GPU Computing

Lecture 3

Young-Ho Gong

광운대학교
KwangWoon University

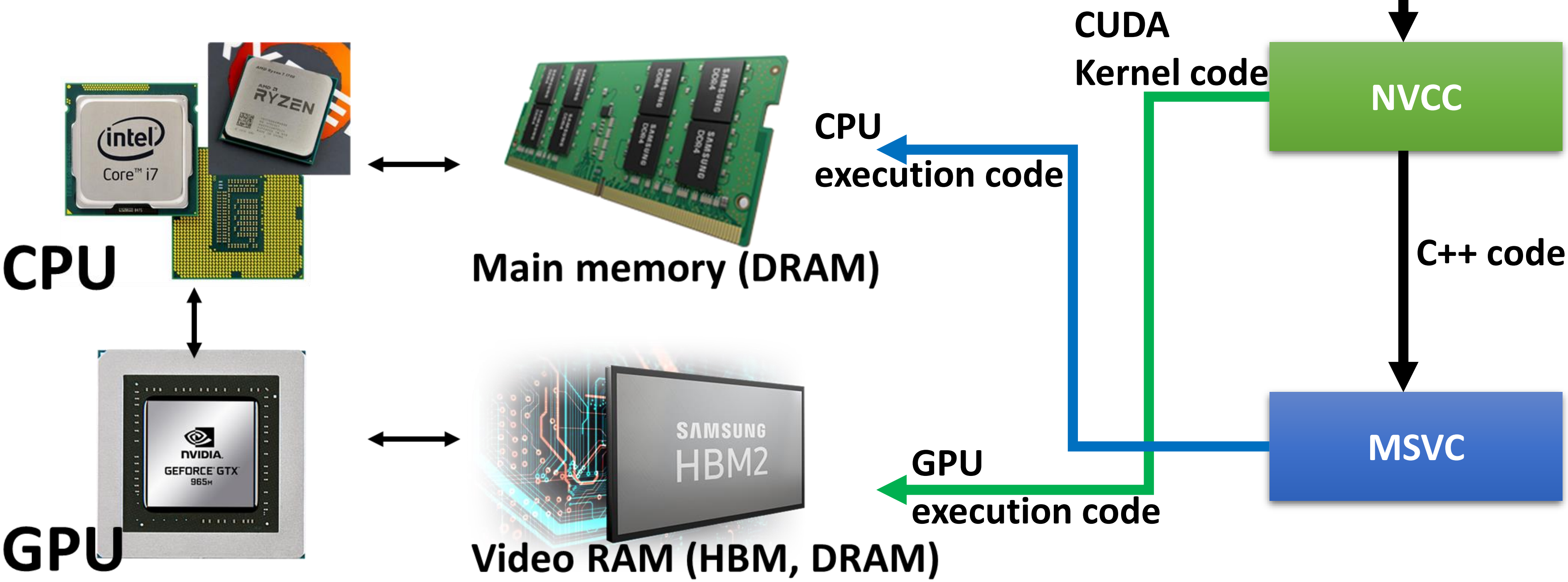
Contents

- Kernel concept
  - the central or most important part of something.

    "this is the kernel of the argument"
- CUDA programming model
- CUDA function declarations
- Vector addition example
- CPU implementation
  - For loop
- Kernel concept
  - Function, as loop-body
- CPU kernel implementation
- CUDA implementation
  - Multiple thread launch
- Kernel launch
  - Example code

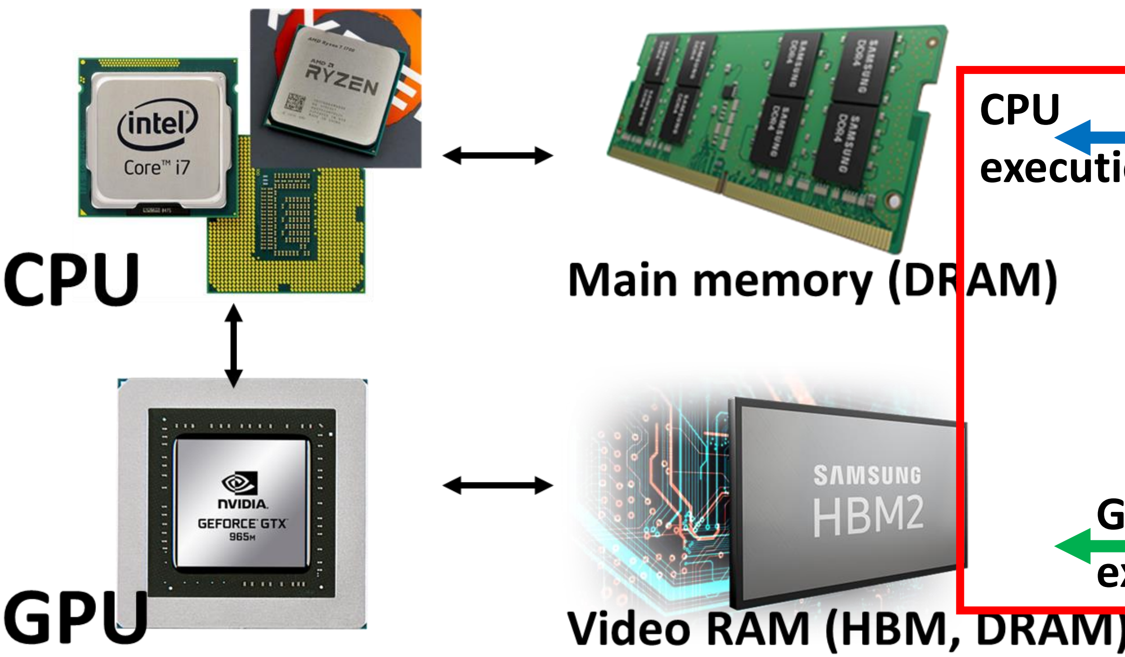
CUDA programming model

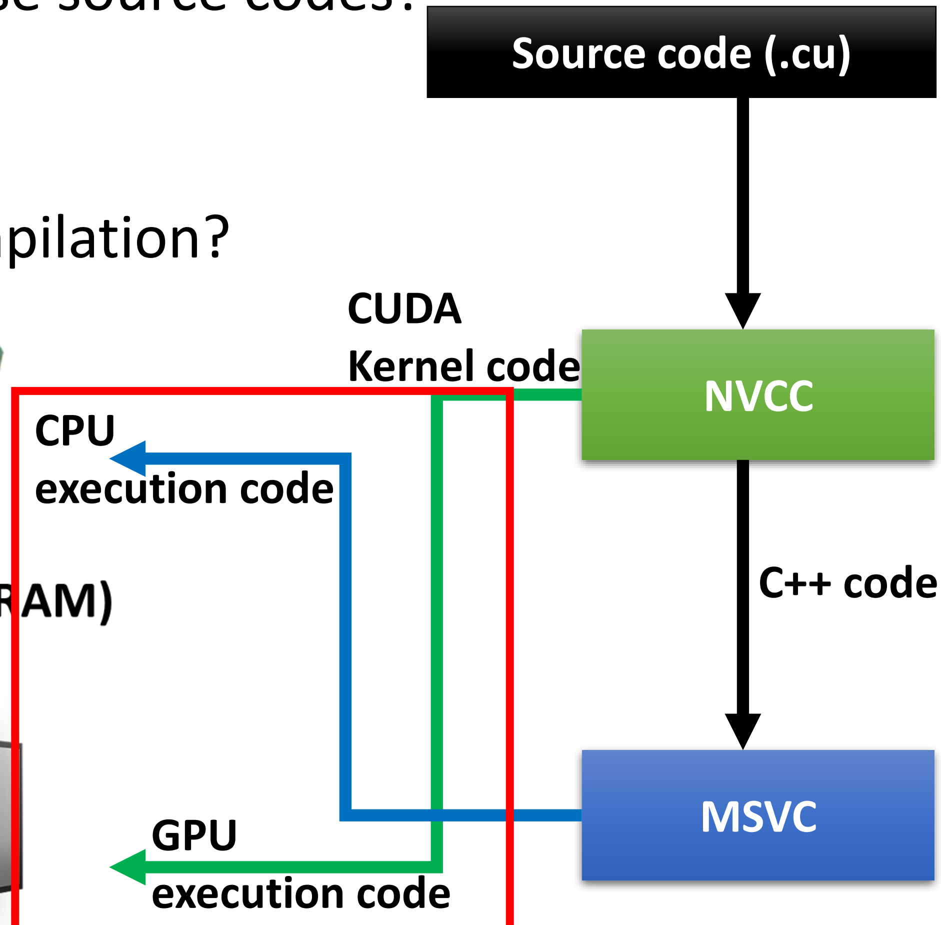
- We have two sets of execution code
  - For CPU (host part), msvc is used
  - For GPU (device part), nvcc is used



CUDA programming model

- At the source code level,
  - How can we distinguish those source codes?
  - Which one is for CPU?
  - Which one is for GPU?
  - And, what is the unit of compilation?





CUDA programming model

- Compilation unit?
  - Function
  - Each function will be assigned to CPU and/or GPU

- How to distinguish them?
  - Use PREFIX for each function

  __host__ : can be called by CPU (default, can be omitted)

  __device__: called from other GPU functions,
  cannot be called by the CPU

  __global__: launched by CPU, (bridge between host/device)
  cannot be called from GPU, must return void

  __host__ and __device__ qualifiers can be combined

More on CUDA Function Declarations

- __global__ defines a kernel function
  - Each "__" consists of two underscore characters
  - A kernel function must return void
- __device__ and __host__ can be used together
  - Compiled twice (!)

	Executed on the:	Only callable from the:
__device__ float DeviceFunc()	Device	Device
__global__ void KernelFunc()	Device	Host
__host__ float HostFunc()	Host	Host

Example Source Code

```
__device__ inline void myAtomicADD(unsigned long long int* addr){
  unsigned long long int oldVal = *address;
  unsigned long long int newVal = oldVal + val;
  unsigned long long int readback;
  while ((readback = atomicCAS(address, oldVal, newVal)) != oldVal){
    oldVal = readback;
    newVal = oldVal + val;
  }
}

__global__ void kernel(unsigned long long int* pCount){
    myAtomicAdd(pCount,1ULL);
}

__host__ int main(void){
    unsigned long long int aCount[1];
    ...
}
```

GPU Executable Functions

- CUDA language == C/C++ with some restrictions:
  - Can only access **GPU memory**
    - In new versions, can access host memory directly, with performance drawbacks.
  - **No recursion**
  - **No dynamic polymorphism**
  - No variable number of arguments
  - No static variables
    - No static variable declarations inside the function

Example: Hello.cu

- Make a new source code, hello.cu
- .cu file = .cpp file + CUDA kernel source code
  - Default function: __host__
- Hello.cu
  -
    ```
    #include <stdio.h>

    int main (){
       printf( "Hello, world!\n" );
       return 0;
    }
    ```
- Result
  - Hello, world!

Programming TIP

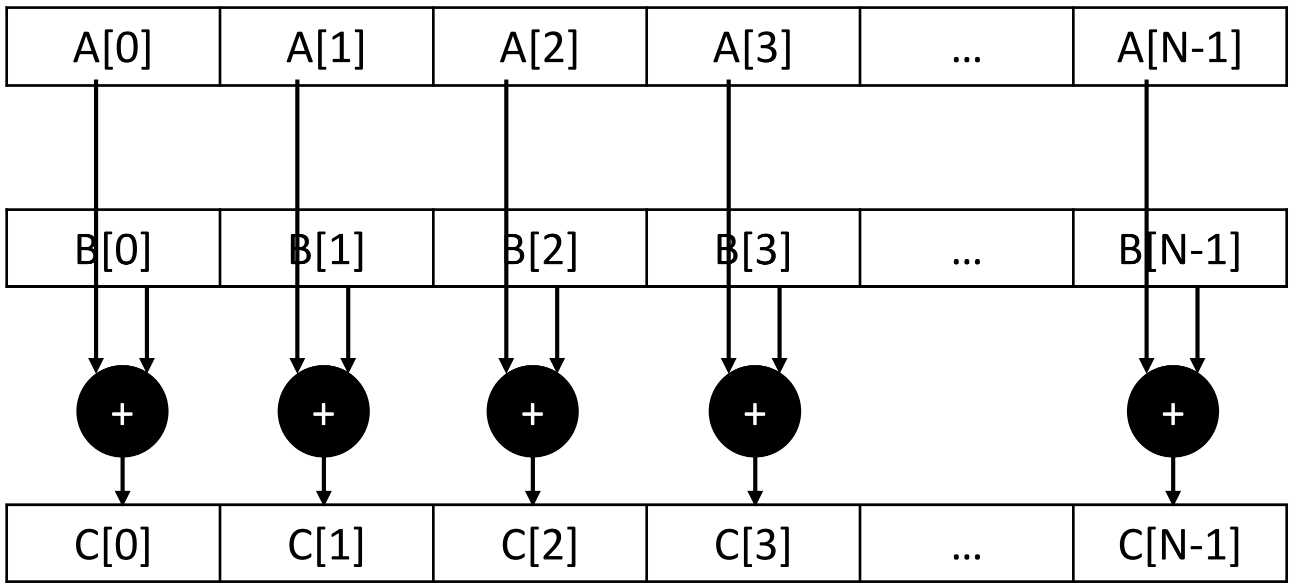
- Sometimes CUDA does not output anything from printf().
- Try to use fflush()
  - Printf → stdio buffer → output
  - Fflush: flush stdio buffer
- Hello.cu

```
#include <stdio.h>

int main (){
   printf( “Hello, world!\n” );
   fflush (stdout);
   return 0;
}
```

Scenario: Vector Addition

- We use arrays for vector representation
  - C[i] = A[i] + B[i]
  - Vector A
  - Vector B
  - Vector C



Vector Addition

- Vector: represented as 1D array
  - const int a[SIZE];
  - const int b[SIZE];
  - int c[SIZE];
- Vector addition: c[…] = a[…] + b[…]
- Serial execution: for-loop
  - E.g., for(i=0;i<SIZE;i++)
    c[i]=a[i]+b[i];
- **A single CPU** does for i=0,1,2,…,SIZE-1
- CUDA execution: parallel kernel execution

Introducing a kernel function

- Converting to a kernel function,

```
void addKernel(int* dst, int aVal, int bVal){
        *dst = aVal + bVal;
}
…
```

- In main function,

```
for (int i=0;i<SIZE;i++)
        addKernel(&c[i],a[i],b[i]);
}
```

- **Kernel function**:
  - Substitute for the **loop body**
  - With proper data values

Introducing a kernel function

- Another view
  - CPU[0] executes addKernel() with its own data
  - CPU[1] executes addKernel() with its own data
  - CPU[2] executes addKernel() with its own data

    …
  - CPU[SIZE-1] executes addKernel() with its own data

- We have a single CPU → sequential execution
- If we have many CPUs? → Parallel execution possible!

CUDA kernel

- CPU kernels
  - With a single CPU
  - For-loop

- Sequential execution
- For-loop
  - CPU[0]
  - CPU[1]
  - CPU[2]

    ...
  - CPU[N-1]

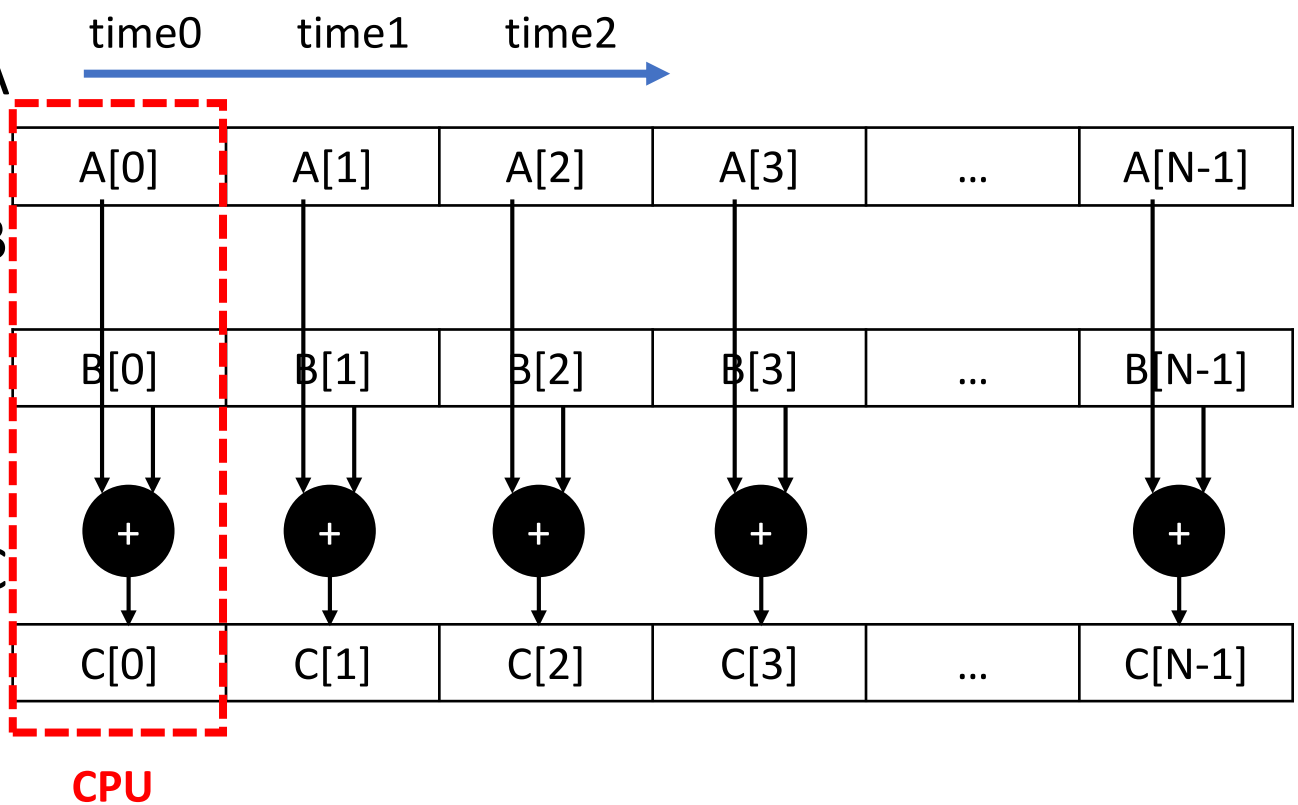
- GPU kernels
  - A set of GPU cores
  - Multiple threads

- Parallel execution
- Kernel launch
  - GPU[0]
  - GPU[1]
  - GPU[2]

    ...
  - GPU[N-1]

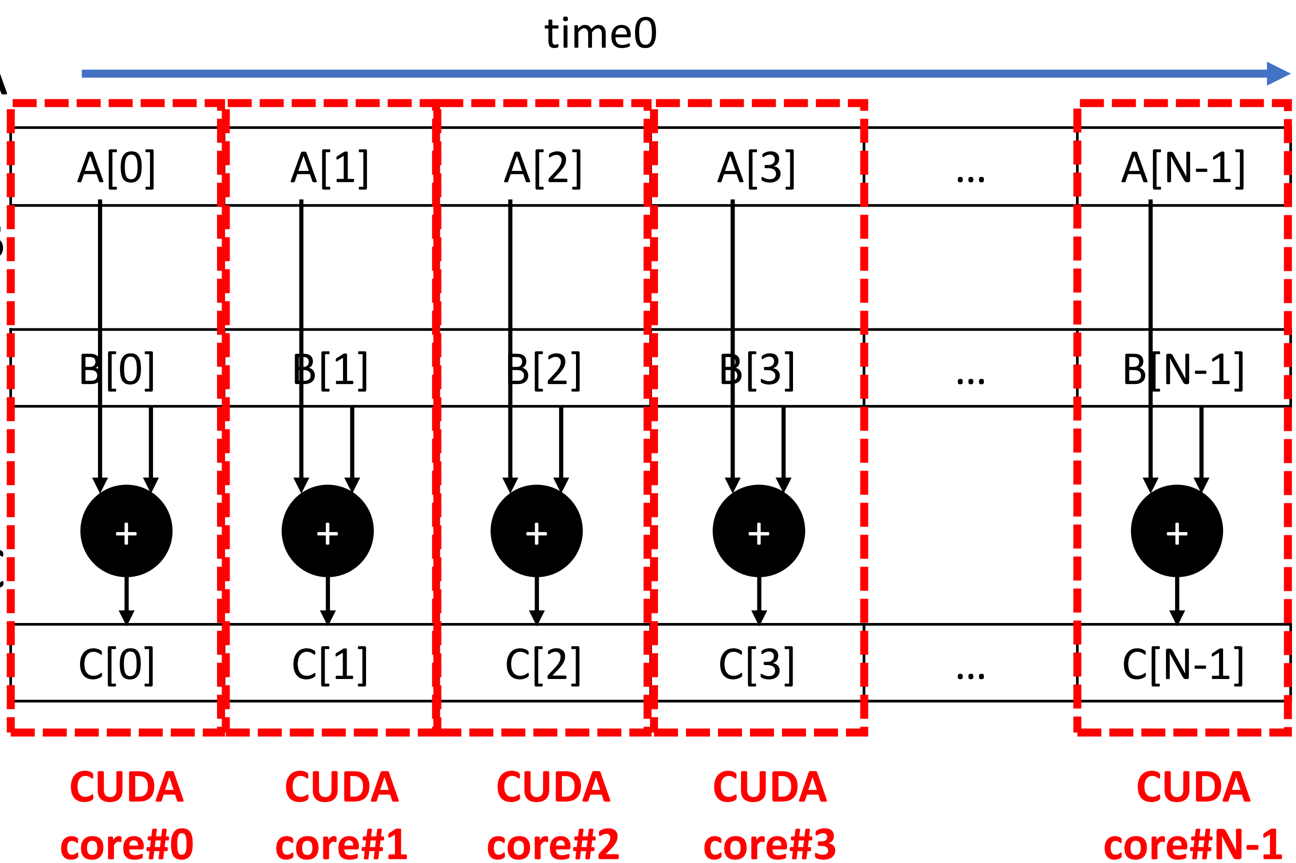
CPU-based vector addition

- A single CPU does an addition
  - C[i] = A[i] + B[i]
  - Vector A
  - Vector B
  - Vector C



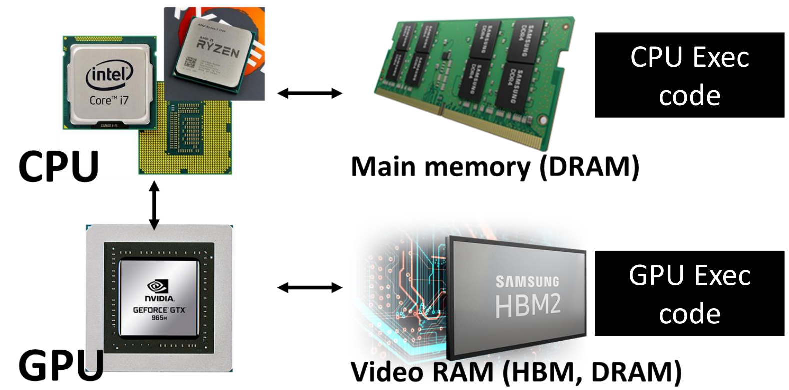
CUDA-based vector addition

- We have many GPU cores
  - They do the addition at the same time!
  - Vector A
  - Vector B
  - Vector C



Scenario: CUDA vector addition

- Step 1: Host-side
  - Make A, B with source data
  - Prepare C for the result
- Step 2: data copy host → device
  - cudaMemcpy from host to device
- Step 3: addition in CUDA
  - **Kernel launch for CUDA device**
  - Result will be stored in device memory (VRAM)
- Step 4: data copy device → host
  - cudaMemcpy from device to host
- Step 5: host-side
  - Print the output data



CUDA kernel launch syntax

- Prepare a CUDA kernel function
  - ```
    __global__ void addKernel(int *c, const int* a, const int* b){
        int i = threadIdx.x;
        c[i] = a[i] + b[i];
    }
    ```
- Kernel launch syntax
  - **addKernel <<<1,SIZE>>>(dev_c, dev_a, dev_b);**
  - **Function_Name<<<# of Blocks,# of Threads>>>(arg1,arg2,…);**
- CUDA view
  - A thread executes addKernel() with **threadIdx.x = 0**
  - A thread executes addKernel() with **threadIdx.x = 1**
  - A thread executes addKernel() with **threadIdx.x = 2**

    …
  - A thread executes addKernel() with **threadIdx.x = SIZE-1**
    ```

Example code

- Kernel function

```
%%cu

#include <stdio.h>

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

Example code

```
int main()
{
    const int SIZE = 5;
    const int a[SIZE] = { 1, 2, 3, 4, 5 };
    const int b[SIZE] = { 10, 20, 30, 40, 50 };
    int c[SIZE] = { 0 };
```

```
    int *dev_a = 0;
    int *dev_b = 0;
    int *dev_c = 0;

    cudaMalloc((void**)&dev_c, SIZE * sizeof(int));
    cudaMalloc((void**)&dev_a, SIZE * sizeof(int));
    cudaMalloc((void**)&dev_b, SIZE * sizeof(int));
```

Example code

```
    cudaMemcpy(dev_a, a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    cudaMemcpy(dev_b, b, SIZE * sizeof(int), cudaMemcpyHostToDevice);
    addKernel<<<1, SIZE>>>(dev_c, dev_a, dev_b);
```

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

```
    printf("{1,2,3,4,5} + {10,20,30,40,50} = {%d,%d,%d,%d,%d}₩n",
        c[0], c[1], c[2], c[3], c[4]);

    cudaFree(dev_c);
    cudaFree(dev_a);
    cudaFree(dev_b);
    return 0;
}
```

Example code (Assignment)

- Result?
  - Quick assignment
    - Write the vector addition code with the SIZE of 5
    - The assignment code should run d[i] = a[i] + b[i] + c[i].
      (NOT the same as the example code)
    - The values of source arrays (a,b, and c) can be initialized with random values.
    - You should capture your source code and the results.

Summary

- Kernel concept

- CUDA programming model
- CUDA function declarations
- Vector addition example
- CPU implementation
  - For loop
- Kernel concept
  - Function, as loop-body
- CPU kernel implementation
- CUDA implementation
  - Multiple thread launch
- Kernel launch
  - Example code and Assignment!

Next step?

- CUDA kernel implementation
  - 2D matrix examples?

Thank you

Any questions?

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

광운대학교
KwangWoon University