Lecture 18

CUDA

이재진
서울대학교 데이터사이언스대학원
서울대학교 공과대학 컴퓨터공학부
http://aces.snu.ac.kr/~jlee

CUDA

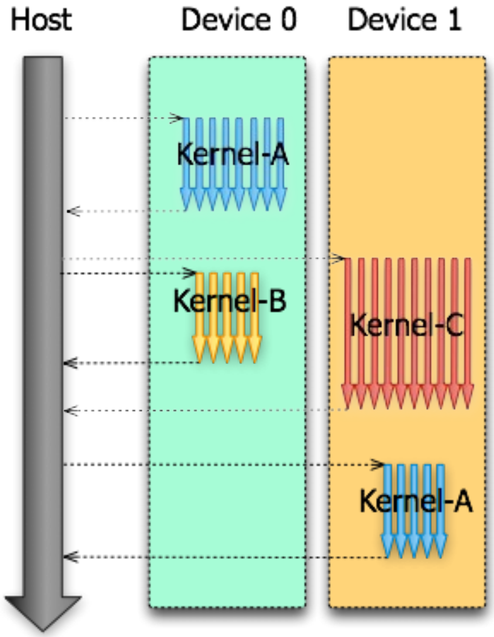
- November 2006, CUDA 1.0 release
- CUDA platform
  - Expose GPU computing for general purpose
  - Retain performance
- CUDA C/C++
  - Based on industry-standard C/C++
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

Heterogeneous Computing

- Host
    - The CPU and its memory (host memory)
- Device
    - The GPU and its memory (device memory)

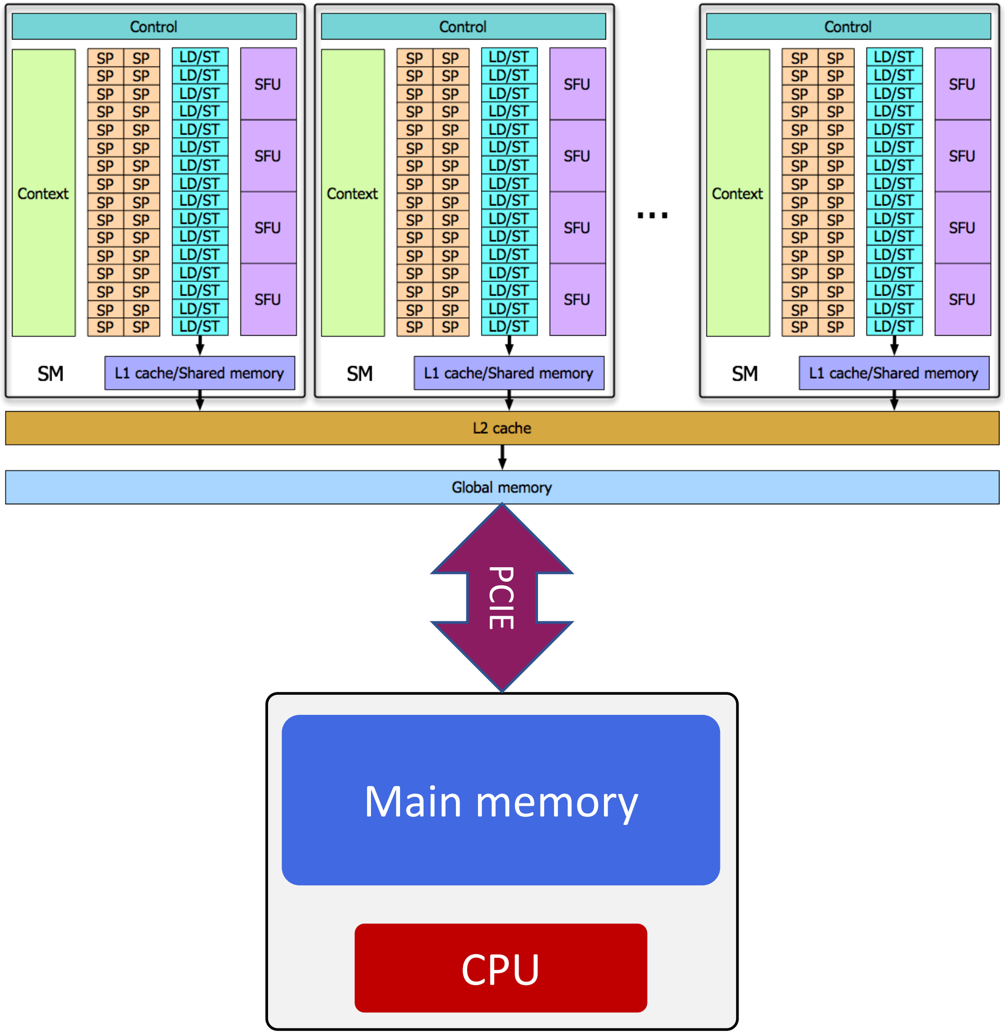
Heterogeneous Computing (cont’d)

- Host program
  - Manages kernel executions
- Kernels
  - Basic unit of executable code (a function) on compute devices
  - When executed, many instances are created
    - Exploit data parallelism
- The host program and kernels all run in parallel



Heterogeneous Computing (cont'd)

- Copy input data from CPU memory to GPU memory
- Load GPU code and execute it, caching data on chip for performance
- Copy results from GPU memory to CPU memory



Hello World! in CUDA

- A program with no device code
- NVIDIA compiler (nvcc) can be used to compile the program

```
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

```
> nvcc hello_world.cu
> a.out
Hello World!
>
```

Hello World! with Device Code

- CUDA C/C++ keyword **`__global__`** indicates a function that runs on the device and is called from the host code
  - Triple angle brackets mark a call from host code to device code
    - Also called a kernel launch
- nvcc separates source code into host and device components
  - Device functions (e.g., **`mykernel()`**) processed by the NVIDIA compiler
  - Host functions (e.g., **`main()`**) processed by the standard host compiler (e.g., gcc)

```
__global__ void mykernel(void) { }

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

Hello World! with Device Code (cont'd)

- `mykernel()` does nothing in this case

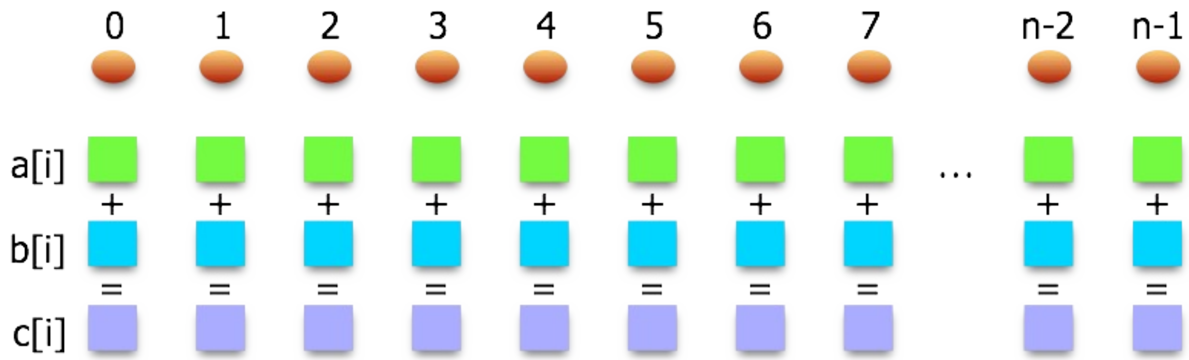
```
__global__ void mykernel(void) { }

int main(void) {
    mykernel<<<1,1>>>();
    printf("Hello World!\n");
    return 0;
}
```

```
> nvcc hello_world.cu
> a.out
Hello World!
>
```

Vector Addition in CUDA

- Data parallel programming model



A Simple Kernel to Add Two Integers

```
__global__ void add(int *a, int *b, int *c) {
    *c = *a + *b;
}
```

- add() runs on the device, so a, b and c must point to the device memory
- We need to allocate memory on the GPU for *a, *b, and *c
- Host and device memory are separate
  - Device pointers point to the GPU memory
    - May be passed to/from the host code
    - May not be dereferenced in the host code
- Host pointers point to the CPU memory
  - May be passed to/from the device code
  - May not be dereferenced in the device code
- Simple CUDA API for handling device memory
  - `cudaMalloc()`, `cudaFree()`, `cudaMemcpy()`

The Host Code to Add Two Integers

```
int main(void) {
    int a, b, c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = sizeof(int);
    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
    // Setup input values
    a = 3;
    b = 4;
    // Copy inputs to device
    cudaMemcpy(d_a, &a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, size, cudaMemcpyHostToDevice);
    // Launch add() kernel on GPU
    add<<<1,1>>>(d_a, d_b, d_c);
    // Copy result back to host
    cudaMemcpy(&c, d_c, size, cudaMemcpyDeviceToHost);
    // Cleanup
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

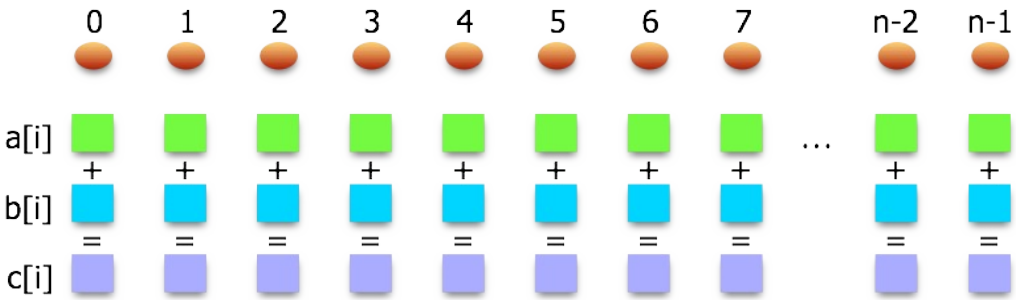
Running `Add()` in Parallel

- Instead of executing add() once, execute N instances of add() in parallel
- Each parallel invocation of add() is referred to as a block
  - The set of blocks is referred to as a grid
  - Each invocation can refer to its block index using blockIdx.x
- By using **`blockIdx.x`** to index into the array, each block handles a different element of the array

```
__global__ void add(int *a, int *b, int *c) {
    c[blockIdx.x] = a[blockIdx.x] + b[blockIdx.x];
}
```

Running `Add()` in Parallel (cont'd)

- Data parallel programming model
  - A set of instructions from the kernel are applied concurrently to each block in the grid
    - SPMD



Running `Add()` in Parallel (cont’d)

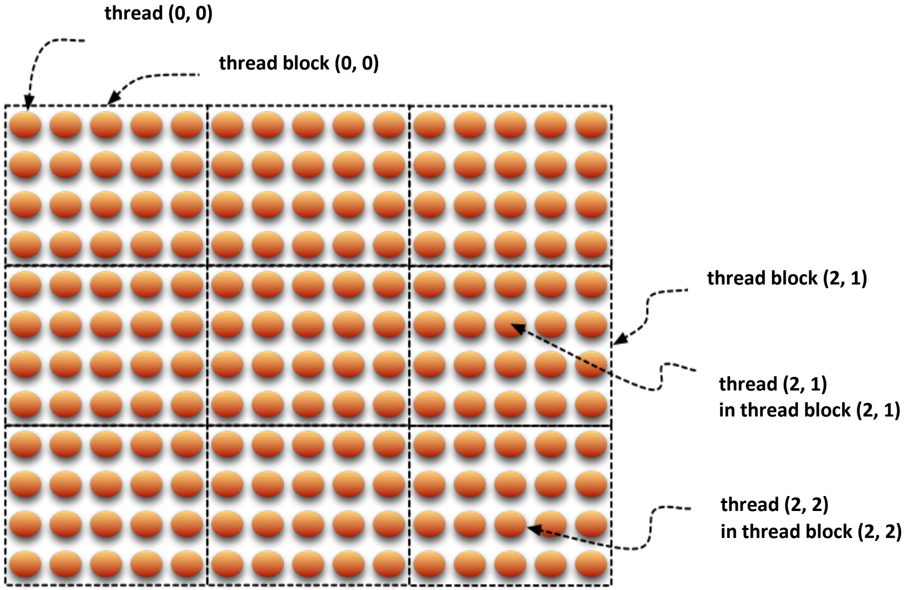
```
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size)
// Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N blocks
    add<<<N,1>>>(d_a, d_b, d_c);
// Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

CUDA Execution Model

- Thread
  - Sequential execution unit
  - All threads execute same sequential program
  - Threads execute in parallel
- Threads Block
  - A group of threads
  - Executes on a single Streaming Multiprocessor (SM)
  - Threads within a block can cooperate
    - Light-weight synchronization (barrier)
    - Data exchange
- Grid
  - A collection of thread blocks
  - Thread blocks of a grid execute across multiple SMs
  - Thread blocks do not synchronize with each other
  - Communication between blocks is expensive

CUDA Execution Model (cont'd)

- Grids map to GPUs
- Blocks map to the Streaming Multiprocessors (SM)
- Threads map to Scalar Processors (SP)
- Warps are groups of (32) threads that execute simultaneously



CUDA Execution Model (cont'd)

- Need to provide each kernel call with values for two key structures:
  - Number of blocks in each dimension
  - Number of threads per block in each dimension

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

- B – a structure that defines the number of blocks in the grid in each dimension
- T – a structure that defines the number of threads in a block in each dimension

CUDA Execution Model (cont'd)

- CUDA Built-In Variables
  - **`blockIdx.x, blockIdx.y, blockIdx.z`**
    - The block ID in the x-axis, y-axis, and z-axis of the block that is executing the given block of code
  - **`threadIdx.x, threadIdx.y, threadIdx.z`**
    - The thread ID in the x-axis, y-axis, and z-axis of the thread that is being executed by this streaming multiprocessor in this particular block
  - **`blockDim.x, blockDim.y, blockDim.z`**
    - The block dimension
    - The number of threads in a block in the x-axis, y-axis, and z-axis

Running `Add()` in Parallel with Multiple Threads

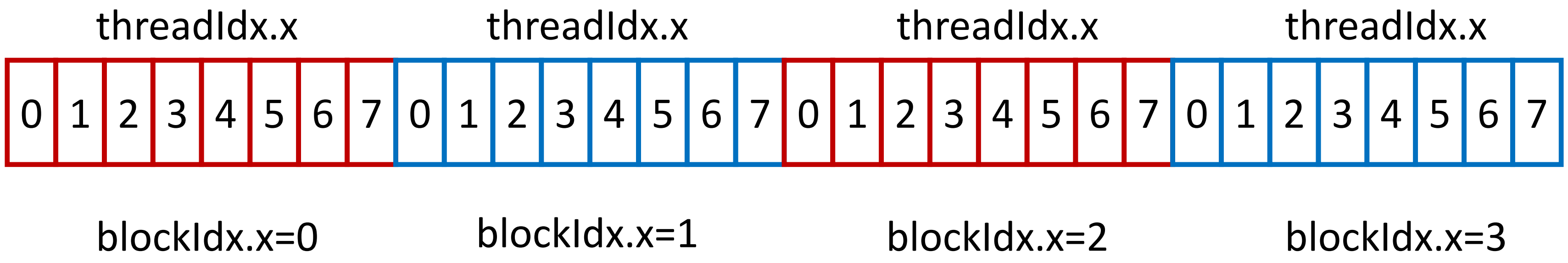
```
__global__ void add(int *a, int *b, int *c) {
    c[threadIdx.x] = a[threadIdx.x] + b[threadIdx.x];
}
```

- A block can be split into parallel threads
- Change add() to use parallel threads instead of parallel blocks

Running `Add()` in Parallel with Multiple Threads (cont'd)

```
#define N 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size)
// Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N blocks
    add<<<1,N>>>(d_a, d_b, d_c);
// Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

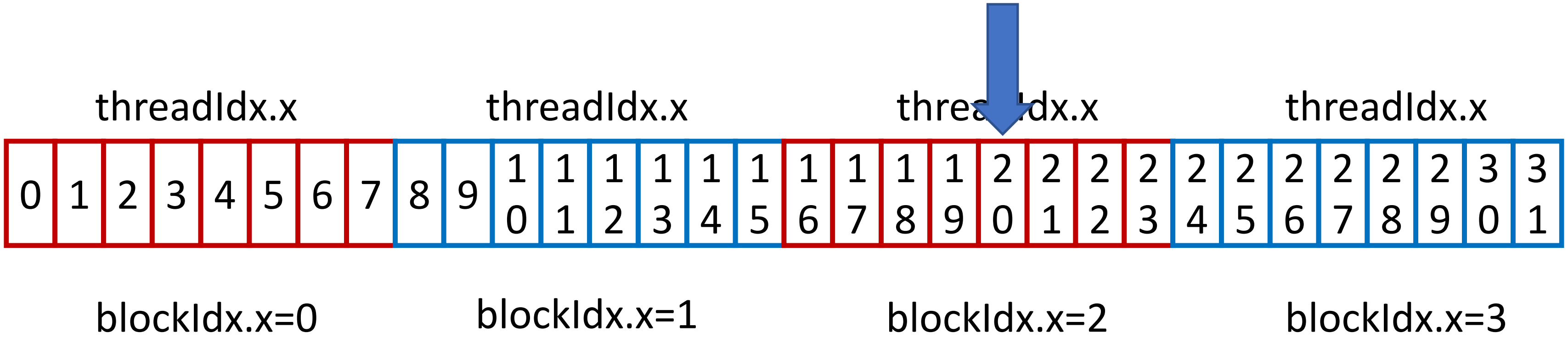
Multiple Threads and Multiple Blocks



threadIdx.x	threadIdx.x	threadIdx.x	threadIdx.x
0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3

- Consider indexing an array with one element per thread
  - 8 threads/block
- With M threads per block, a unique index for each thread is given by
  - index = threadIdx.x + blockIdx.x * M

Multiple Threads and Multiple Blocks (cont'd)



threadIdx.x	threadIdx.x	threadIdx.x	threadIdx.x
0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23	24 25 26 27 28 29 30 31
blockIdx.x=0	blockIdx.x=1	blockIdx.x=2	blockIdx.x=3

- index = threadIdx.x + blockIdx.x * M

  = 4 + 2 * 8

  = 20

- index = threadIdx.x + blockIdx.x * blockDim.x

Multiple Threads and Multiple Blocks (cont'd)

```
__global__ void add(int *a, int *b, int *c) {
   int index = threadIdx.x + blockIdx.x * blockDim.x;
   c[index] = a[index] + b[index];
}
```

Multiple Threads and Multiple Blocks (cont'd)

```
#define N (2048*2048)
#define THREADS_PER_BLOCK 512
int main(void) {
    int *a, *b, *c; // host copies of a, b, c
    int *d_a, *d_b, *d_c; // device copies of a, b, c
    int size = N * sizeof(int);
// Alloc space for device copies of a, b, c
    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);
// Alloc space for host copies of a, b, c and setup input values
    a = (int *)malloc(size); random_ints(a, N);
    b = (int *)malloc(size); random_ints(b, N);
    c = (int *)malloc(size)
// Copy inputs to device
    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);
// Launch add() kernel on GPU with N blocks
    add<<<N/THREADS_PER_BLOC, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
// Copy result back to host
    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);
// Cleanup
    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```

Handling Arbitrary Vector Sizes

```
__global__ void add(int *a, int *b, int *c, int n) {
   int index = threadIdx.x + blockIdx.x * blockDim.x;
   if (index < n)
       c[index] = a[index] + b[index];
}
```

```
 // N: the number of array elements
 // M: blockDim.x
 add<<< (N + M - 1)/M, M >>>(d_a, d_b, d_c, N)
```