SSE3054 Multicore Systems

CUDA Thread 1

Prof. Seokin Hong

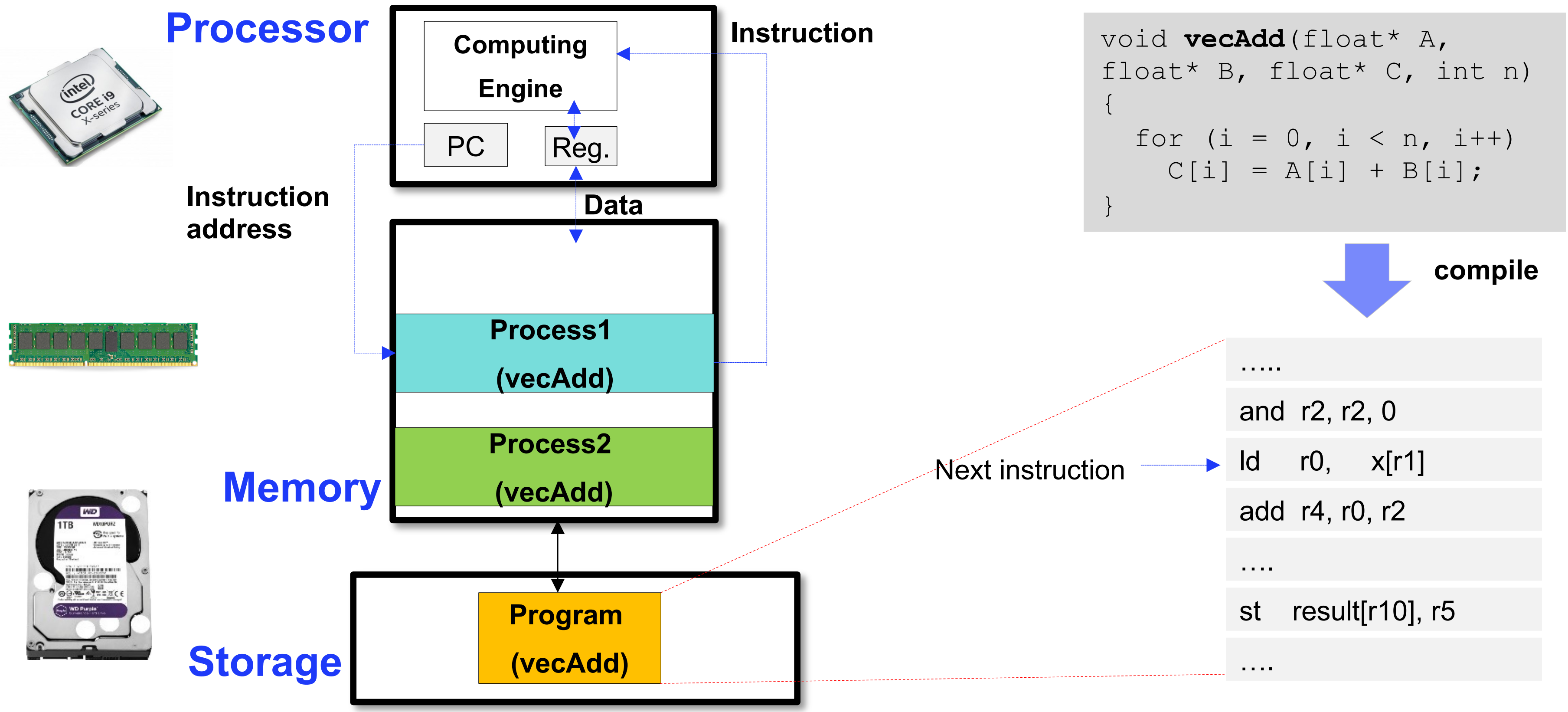
Agenda

- What is Thread?
- CUDA Thread Organization
- Extended VecAdd.cu
- Matrix Addition Kernel
- Mapping Threads to Multidimensional Data
- Synchronization and Transparent Scalability
- Resource Assignment
- Querying Device Properties
- Thread Scheduling and Latency Tolerance

What is Thread?

Process and Thread

- **Process**
  - **An instance of a computer program that is being executed**
  - Program code + Execution state (address of the next instruction, register state, memory contents)

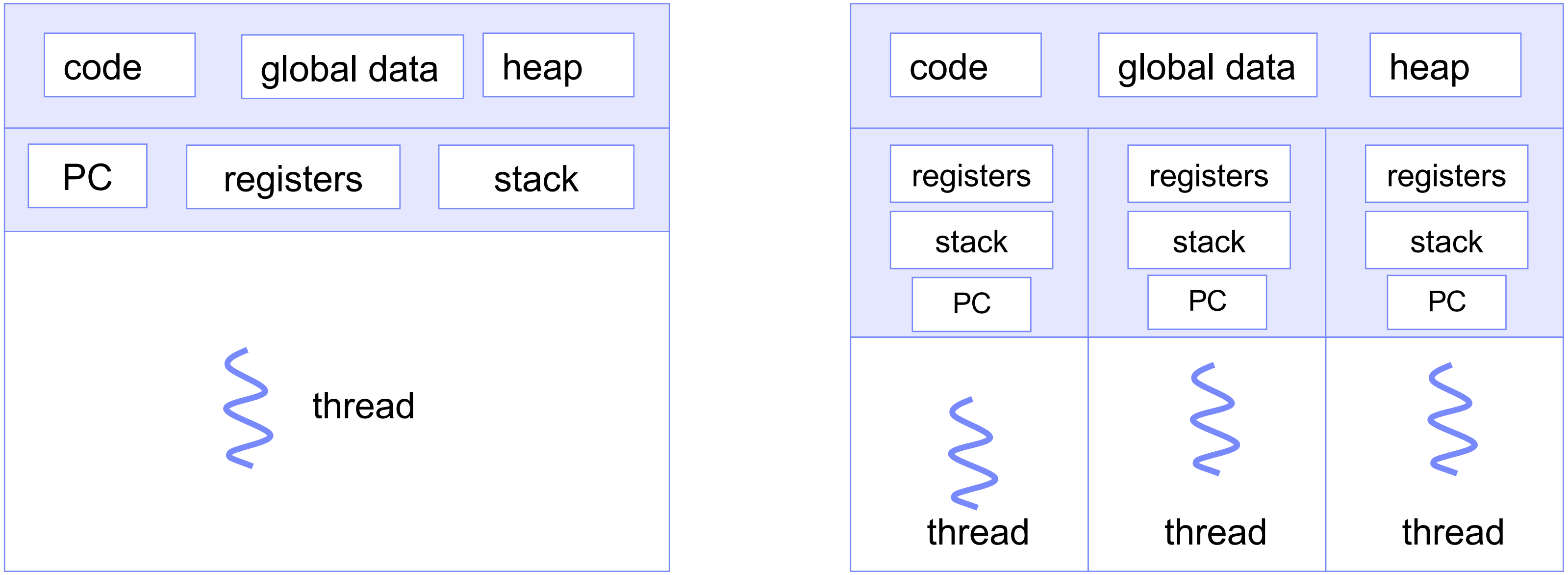


```
void vecAdd(float* A,
float* B, float* C, int n)
{
  for (i = 0, i < n, i++)
    C[i] = A[i] + B[i];
}
```

```
.....
and  r2, r2, 0
ld    r0,    x[r1]
add  r4, r0, r2
....
st    result[r10], r5
....
```

Process and Thread (Cont'd)

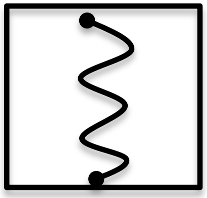
- Process
  - **An instance of a computer program that is being executed**
  - Program code + Execution state (address of the next instruction, register state, memory contents)
- **Thread**
  - **A flow of execution within a process**
  - Execution state (address of the next instruction, register state, stack)



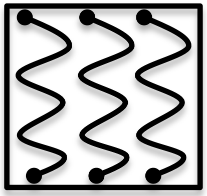
single-threaded process

multithreaded process

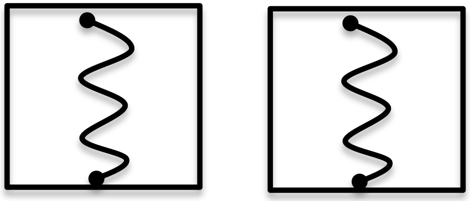
Process and Thread (Cont'd)



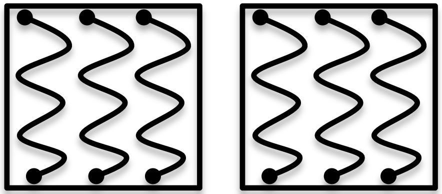
one process,
one thread



one process,
multiple threads



multiple processes,
one thread per process

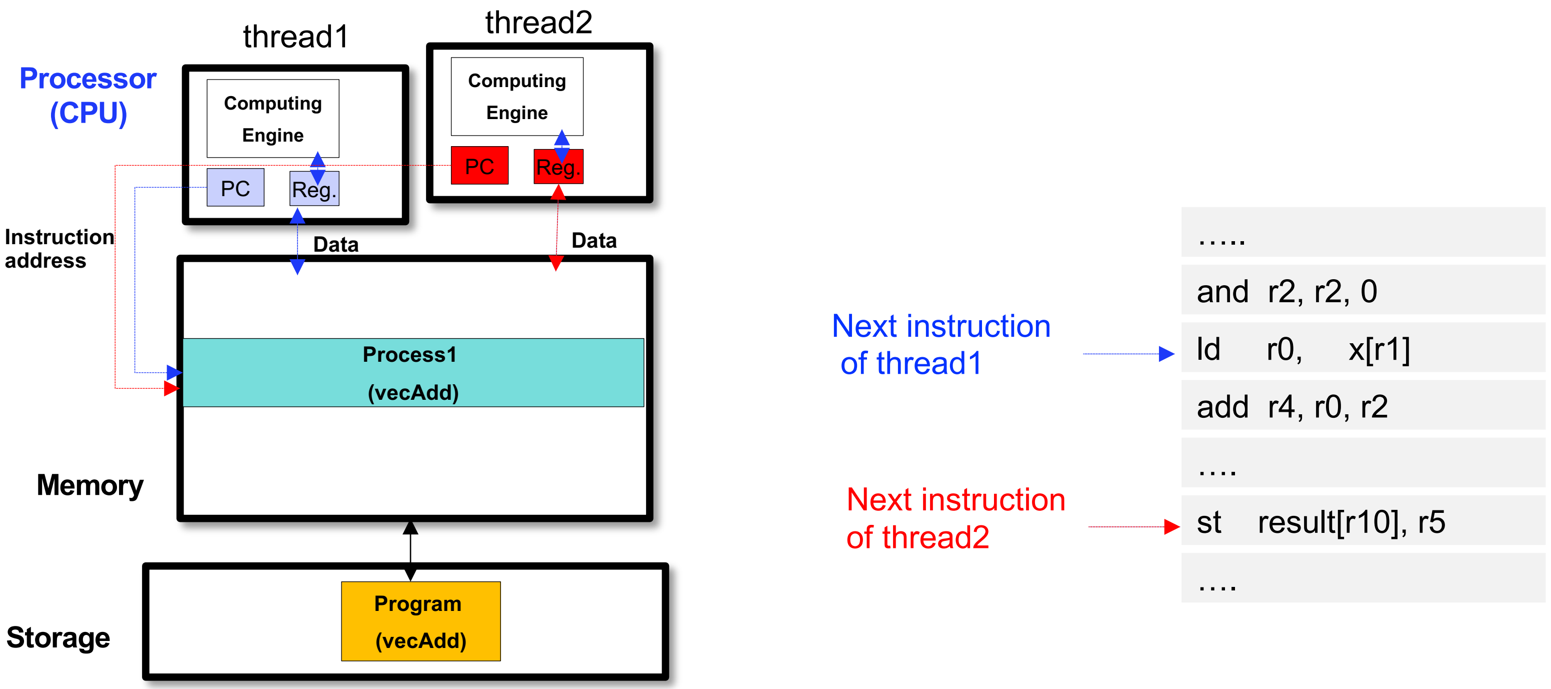


multiple processes,
multiple threads per process

Slide Credit: Slides are modified from Prof Baek's slides

Process and Thread (Cont'd)

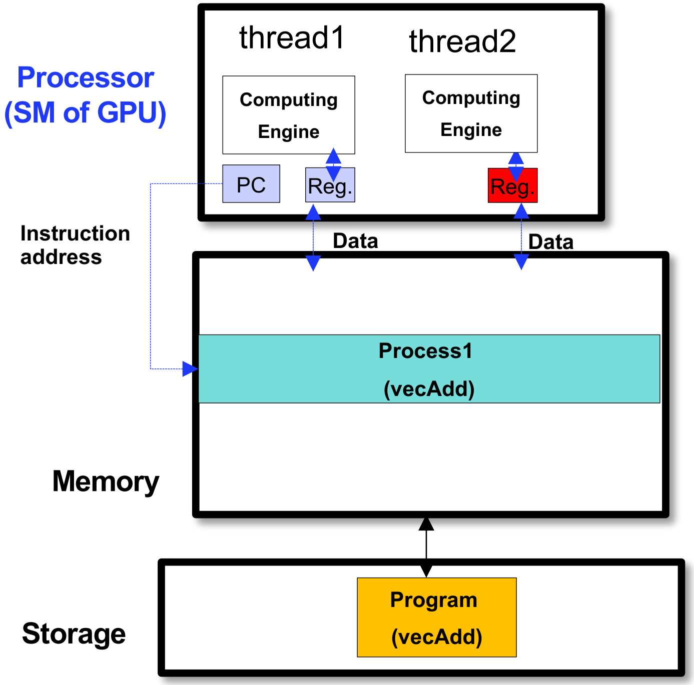
- Process
  - An instance of a computer program that is being executed
  - Program code + Execution state (address of the next instruction, register state, memory contents)
- Thread
  - **A flow of execution within a process**
  - Execution state (address of the next instruction, register state)



```
…..
and  r2, r2, 0
ld     r0,     x[r1]
add  r4, r0, r2
….
st    result[r10], r5
….
```

Process and Thread (Cont’d)

- Process
  - An instance of a computer program that is being executed
  - Program code + Execution state (address of the next instruction, register state, memory contents)
- **Thread**
  - **A flow of execution within a process**
  - Execution state (address of the next instruction, register state)

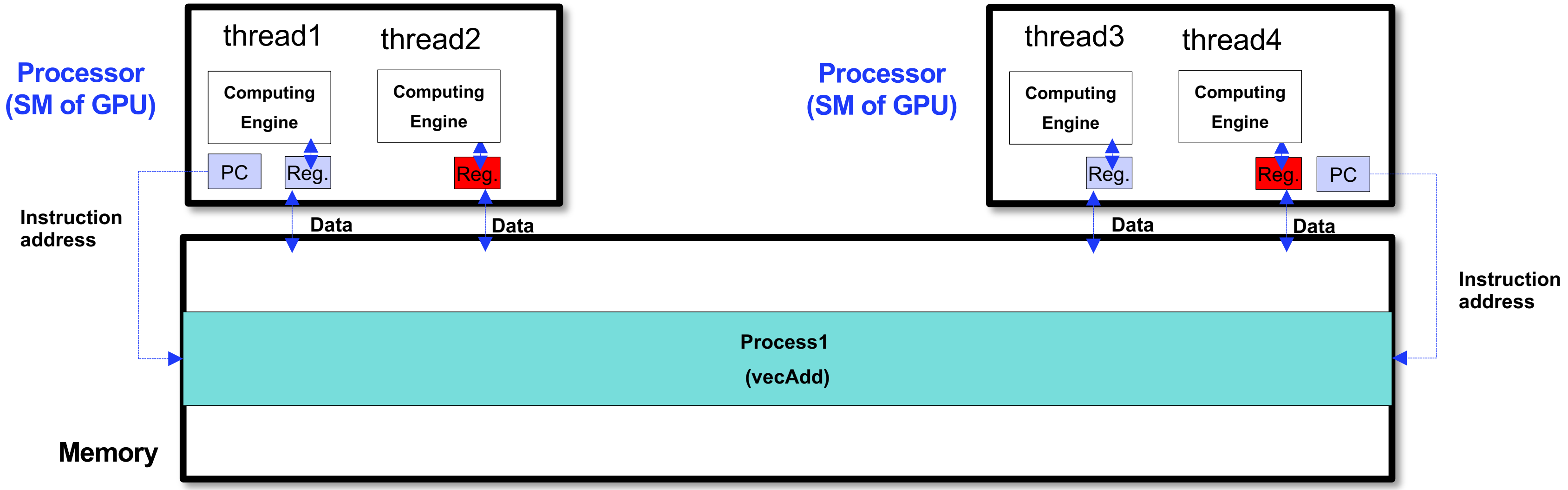


Next instruction of thread1 and thread2 →

```
…..
and  r2, r2, 0
ld     r0,      x[r1]
add  r4, r0, r2
….
st    result[r10], r5
….
```

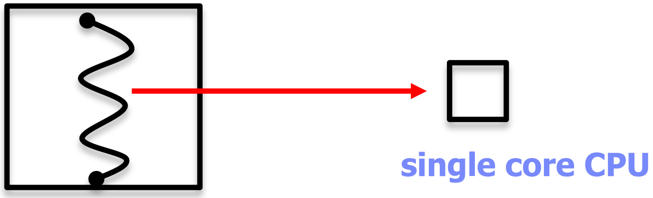
Process and Thread (Cont'd)

- Process
  - An instance of a computer program that is being executed
  - Program code + Execution state (address of the next instruction, register state, memory contents)
- Thread
  - **A flow of execution within a process**
  - Execution state (address of the next instruction, register state)

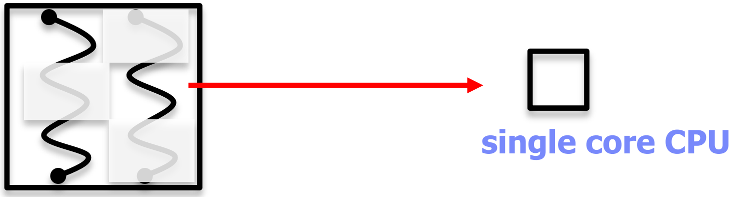


Thread Execution

- On Single Core Processors
  - single thread → Serial processing

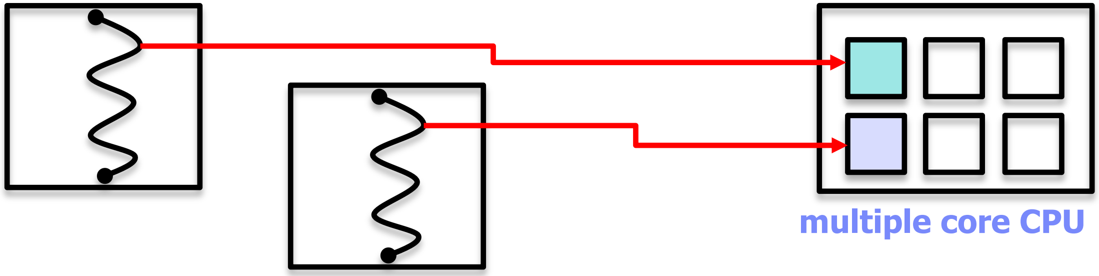


  - multiple thread → time sharing (concurrent processing)

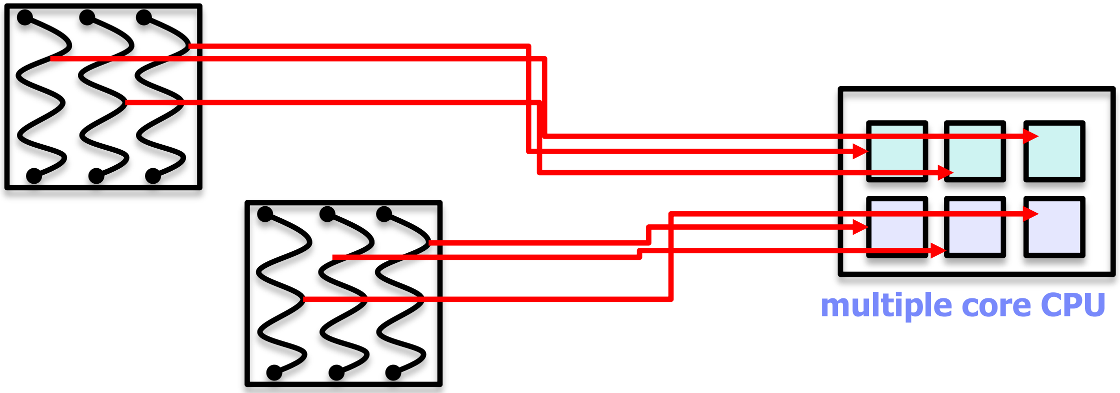


Thread Execution on Different Types of Processor (Cont'd)

- On Multicore Processors
  - single thread, multiple process → parallel processing



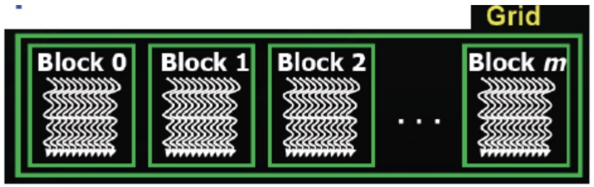
  - multiple thread → parallel processing



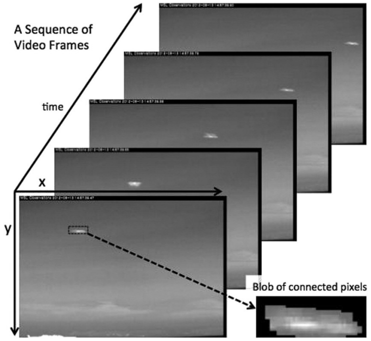
CUDA Thread Organization

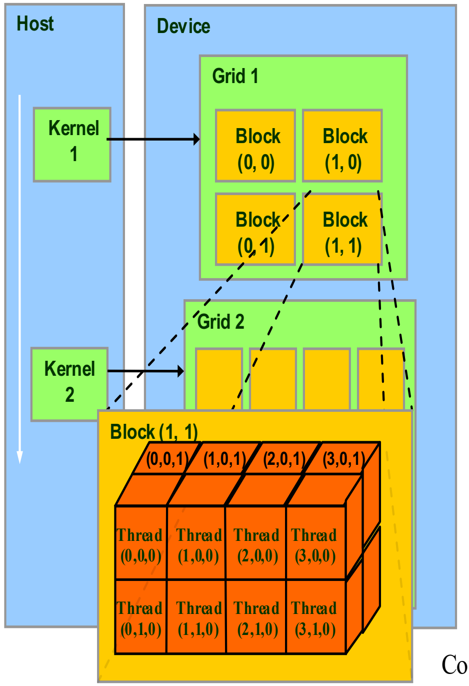
Hierarchy of Threads

- Kernel is composed of many threads
  - All threads execute same sequential program
  - Use parallel threads rather than sequential loop
- Threads are grouped into thread blocks
  - Threads in block can share data with a shared memory
- Blocks are grouped into grids



- Why hierarchy of threads?
  - Real-world systems(data) are organized hierarchically
    - Video file
      - Pixel
      - Image : multiple pixels
      - Video : A sequence of frames

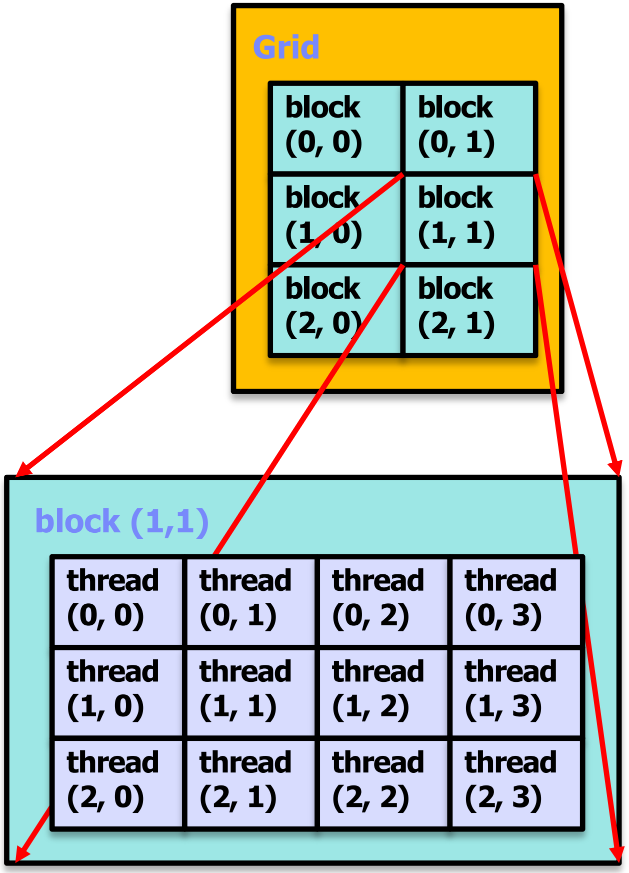




Courtesy: NDVIA

IDs and Dimensions

- Threads and blocks have unique IDs
- **Threads:**
  - **threadIdx**: 1D, 2D, or 3D
  - unique within a block
- **Blocks:**
  - **blockIdx**: 1D, 2D, or 3D
  - unique within a grid
- **Dimensions set at launch**
  - Can be unique for each grid
  - **blockDim**: dimension of block
  - **gridDim**: dimension of grid



Grid

block (0, 0)	block (0, 1)
block (1, 0)	block (1, 1)
block (2, 0)	block (2, 1)

block (1,1)

thread (0, 0)	thread (0, 1)	thread (0, 2)	thread (0, 3)
thread (1, 0)	thread (1, 1)	thread (1, 2)	thread (1, 3)
thread (2, 0)	thread (2, 1)	thread (2, 2)	thread (2, 3)

Each block and thread is labelled with (y, x)

Slide Credit: Slides are modified from Prof Baek's slides

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 (x, y, z):
  - dim3 blocks1D(5);
  - dim3 blocks2D(5, 5);
  - dim3 blocks3D(5, 5, 5);

Slide Credit: Slides are modified from Prof Baek’s slides

Configuring Thread Organization

- A kernel function must be called with an execution configuration:
  - **Example1**

```
KernelFunc<<< blocks, threads >>>(...);
```

Or

  - **Example2**
    - Launch the KernelFunc() kernel function, and generate a 1D grid that consists of 32 blocks, each of which consists of 128 threads

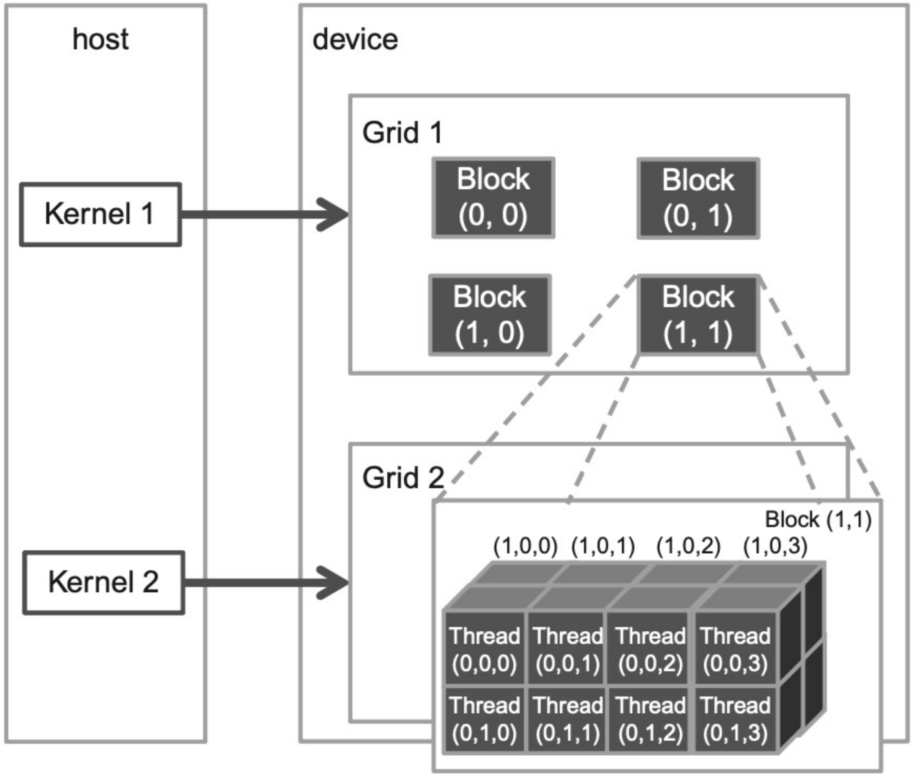
```
dim3   DimGrid(32, 1, 1);
dim3   DimBlock(128, 1, 1);
KernelFunc<<< DimGrid, DimBlock >>>(...);
```

    - **dim3 :** a C struct with three unsigned integer fields: x, y, z
    - Total number of threads in the grid is 128*32=4096

Configuring Thread Organization

- A kernel function must be called with an execution configuration:
  - **Example3**

```
dim3   DimGrid(2, 2, 1);
dim3   DimBlock(4, 2, 2);
KernelFunc<<< DimGrid, DimBlock >>>(...);
```



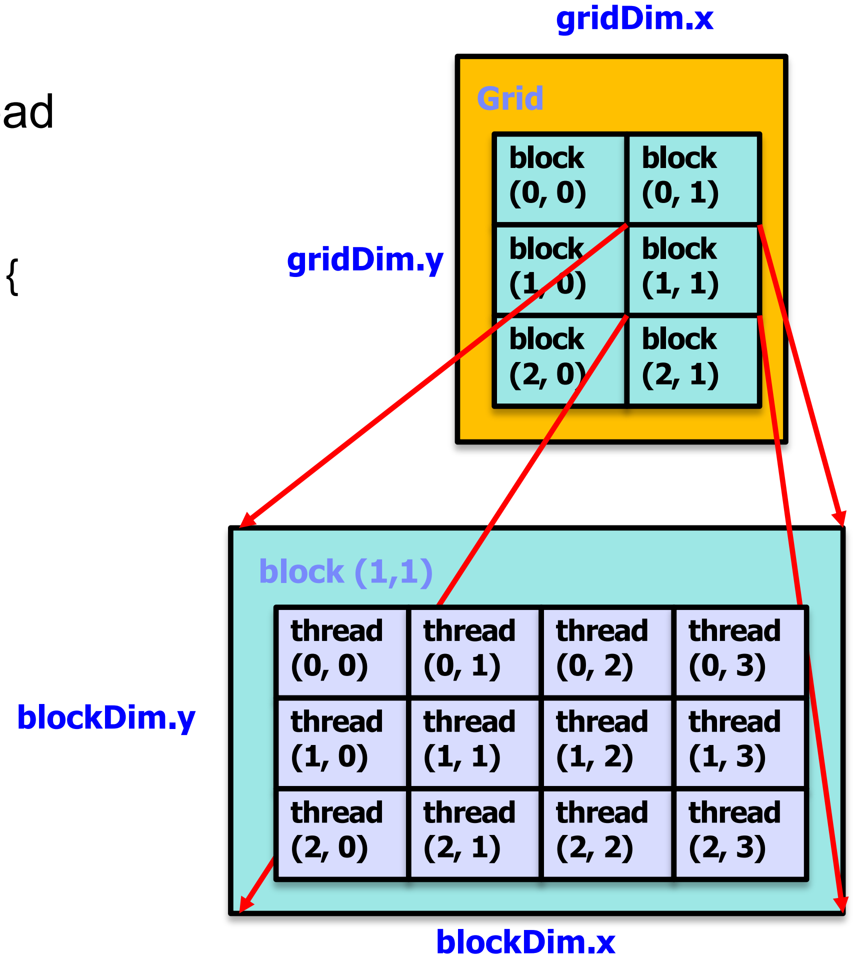
Kernel with 2D Indexing

dim3   DimGrid(100, 50); → gridDim in kernel

dim3   DimBlock(4, 8, 8); → blockDim in kernel

blockIdx, threadIdx : unique for each thread

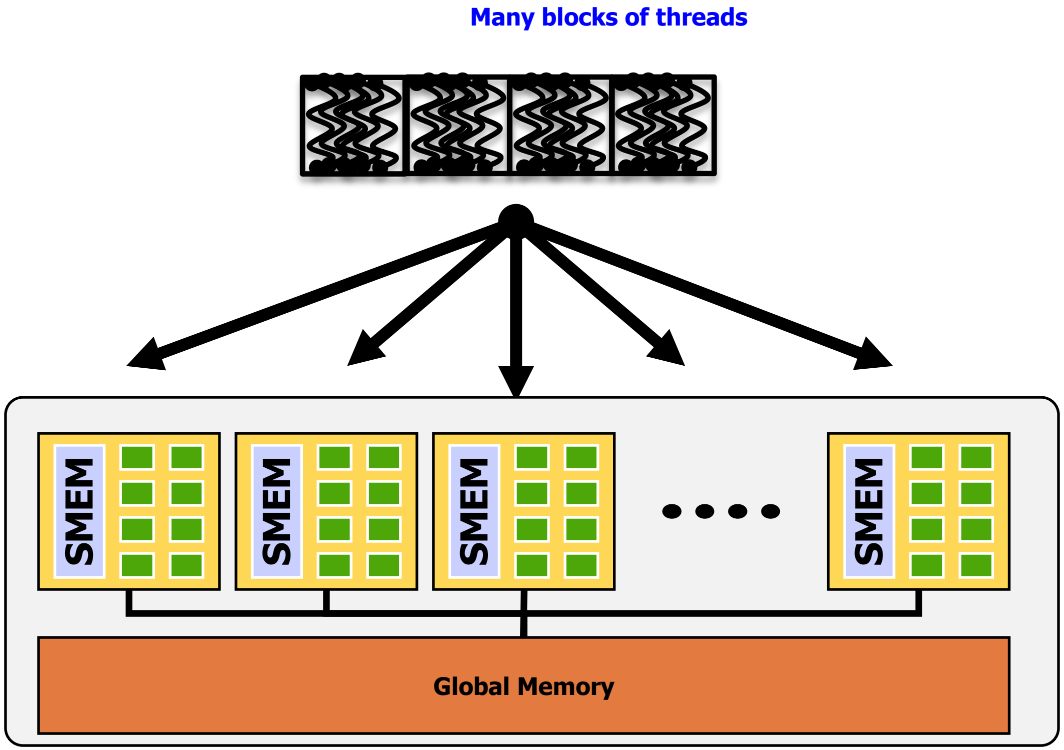
```
__global__ void kernel( int *a, int dimx, int dimy ) {
    int ix   = blockIdx.x*blockDim.x + threadIdx.x;
    int iy   = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = iy*dimx + ix;
    a[idx]  = a[idx]+1;
}
```



Slide Credit: Slides are modified from Prof Baek's slides

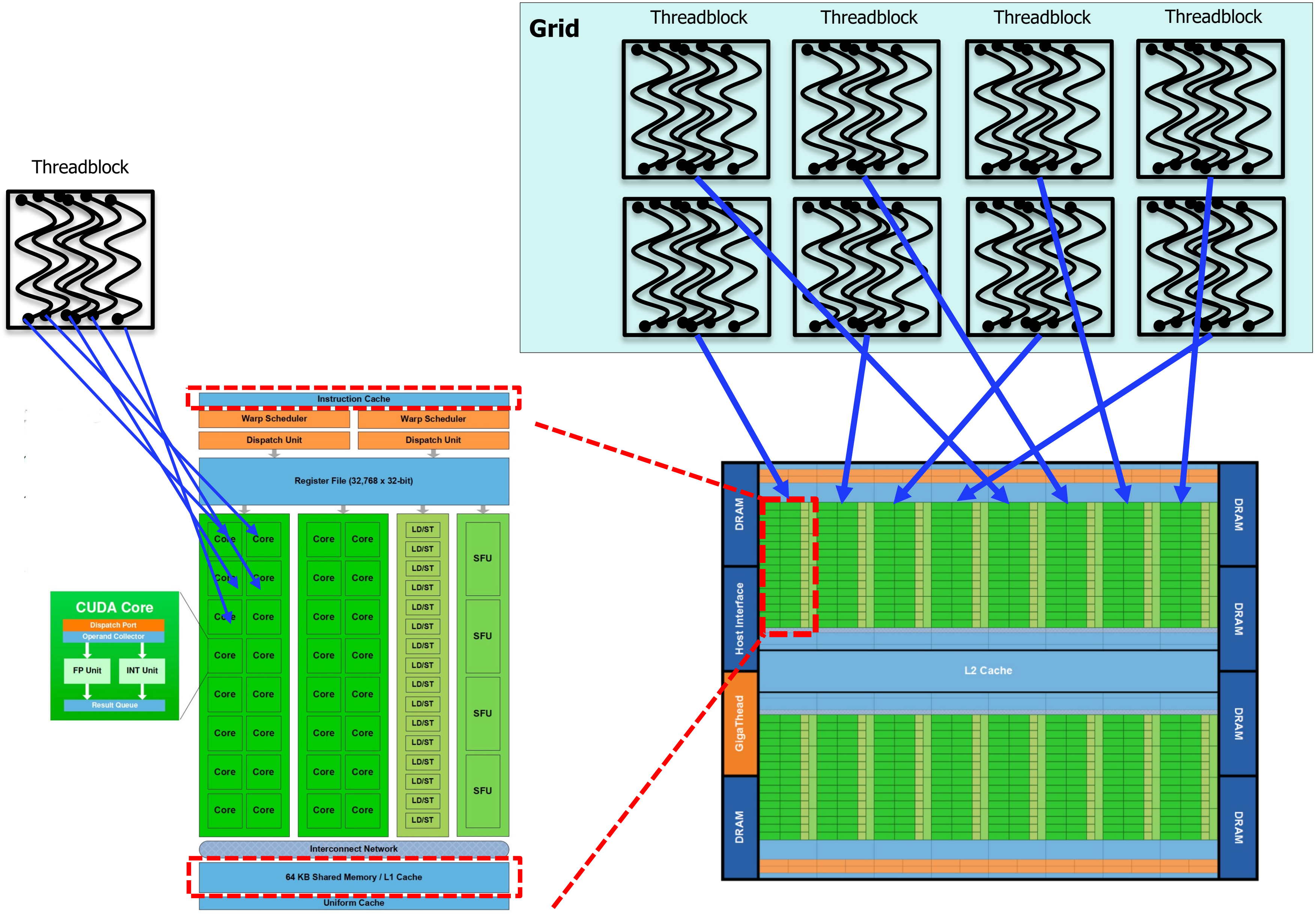
Executing threads on GPU

- Each thread block goes to a SM
  - Threads running on the same SM can share data through shared memory (SMEM)
    - Shared memory is much faster than the global memory



Slide Credit: Slides are modified from Prof Baek's slides

Executing threads on GPU



SM (Streaming Multiprocessor)

GPU

Extending Vector Addition Program
for scalable parallel exectution

VecAdd_ext.cu

Host code

```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<1, n>>>(A_d, B_d, C_d);

   // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
   // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

```
void vecAdd(int* A, int* B, int* C, int n)
{
    int size = n * sizeof(int);
    int* A_d=0;
    int* B_d=0;
    int* C_d=0;
    // Allocate device memory
    cudaMalloc((void **) &A_d, size);
    cudaMalloc((void **) &B_d, size);
    cudaMalloc((void **) &C_d, size);

   // Transfer A and B to device memory
    cudaMemcpy(A_d, A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(B_d, B, size, cudaMemcpyHostToDevice);

    addKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d,n);

   // Transfer C from device to host
    cudaMemcpy(C, C_d, size, cudaMemcpyDeviceToHost);
   // Free device memory for A, B, C
    cudaFree(A_d); cudaFree(B_d); cudaFree (C_d);
}
```

- ceil(x): maps x to the least integer greater than or equal to x
  ex) ceil(4.5) = 5

Extended Vector Addition Kernel

```
__global__
void addKernel(int* A_d, int* B_d, int* C_d)
{
     // each thread knows its own index
     int i = threadIdx.x;
    C_d[i] = A_d[i] + B_d[i];
}
```

```
__global__
void addKernel(int* A_d, int* B_d, int* C_d, int n)
{
     int i= blockDim.x*blockIdx.x + threadIdx.x;
    if(i<n) C_d[i] = A_d[i] + B_d[i];
}
```

Host code

```
int main(void) {
  const int SIZE = 2048;
   int a[SIZE];
   int b[SIZE];
   int c[SIZE];

   for(int i=0;i<SIZE;i++)
   {
     a[i]=i;
     b[i]=i;
   }
   vecAdd(a,b,c, SIZE );

    // print the result
  for(int i=0;i<SIZE;i++)
     printf("%d\n", c[i]);

    // done
   return 0;
}
```

Scalable Parallel Execution

- **addKernel**<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d,n);
  - The number of threads per block: 256
  - The number of blocks can be different depending on the total number of threads
    - If n is 750, 3 blocks will be used
    - If n is 4000, 16 thread blocks will be used
    - If n is 2000000, 7813 blocks will be used
  - All the thread blocks operate on different parts of the vectors
  - The thread block can be executed in any arbitrary order
    - Programmers must not make any assumptions regarding execution order
  - **CUDA supports Scalable Parallel Execution**
    - A small GPU with a small amount of SM may execute only one or two of the threadd blocks in parallel
    - A larger GPU may execute 64 or 128 blocks in parallel

    → So, same code unrs at lower speed on small GPU and higher speed on larger GPU

Matrix Addition Kernel

Slide Credit: Slides are modified from Prof Baek’s slides

Matrix Addition

- We want to add matrix A and matrix B

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

- **We need a two-dimensional kernel** !
  - ○ $c_{ij} = a_{ij} + b_{ij}$

Memory Layout of Matrix

- row-major matrix storage
  - logical layout:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix}$$

  - physical layout: 1D array $\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{10} & a_{11} & a_{12} & a_{20} & a_{21} & a_{22} \end{bmatrix}$
  - re-interpret: $\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 \end{bmatrix}$

- index change:
  - idx = y * WIDTH + x

matadd-host.cu

```
#include <iostream>


int main(void) {
    // host-side data
    const int WIDTH = 5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };

// make a, b matrices
    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;
            }
    }
}
```

$$\begin{array}{cc} & \xrightarrow{\quad\quad\quad x \quad\quad\quad} \\ y\Big\downarrow & A = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 10 & 11 & 12 & 13 & 14 \\ 20 & 21 & 22 & 23 & 24 \\ 30 & 31 & 32 & 33 & 34 \\ 40 & 41 & 42 & 43 & 44 \end{bmatrix} \end{array}$$

matadd-host.cu (cont’d)

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

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 \\ 10 & 11 & 12 & 13 & 14 \\ 20 & 21 & 22 & 23 & 24 \\ 30 & 31 & 32 & 33 & 34 \\ 40 & 41 & 42 & 43 & 44 \end{bmatrix} + \begin{bmatrix} 0 & 100 & 200 & 300 & 400 \\ 1000 & 1100 & 1200 & 1300 & 1400 \\ 2000 & 2100 & 2200 & 2300 & 2400 \\ 3000 & 3100 & 3200 & 3300 & 3400 \\ 4000 & 4100 & 4200 & 4300 & 4400 \end{bmatrix}$$

matadd-host.cu (cont’d)

```
    // print the result
    for (int y = 0; y < WIDTH; ++y) {
            for (int x = 0; x < WIDTH; ++x) {
                    printf("%5d", c[y][x]);
            }
            printf("\n");
    }
    // done
    return 0;
}
```

matadd-host.cu: a Complete Version

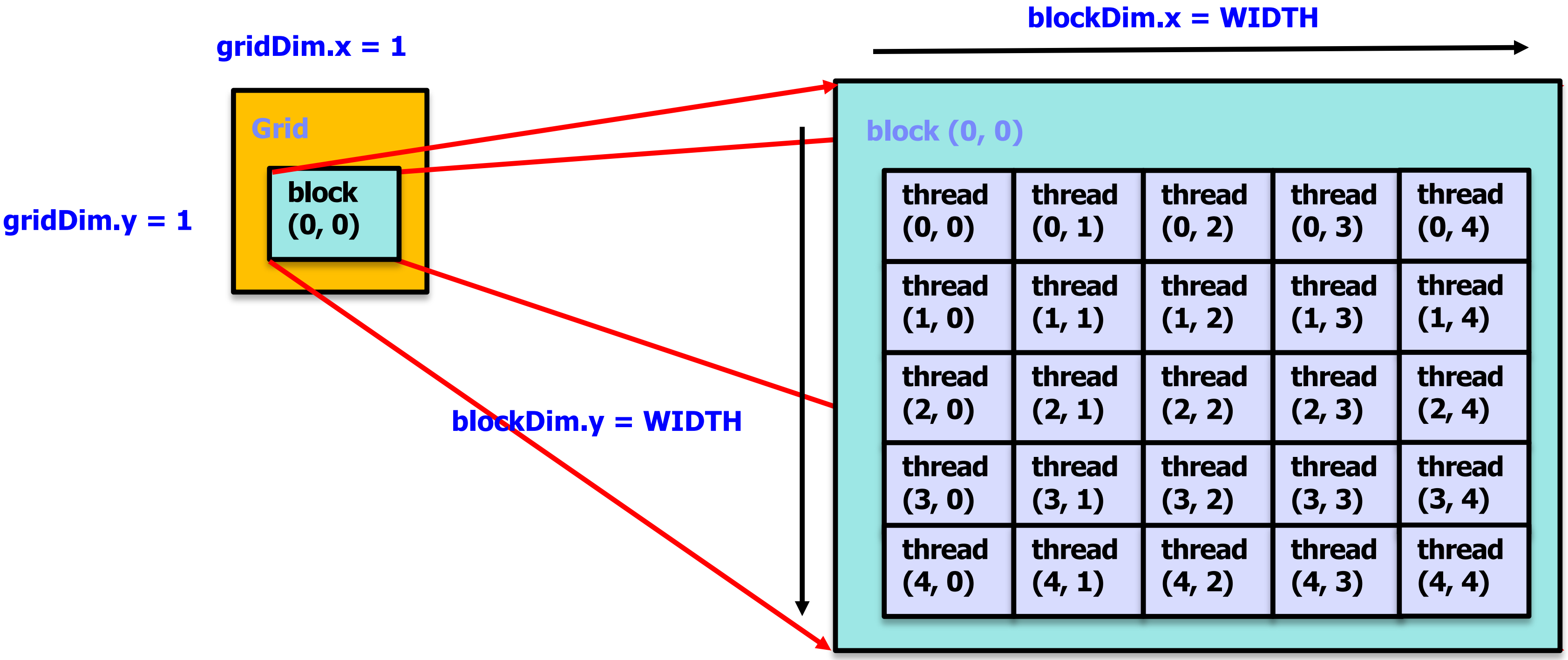
A Complete Version

```
#include <iostream>

int main(void) {
    // host-side data
    const int WIDTH = 5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    // make a, b matrices
    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;
            }
    }
    // calculate
    for (int y = 0; y < WIDTH; ++y) {
            for (int x = 0; x < WIDTH; ++x)
                    c[y][x] = a[y][x] + b[y][x];

    }
    // print the result
    for (int y = 0; y < WIDTH; ++y) {
            for (int x = 0; x < WIDTH; ++x)
                    printf("%5d", c[y][x]);
            printf("\n");
    }
    // done
    return 0;
}
```

```
$ nvcc matadd-host.cu –o matadd-host
$ ./matadd-host
```

matadd-dev.cu

- Let’s use GPU!
- Thread Organization
  - Matrix → 2D
  - Small size matrix → a single block



gridDim.x = 1

gridDim.y = 1

Grid

block (0, 0)

blockDim.x = WIDTH

blockDim.y = WIDTH

block (0, 0)

thread (0, 0)	thread (0, 1)	thread (0, 2)	thread (0, 3)	thread (0, 4)
thread (1, 0)	thread (1, 1)	thread (1, 2)	thread (1, 3)	thread (1, 4)
thread (2, 0)	thread (2, 1)	thread (2, 2)	thread (2, 3)	thread (2, 4)
thread (3, 0)	thread (3, 1)	thread (3, 2)	thread (3, 3)	thread (3, 4)
thread (4, 0)	thread (4, 1)	thread (4, 2)	thread (4, 3)	thread (4, 4)

Image from http://developer.amd.com/zones/OpenCLZone/courses/pages/Introductory-OpenCL-SAAHPC10.aspx

matadd-dev.cu (cont'd)

- **CUDA kernel**

```
#include <iostream>

// kernel program for the device (GPU): compiled by NVCC
__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;      // [y][x] = y * WIDTH + x;
                c[i] = a[i] + b[i];
}
```

matadd-dev.cu (cont’d)

- **Host code**

```
int main(void) {
    // host-side data
    const int WIDTH = 5;
    int a[WIDTH][WIDTH];
    int b[WIDTH][WIDTH];
    int c[WIDTH][WIDTH] = { 0 };
    // make a, b matrices
    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;
            }
    }
```

matadd-dev.cu (cont’d)

- **Host code (cont’d)**

```
// device-side data
int *dev_a = 0;
int *dev_b = 0;
int *dev_c = 0;
// allocate device memory
cudaMalloc((void**)&dev_a, WIDTH * WIDTH * sizeof(int));
cudaMalloc((void**)&dev_b, WIDTH * WIDTH * sizeof(int));
cudaMalloc((void**)&dev_c, WIDTH * WIDTH * sizeof(int));

// copy from host to device
cudaMemcpy(dev_a, a, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice);
```

matadd-dev.cu (cont'd)

- **Host code (cont'd)**

```
 dim3   dimGrid(1, 1, 1);
 dim3   dimBlock(WIDTH, WIDTH, 1); // x, y, z
 addKernel<<<dimGrid, dimBlock>>>(dev_c, dev_a, dev_b);

// copy from device to host
cudaMemcpy(c, dev_c, WIDTH * WIDTH * sizeof(int),
            cudaMemcpyDeviceToHost);

// free device memory
cudaFree(dev_c);
cudaFree(dev_a);
cudaFree(dev_b);
```

matadd-dev.cu (cont’d)

- **Host code (cont’d)**

```
    // print the result
    for (int y = 0; y < WIDTH; ++y) {
            for (int x = 0; x < WIDTH; ++x) {
                    printf("%5d", c[y][x]);
            }
            printf("\n");
    }
    // done
    return 0;
}
```

matadd-dev.cu: a Complete Version

```
#include <iostream>
#include "common.h"

// kernel program for the device (GPU): compiled by NVCC
__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;     // [y][x] = y * WIDTH + x;
  c[i] = a[i] + b[i];
}

// main program for the CPU: compiled by MS-VC++
int main(void) {
  // host-side data
  const int WIDTH = 5;
  int a[WIDTH][WIDTH];
  int b[WIDTH][WIDTH];
  int c[WIDTH][WIDTH] = { 0 };
  // make a, b matrices
  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;
    }
  }
  // device-side data
  int* dev_a = 0;
  int* dev_b = 0;
  int* dev_c = 0;
```

```
  // allocate device memory
  CUDA_CHECK( cudaMalloc((void**)&dev_a, WIDTH * WIDTH * sizeof(int)) );
  CUDA_CHECK( cudaMalloc((void**)&dev_b, WIDTH * WIDTH * sizeof(int)) );
  CUDA_CHECK( cudaMalloc((void**)&dev_c, WIDTH * WIDTH * sizeof(int)) );
  // copy from host to device
  CUDA_CHECK( cudaMemcpy(dev_a, a, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice) );
  CUDA_CHECK( cudaMemcpy(dev_b, b, WIDTH * WIDTH * sizeof(int), cudaMemcpyHostToDevice) );
  // launch a kernel on the GPU with one thread for each element.
  dim3 dimBlock(WIDTH, WIDTH, 1); // x, y, z
  addKernel <<< 1, dimBlock>>>(dev_c, dev_a, dev_b);    // dev_c = dev_a + dev_b;
  CUDA_CHECK( cudaPeekAtLastError() );
  // copy from device to host
  CUDA_CHECK( cudaMemcpy(c, dev_c, WIDTH * WIDTH * sizeof(int), cudaMemcpyDeviceToHost) );
  // free device memory
  CUDA_CHECK( cudaFree(dev_c) );
  CUDA_CHECK( cudaFree(dev_a) );
  CUDA_CHECK( cudaFree(dev_b) );
  // print the result
  for (int y = 0; y < WIDTH; ++y) {
    for (int x = 0; x < WIDTH; ++x) {
      printf("%5d", c[y][x]);
    }
    printf("\n");
  }
  // done
  return 0;
}
```

matadd-dev.cu: a Complete Version

common.h

```
#ifdef DEBUG // debug mode
#define CUDA_CHECK(x)       do {\
                    (x); \
                    cudaError_t e = cudaGetLastError(); \
                    if (cudaSuccess != e) { \
                                        printf("cuda failure %s at %s:%d\n", \
                                            cudaGetErrorString(e), \
                                            __FILE__, __LINE__); \
                                        exit(1); \
                    } \
} while (0)
#else
#define CUDA_CHECK(x)       (x)                 // release mode
#endif
```

Next?

- What is Thread?
- CUDA Thread Organization
- Extended VecAdd.cu
- Maxtrix Addition Kernel
- Mapping Threads to Multidimensional Data
- Synchronization and Transparent Scalability
- Resource Assignment
- Querying Device Properties
- Thread Scheduling and Latency Tolerance