CUDA Threads

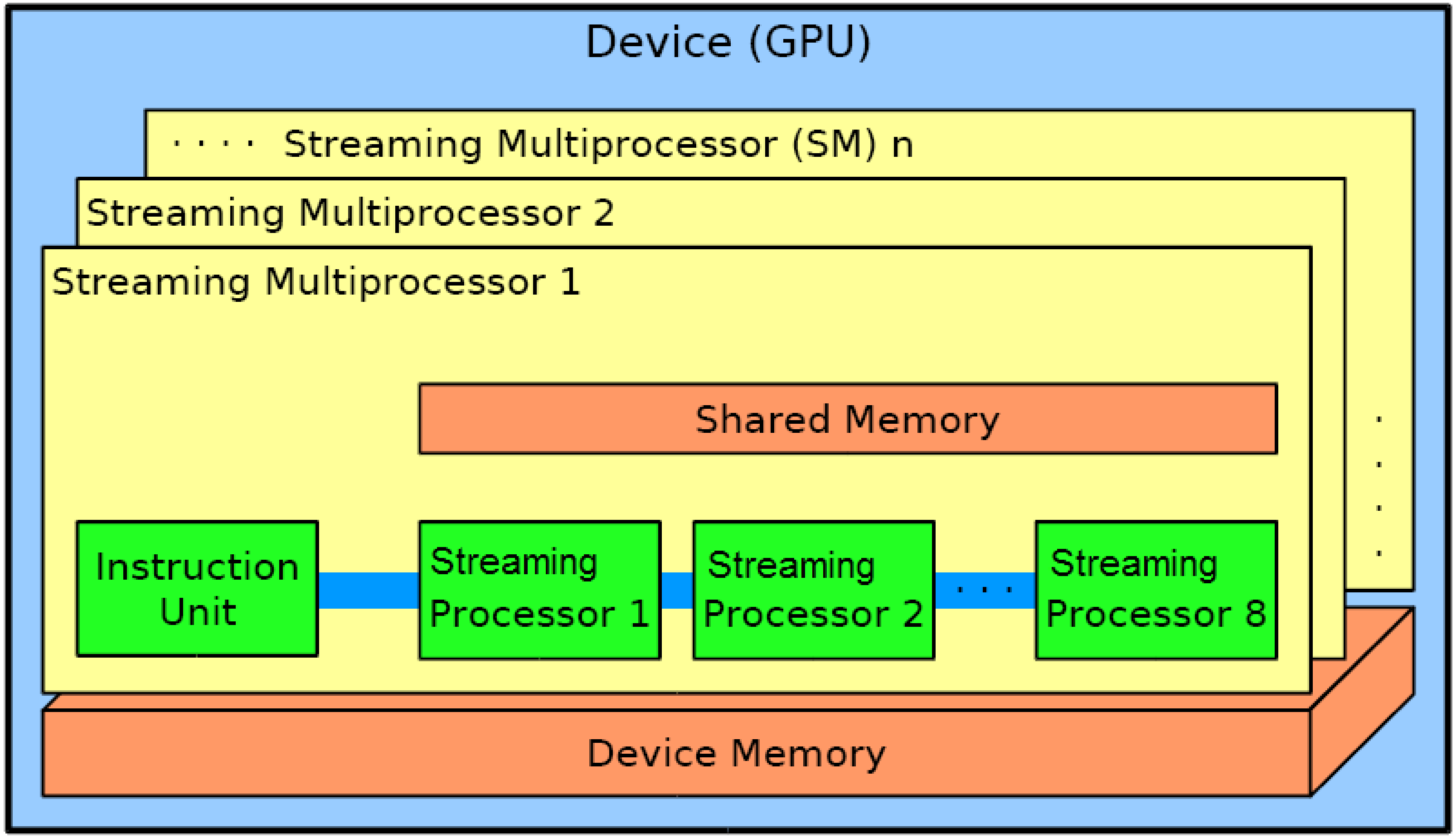
Sudsanguan Ngamsuriyaroj
Ekasit Kijsipongse
Ittipon Rassameeroj

Semester 1/2022

Topics

- CUDA Kernel and Memory
- Grid and Thread Blocks
- 2D/3D Grids
- Case Study: Matrix Multiplication

GPU Architecture



Kernel Functions

- Functions that are executed on GPU
- They are C functions with some restrictions
  - Must return void
  - No variable number of arguments
  - No access to host memory and host functions
  - No static variables
  - ~~No recursion~~ Solved by dynamic parallelism.
- Must be declared with a qualifier
  - **`__global__`** called by host
  - **`__device__`** called only by other kernels

Launching Kernels

- Modified C function call syntax:

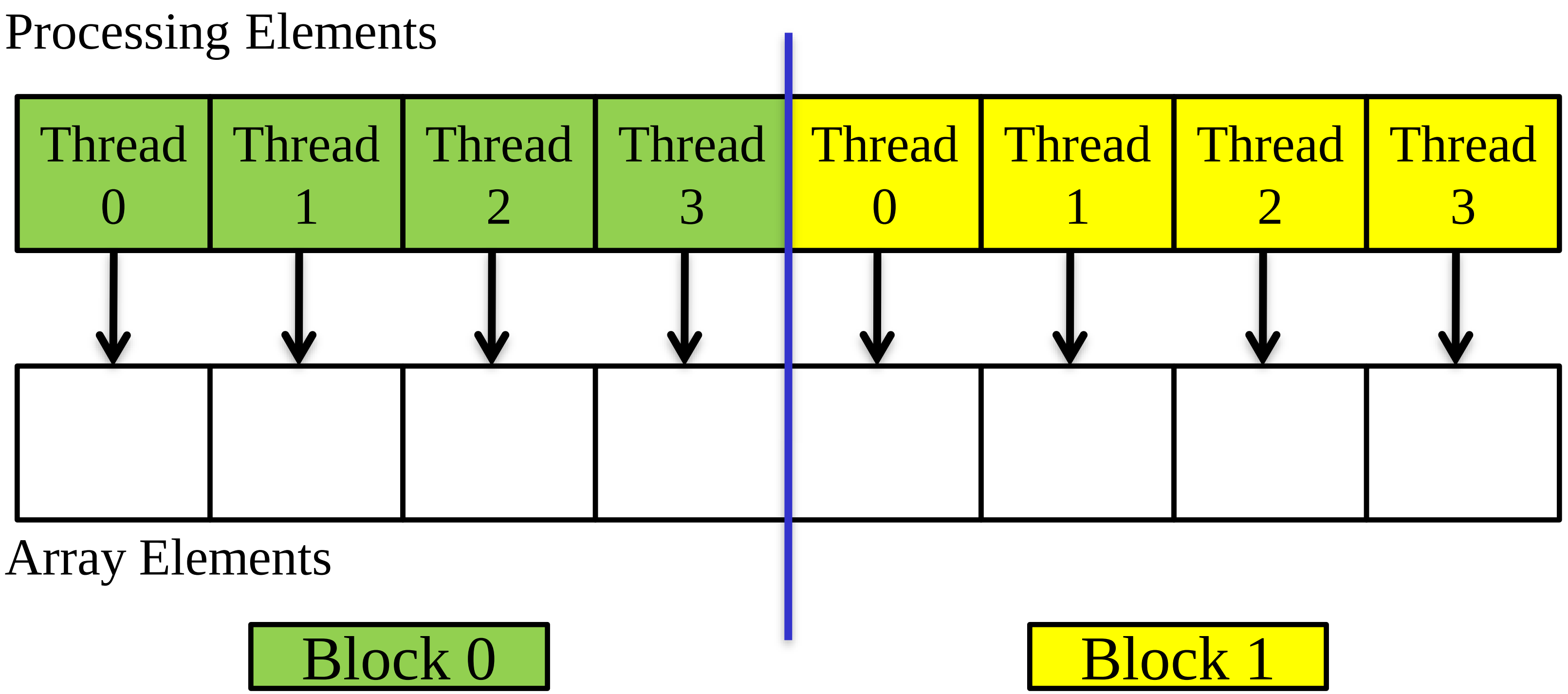
```
kernel<<< dimGrid, dimBlock >>>(args);
```

- Execution Configuration (<<< >>>)
  - dimGrid = dimension and size of grid (number of blocks)
  - dimBlock = dimension and size of thread block (number of threads in a block)

Thread Block

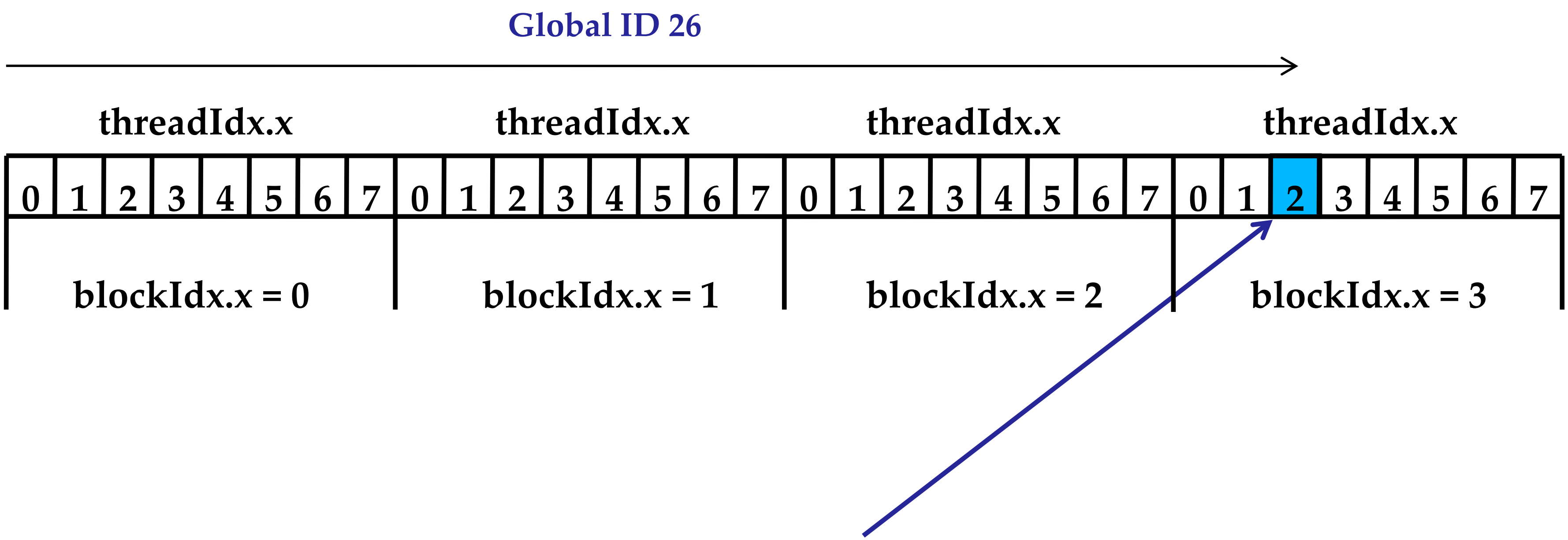
- Threads are organized into blocks
- 1 block has up to 1024 threads (depending on model)
- Each threads has a unique thread ID within a block
- Each blocks also has a unique block ID
- Each thread uses **block ID and thread ID** to decide what data to work on
- Device memory is shared by all threads.
- Shared memory is shared among threads in the same block.
- Threads in the same block can synchronize while threads in different blocks cannot cooperate

Thread Block Example



Calculating Global Thread ID -- x direction

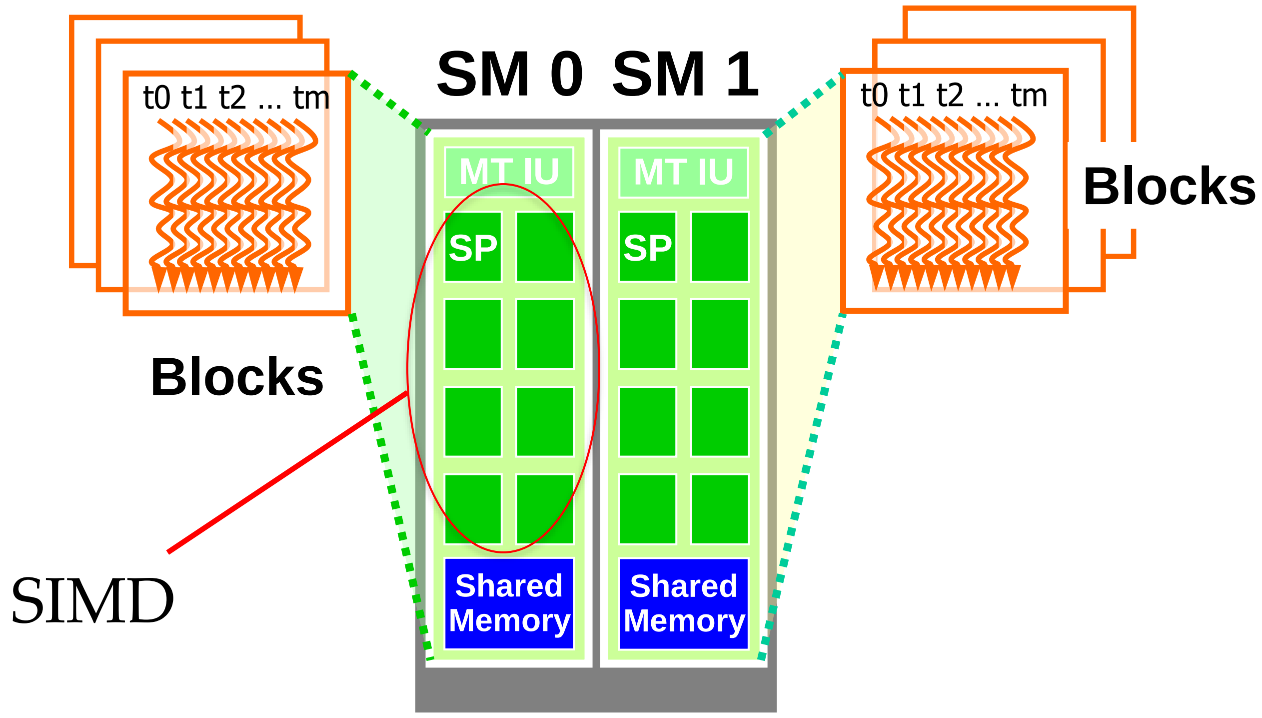
4 blocks, each having 8 threads



Global thread ID = **blockIdx.x * blockDim.x + threadIdx.x**
= 3 * 8 + 2 = thread 26 with linear global addressing

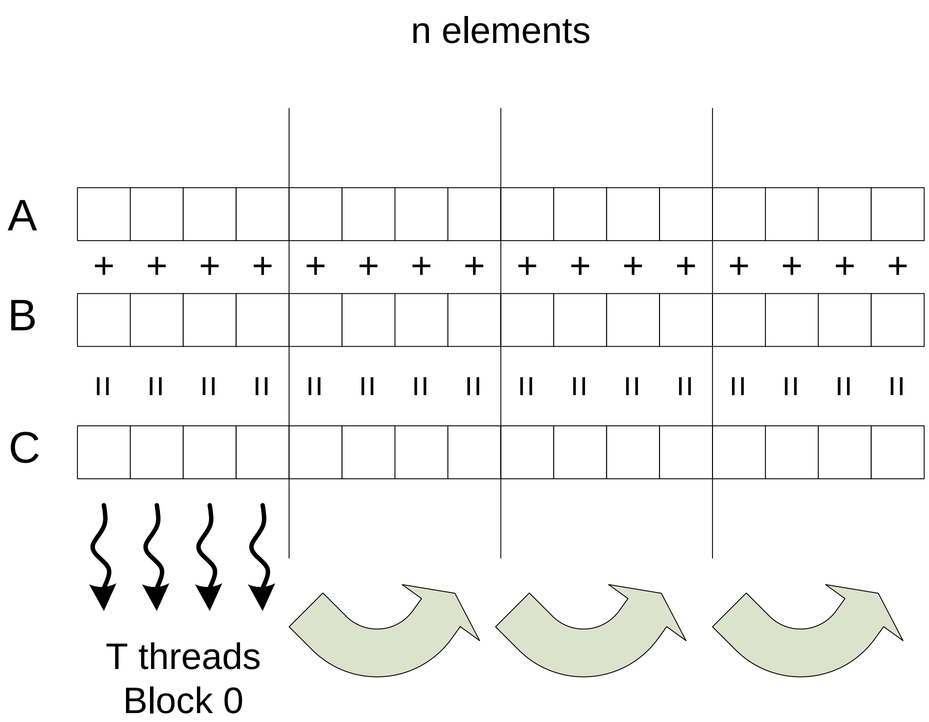
Thread Block Execution

- Each block is dispatched to an SM for execution
- Each block can execute in any order relative to other blocks!



Vector addition when n > T

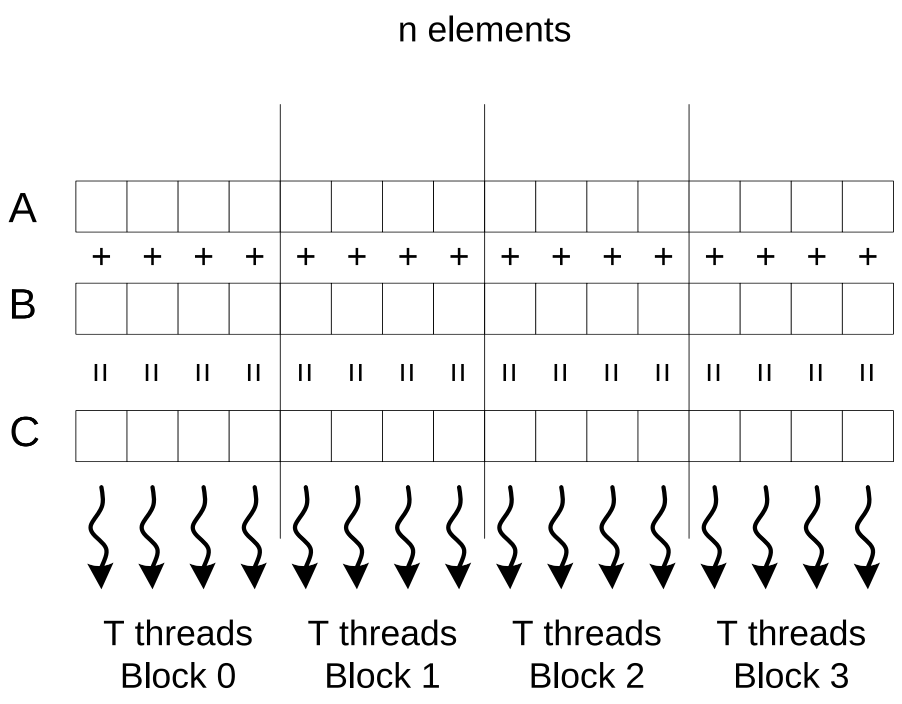
- Use 1 block of T threads (vecAdd2.cu)
- If 1 block is active, low GPU utilization



Recall the previous version from last lecture

Vector addition using when n > T (Improved)

- Use multiple blocks, each of T threads
- Multiple blocks can be executed in parallel on SMs
- Higher utilization



Example with multiple blocks

```
#define n 1024     // size of vectors
#define T 256      // number of threads per block

__global__ void vecAdd(int *A, int *B, int *C) {

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

   C[i] = A[i] + B[i];
}

int main (int argc, char *argv[] ) {
   …

   vecAdd<<<n/T, T>>>(devA, devB, devC);     // assume n/T is an integer

   …

}
```

Number of threads in a block

Number of blocks

vecAdd3.cu

```
#include <stdio.h>

#define n 1024        /* n must be a multiple of T */
#define T 256

__global__ void vecAdd(float *A, float *B, float *C) {
        int i;

        i = blockIdx.x*blockDim.x + threadIdx.x;
        C[i] = A[i] + B[i];
    }
}

int main (int argc, char *argv[] ) {

    int i;
    int size = n *sizeof(float);
    float  a[n], b[n], c[n], *devA, *devB, *devC;

    for (i=0; i < n; i++) {
        a[i] = 1; b[i] = 2;
    }
```

```
    cudaMalloc( (void**)&devA, size);
    cudaMalloc( (void**)&devB, size);
    cudaMalloc( (void**)&devC, size);

    cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);

    int nblocks = n/T;

    vecAdd<<<nblocks, T>>>(devA, devB, devC);

    cudaMemcpy( c, devC, size, cudaMemcpyDeviceToHost);
    cudaFree( devA);
    cudaFree( devB);
    cudaFree( devC);

    for (i=0; i < n; i++) {
       printf("%f ",c[i]);
    }
    printf("\n");

}
```

vecAdd4.cu: If n/T not necessarily an integer

```
#define n 1029     // size of vectors
#define T 256      // number of threads per block

__global__ void vecAdd(int *A, int *B, int *C) {

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

   if (i < n)                    // allows for more threads than vector elements
       C[i] = A[i] + B[i];     // some unused
}

int main (int argc, char *argv[] ) {

  int blocks = (n + T - 1) / T;   // efficient way of rounding to next integer
   …
   vecAdd<<<blocks, T>>>(devA, devB, devC);
   …

}
```

CUDA Programming Model

int main() {

CPU Serial Code

GPU Parallel Kernel

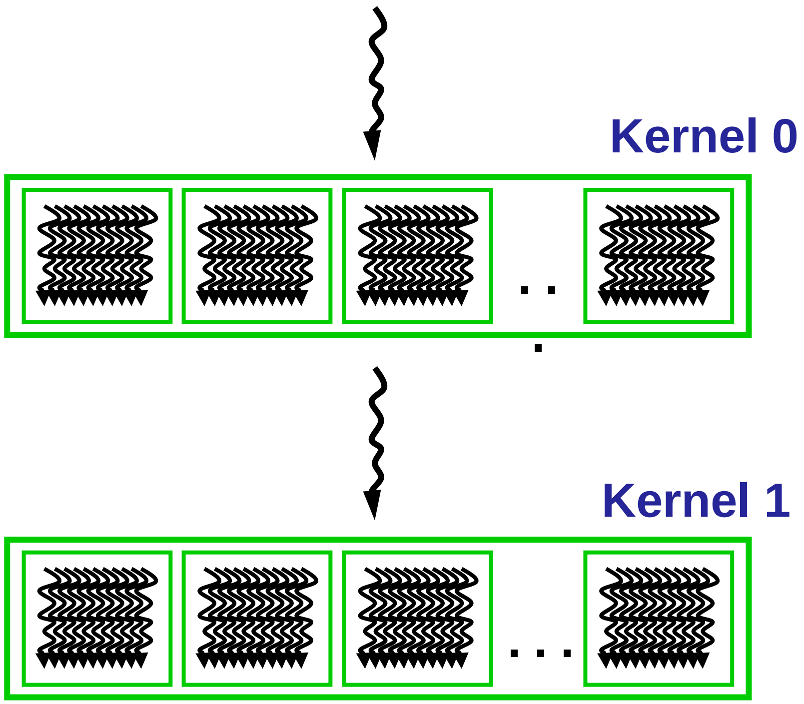
KernelA<<< nBlk, nT >>>(args);

CPU Serial Code

GPU Parallel Kernel

KernelB<<< nBlk, nT >>>(args);

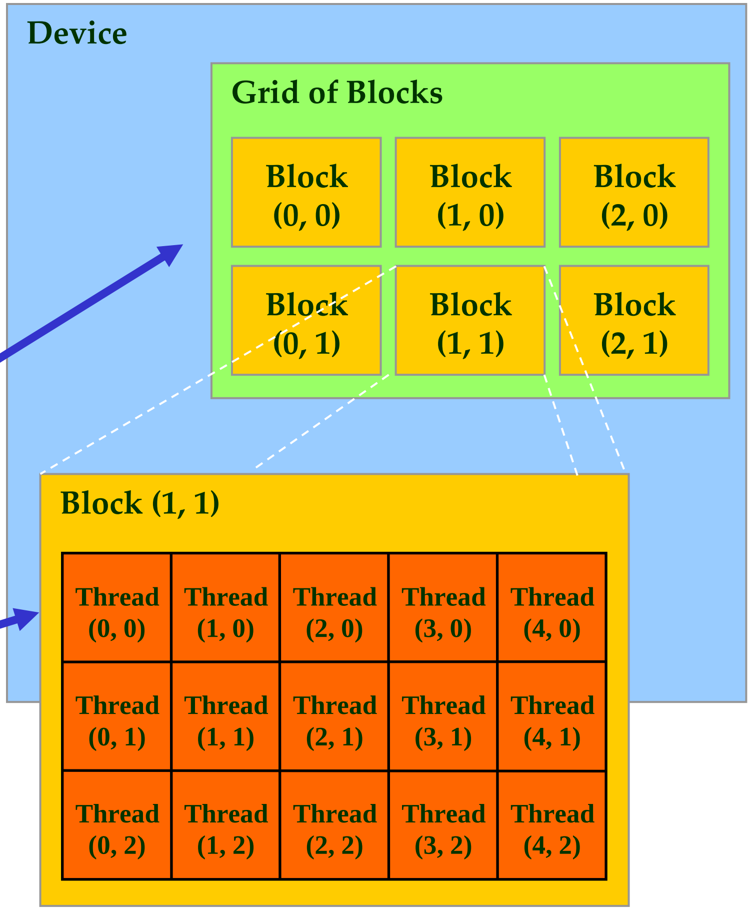
}



Block and Thread Shapes

- Blocks and Threads can be organized in 1D, 2D, or 3D to fit to applications

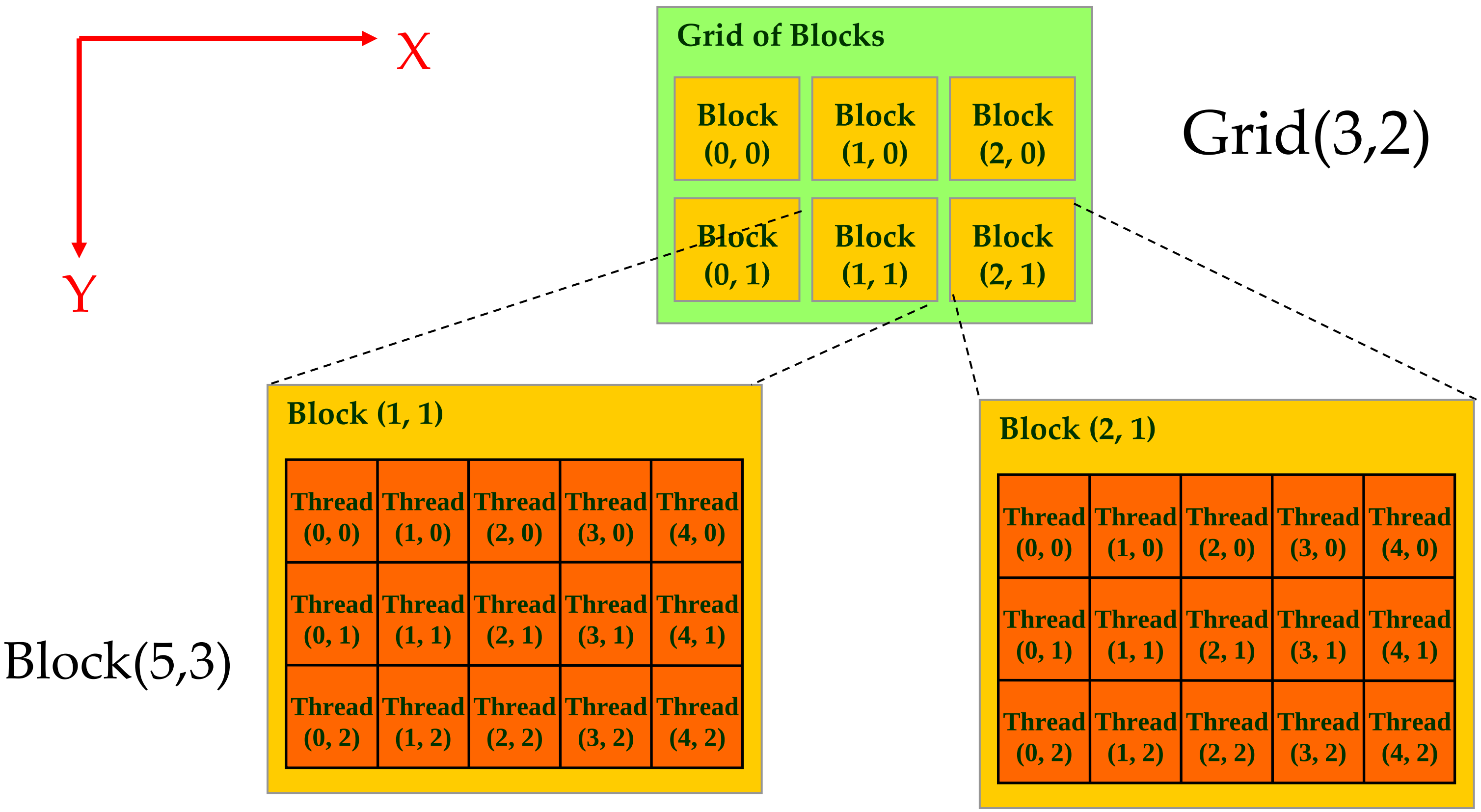
- Threads and blocks have IDs in each direction
  - Block ID: 1D or 2D (blockIdx.x, blockIdx.y)
  - Thread ID: 1D, 2D, or 3D (threadIdx.x, threadIdx.y, threadIdx.z)



Defining Grid/Block Dimensions

(X,Y)

```
dim3    DimGrid(3,2);     // 6 blocks
dim3    DimBlock(5,3);    // 15 threads per block
```



Built-in Variables for Grid/Block Dimensions

(X,Y)

```
dim3    DimGrid(3,2);     // 6 blocks
dim3    DimBlock(5,3);    // 15 threads per block

KernelFunc<<< DimGrid, DimBlock >>>(...);
```

depends on GPU model

Number of threads in a block = 5 *3 = 15 ≤ 1024
Full global thread ID in x and y dimensions can be computed by:

x = blockIdx.x * blockDim.x + threadIdx.x;
y = blockIdx.y * blockDim.y + threadIdx.y;

Memory Layout of a Matrix in C

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

A matrix is represented by 2D array.

M

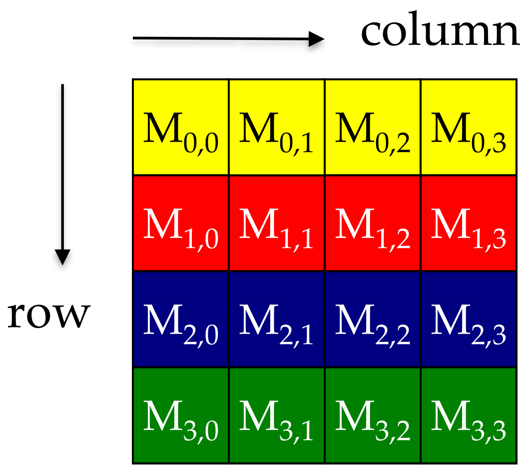
Row-major Order

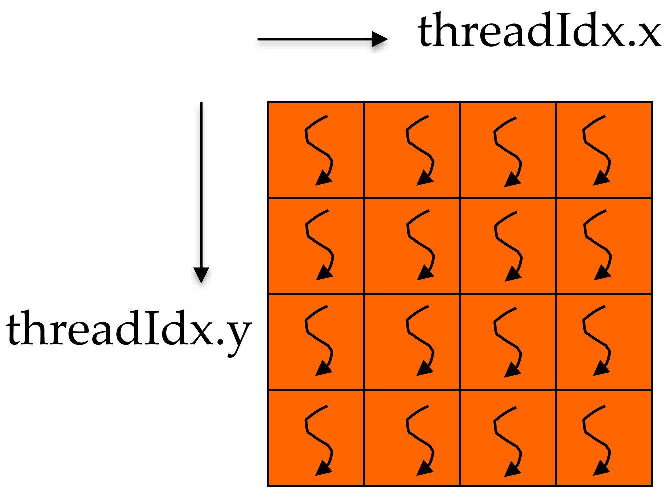
$M_{0,0}$	$M_{0,1}$	$M_{0,2}$	$M_{0,3}$	$M_{1,0}$	$M_{1,1}$	$M_{1,2}$	$M_{1,3}$	$M_{2,0}$	$M_{2,1}$	$M_{2,2}$	$M_{2,3}$	$M_{3,0}$	$M_{3,1}$	$M_{3,2}$	$M_{3,3}$

offset = row*rowsize+column

e.g. M[2][1] = M[2*4+1] = M[9]

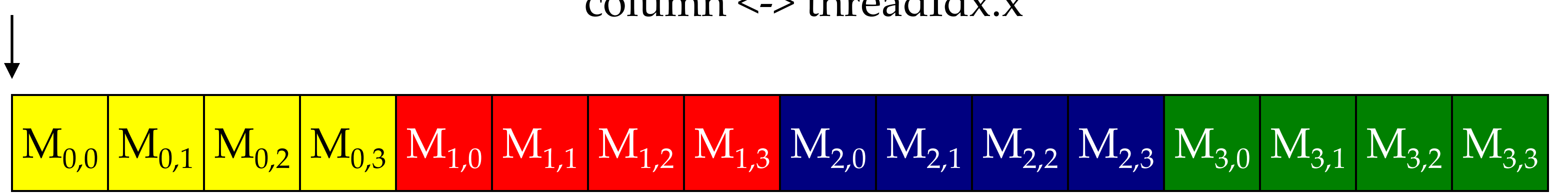
Mapping Matrix to CUDA Threads





row <-> threadIdx.y
column <-> threadIdx.x

M



offset = row*rowsize+column
offset = threadIdx.y*rowsize + threadIdx.x

MatAdd.cu: example using 2-D grid with a 2-D thread block to add two matrices

```
#define N 16     // size of matrix

__global__ void addMatrix (int *a, int *b, int *c) {
        int i = threadIdx.y;                    // row
        int j = threadIdx.x;                    // column
        int index = i*N + j;                    // find offset
        c[index]= a[index] + b[index]; // c[i][j] = a[i][j]+b[i][j];
}

int main(int argc, char *argv[]) {
        ...
        dim3 dimBlock (N,N);      // 16x16 threads
        dim3 dimGrid (1, 1);      // 1x1 blocks

        addMatrix<<<dimGrid, dimBlock>>>(devA, devB, devC);
        …
}
```

matAdd.cu

```
#include <stdio.h>

#define N 16    // size of N x N matrix

__global__ void addMatrix (float *a, float *b, float *c) {
    int i = threadIdx.y;
    int j = threadIdx.x;
    int index = i*N + j;
    c[index]= a[index] + b[index];
}

int main (int argc, char *argv[] ) {

    int i,j;
    int size = N * N *sizeof(float);
    float  a[N][N], b[N][N], c[N][N], *devA, *devB, *devC;

    for (i=0; i < N; i++) {
        for (j=0; j < N; j++) {
            a[i][j] = 1; b[i][j] = 2;
        }
    }

    cudaMalloc( (void**)&devA, size);
    cudaMalloc( (void**)&devB, size);
    cudaMalloc( (void**)&devC, size);
```

```
cudaMemcpy( devA, a, size, cudaMemcpyHostToDevice);
cudaMemcpy( devB, b, size, cudaMemcpyHostToDevice);

dim3 dimBlock (N,N);     // 16x16 threads
dim3 dimGrid (1,1);      // 1x1 blocks

addMatrix<<<dimGrid, dimBlock>>>(devA, devB, devC);

cudaMemcpy( c, devC, size, cudaMemcpyDeviceToHost);
cudaFree( devA);
cudaFree( devB);
cudaFree( devC);

for (i=0; i < N; i++) {
    for (j=0; j < N; j++) {
        printf("%.2f ",c[i][j]);
    }
    printf("\n");
}

}
```

MatAdd2.cu (When N > T)

```
#define N 64    // size of matrix

__global__ void addMatrix (int *a, int *b, int *c) {
        int i =blockIdx.y*blockDim.y+threadIdx.y;
        int j = blockIdx.x*blockDim.x+threadIdx.x;
        int index = i*N + j;
        c[index]= a[index] + b[index];
}

int main(int argc, char *argv[]) {
        ...
        dim3 dimBlock (16,16);     // 16x16 threads (T = 16)
        dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);    // 4x4 blocks

        addMatrix<<<dimGrid, dimBlock>>>(devA, devB, devC);
        …
}
```

matAdd3.cu: n/T may not be an integer

```
__global__ void addMatrix (int *a, int *b, int *c) {
        int i =blockIdx.y*blockDim.y+threadIdx.y;
        int j = blockIdx.x*blockDim.x+threadIdx.x;
        int index = i*N + j;
        if ( i < N && j < N)  c[index]= a[index] + b[index];
}

int main(int argc, char *argv[]) {
        ...
        dim3 dimBlock (16,16);     // 16x16 threads
        dim3 dimGrid ((N+16-1)/16, (N+16-1)/16);

        addMatrix<<<dimGrid, dimBlock>>>(devA, devB, devC);
        …
}
```

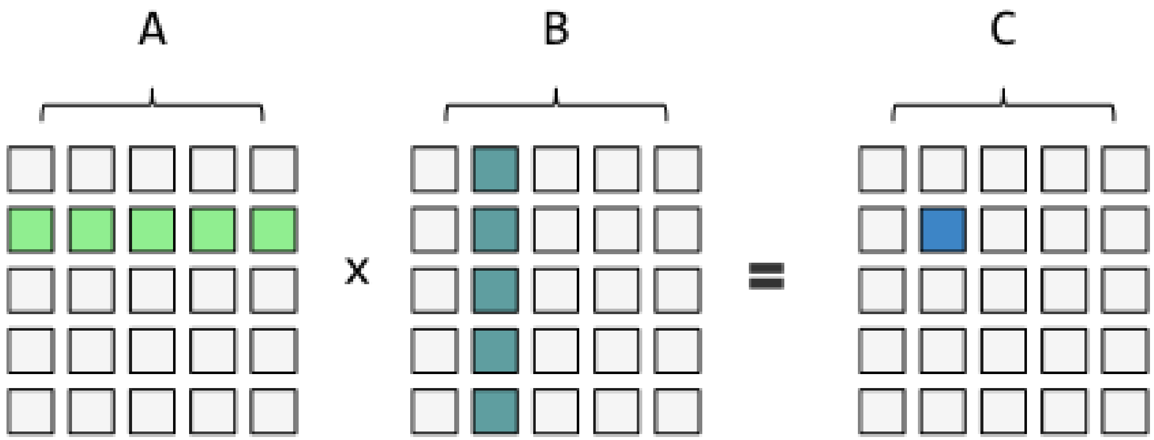
Case Study: Matrix Multiplication

- A simple matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Thread ID usage
  - Assume square matrix for simplicity

$$\begin{pmatrix} 5 & 2 & 6 & 1 \\ 0 & 6 & 2 & 0 \\ 3 & 8 & 1 & 4 \\ 1 & 8 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 7 & 5 & 8 & 0 \\ 1 & 8 & 2 & 6 \\ 9 & 4 & 3 & 8 \\ 5 & 3 & 7 & 9 \end{pmatrix} = \begin{pmatrix} 96 & 68 & 69 & 69 \\ 24 & 56 & 18 & 52 \\ 58 & 95 & 71 & 92 \\ 90 & 107 & 81 & 142 \end{pmatrix}$$

Matrix Multiplication

MATRIX MULTIPLICATION



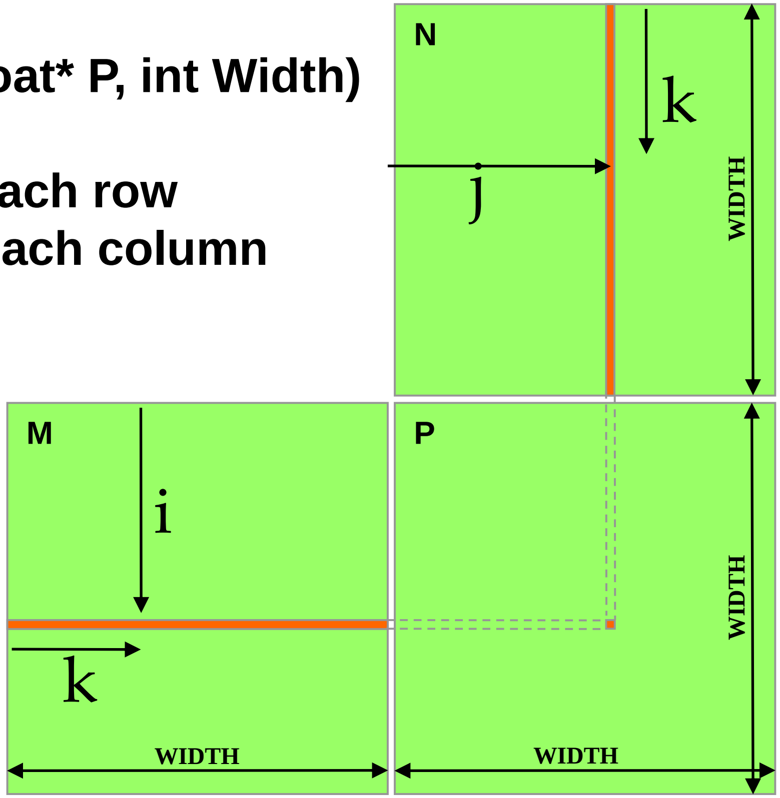

```
C[i][j] = sum(A[i][k] * B[k][j]) for k = 0 ... n
```

```
In our case:
C[1][1] =>
A[1][0]*B[0][1] + A[1][1]*B[1][1] + A[1][2]*B[2][1] + A[1][3]*B[3][1] + A[1][4]*B[4][1]
```

Matrix Multiplication
A Simple Host Version in C

// Matrix multiplication on the (CPU) host: sequential version

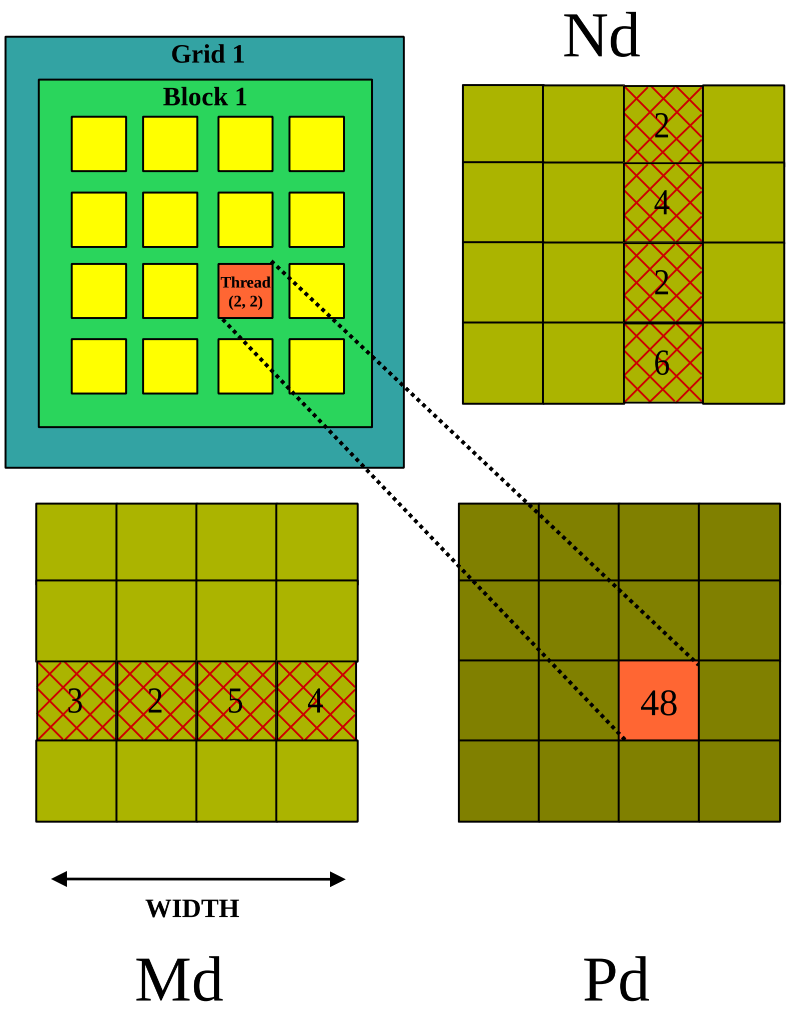
```
void MatrixMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; ++i)          // for each row
        for (int j = 0; j < Width; ++j) {    // for each column
            sum = 0.0;
            for (int k = 0; k < Width; ++k) {
                a = M[i * Width + k];   // M[i][k]
                b = N[k * Width + j];   // N[k][j]
                sum += a * b;
            }
            P[i * Width + j] = sum;   // P[i][j]
        }
}
```



P = M * N of size WIDTH x WIDTH

matmul.cu: For small matrix

- One thread block used
  - Each thread computes one element of Pd
- Each thread
  - Loads a row of matrix Md
  - Loads a column of matrix Nd
  - Perform one multiply and addition for each pair of Md and Nd elements
- Size of matrix limited by the number of threads allowed in a thread block



Elements of M and N used by a Thread

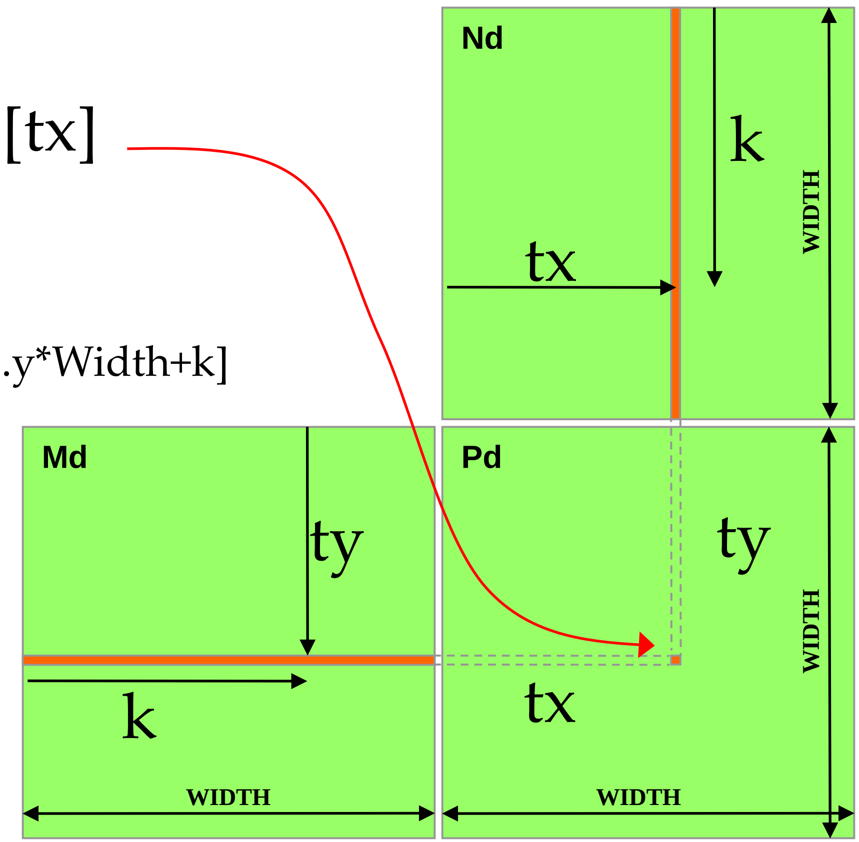
Nelement = Nd[k*Width+threadIdx.x]

N[k][tx]

Find Pd[ty][tx]

Melement = Md[threadIdx.y*Width+k]

M[ty][k]



matMul.cu

Passing a pointer to array

Passing a primitive value

```
#include <stdio.h>

#define Width 16      // size of Width x Width matrix

__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int ncols) {

    // Pvalue is used to store the element of the output matrix
    // that is computed by the thread

    float Pvalue = 0;
    for (int k = 0; k < ncols; ++k) {
        float Melement = Md[threadIdx.y*ncols+k];
        float Nelement = Nd[k*ncols+threadIdx.x];
        Pvalue += Melement * Nelement;
    }

    Pd[threadIdx.y*ncols+threadIdx.x] = Pvalue;
}
```

M[ty][k]
N[k][tx]

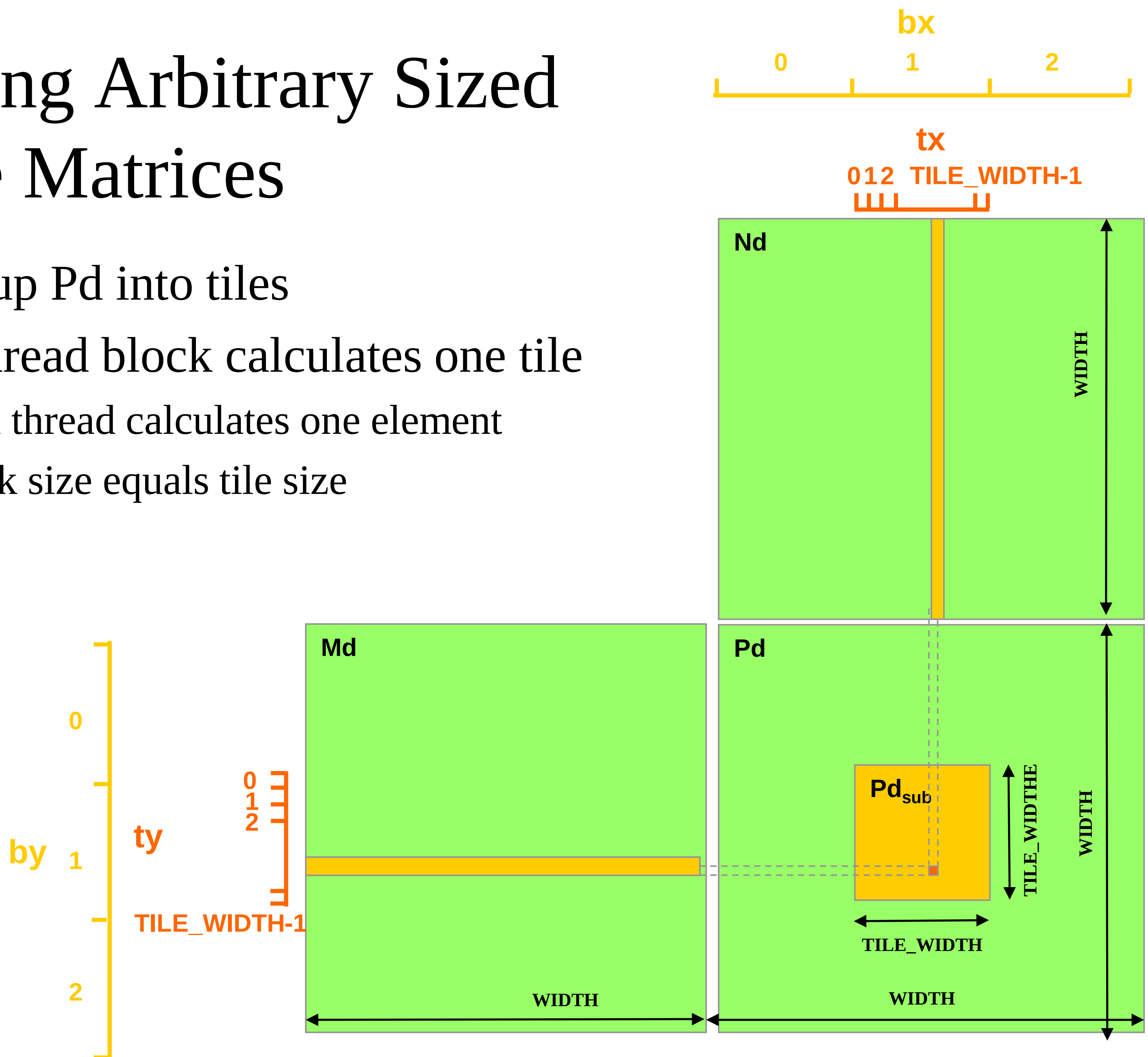
P[ty][tx]

```c
int main (int argc, char *argv[] ) {

    int i,j;
    int size = Width * Width * sizeof(float);
    float M[Width][Width],N[Width][Width],P[Width][Width];
    float* Md, *Nd, *Pd;

    for (i=0; i < Width; i++) {
        for (j=0; j < Width; j++) {
           M[i][j] = 1; N[i][j] = 2;
        }
    }

    cudaMalloc( (void**)&Md, size);
    cudaMalloc( (void**)&Nd, size);
    cudaMalloc( (void**)&Pd, size);

    cudaMemcpy( Md, M, size, cudaMemcpyHostToDevice);
    cudaMemcpy( Nd, N, size, cudaMemcpyHostToDevice);

    // Setup the execution configuration
    dim3 dimBlock(Width, Width);
    dim3 dimGrid(1, 1);

    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

```
    // Read P from the device
    cudaMemcpy(P, Pd, size, cudaMemcpyDeviceToHost);

    // Free device matrices
    cudaFree(Md); cudaFree(Nd); cudaFree (Pd);

    for (i=0; i < Width; i++) {
        for (j=0; j < Width; j++) {
            printf("%.2f ",P[i][j]);
        }
        printf("\n");
    }


}
```

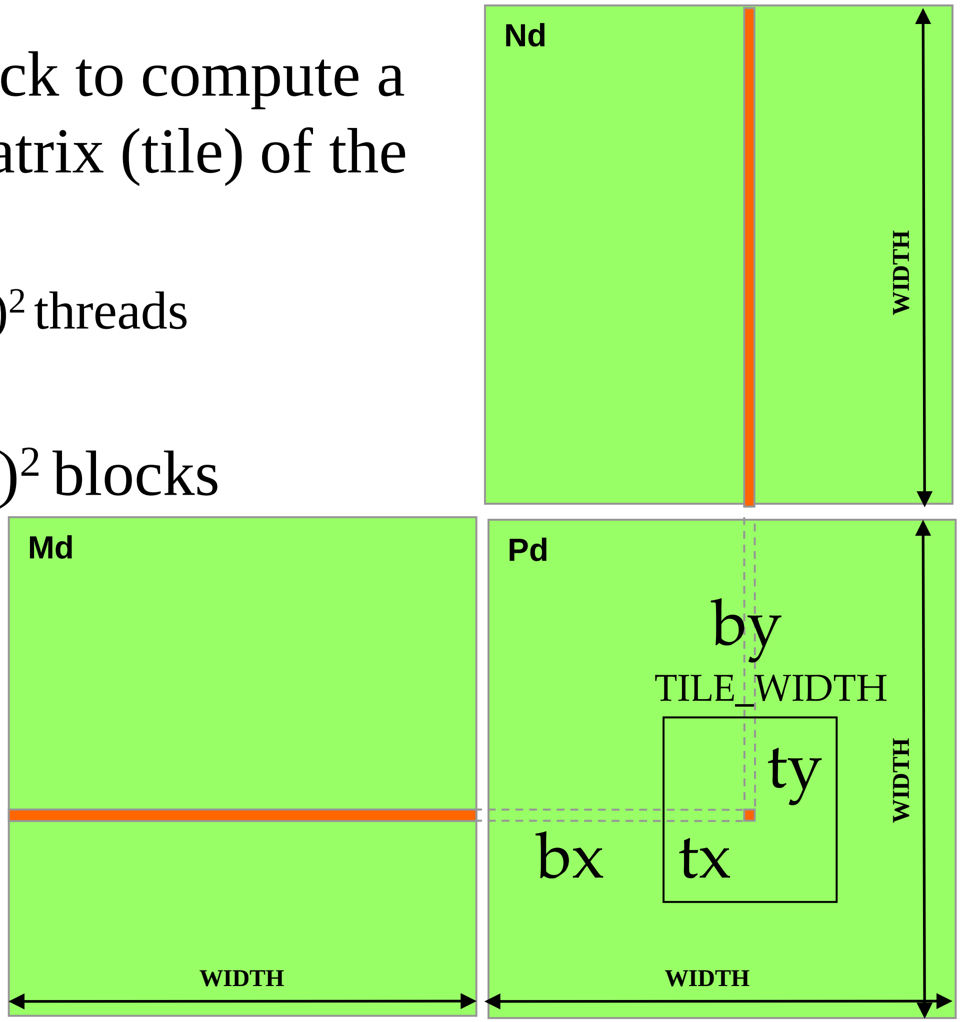
Handling Arbitrary Sized Square Matrices

- Break-up Pd into tiles
- Each thread block calculates one tile
  - Each thread calculates one element
  - Block size equals tile size



Matrix Multiplication Using Multiple Thread Blocks

- Have each 2D thread block to compute a (TILE_WIDTH)^2 sub-matrix (tile) of the result matrix
  - Each has (TILE_WIDTH)^2 threads
- Generate a 2D Grid of (WIDTH/TILE_WIDTH)^2 blocks



matmul2.cu: Matrix Multiplication Kernel using Multiple Blocks

```
__global__ void MatrixMulKernel(float* Md, float* Nd, float* Pd, int ncols)
{
  // Calculate the row index of the Pd element and M
  int Row = blockIdx.y*blockDim.y + threadIdx.y;

 // Calculate the column idenx of Pd and N
  int Col = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;

  // each thread computes one element of the block sub-matrix
  for (int k = 0; k < ncols; ++k)
    Pvalue += Md[Row*ncols+k] * Nd[k*ncols+Col];

  Pd[Row*ncols+Col] = Pvalue;
}
```

matmul2.cu: Matrix Multiplication Kernel using Multiple Blocks (Cont.)

```
#define Width  64     // must be multiple of TILE_WIDTH
#define TILE_WIDTH 16

int main(int argc, char *argv[]) {
    …
    // Setup the execution configuration
    dim3 dimGrid(Width/TILE_WIDTH, Width/TILE_WIDTH);
    dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);
    …
    // Launch the device computation threads!
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
    …
}
```

Q & A