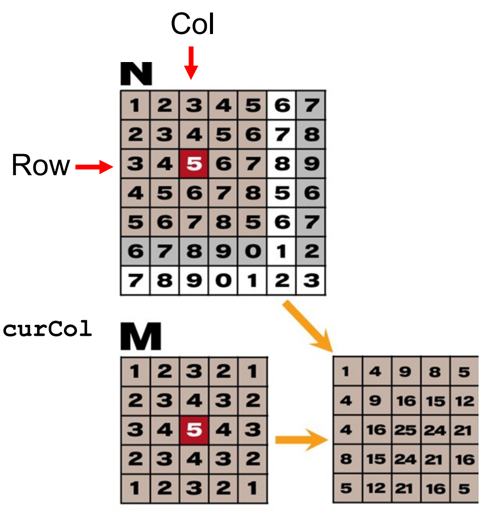
## 2D Convolution



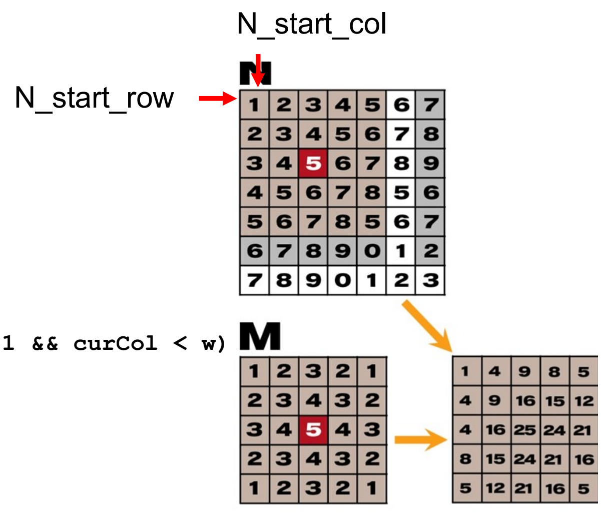
## 2D Convolution – Ghost Cells



```
 __global__
  void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask,
unsigned char * out, int maskwidth, int w, int h) {
      int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
      int Row  = blockIdx.y * blockDim.y + threadIdx.y;

      if (Col < w && Row < h) {
          int pixVal = 0;
          N_start_col  = Col -   (maskwidth/2);
          N_start_row = Row - (maskwidth/2);
          // Get the of the surrounding box
          for(int j = 0; j < maskwidth; ++j) {
              for(int k = 0; k < maskwidth; ++k) {
                  int curRow = N_Start_row + j;
                  int curCol =   N_start_col  + k;
                  // Verify we have a valid image pixel
                  if(curRow > -1 && curRow < h && curCol > -1 && curCol
{
                      pixVal += in[curRow * w + curCol] *
mask[j*maskwidth+k];
                  }
              }
          }
          // Write our new pixel value out
          out[Row * w + Col] = (unsigned char)(pixVal);
      }
  }
```



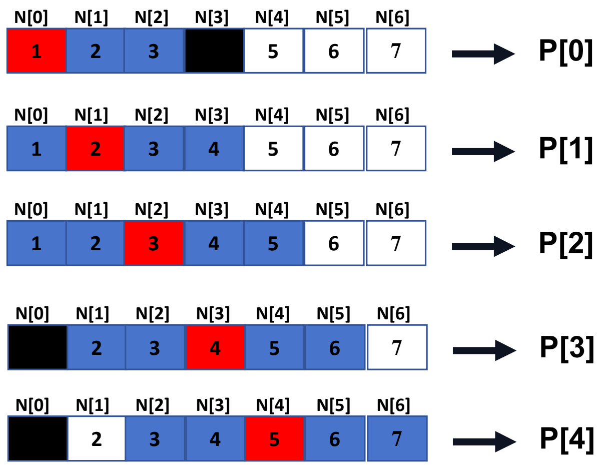
```
  __global__
   void convolution_2D_basic_kernel(unsigned char * in, unsigned char * mask,
 unsigned char * out,
         int maskwidth, int w, int h) {
       int Col  =   blockIdx.x * blockDim.x + threadIdx.x;
       int Row  = blockIdx.y * blockDim.y + threadIdx.y;

       if (Col < w && Row < h) {
           int pixVal = 0;

           N_start_col  = Col -   (maskwidth/2);
           N_start_row = Row - (maskwidth/2);
           // Get the of the surrounding box
           for(int j = 0; j < maskwidth; ++j) {
               for(int k = 0; k < maskwidth; ++k) {
                   int curRow = N_Start_row + j;
                   int curCol =   N_start_col  + k;
                   // Verify we have a valid image pixel
                   if(curRow > -1 && curRow < h && curCol > -1 && curCol < w)
 {
                       pixVal += in[curRow * w + curCol] *
 mask[j*maskwidth+k];
                   }
               }
           }
           // Write our new pixel value out
           out[Row * w + Col] = (unsigned char)(pixVal);
       }
   }
```



## 讲授内容：Parallel Computation Patterns (Stencil)

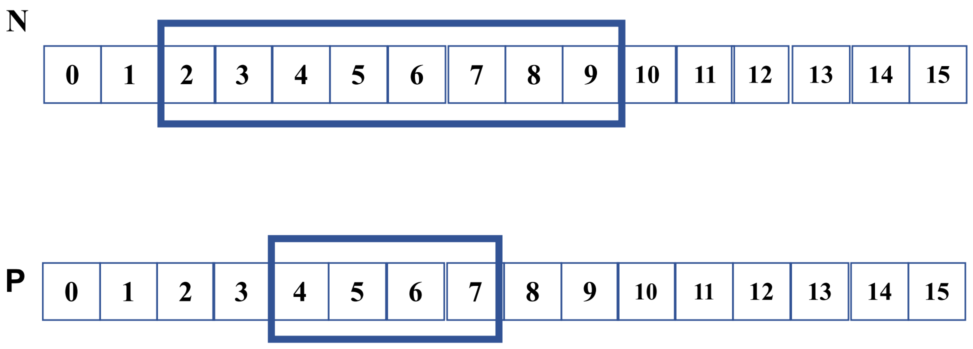
## Tiling Opportunity Convolution

– Calculation of adjacent output elements involve shared input elements
  – E.g., N[2] is used in calculation of P[0], P[1], P[2], P[3], P[4] and P[5] assuming a 1D convolution Mask_Width of width 5
– We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses

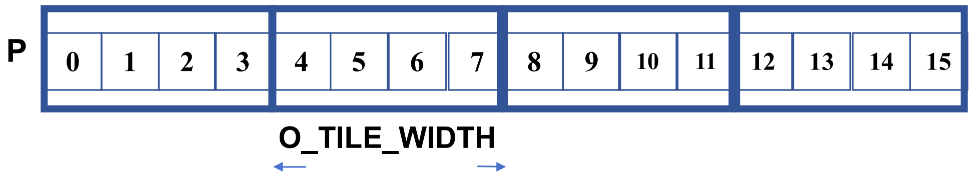


## Input Data Needs

- Assume that we want to have each block to calculate T output elements
  - T + Mask_Width -1 input elements are needed to calculate T output elements
  - T + Mask_Width -1 is usually not a multiple of T, except for small T values
  - T is usually significantly larger than Mask_Width



## Definition – output tile



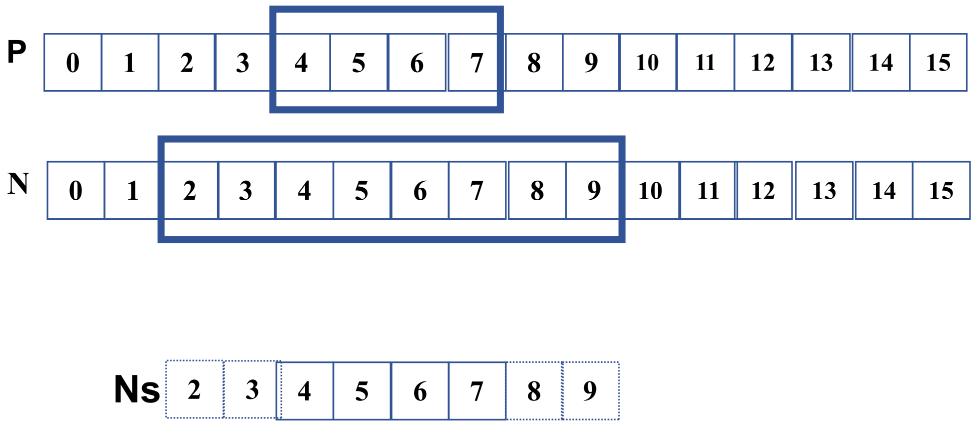
Each thread block calculates an output tile

Each output tile width is O_TILE_WIDTH

For each thread,

O_TILE_WIDTH is 4 in this example

## Definition - Input Tiles

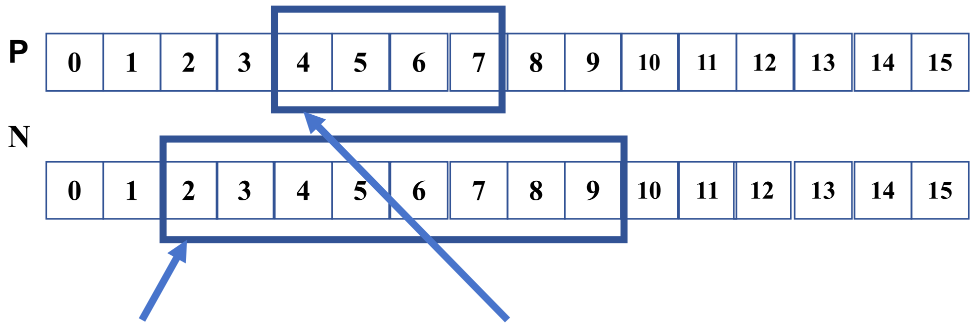


**Each input tile has all values needed to calculate the corresponding output tile.**

## Two Design Options

- **Design 1: The size of each thread block matches the size of an output tile**
  - All threads participate in calculating output elements
  - blockDim.x would be 4 in our example
  - Some threads need to load more than one input element into the shared memory
- **Design 2: The size of each thread block matches the size of an input tile**
  - Some threads will not participate in calculating output elements
  - blockDim.x would be 8 in our example
  - Each thread loads one input element into the shared memory

## Thread to Input and Output Data Mapping



Thread 0 reads this    Thread 0 writes this

For each thread,
`Index_i = index_o – n`

were n is Mask_Width /2
n is 2 in this example

## All Threads Participate in Loading Input Tiles

```
float output = 0.0f;
if((index_i >= 0) && (index_i < Width)) {
  Ns[tx] = N[index_i];
}
else{
  Ns[tx] = 0.0f;
}
```

## Some threads do not participate in calculating output

```
  if (threadIdx.x < O_TILE_WIDTH){
     output = 0.0f;
     for(j = 0; j < Mask_Width; j++) {
         output += M[j] *
Ns[j+threadIdx.x];
     }
     P[index_o] = output;
  }
```

– Only Threads 0 through O_TILE_WIDTH-1 participate in calculation of output.

## Setting Block Size

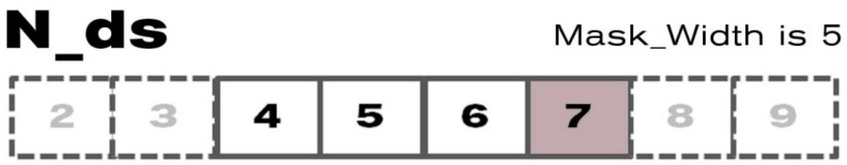
```
#define O_TILE_WIDTH 1020
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)

dim3 dimBlock(BLOCK_WIDTH, 1, 1);

dim3 dimGrid((Width-1)/O_TILE_WIDTH+1, 1, 1)
```

The Mask_Width is 5 in this example.

In general, block width should be

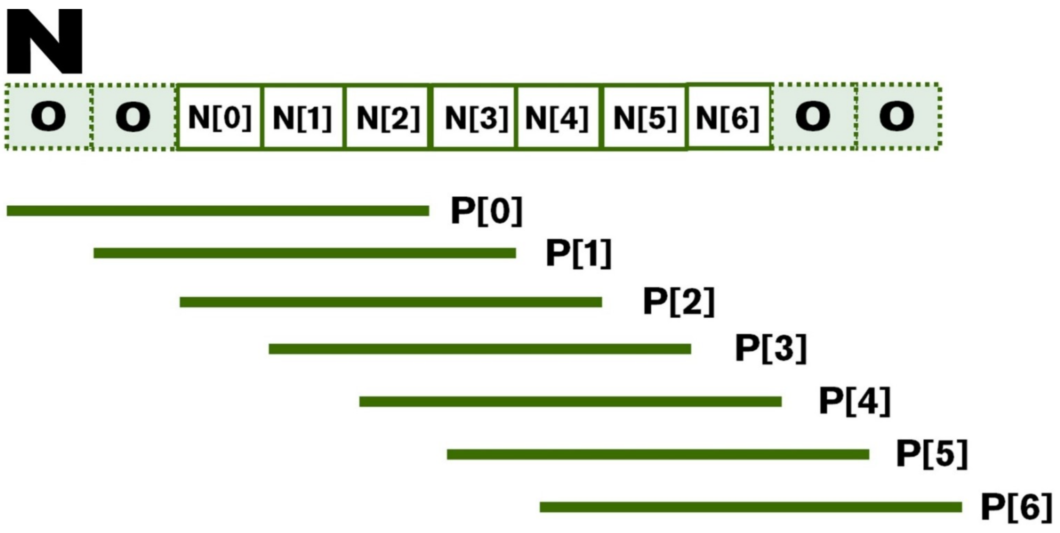
```
  output tile width + (mask width-1)
```

## Shared Memory Data Reuse



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)

## Ghost Cells

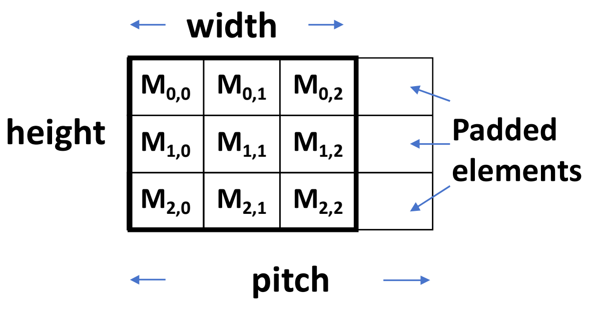


## 讲授内容：Parallel Computation Patterns (Stencil)

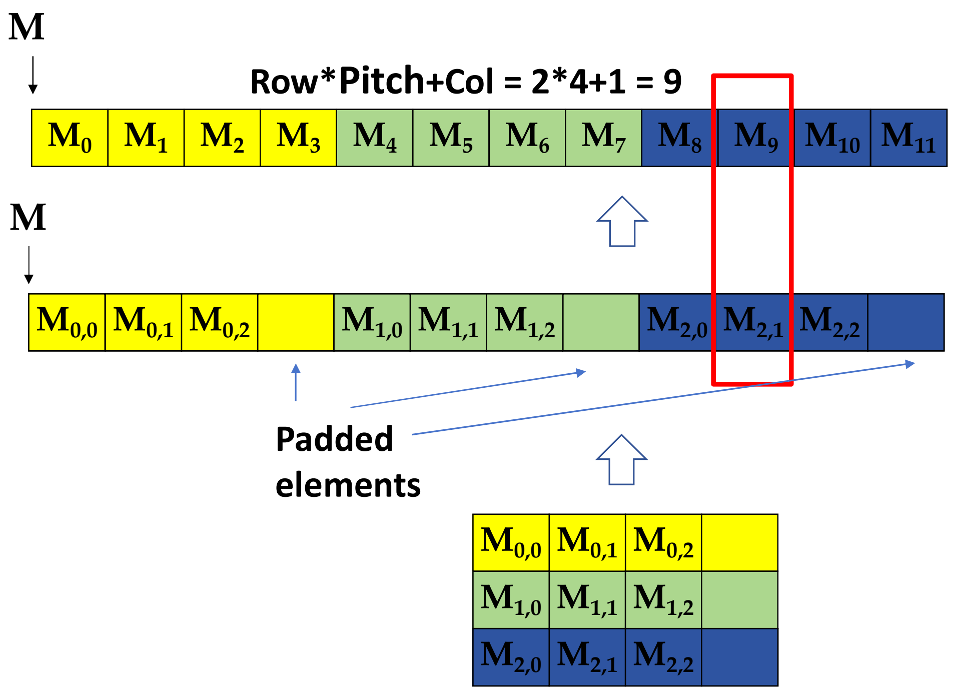
## 2D Image Matrix with Automated Padding

- It is sometimes desirable to pad each row of a 2D matrix to multiples of DRAM bursts
  - So each row starts at the DRAM burst boundary
  - Effectively adding columns
  - This is usually done automatically by matrix allocation function
  - Pitch can be different for different hardware
- Example: a 3X3 matrix padded into a 3X4 matrix

Height is 3
Width is 3
Channels is 1 (See MP Description)
Pitch is 4



## Row-Major Layout with Pitch



M
Row*Pitch+Col = 2*4+1 = 9
$M_0$ $M_1$ $M_2$ $M_3$ $M_4$ $M_5$ $M_6$ $M_7$ $M_8$ $M_9$ $M_{10}$ $M_{11}$
M
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$ $M_{1,0}$ $M_{1,1}$ $M_{1,2}$ $M_{2,0}$ $M_{2,1}$ $M_{2,2}$
Padded elements
$M_{0,0}$ $M_{0,1}$ $M_{0,2}$
$M_{1,0}$ $M_{1,1}$ $M_{1,2}$
$M_{2,0}$ $M_{2,1}$ $M_{2,2}$

## Image Matrix Type in this Course

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

## wbImage_t API Function for Your Lab

```
wbImage_t  wbImage_new(int height, int width, int channels)
wbImage_t  wbImport(char * File);

void wbImage_delete(wbImage_t img)


int wbImage_getWidth(wbImage_t img)
int wbImage_getHeight(wbImage_t img)
int wbImage_getChannels(wbImage_t img)
int wbImage_getPitch(wbImage_t img)

float *wbImage_getData(wbImage_t img)
```

For simplicity, the pitch of all matrices are set to be width * channels (no padding) for our labs.
The use of all API functions has been done in the provided host code.

## Setting Block Size

```
#define O_TILE_WIDTH 12
#define BLOCK_WIDTH (O_TILE_WIDTH + 4)

dim3 dimBlock(BLOCK_WIDTH,BLOCK_WIDTH);
dim3 dimGrid((wbImage_getWidth(N)-
1)/O_TILE_WIDTH+1,    (wbImage_getHeight(N)-
1)/O_TILE_WIDTH+1, 1)
```

**In general, BLOCK_WIDTH should be** `O_TILE_WIDTH + (MASK_WIDTH-1)`

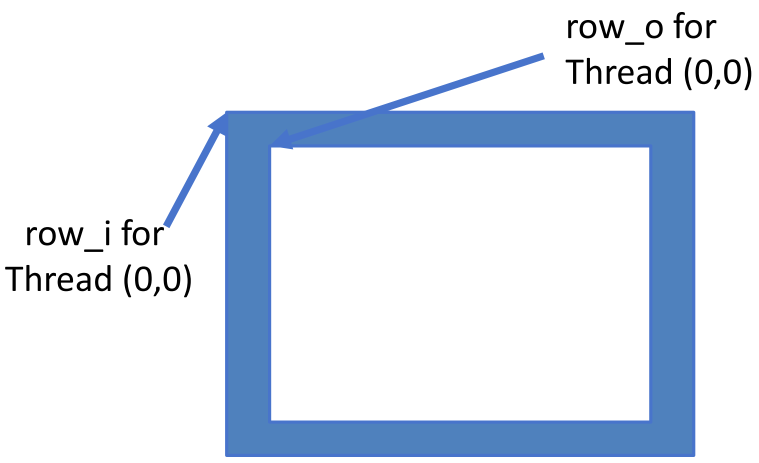
## Using constant memory and caching for Mask

- Mask is used by all threads but not modified in the convolution kernel
  - All threads in a warp access the same locations at each point in time
- CUDA devices provide constant memory whose contents are aggressively cached
  - Cached values are broadcast to all threads in a warp
  - Effectively magnifies memory bandwidth without consuming shared memory
- Use of const __restrict__ qualifiers for the mask parameter informs the compiler that it is eligible for constant caching, for example:

```
__global__ void convolution_2D_kernel(float *P,
    float *N, height, width, channels,
    const float __restrict__ *M) {
```

## Shifting from output coordinates to input coordinate

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

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



## Taking Care of Boundaries (1 channel example)

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

**Use of width here is OK since pitch is set to width for this MP.**

## Some threads do not participate in calculating output. (1 channel example)

```
    float output = 0.0f;
    if(ty < O_TILE_WIDTH && tx <
  O_TILE_WIDTH){
        for(i = 0; i < MASK_WIDTH; i++) {
          for(j = 0; j < MASK_WIDTH; j++) {
            output += M[i][j] *
  Ns[i+ty][j+tx];
        }
    }
```

## Some threads do not write output (1 channel example)

```
if(row_o < height && col_o <
width)
    data[row_o*width + col_o] =
output;
```

**You need to write the kernel for a 3-channel (RGB) image.**
**See more details in the Lab MP Description.**

## 讲授内容：Parallel Computation Patterns (Stencil)

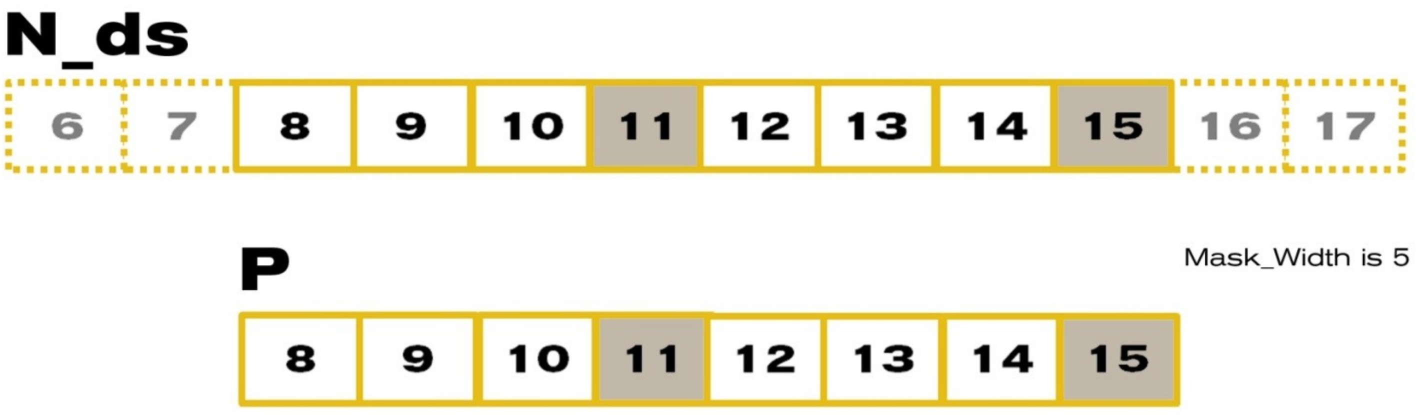
① **Convolution**

② **Tiled Convolution**

③ **Tile Boundary Conditions**

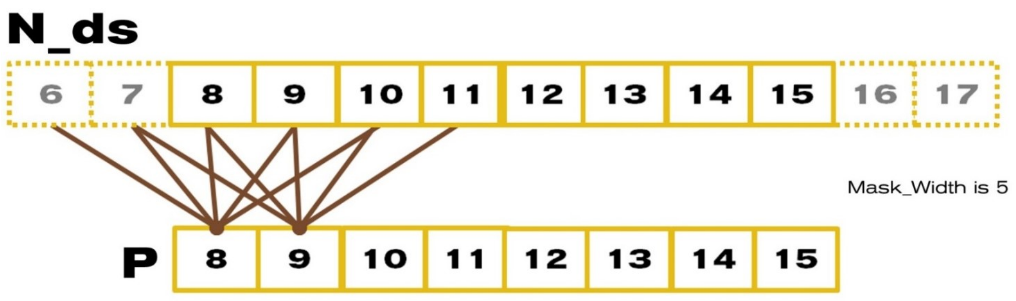
④ **Analyzing Data Reuse in Tiled Convolution**

## An 8-element Convolution Tile



**For Mask_Width=5, we load 8+5-1=12 elements**
**(12 memory loads)**

## Each output P element uses 5 N elements



**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] use 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]**

[redacted]

## A simple way to calculate tiling benefit

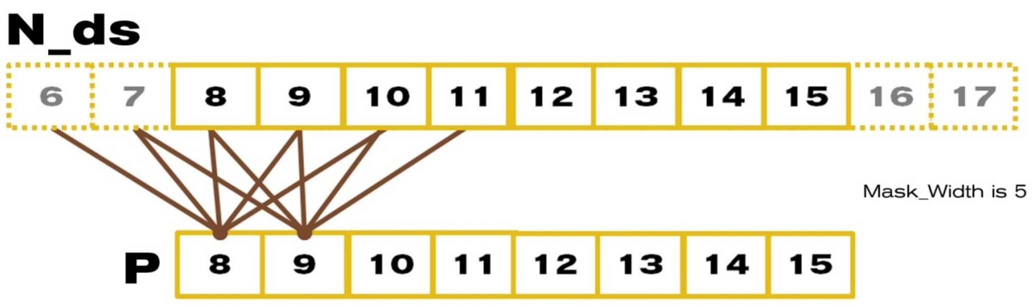
- (8+5-1)=12 elements loaded
- 8*5 global memory accesses replaced by shared memory accesses
- This gives a bandwidth reduction of 40/12=3.3

## In General, for 1D TILED CONVOLUTION

- `O_TILE_WIDTH+MASK_WIDTH -1` elements loaded for each input tile
- `O_TILE_WIDTH*MASK_WIDTH` global memory accesses replaced by shared memory accesses
- This gives a reduction factor of `(O_TILE_WIDTH*MASK_WIDTH)/(O_TILE_WIDTH+MASK_WIDTH-1)`

This ignores ghost elements in edge tiles.

## Another Way to Look at Reuse



N[6] is used by P[8] (1X)
N[7] is used by P[8], P[9] (2X)
N[8] is used by P[8], P[9], P[10] (3X)
N[9] is used by P[8], P[9], P[10], P[11] (4X)
N10 is used by P[8], P[9], P[10], P[11], P[12] (5X)
… (5X)
N[14] is used by P[12], P[13], P[14], P[15] (4X)
N[15] is used by P[13], P[14], P[15] (3X)

[redacted]

## Another Way to Look at Reuse

The total number of global memory accesses (to the (8+5-1)=12 N elements) replaced by shared memory accesses is:

$$\begin{aligned} & 1+2+3+4+5*(8\text{-}5\text{+}1)+4+3+2+1 \\ &= 10+20+10 \\ &= 40 \end{aligned}$$

So the reduction is:

$$40/12 = 3.3$$

## In General, for 1D

– The total number of global memory accesses to the input tile can be calculated as

```
1 + 2+…+ MASK_WIDTH-1 + MASK_WIDTH*(O_TILE_WIDTH-
     MASK_WIDTH+1) + MASK_WIDTH-1 + …+ 2 + 1

   = MASK_WIDTH * (MASK_WIDTH-1) + MASK_WIDTH *

                 (O_TILE_WIDTH-MASK_WIDTH+1)

    =  MASK_WIDTH * O_TILE_WIDTH
```

– For a total of O_TILE_WIDTH + MASK_WIDTH -1 input tile elements

## Examples of Bandwidth Reduction for 1D

**The reduction ratio is:**
MASK_WIDTH * (O_TILE_WIDTH)/(O_TILE_WIDTH+MASK_WIDTH-1)

| O_TILE_WIDTH | 16 | 32 | 64 | 128 | 256 |
|---|---|---|---|---|---|
| MASK_WIDTH= 5 | 4.0 | 4.4 | 4.7 | 4.9 | 4.9 |
| MASK_WIDTH = 9 | 6.0 | 7.2 | 8.0 | 8.5 | 8.7 |

## For 2D Convolution Tiles

- (O_TILE_WIDTH+MASK_WIDTH-1)$^2$ input elements need to be loaded into shared memory
- The calculation of each output element needs to access MASK_WIDTH$^2$ input elements
- O_TILE_WIDTH$^2$ * MASK_WIDTH$^2$ global memory accesses are converted into shared memory accesses
- The reduction ratio is

O_TILE_WIDTH$^2$ * MASK_WIDTH$^2$ / (O_TILE_WIDTH+MASK_WIDTH-1)$^2$

## Bandwidth Reduction for 2D

**The reduction ratio is:**
`O_TILE_WIDTH² * MASK_WIDTH² / (O_TILE_WIDTH+MASK_WIDTH-1)²`

| O_TILE_WIDTH | 8 | 16 | 32 | 64 |
|---|---|---|---|---|
| MASK_WIDTH = 5 | 11.1 | 16 | 19.7 | 22.1 |
| MASK_WIDTH = 9 | 20.3 | 36 | 51.8 | 64 |

**Tile size has significant effect on of the memory bandwidth reduction ratio.**
**This often argues for larger shared memory size.**

THANKS