Convolution with Constant & Shared Memory

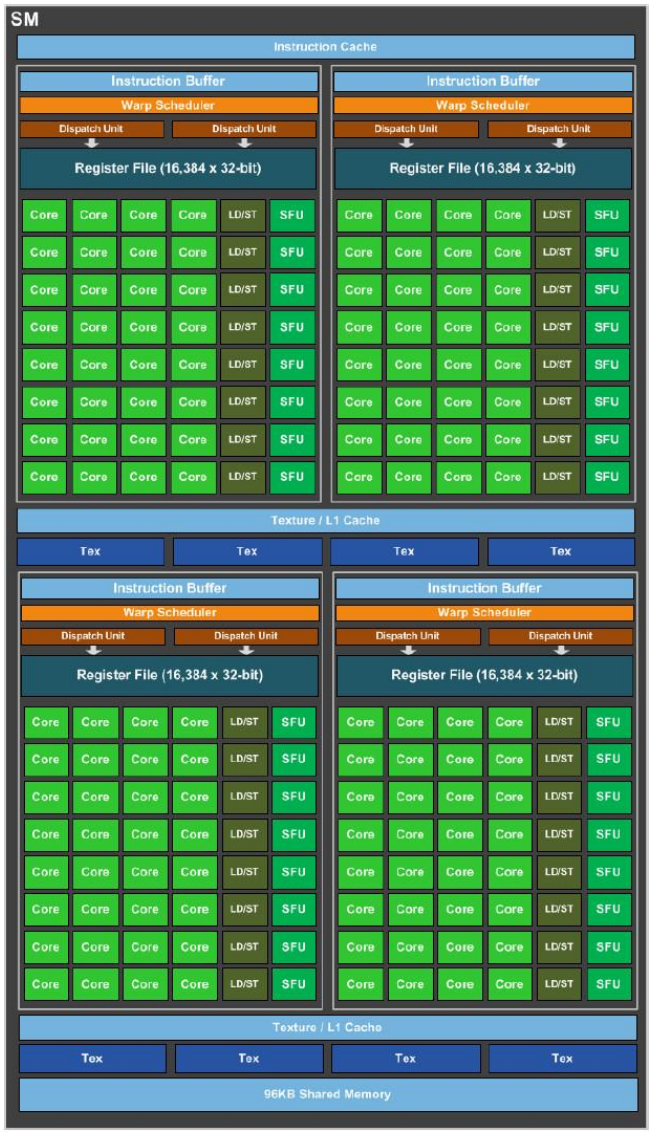
KOREA UNIVERSITY

KUCG

In This Lecture

- Quick Review of Previous Lectures
- Convolution Algorithm Basic
- Memory Hierarchies
- Convolution with Constant Memory
- Tiled Convolution
- Evaluating Tiled Convolution

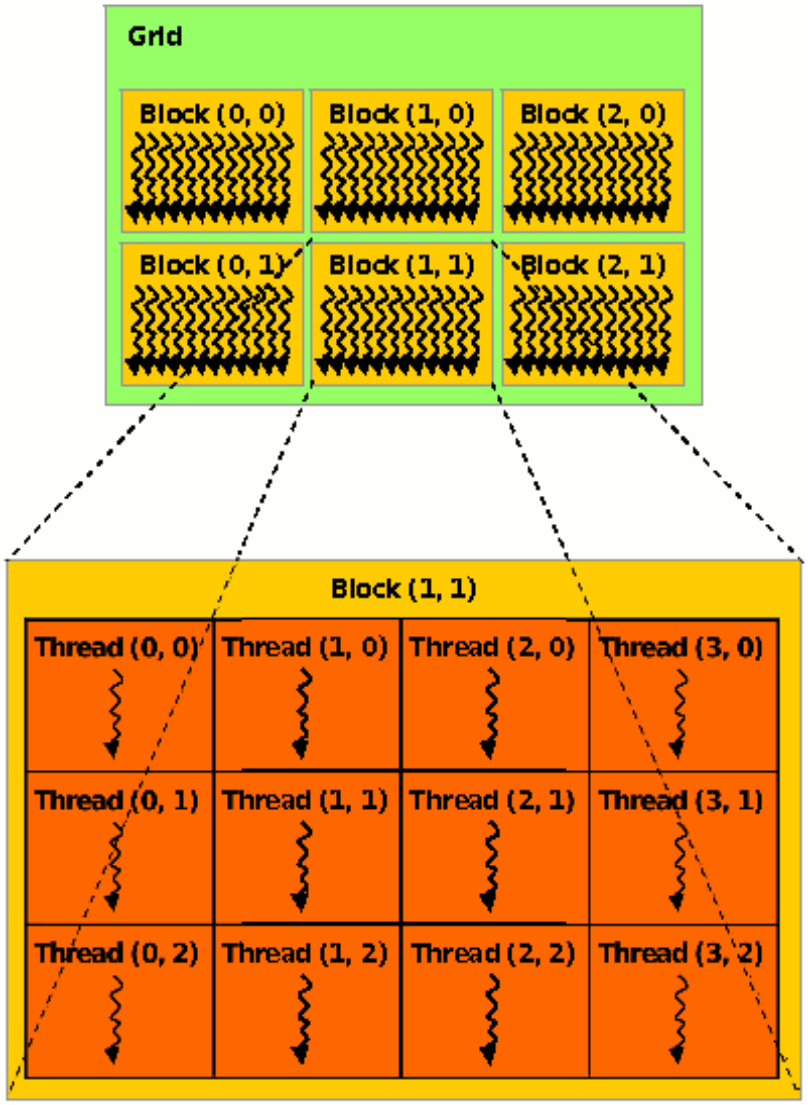
GPU Hardware Architecture (NVIDIA GP104 Pascal, GTX 1080)



20 SMs, each has 128 CUDA cores (SP)

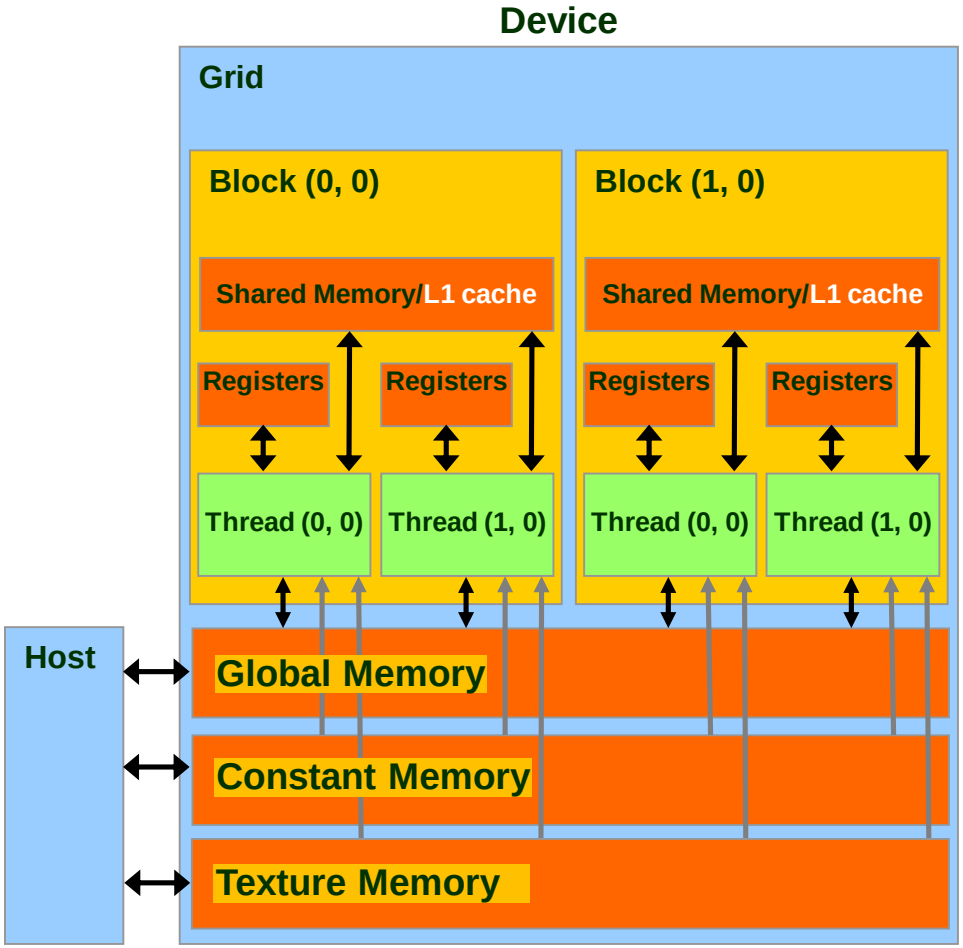
Unit of Computation for CUDA

- **Thread**
  - Smallest/Primary Unit of Computation
  - Maps to Streaming Processor (i.e. CUDA Core)
- **Warp**
  - A Group of 32 Threads Processed as a Unit by Hardware
- **Block**
  - Multidimensional Group of Threads
  - Maps to Streaming Multiprocessor
- **Grid**
  - Multidimensional Group of Blocks
  - 1 Grid per GPU



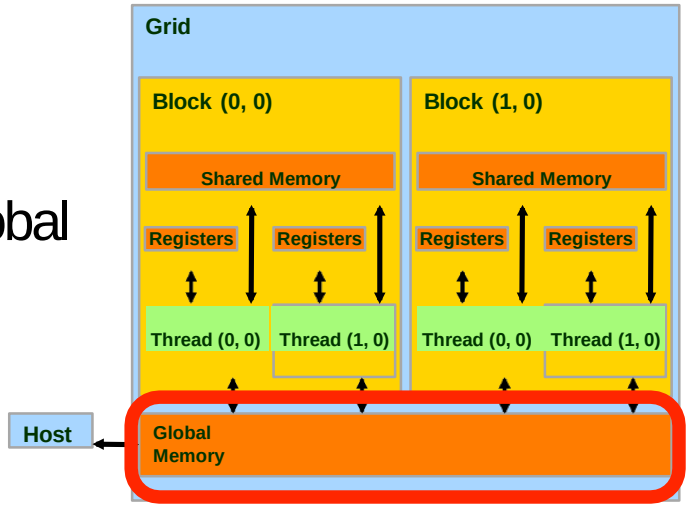
CUDA Memory Structure

- **Registers**
  - Used by a single thread
- **Shared Memory**
  - Shared by all threads in a block
- **Global Memory**
  - R/W by thread & host
- **Constant Memory**
  - Read only by device, initialized by host
- **Texture Memory**
  - Read only by device, initialized by host



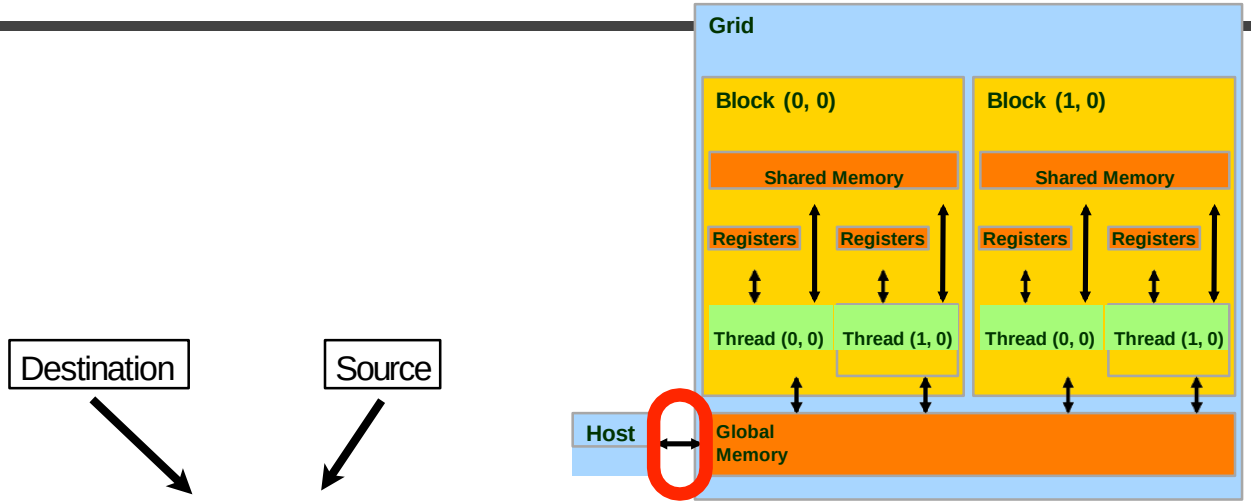
Allocating and Deallocating Device Memory Space

- Allocating Memory: allocates object in device global memory and returns pointer to it.
  - `int *dev_C;`
  - `int size = N *sizeof(int);`
  - `cudaMalloc((void**)&dev_C, size);`



- Deallocating Memory: free object from device global memory
  - `cudaFree(dev_C)`

Transferring Data via cudaMemcpy



- `cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);`

- `cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);`

- "dev_a" and "dev_c" are pointers to device data
- "a" and "c" are pointers to host data
- "size" is the size of the data
- "cudaMemcpyHostToDevice" and "cudaMemcpyDeviceToHost" tells cudaMemcpy the source and estination of the operation.

Dimension & Indexing (1)

- **Built-In CUDA Variables for Block/Thread Dimension**

```
myKernel<<< B,T >>>(arg1, arg2, ... );
```

  - `B`: a structure defining the number and dimension of blocks in a grid
  - `T`: a structure defining the number and dimension of threads in a block

```
dim3 grid(16,9,4);
dim3 block(32,16,4);
```

- Keywords when retrieving each dimension

`gridDim.x` → 16 `gridDim.y` → 9 `gridDim.z` → 4

`blockDim.x` → 32 `blockDim.y` → 16 `blockDim.z` → 4

Dimension & Indexing (2)

- Built-In CUDA Variables for Indexing within Kernel Function

```
__global__ void myKernel(arg1, arg2, ... )
{
    . . .

    int bx = blockIdx.x;
    int by = blockIdx.y;
    int bz = blockIdx.z;

    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int tz = threadIdx.z;

    . . .
}
```

Retrieving a Unique Location from CUDA Indices

- **Normally each thread/block having multidimensional indices need to access 1D arrays**
  - How each thread in a multidimensional nested grid structure can find the unique index of the multidimensional data flattened to 1D array from its blockId and threadId using the same expression for all threads?

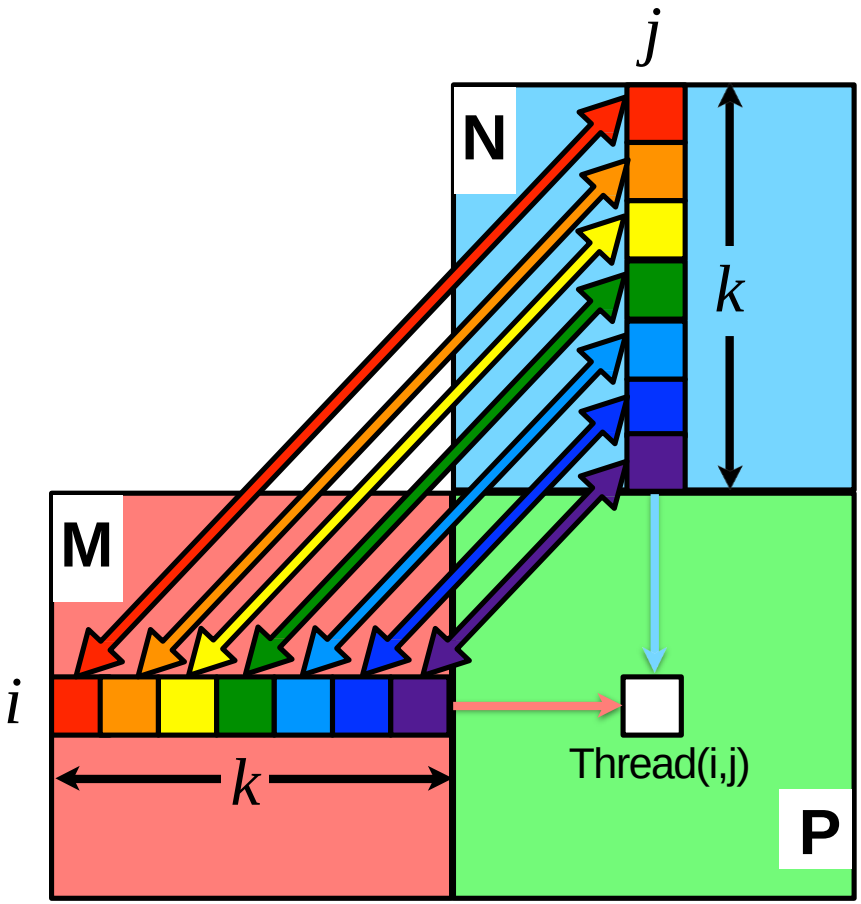
```
uniq x coord → blockIdx.x*blockDim.x + threadIdx.x;
uniq y coord → blockIdx.y*blockDim.y + threadIdx.y;
uniq z coord → blockIdx.z*blockDim.z + threadIdx.z;
```

```
int col = blockIdx.x*blockDim.x + threadIdx.x;
int row = blockIdx.y*blockDim.y + threadIdx.y;
int dep = blockIdx.z*blockDim.z + threadIdx.z;
int index = col + row*NUMCOL + dep*NUMCOL*NUMROW;
int a[row][col][dep] = A[index];
```

Using Indices within a Kernel

```
// Vector Summation (1D data)
__global__ void vectorAdd (int *A, int *B, int *C)
{
    int index = blockIdx.x * blockDim.x + threadIdx.x;

    if (index < NUMSIZE)  C[index] = A[index] + B[index];
}

// Matrix Summation (2D data)
__global__ void matrixAdd (int *A, int *B, int *C)
{
    int col = blockIdx.x * blockDim.x + threadIdx.x;
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    int index = col + row * NUMCOL;

    if (col < NUMCOL && row < NUMROW)  C[index] = A[index] + B[index];
}
```

Parallel (Square) Matrix Multiplication



The thread (of index i,j) computes the inner product of the vectors i and j shown in the image.

We can use multiple blocks when dealing with large sized matrices which do not fit into the max number of threads per SM.

Tiled Matrix Multiplication Using Shared Memory

- **Reduce Global Memory Access**
  - Global Memory is slow
- **Use On-chip Memory**
  - We can use Shared Memory
  - Small size, but much faster !
  - Thus the data should be partitioned (tiled)
- **Quick review using cartoon . . .**

Convolution Basic

Objective

- **Convolution, an important parallel computation pattern**
  - Widely used in signal/image/video processing
  - Foundational to stencil computation used in many science and engineering area
- **Important techniques**
  - Taking advantage of cache memories

Convolution Computation

- **Each output element is a weighted sum of neighboring input elements**

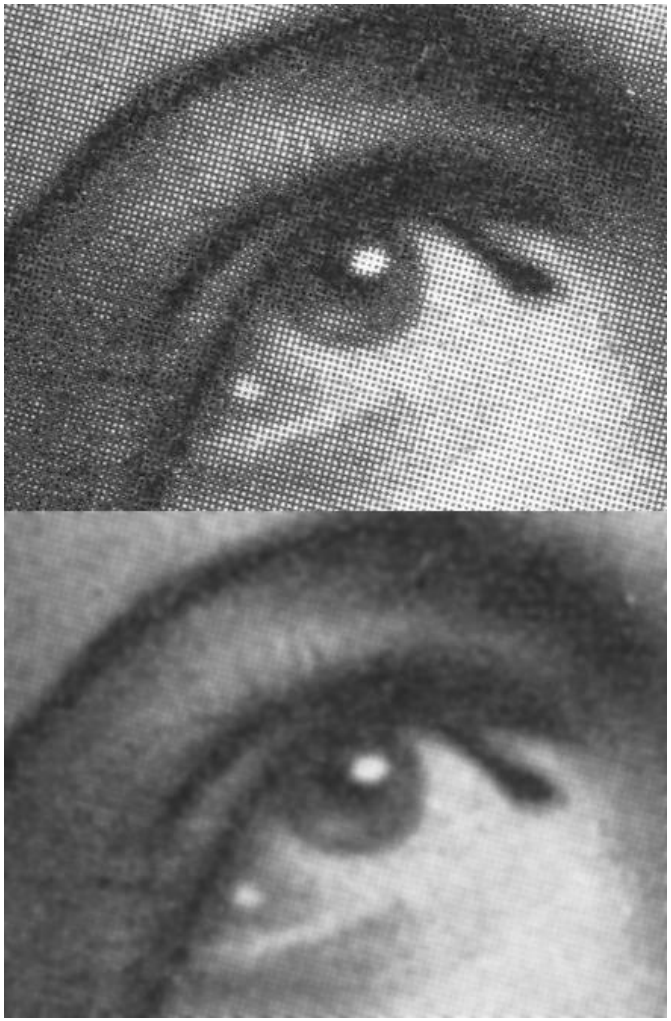
- **The weights are defined as a convolution kernel (mask)**
  - The same convolution mask is typically used for all elements of the array.

Gaussian Blur

$$\frac{1}{273}$$

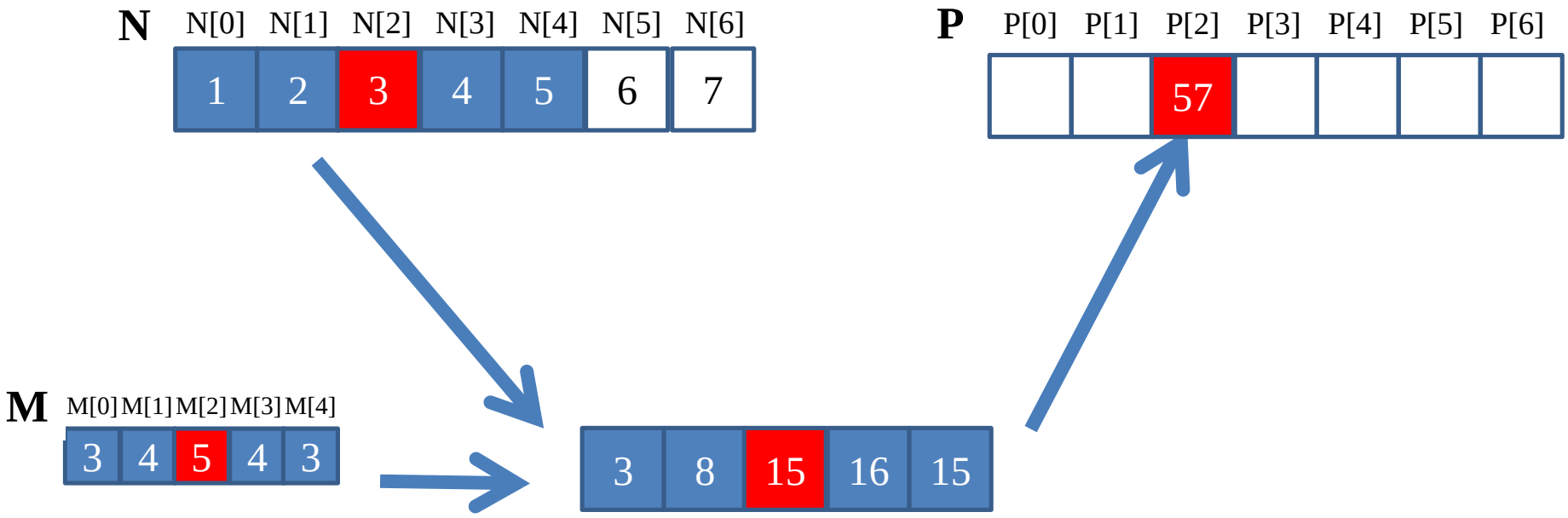
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Simple Integer Gaussian Kernel



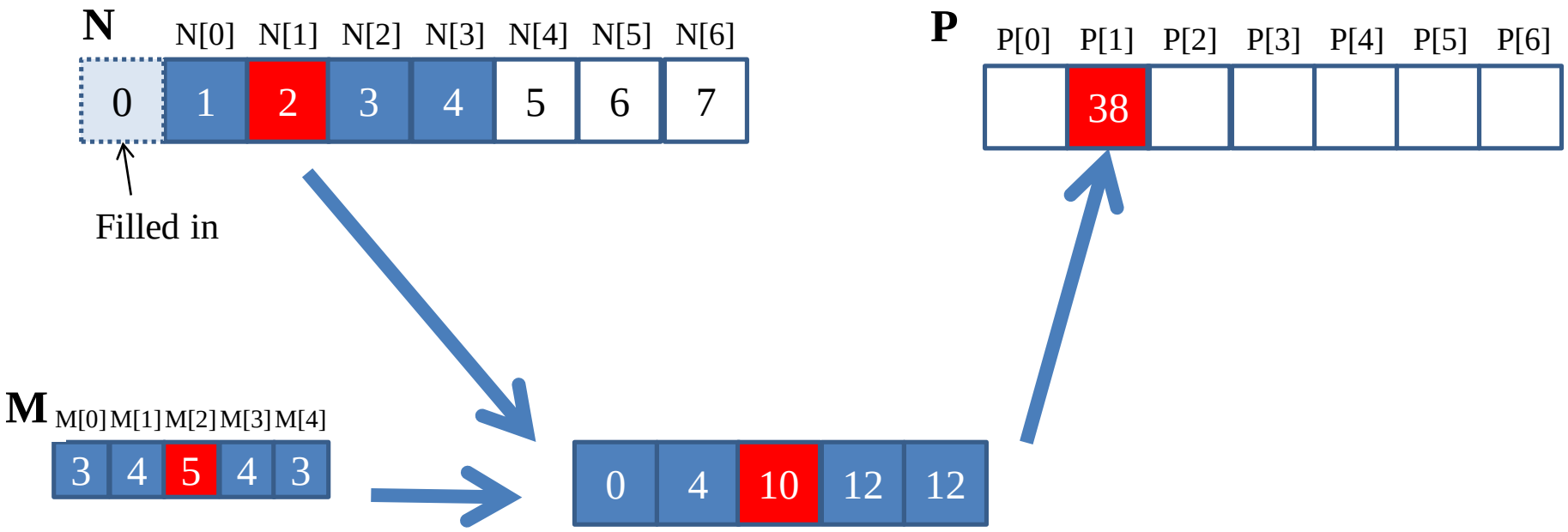
1D Convolution Example

- **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 Boundary Condition

- **Calculation of output elements near the boundaries (beginning and end) of the input array need to deal with "ghost" elements**
  - Different policies (zero, boundary replication, mirror, etc.)



A 1D Convolution Kernel with Boundary Condition Handling

- All elements outside input data is set to 0

```
__global__ void basic_1D_conv(float *N, float *M, float *P, int Mask_Width, int Width)
{

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;

  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;
  }
}
```

A 1D Convolution Kernel with Boundary Condition Handling

- All elements outside input data is set to 0

```
__global__ void basic_1D_conv(float *N, float *M, float *P, int Mask_Width, int Width)
{

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  int N_start_point = i - (Mask_Width/2);
  float Pvalue = 0;

  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;
  }
}
```

Source of bad performance:
1 floating-point operation per global memory access

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

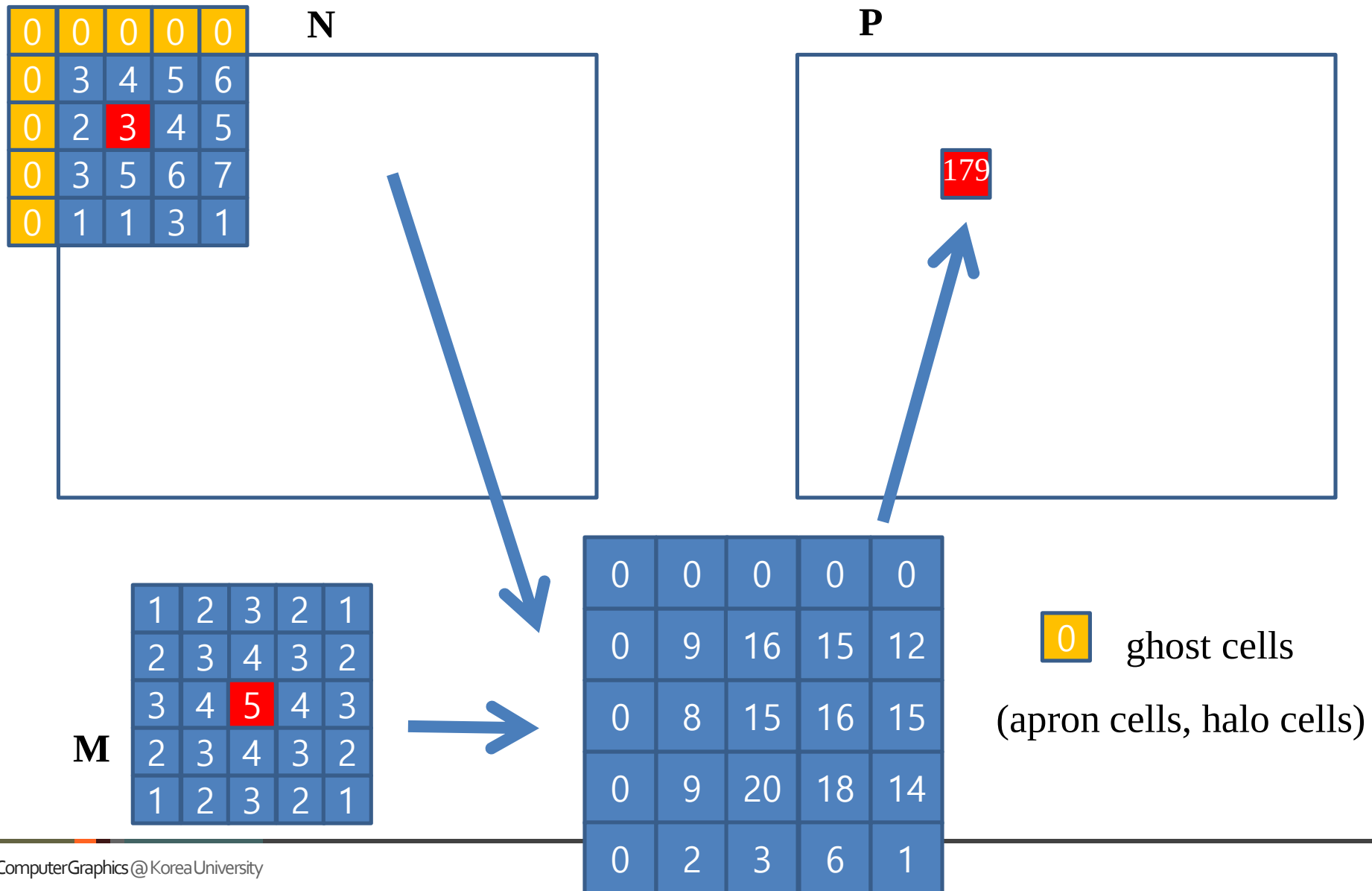
		321				

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 – Ghost Cells

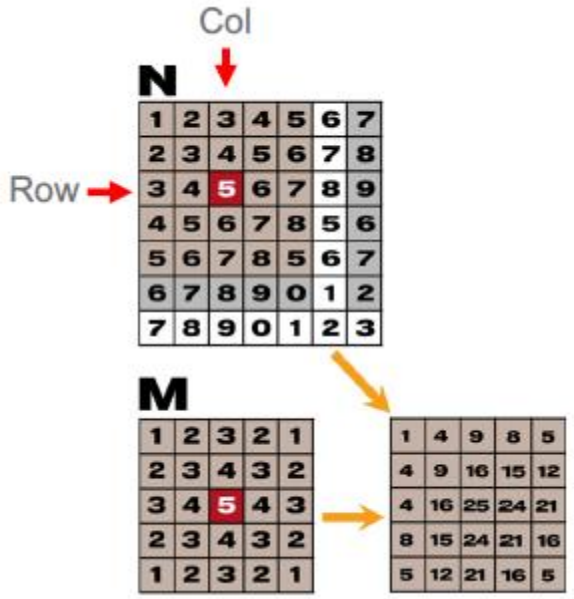


2D Convolution Example

```
__global__
void convolution_2D_basic_kernel(int* N, int* M, int* P, 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);

        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 += N[curRow * w + curCol] * M[j*maskwidth+k];
                }
            }
        }
        // Write our new pixel value out
        P[Row * w + Col] = pixVal;
    }
}
```

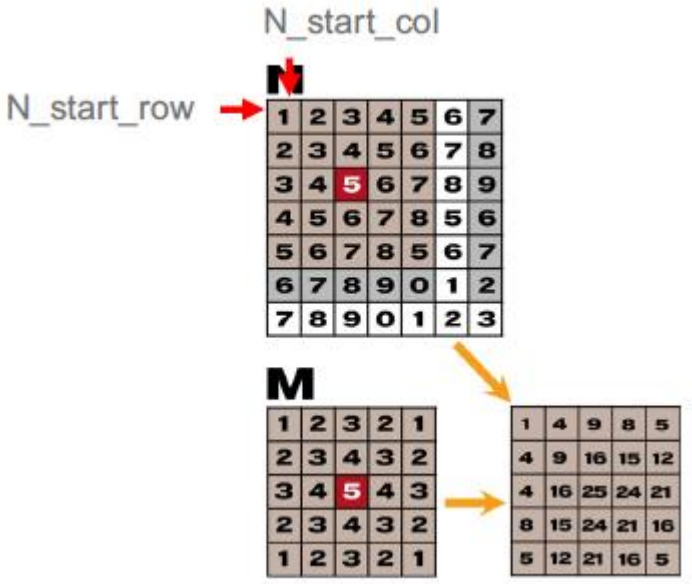
2D Convolution Example

```
__global__
void convolution_2D_basic_kernel(int* N, int* M, int* P, 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);

        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 &&
                    pixVal += N[curRow * w + curCol] * M[j*mas
                }
            }
        }
        // Write our new pixel value out
        P[Row * w + Col] = pixVal;
    }
}
```



2D Convolution Example

```
__global__
void convolution_2D_basic_kernel(int* N, int* M, int* P, 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);

        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 >
                    pixVal += N[curRow * w + curCol] * M
                }
            }
        }
        // Write our new pixel value out
        P[Row * w + Col] = pixVal;
    }
}
```



2D Convolution Example

```
__global__
void convolution_2D_basic_kernel(int* N, int* M, int* P, 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);

        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 += N[curRow * w + curCol] * M[j*maskwidth+k];
                }
            }
        }
        // Write our new pixel
        P[Row * w + Col] = pixV
    }
}
```

Source of bad performance:
1 floating-point operation per global memory access

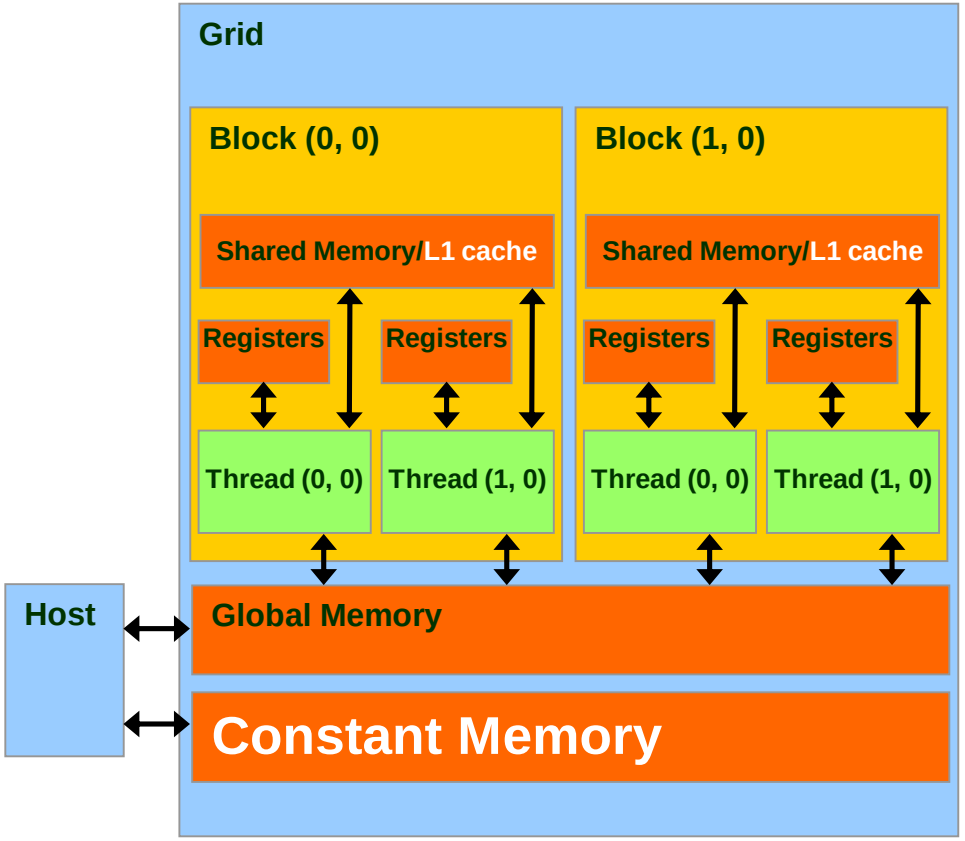
Access Pattern for M

- M is referred to as mask (a.k.a. kernel, filter, etc.)
- Calculation of all output P elements need M
- Total of O(PxM) reads of M
- M is not changed during kernel
- M elements are accessed in the same order when calculating all P elements

Memory Hierarchies

Programmer View of CUDA Memories

- **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)**



General Memory Hierarchies

- If every time we needed a piece of data, we had to go to main memory to get it, computers would take a lot longer to do anything
- On today's processors (CPU), main memory accesses take hundreds of cycles
- One solution: Caches

Cache

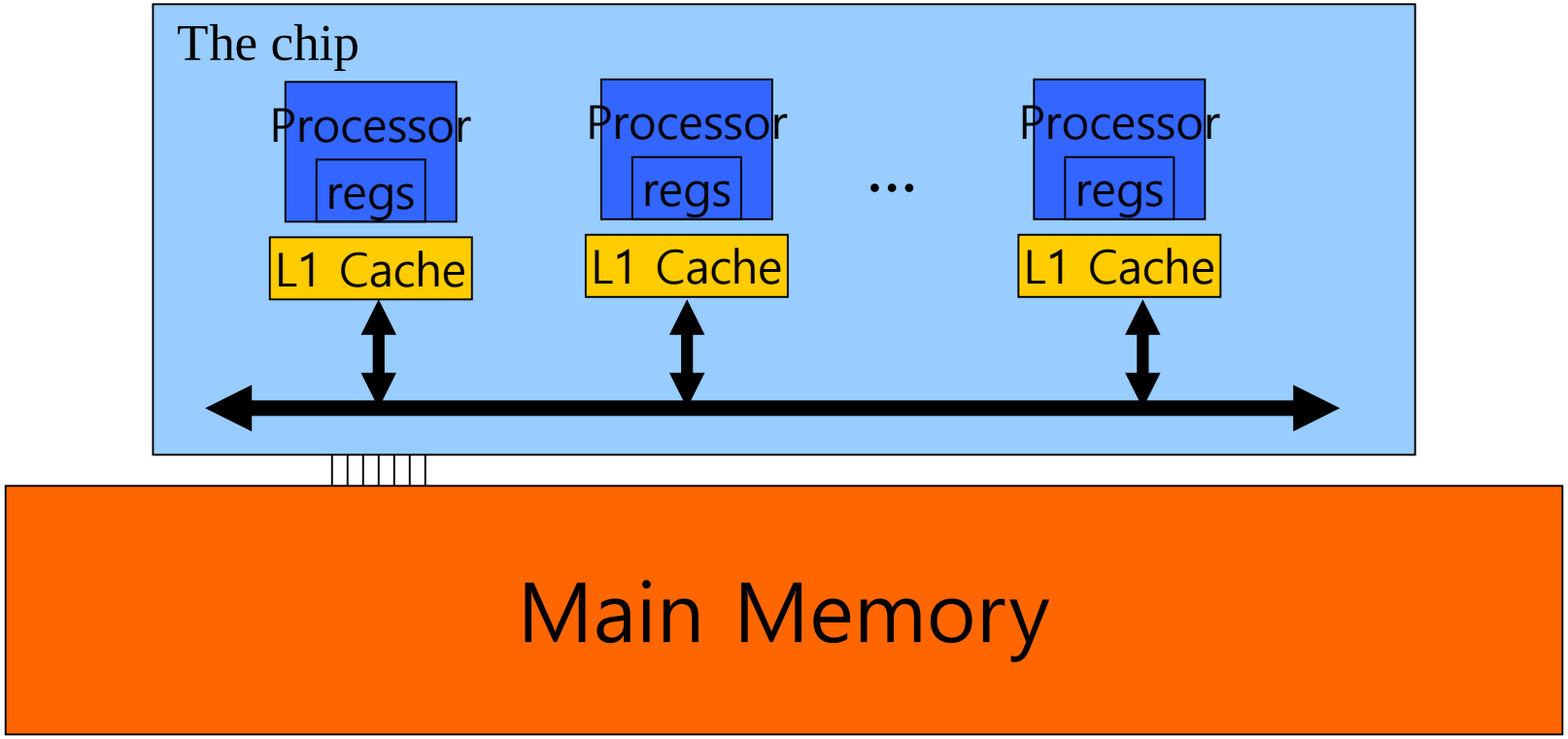
- Cache is unit of volatile memory storage
- A cache is an “array” of cache lines
- Cache line can usually hold data from several consecutive memory addresses
- When data is requested from memory, an entire cache line is loaded into the cache, in an attempt to reduce main memory requests

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
- It requires explicit data transfer instructions, whereas cache doesn't

Cache Coherence Protocol

- A mechanism for caches to propagate updates by their local processor to other caches (processors)



CPU and GPU have different caching philosophy

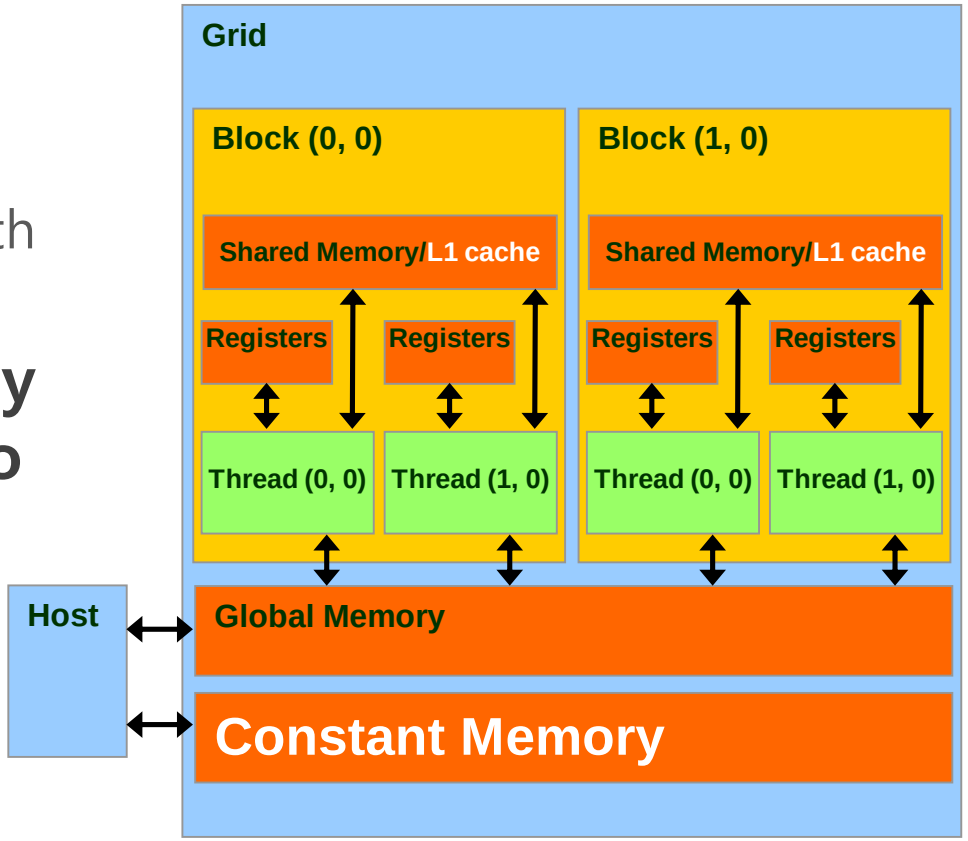
- **CPU L1 caches are usually coherent**
  - L1 is also replicated for each core
  - Changeable data can be cached in L1
  - Updates to local cache copy invalidates copies in other caches
  - Expensive in terms of hardware and disruption of services
- **GPU L1 caches are usually incoherent**
  - Avoid caching for changing data (Constant Memory)

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.

More on Constant Caching

- **Each SM has its own L1 cache**
  - Low latency, high bandwidth access by all threads
- **However, there is no way for threads in one SM to update the L1 cache in other SMs**
  - No L1 cache coherence



This is not a problem if a variable is NOT modified by a kernel.

Convolution with Constant Memory

Improving convolution kernel

- Use tiling for the N array element (will see later)
- Use constant memory for the M mask
  - it's typically small and is not changed
  - can be read by all threads in the grid

```
#define MAX_MASK_WIDTH 10

__constant__ float M[MAX_MASK_WIDTH];
```

Global Variable

```
cudaMemcpyToSymbol(M, h_M, Mask_Width*sizeof(float));
```

Host Memory copy to Constant Memory

1D Convolution with Constant Memory

```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int N_start_point = i - (Mask_Width/2);

    float Pvalue = 0;
    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;
    }
}
```

1D Convolution with Constant Memory

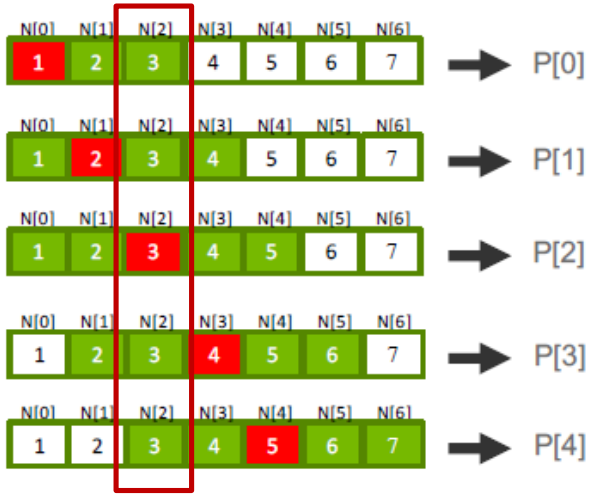
```
__global__ void convolution_1D_basic_kernel(float *N, float *P, int Mask_Width, int Width)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int N_start_point = i - (Mask_Width/2);

    float Pvalue = 0;
    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;
    }
}
```

2 floating-point operations per global memory access(N)

Tiling & Convolution

Tiling & Convolution

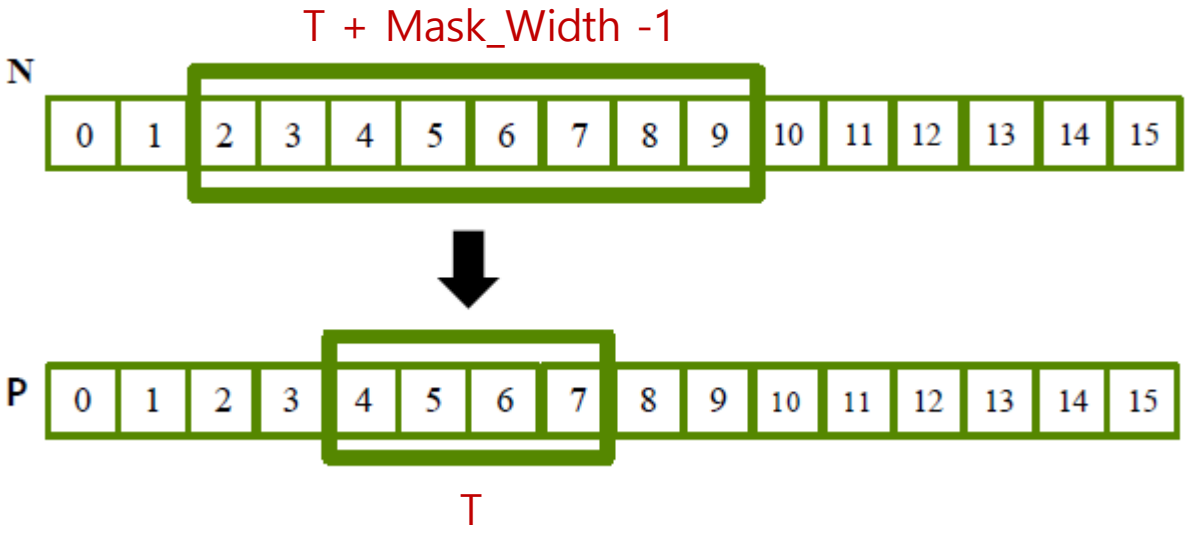
- Calculation of adjacent output elements involve shared input elements
  - E.g., N[2] is used in calculation of P[0], P[1], P[2], P[3], and P[4] assuming a 1D convolution Mask_Width of 5



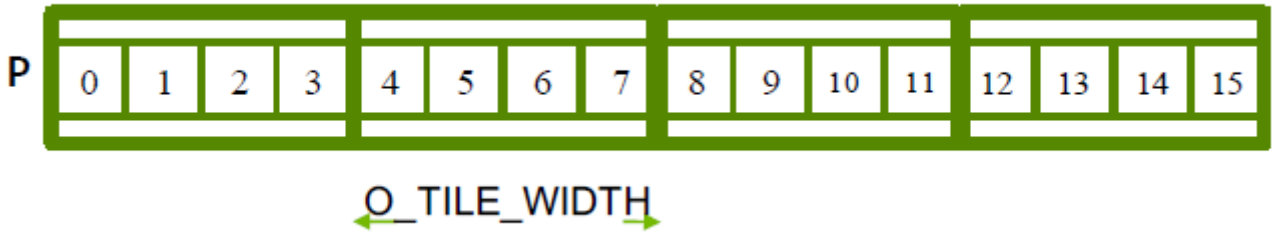
- We can load all the input elements required by all threads in a block into the shared memory to reduce global memory accesses

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

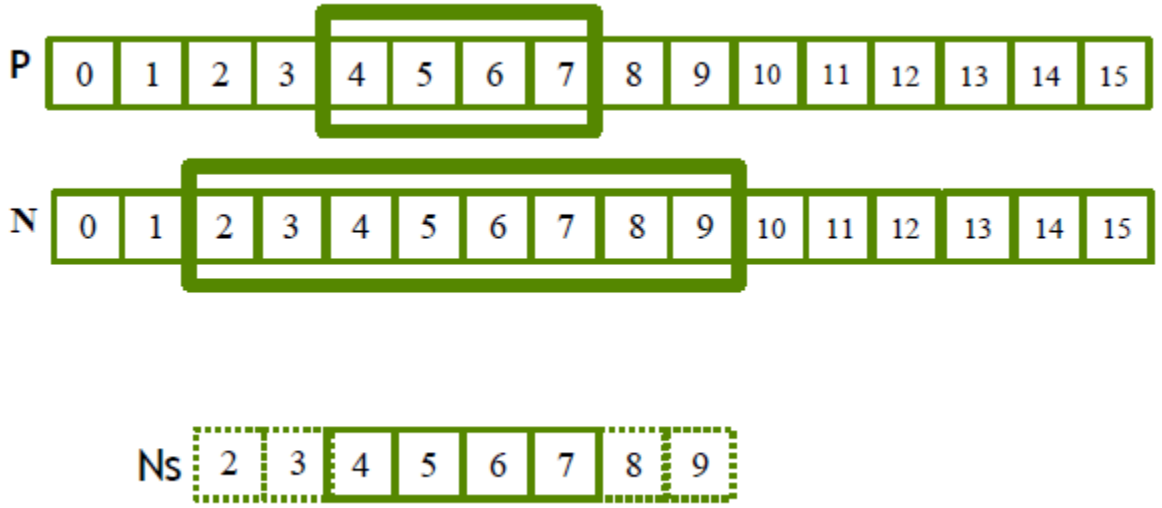


Definition – output tile



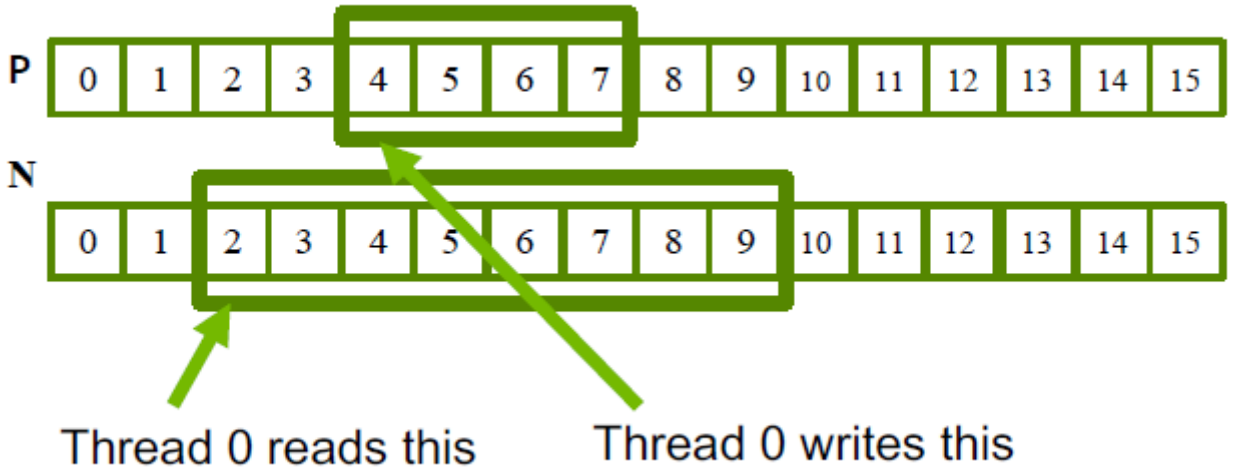
- Each thread block calculates an output tile
- Each output tile width is O_TILE_WIDTH
  - O_TILE_WIDTH is 4 in this example

Definition - Input Tiles



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

Thread to Input and Output Data Mapping



- For each thread:
  - index_i = index_o – n
- where n is (Mask_Width-1)/2
  - n is 2 in this example (because Mask_Width is 5)

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

Design#1: 1D Tiled Convolution Kernel

```
__global__ void convolution(int* N, int* P, int numPhase)
{                                                  // (int)ceil((float)I_TILE_WIDTH/O_TILE_WIDTH)
    __shared__ int Nds[I_TILE_WIDTH];

    int tx = threadIdx.x;
    int index_o = blockIdx.x * O_TILE_WIDTH + tx;
    int index_i = index_o-(MASK_WIDTH/2);
    int pValue = 0;

    for(int i=0; i<numPhase; i++){
        int ds_i = tx + i*O_TILE_WIDTH;
        if(ds_i < I_TILE_WIDTH){
            if((index_i >= 0) && (index_i < ARRAY_SIZE)) {
                Nds[ds_i] = N[index_i];
            }else{
                Nds[ds_i] = 0;
            }
        }
        index_i += O_TILE_WIDTH;
    }
    __syncthreads();

    if (index_o<ARRAY_SIZE){
        for(int i = 0; i < MASK_WIDTH; i++) {
            pValue += M[i] * Nds[i+tx];
        }
        P[index_o] = pValue;
    }
}
```

Design#1: 1D Tiled Convolution main

```
#define ARRAY_SIZE      128
#define MASK_WIDTH      5
#define O_TILE_WIDTH    4
#define I_TILE_WIDTH    O_TILE_WIDTH+MASK_WIDTH-1

__constant__ int M[MASK_WIDTH];
……
int main(){
……
    // Set Grid/Block Dimensions
    int T = O_TILE_WIDTH;
    int B = (int)ceil((float)ARRAY_SIZE/O_TILE_WIDTH);

    // Launch Kernel
    convolution<<<B, T>>>(d_N,d_P,(int)ceil((float)I_TILE_WIDTH/O_TILE_WIDTH));
……

}
```

Practice: Tiled 1D Convolution Design #2

- **Copy Sample Skeleton Code**
  - cp –r /home/share/19_Convolution ./[FolderName]
  - cd [FolderName]
- **Notepad: TiledConvolution.cu 코드 Kernel function 완성**
- **Compile & run program**
  - make
  - ./EXE

Design#2: 1D Tiled Convolution Kernel

```
__global__ void convolution(int* N, int* P)
{
    __shared__ int Nds[I_TILE_WIDTH];

    int tx = threadIdx.x;
    int index_o = blockIdx.x * O_TILE_WIDTH + tx;
    int index_i = index_o-(MASK_WIDTH/2);
    int pValue = 0;

    if(index_i >= 0 && index_i < ARRAY_SIZE) {
        Nds[tx] = N[index_i];
    }else{
        Nds[tx] = 0;
    }
    __syncthreads();

    if (tx < O_TILE_WIDTH && index_o < ARRAY_SIZE){

        for(int i = 0; i < MASK_WIDTH; i++) {
            pValue += M[i] * Nds[i+tx];
        }

        P[index_o] = pValue;

    }

}
```

Design#2: 1D Tiled Convolution main

```
#define ARRAY_SIZE      128
#define MASK_WIDTH      5
#define O_TILE_WIDTH    4
#define I_TILE_WIDTH    O_TILE_WIDTH+MASK_WIDTH-1

__constant__ int M[MASK_WIDTH];
……
int main(){
……
    // Set Grid/Block Dimensions
    int T = I_TILE_WIDTH;
    int B = (int)ceil((float)ARRAY_SIZE/O_TILE_WIDTH);

    // Launch Kernel
    convolution<<<B, T>>>(d_N,d_P);
……

}
```

Design#2: 2D Tiled Convolution Kernel

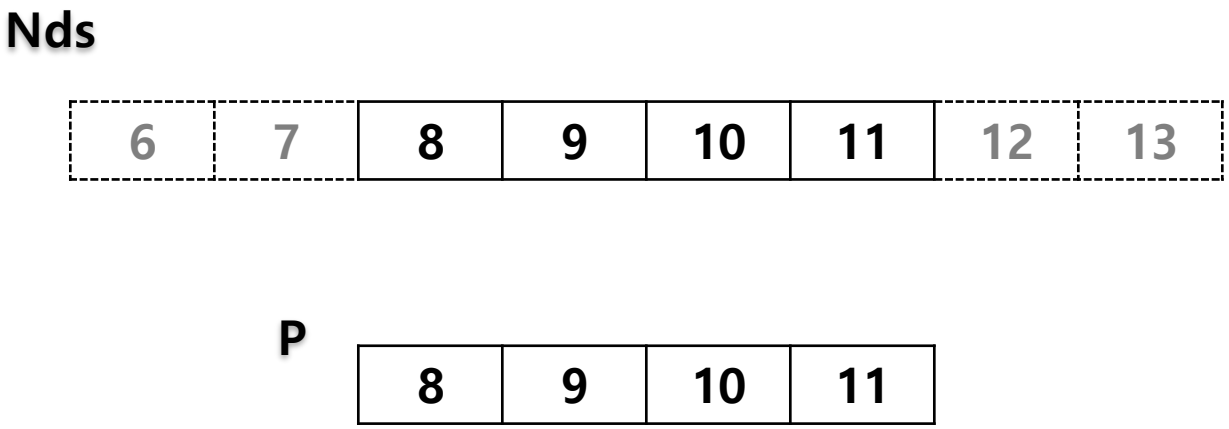
```
__global__ void convolution(int* N, int* P)
{
    __shared__ int Nds[I_TILE_WIDTH][I_TILE_WIDTH];

    int tx = threadIdx.x; int ty = threadIdx.y;
    int Col_o = blockIdx.x * O_TILE_WIDTH + tx;
    int Row_o = blockIdx.y * O_TILE_WIDTH + ty;
    int Col_i = Col_o-(MASK_WIDTH/2);
    int Row_i = Row_o-(MASK_WIDTH/2);
    int pValue = 0;

    if(Col_i >= 0 && Col_i < ARRAY_WIDTH && Row_i >= 0 && Row_i < ARRAY_HEIGHT) {
        Nds[ty][tx] = N[Row_i*ARRAY_WIDTH+Col_i];
    }else{
        Nds[ty][tx] = 0;
    }
    __syncthreads();

    if (tx<O_TILE_WIDTH && ty<O_TILE_WIDTH && Col_o<ARRAY_WIDTH && Row_o<ARRAY_HEIGHT){

        for(int i = 0; i < MASK_WIDTH; i++) {
            for(int j = 0; j < MASK_WIDTH; j++){
                pValue += M[i][j] * Nds[i+ty][j+tx];
            }
        }
        P[Row_o*ARRAY_WIDTH+Col_o] = pValue;
    }

}
```

Design#2: 2D Tiled Convolution main

```
#define ARRAY_WIDTH     1024
#define ARRAY_HEIGHT    512
#define MASK_WIDTH      5
#define O_TILE_WIDTH    8
#define I_TILE_WIDTH    O_TILE_WIDTH+MASK_WIDTH-1

__constant__ int M[MASK_WIDTH];
……
int main(){
……
    // Set Grid/Block Dimensions
    dim3 T(I_TILE_WIDTH,I_TILE_WIDTH);
    dim3 B((int)ceil((float)ARRAY_WIDTH/O_TILE_WIDTH),
           (int)ceil((float)ARRAY_HEIGHT/O_TILE_WIDTH));

    // Launch Kernel
    convolution<<<B, T>>>(d_N,d_P);
……

}
```

Evaluating Tiling Efficiency

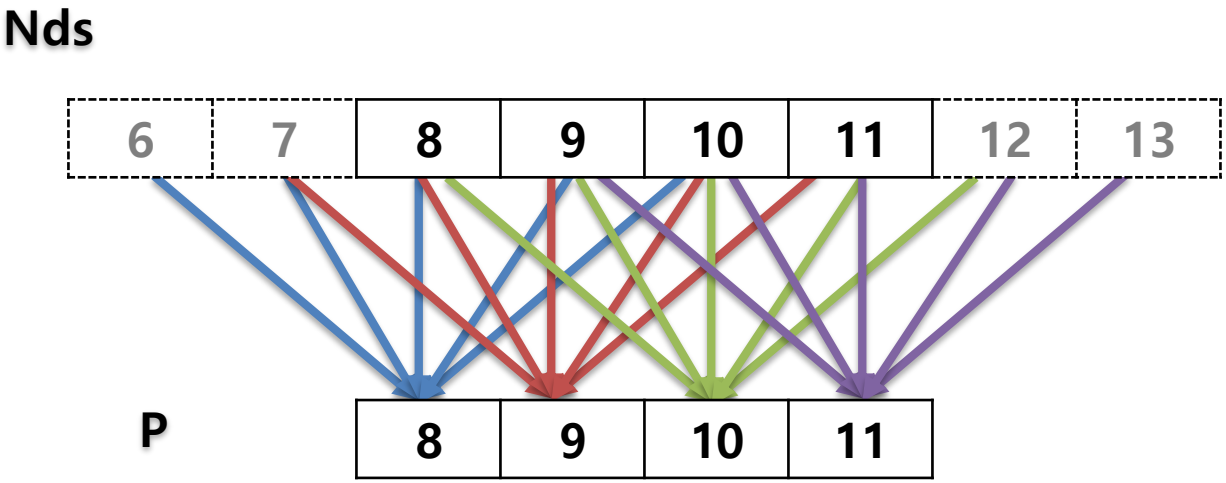
Example1: An 4-elements Convolution Tile



For Mask_Width=5, we load 4+5-1=8 elements
(8 memory loads)

Example1: Evaluating Re-use

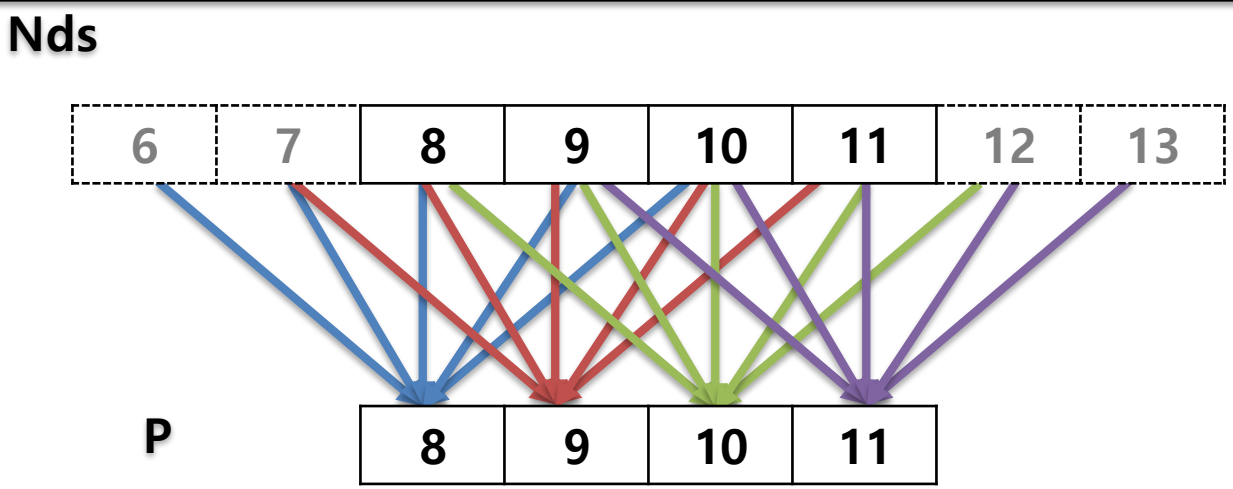
- P[8] uses N[6], N[7], N[8], N[9], N[10]
- P[9] uses N[7], N[8], N[9], N[10], N[11]
- P[10] use N[8], N[9], N[10], N[11], N[12]
- P[11] use N[9], N[10], N[11], N[12], N[13]



Example1: Evaluating Re-use

- 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[11] use N[9], N[10], N[11], N[12], N[13]

(4+5-1)=8 elements loaded
4*5 global memory accesses replaced by shared memory accesses
This gives a bandwidth reduction of 20/8 = 2.5



Example2: An 8-elements Convolution Tile

Nds

P

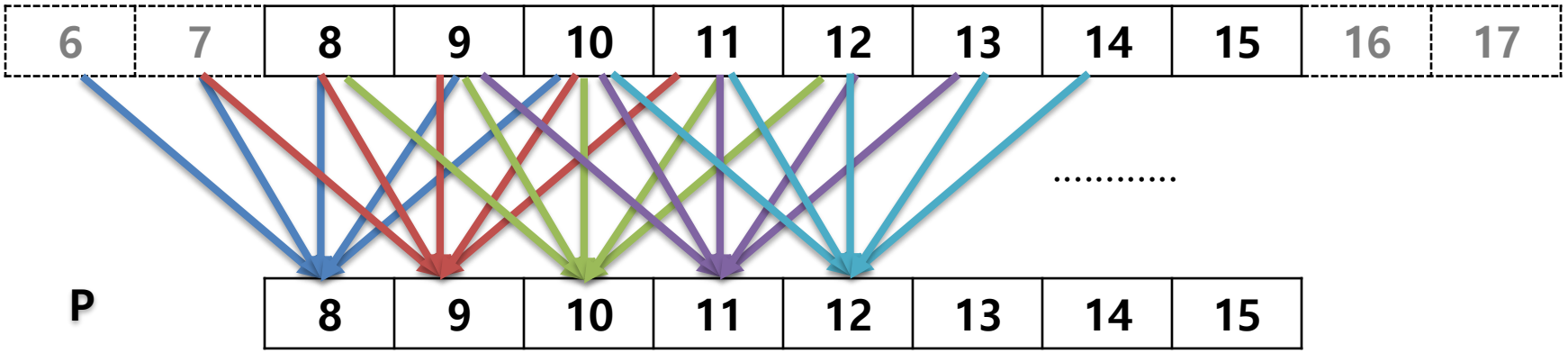
For Mask_Width=5, we load 8+5-1=12 elements
(12 memory loads)

Example2: Evaluating Reuse

- 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]

.....

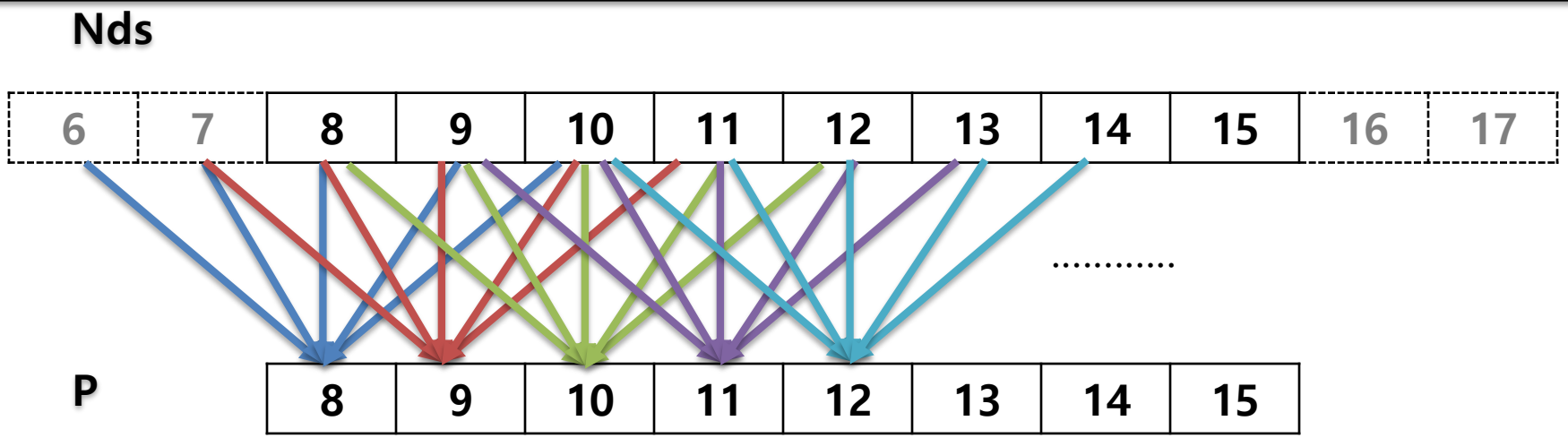
- P[14] uses N[12], N[13], N[14], N[15], N[16]
- P[15] uses N[13], N[14], N[15], N[16], N[17]



Example2: Evaluating Reuse

- 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]

(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



Evaluating Reuse: General 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

$$\frac{\text{O\_TILE\_WIDTH} \times \text{MASK\_WIDTH}}{\text{O\_TILE\_WIDTH}+\text{MASK\_WIDTH}-1}$$

  - This ignores ghost elements in edge tiles.

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

Evaluating Reuse: 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
- $O\_TILE\_WIDTH^2 \times MASK\_WIDTH^2$ global memory accesses are converted into shared memory accesses
- The reduction ratio is

$$\frac{O\_TILE\_WIDTH^2 \times MASK\_WIDTH^2}{(O\_TILE\_WIDTH+MASK\_WIDTH-1)^2}$$

Bandwidth Reduction for 2D

- The reduction ratio is

$$\frac{\text{O\_TILE\_WIDTH}^2 \times \text{MASK\_WIDTH}^2}{(\text{O\_TILE\_WIDTH}+\text{MASK\_WIDTH}-1)^2}$$

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 the memory bandwidth reduction ratio.
- This often argues for larger shared memory size