GPU Computing

Lecture 4

Young-Ho Gong

광운대학교
KwangWoon University

Contents

- CUDA Kernel launch

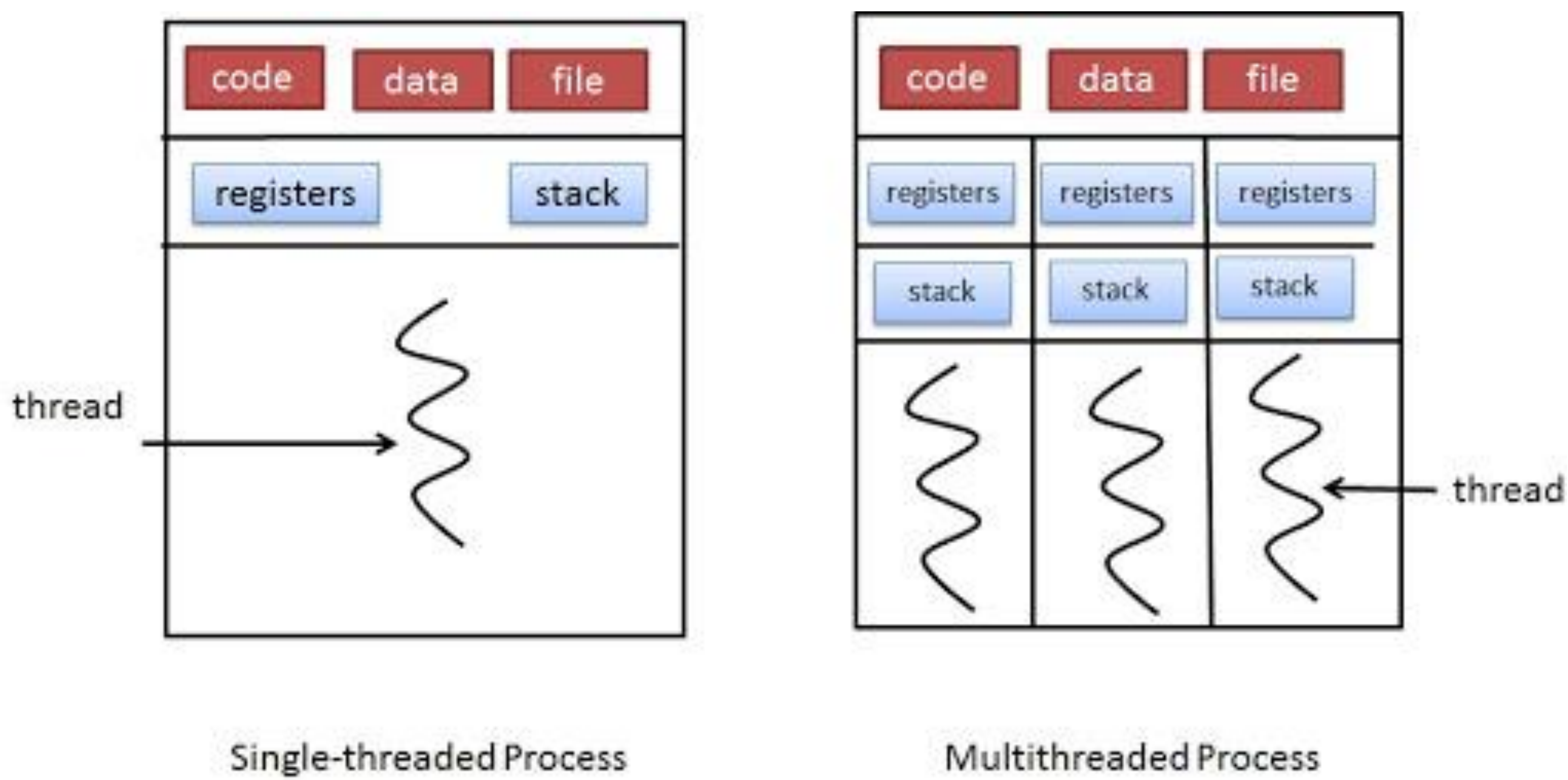
- Process and Thread
- CUDA programming model
- Kernel launch
- Pre-defined variables
- CUDA architecture

Process and Thread

- Computer process
  - An instance of a computer program that is being executed
  - Program code + current activity (or status data)
- Thread
  - A control flow in a computer process
  - The smallest sequence of instructions that can be managed independently by an OS scheduler

Operating System Supports

- Processes and threads are supported by operating systems



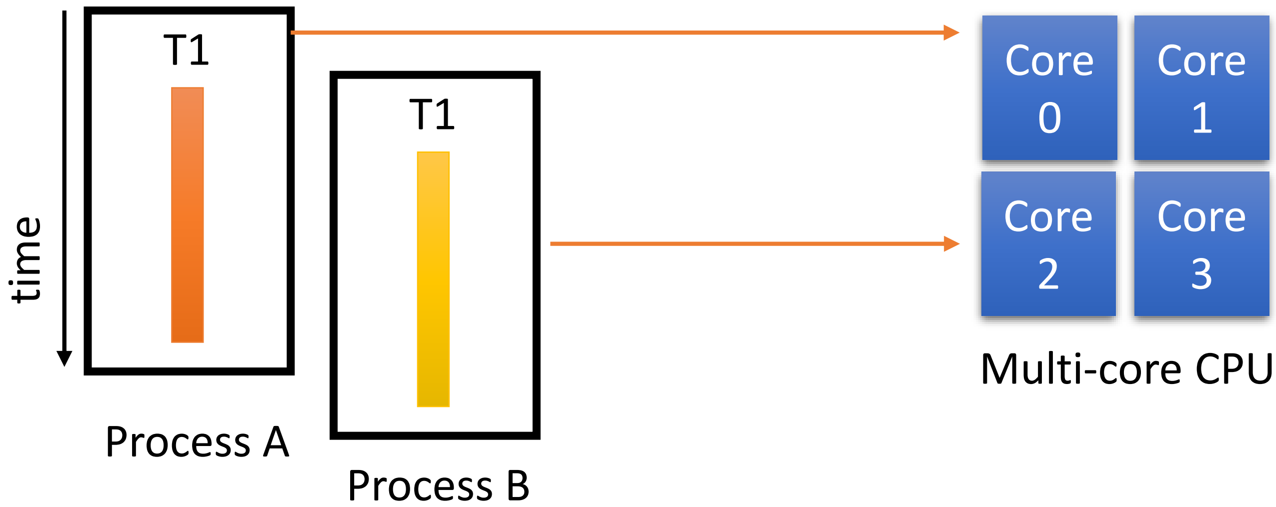
Single-core Processors

- Single thread

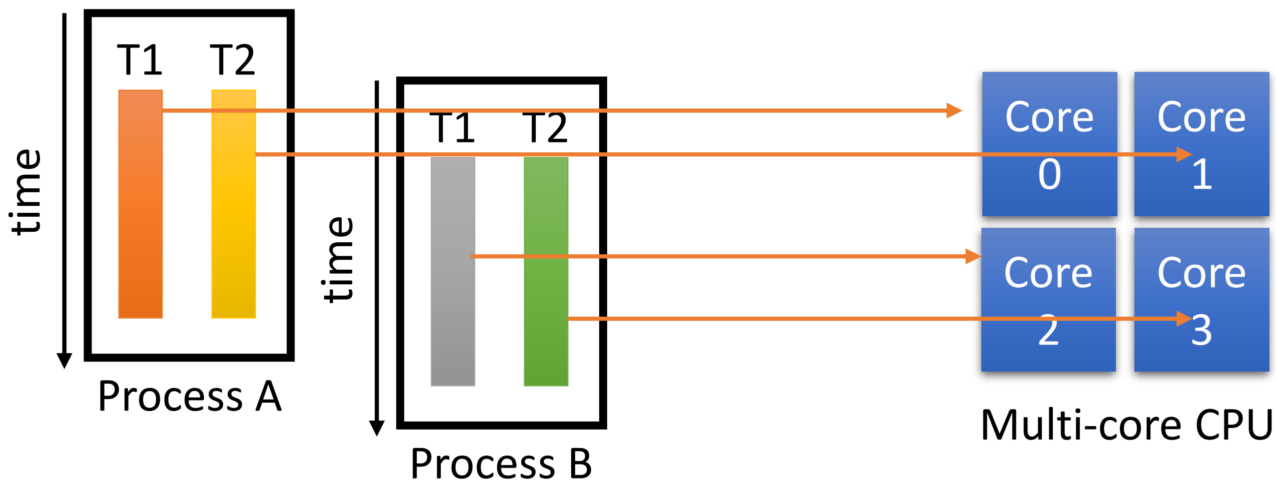
- Multiple thread → time sharing

Multi-core Processors

- Single thread, multiple process → Parallel processing



- Multiple thread → Parallel processing



CUDA programming model

- Parallel code (kernel) is launched and executed on a device by many threads
  - Multiple threads → ~ 10 threads
  - Many threads → > 1,000 threads

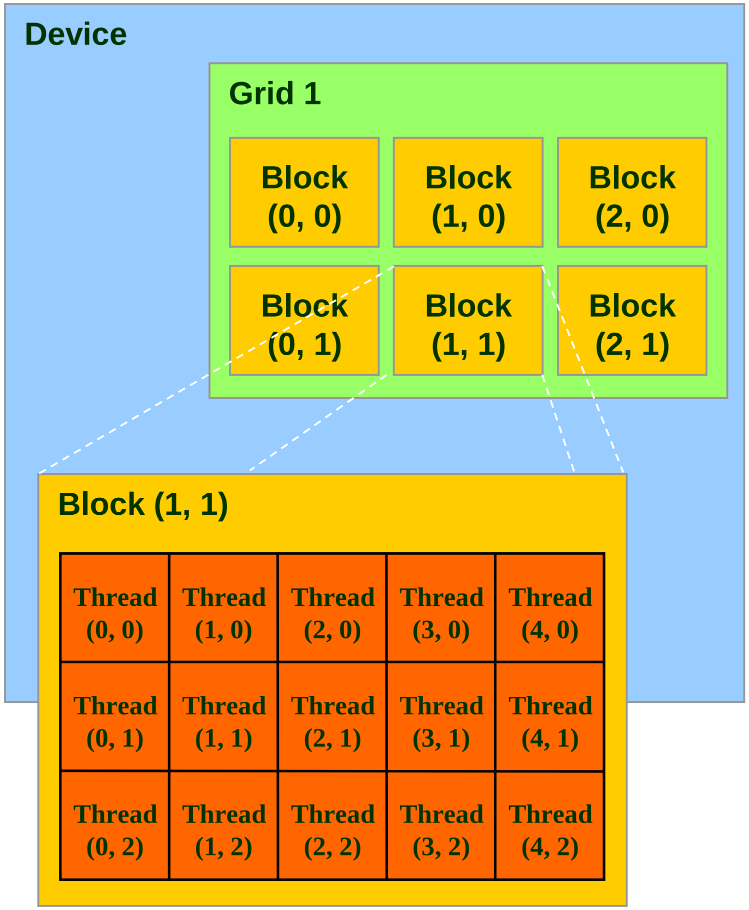
- On the many-core GPUs
  - Multi-core CPU → < 10 cores
  - Many-core GPU → > 1,000 cores (called CUDA cores)

CUDA programming model

- Many threads on many-core
  - For example,
    - 1,000,000 threads on 1,000 cores
- Launches are hierarchical: threads → blocks → grids
  - Threads are grouped into blocks
  - Blocks are grouped into grids
- Familiar serial code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

IDs and Dimensions

- Threads:
  - 3D IDs, unique within a block
- Blocks:
  - 2D IDs, unique within a grid
- Dimensions set at launch
  - Can be unique for each grid
- Built-in variables:
  - threadIdx, blockIdx
  - blockDim, gridDim



Calling a Kernel Function

- A kernel function must be called with an execution configuration:

```
__global__ void KernelFunc(...);
dim3  DimGrid(100,50); // 5000 thread blocks
dim3  DimBlock(4,8,8);  // 256 threads per block

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

CUDA Pre-defined Data Types

- Vector types
  - char1, uchar1, short1, ushort1, int1, uint1, long1, ulong1, float1
  - char2, uchar2, short2, ushort2, int2, uint2, long2, ulong2, float2
  - char3, uchar3, short3, ushort3, int3, uint3, long3, ulong3, float3
  - char4, uchar4, short4, ushort4, int4, uint4, long4, ulong4, float4
  - longlong1, ulonglong1, double1
  - longlong2, ulonglong2, double2
  - dim3
- Components are accessible as variable.x, variable.y, variable.z variable.w
  - As is in OpenGL graphics library
  - We can consider it as a coordinate value: (x,y,z)

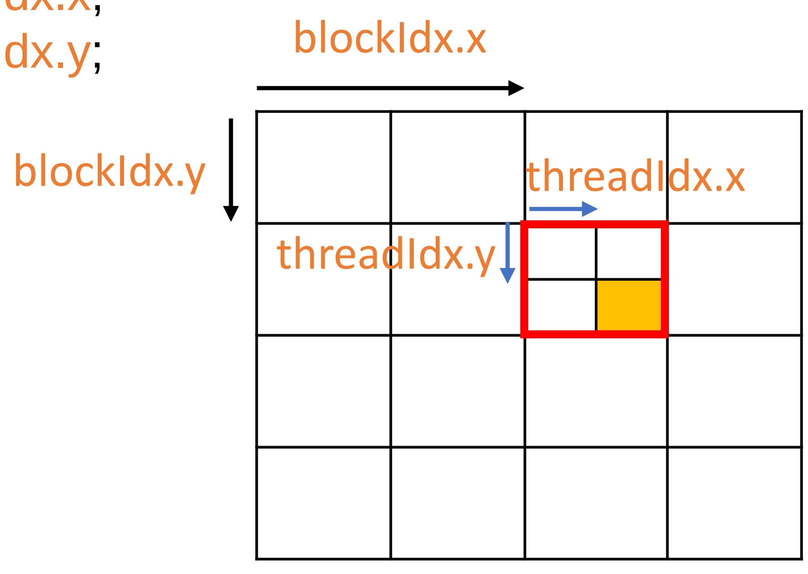
CUDA Pre-defined Variables

- Pre-defined variables
  - dim3 gridDim dimensions of grid
  - dim3 blockDim dimensions of block
  - uint3 blockIdx block index within grid
  - uint3 threadIdx thread index within block
  - int warpSize number of threads in warp
- Dim3 can take 1, 2, or 3 arguments:
  - dim3 blocks1D (5);
  - dim3 blocks2D (5,5);
  - dim3 blocks3D (5,5,5);

Kernel with 2D Indexing

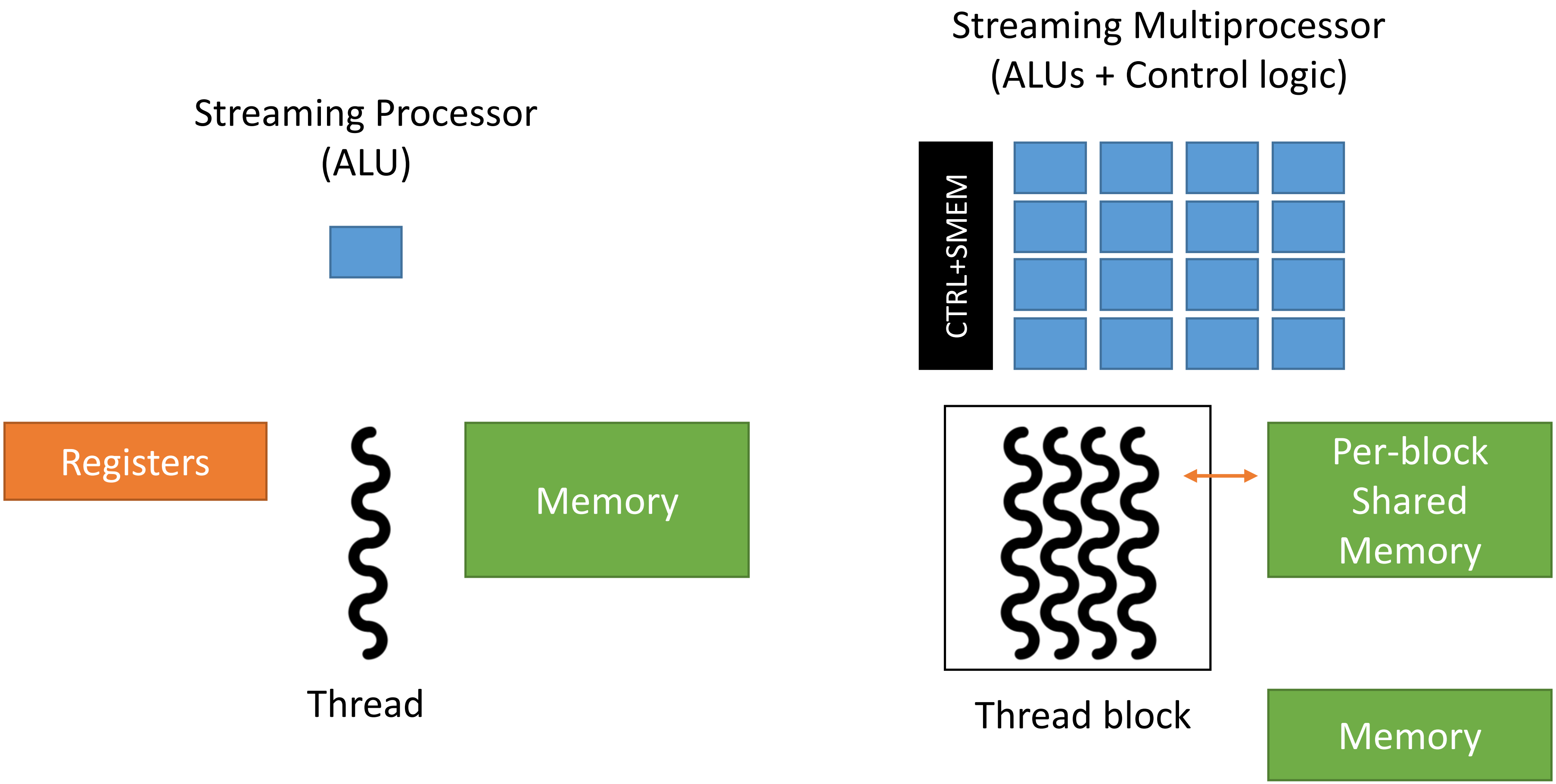
- dim3 DimGrid(4,4); → gridDim in kernel
  - Number of blocks in grid
- dim3 DimBlock(2,2); → blockDim in kernel
  - Number of threads in a block
- blockIdx, threadIdx: unique for each thread

```
__global__ void kernel( int *a, int dimx, int dimy )
{
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;

    a[idx]  = a[idx]+1;
}
```



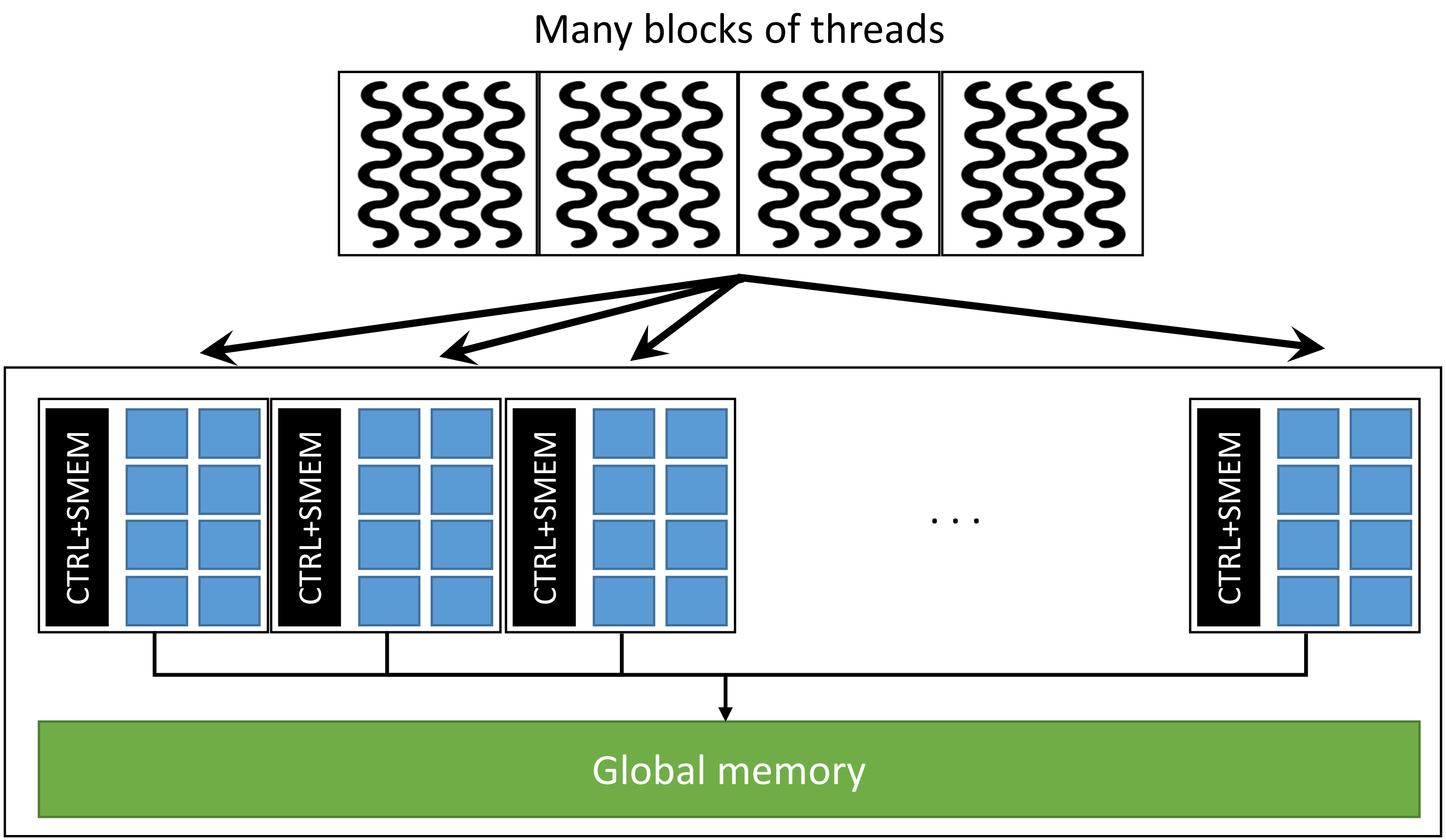
CUDA Architecture for Threads

- CUDA Streaming Multiprocessors



CUDA Architecture for Threads

- Each thread block goes to a SM
- Thread block queue!



Intermediate summary

- CUDA Kernel Launch

- Process and Thread
- CUDA Programming Model
- Kernel Launch
- Pre-defined variables
- CUDA Architecture

- Matrix Addition/Multiplication

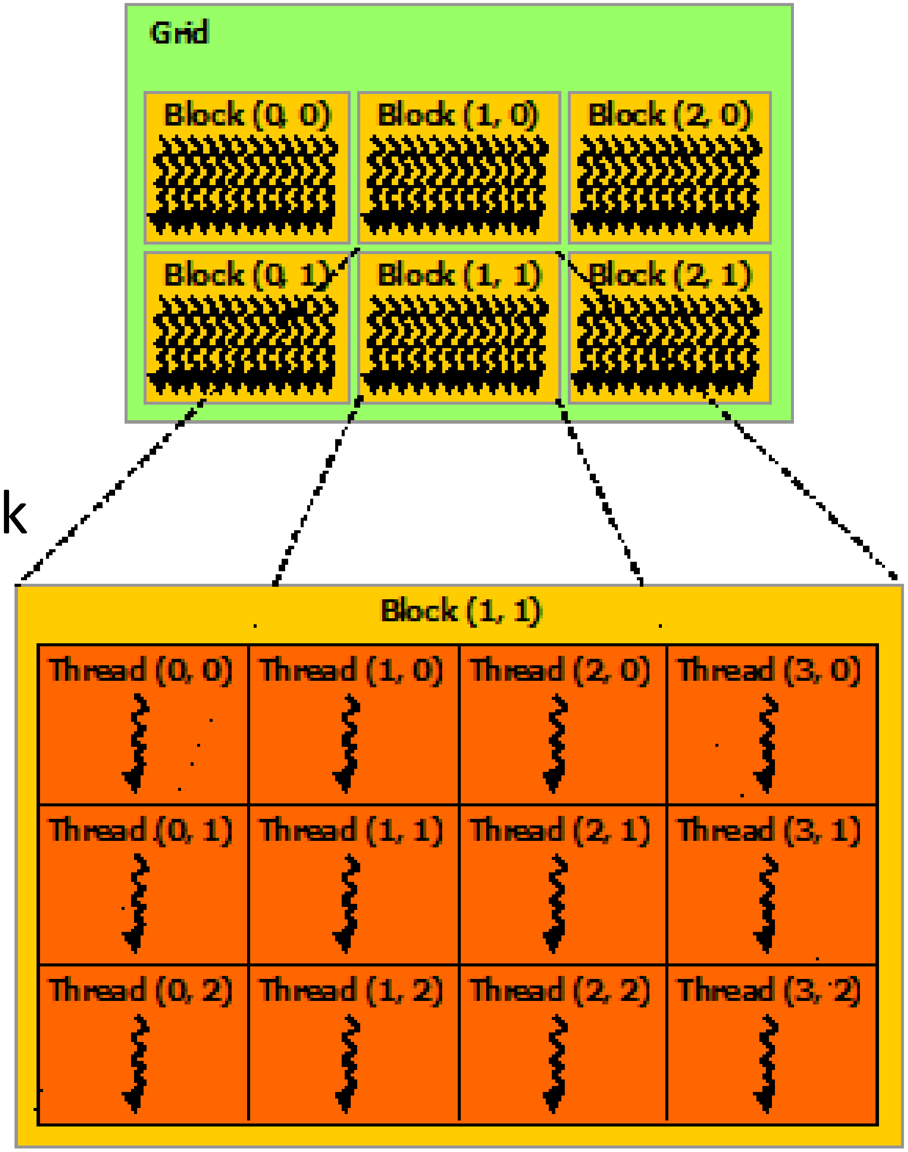
Matrix Addition

- CUDA 2D Kernel

- CUDA Thread Hierarchy
- Matrix Addition Example
- Host Implementation
  - Double for-loop
  - Kernel version
- CUDA Implementation
  - 2D Kernel

CUDA Thread Hierarchy

- Grid:
  - Top level
- Blocks:
  - 2D IDs, unique within a grid
- Threads:
  - 3D IDs, unique within a block



Example: Matrix Addition

- We want to add matrix A and matrix B

$$C = A + B = \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} + \begin{pmatrix} B_{00} & B_{01} & B_{02} & B_{03} & B_{04} \\ B_{10} & B_{11} & B_{12} & B_{13} & B_{14} \\ B_{20} & B_{21} & B_{22} & B_{23} & B_{24} \\ B_{30} & B_{31} & B_{32} & B_{33} & B_{34} \\ B_{40} & B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix}$$

- We need a two-dimensional kernel!
  - $C_{ij} = A_{ij} + B_{ij}$

Memory Access to the Matrix

- Row-major matrix storage
- Logical layout

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$$

- Physical layout: 1D array

$$[\ A_{00}\ A_{01}\ A_{02}\ A_{03}\ A_{04}\ A_{10}\ A_{11}\ A_{12} \ldots\ ]$$

- Re-interpreted layout: 1D array with single index

$$[\ A_0\ A_1\ A_2\ A_3\ A_4\ A_5\ A_6\ A_7 \ldots\ ]$$

- Index change:
  - *Idx = y * WIDTH + x*
  - *In this case, WIDTH = 5*

Matrix Addition in Host (CPU)

- Matrix addition using double for-loop in C++

```cpp
#include <stdio.h>

int main(){
    const int WIDTH=5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            a[y][x] = y*10+x;
            b[y][x] = (y*10+x)*100;
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            c[y][x] = a[y][x] + b[y][x];
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            printf("%5d ", c[y][x]);
        }
        printf("\n");
    }

    return 0;
}
```

Matrix Addition in Host (CPU)

- Matrix addition using Kernel function in C++

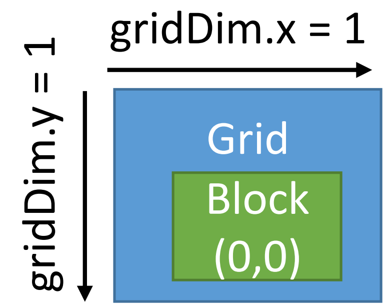
```
#include <stdio.h>

int main(){
    const int WIDTH=5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            a[y][x] = y*10+x;
            b[y][x] = (y*10+x)*100;
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            c[y][x] = a[y][x] + b[y][x];
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            printf("%5d ", c[y][x]);
        }
        printf("\n");
    }

    return 0;
}
```

```
void add(const int x, const int y, const int WIDTH, int* c, const int* a, const int* b){
    int i = y * (WIDTH) + x; // index calculation
    c[i] = a[i] + b[i];
}
```

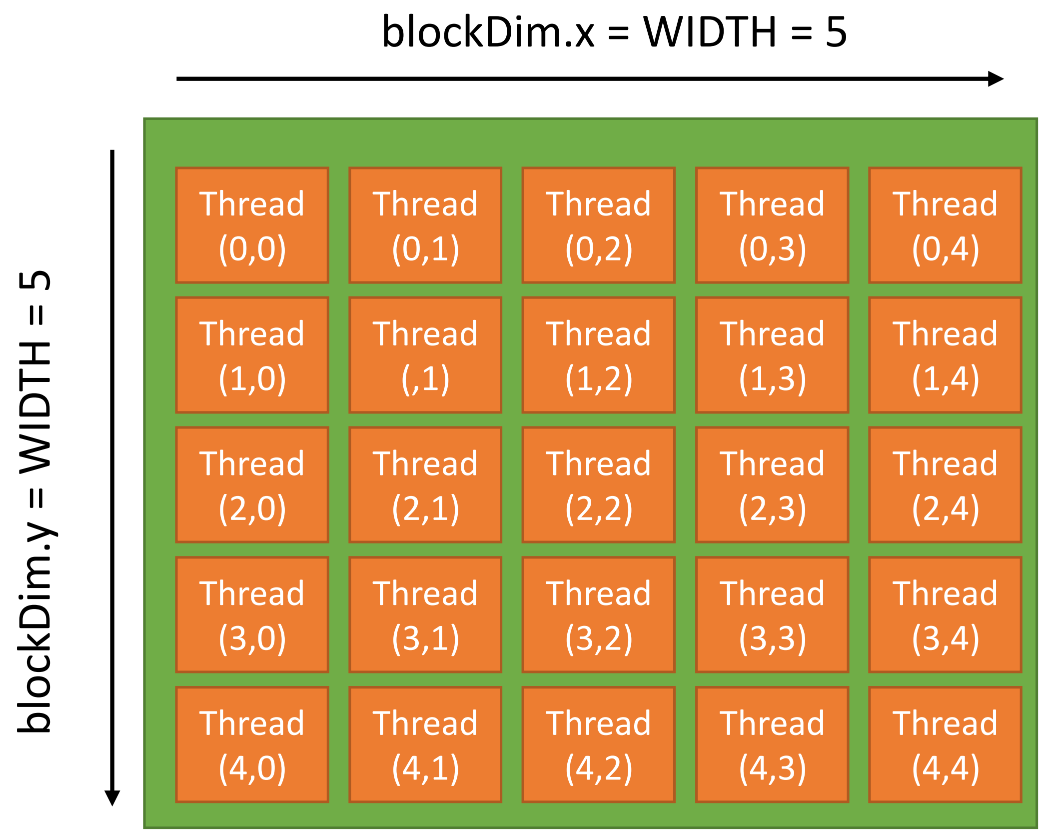
```
int main(){
    const int WIDTH=5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            a[y][x] = y*10+x;
            b[y][x] = (y*10+x)*100;
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            //c[y][x] = a[y][x] + b[y][x];
            add(x,y, WIDTH, (int*)c, (int*)a, (int*)b);
        }
    }

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            printf("%5d ", c[y][x]);
        }
        printf("\n");
    }

    return 0;
}
```

Thread Layout

- Matrix → 2D Layout
- Small size matrix → a single block!



$$\begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix}$$



Matrix Addition in Device (GPU)

- Kernel function

```
#include <stdio.h>
__global__ void addKernel(int* c, const int* a, const int* b){
    int x = threadIdx.x;
    int y = threadIdx.y;
    int i = y * (blockDim.x) + x; // index = y * WIDTH + x
    c[i] = a[i] + b[i];
}
```

```
int main(){
    const int WIDTH=5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };

    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            a[y][x] = y*10+x;
            b[y][x] = (y*10+x)*100;
        }
    }
```

Matrix Addition in Device (GPU)

- Declaration of arrays for device (GPU)

```
int *dev_a, *dev_b, *dev_c = 0; // GPU does not know the array structure of dev_a, dev_b, dev_c
cudaMalloc((void**)&dev_a, WIDTH*WIDTH*sizeof(int)); // Memory allocation (WIDTH*WIDTH)
cudaMalloc((void**)&dev_b, WIDTH*WIDTH*sizeof(int)); // Memory allocation (WIDTH*WIDTH)
cudaMalloc((void**)&dev_c, WIDTH*WIDTH*sizeof(int)); // Memory allocation (WIDTH*WIDTH)

cudaMemcpy(dev_a, a, WIDTH*WIDTH*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, WIDTH*WIDTH*sizeof(int), cudaMemcpyHostToDevice);
```

- Kernel launch using DimBlock and copy the results

```
dim3 DimBlock(WIDTH,WIDTH);
addKernel <<<1,DimBlock>>>(dev_c,dev_a,dev_b);

cudaMemcpy(c, dev_c, WIDTH*WIDTH*sizeof(int), cudaMemcpyDeviceToHost);
```

Matrix Multiplication

- CUDA 2D Kernel

- Matrix Multiplication Example
- Host Implementation
  - Triple for-loop
  - Kernel version
- CUDA Implementation
  - 2D Kernel
  - Kernel has a for-loop!

Example: Matrix Multiplication

- We want to multiply matrix A and matrix B
  - For simplicity, we assume square matrices

$$C = A \cdot B = \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} & B_{02} & B_{03} & B_{04} \\ B_{10} & B_{11} & B_{12} & B_{13} & B_{14} \\ B_{20} & B_{21} & B_{22} & B_{23} & B_{24} \\ B_{30} & B_{31} & B_{32} & B_{33} & B_{34} \\ B_{40} & B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix}$$

Example: Matrix Multiplication

- C_{ij} = dot product of $A_{i*} + B_{*j}$

$$C_{31} = \begin{pmatrix} A_{00} & A_{01} & A_{02} & A_{03} & A_{04} \\ A_{10} & A_{11} & A_{12} & A_{13} & A_{14} \\ A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ A_{30} & A_{31} & A_{32} & A_{33} & A_{34} \\ A_{40} & A_{41} & A_{42} & A_{43} & A_{44} \end{pmatrix} \cdot \begin{pmatrix} B_{00} & B_{01} & B_{02} & B_{03} & B_{04} \\ B_{10} & B_{11} & B_{12} & B_{13} & B_{14} \\ B_{20} & B_{21} & B_{22} & B_{23} & B_{24} \\ B_{30} & B_{31} & B_{32} & B_{33} & B_{34} \\ B_{40} & B_{41} & B_{42} & B_{43} & B_{44} \end{pmatrix}$$

- $C_{ij} = \sum_{k=1}^{n} A_{ik}\, Bkj$

```
int sum = 0;
for (int k = 0; k<WIDTH; k++){
        sum += a[i][k] * b[k][j];
}
c[i][j] = sum;
```

Matrix Multiplication in Host (CPU)

- Initialization

```c
#include <stdio.h>

int main(){
    const int WIDTH=5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    for (int y=0; y<WIDTH; y++){
        for (int x=0; x<WIDTH; x++){
            a[y][x] = y+x;
            b[y][x] = y+x;
        }
    }
```

- Matrix multiplication

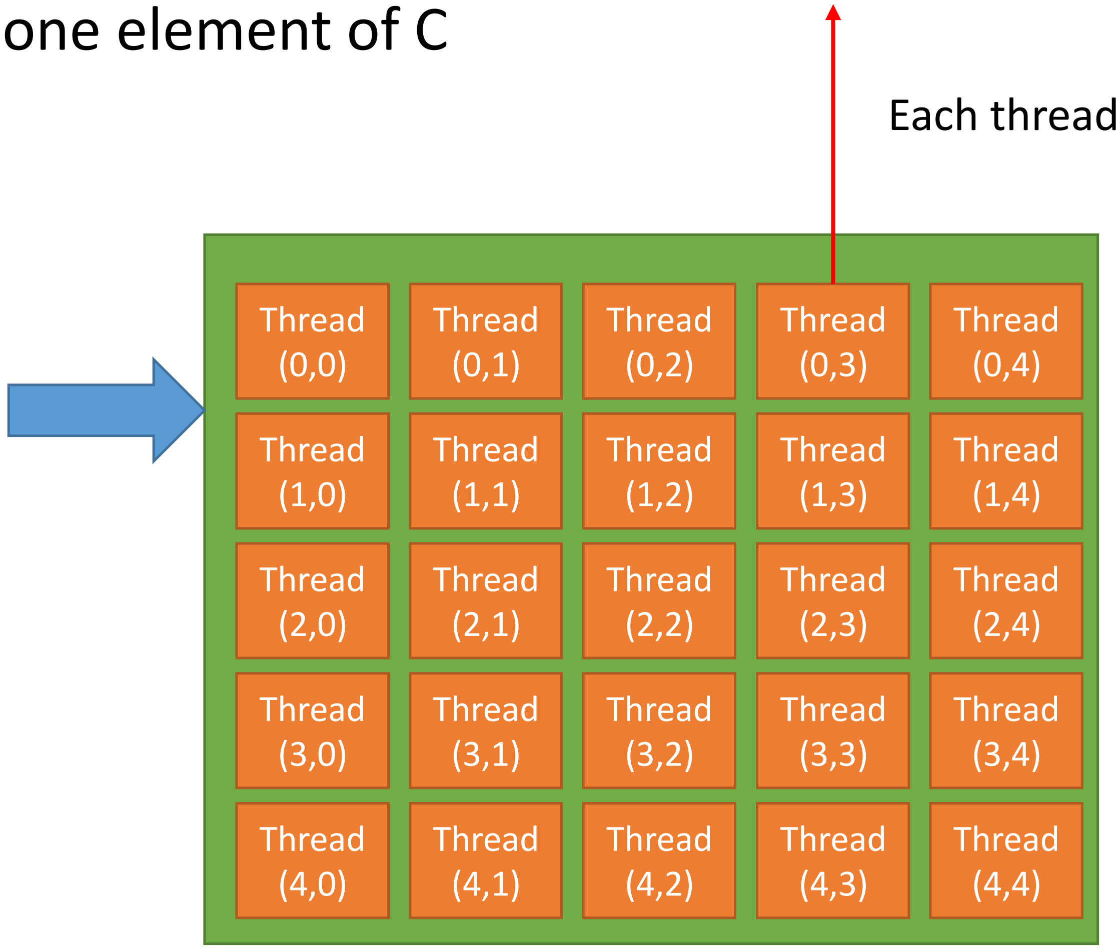
```c
for (int y=0; y<WIDTH; y++){
    for (int x=0; x<WIDTH; x++){
        int sum = 0;
        for (int k=0; k<WIDTH;k++){
            sum += a[y][k] * b[k][x];
        }
        c[y][x] = sum;
    }
}
```

CUDA Matrix Multiplication

- C = A*B of size WIDTH x WIDTH
- Thread hierarchy design
  - One thread handles one element of C

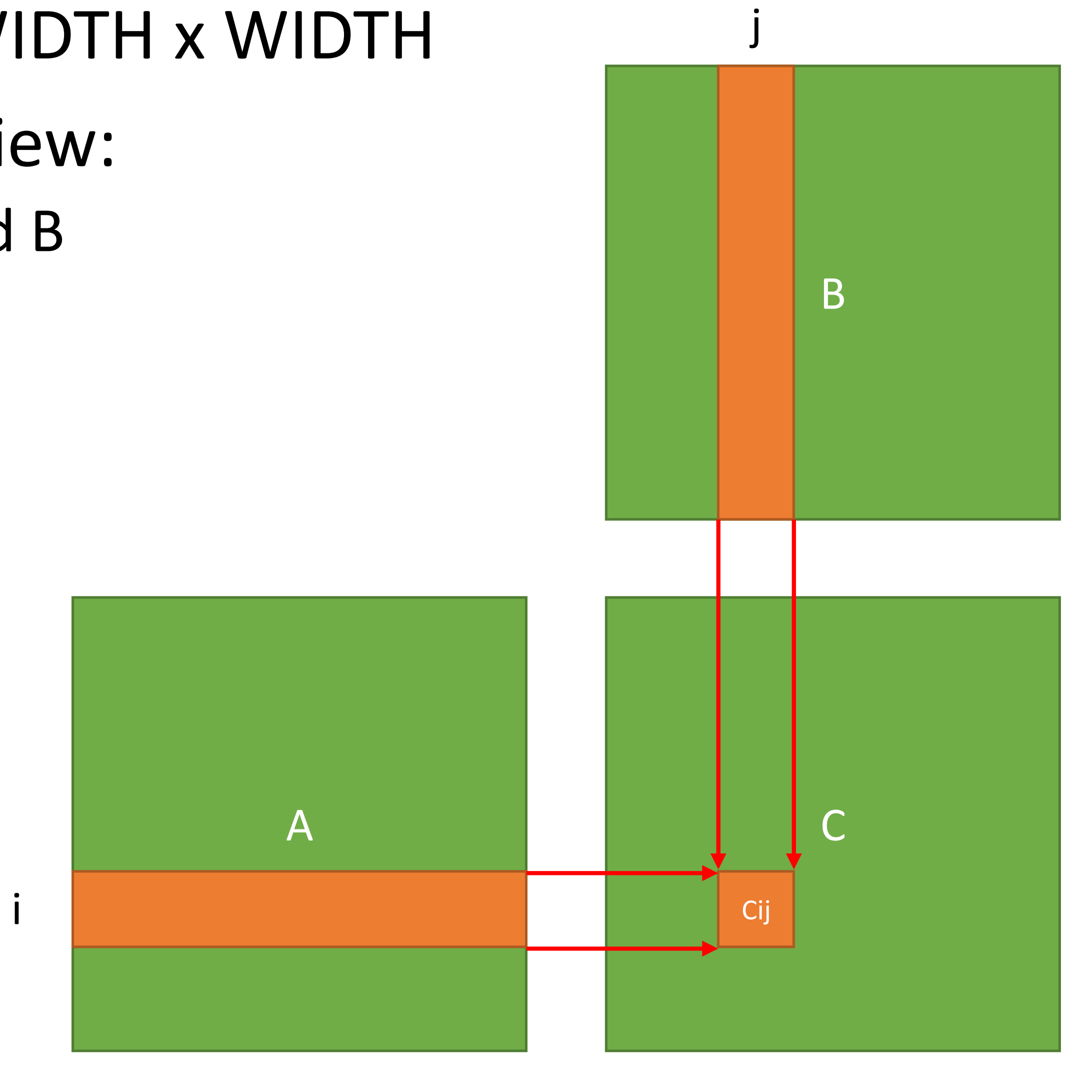
$$C_{ij} = \sum_{k=1}^{n} A_{ik} Bkj$$

$$\begin{pmatrix} C_{00} & C_{01} & C_{02} & C_{03} & C_{04} \\ C_{10} & C_{11} & C_{12} & C_{13} & C_{14} \\ C_{20} & C_{21} & C_{22} & C_{23} & C_{24} \\ C_{30} & C_{31} & C_{32} & C_{33} & C_{34} \\ C_{40} & C_{41} & C_{42} & C_{43} & C_{44} \end{pmatrix}$$



CUDA Matrix Multiplication: Memory usage

- C = A*B of size WIDTH x WIDTH
- Memory usage view:
  - Read from A and B
  - To calculate
- Using:
  - A[i*WIDTH + k]
  - B[k*WIDTH + j]



Matrix Multiplication in Device (GPU)

- Assignment
  - Write CUDA code with Kernel function for matrix multiplication
  - Print the result

Summary

- CUDA 2D Kernel

- Matrix multiplication example
- Host implementation
  - Triple for-loop
  - Kernel version
- CUDA implementation
  - 2D Kernel
  - Kernel has a for-loop

Next step?

- CUDA thread again
  - Hardware considerations?

Thank you

Any questions?

E-mail: yhgong@kw.ac.kr
Lab: https://sites.google.com/view/yhgong/

광운대학교
KwangWoon University