GPGPU Basics

In This Lecture

- **GPGPU Introduction**
- **CUDA Basics**
  - CUDA Overview
  - CUDA Kernel
  - CUDA Data Transfer
  - Recall & Remind: Vector Addition

GPUs vs CPUs

CPUs vs. GPUs

- **The design of a CPU is optimized for sequential code performance.**
  - out-of-order execution, branch-prediction
  - large cache memories to reduce latency in memory access
  - multi-core

- **GPUS**
  - many-core
  - massive floating point computations for video games
  - much larger bandwidth in memory access
  - no branch prediction or too much control logic: just compute

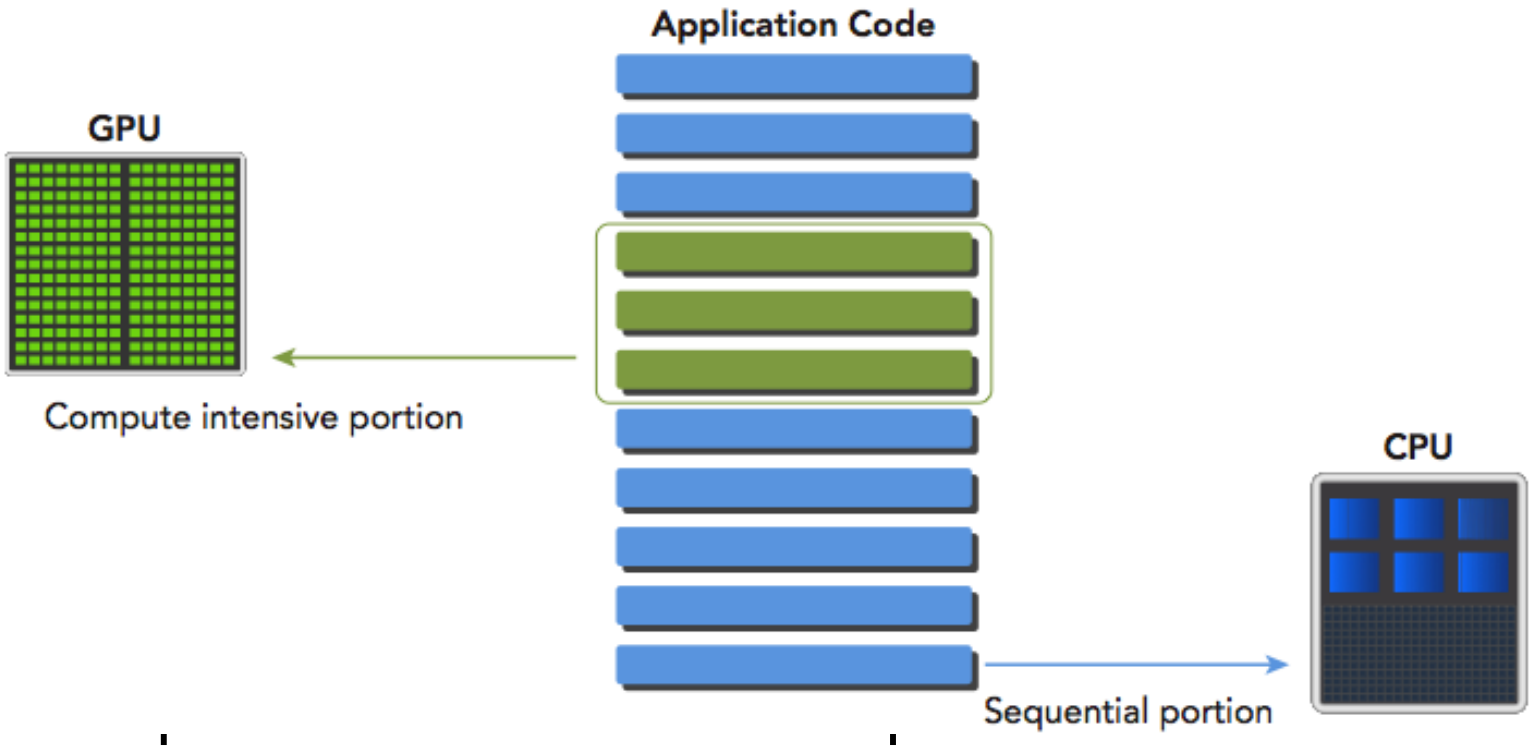
CPUs and GPUs

- GPUs are designed as numeric computing engines, and they will not perform well on some tasks on which CPUs are designed to perform well;
- One should expect that most applications will use both CPUs and GPUs, executing the sequential parts on the CPU and numerically intensive parts on the GPUs.
- We are going to deal with heterogeneous architectures: CPUs + GPUs.

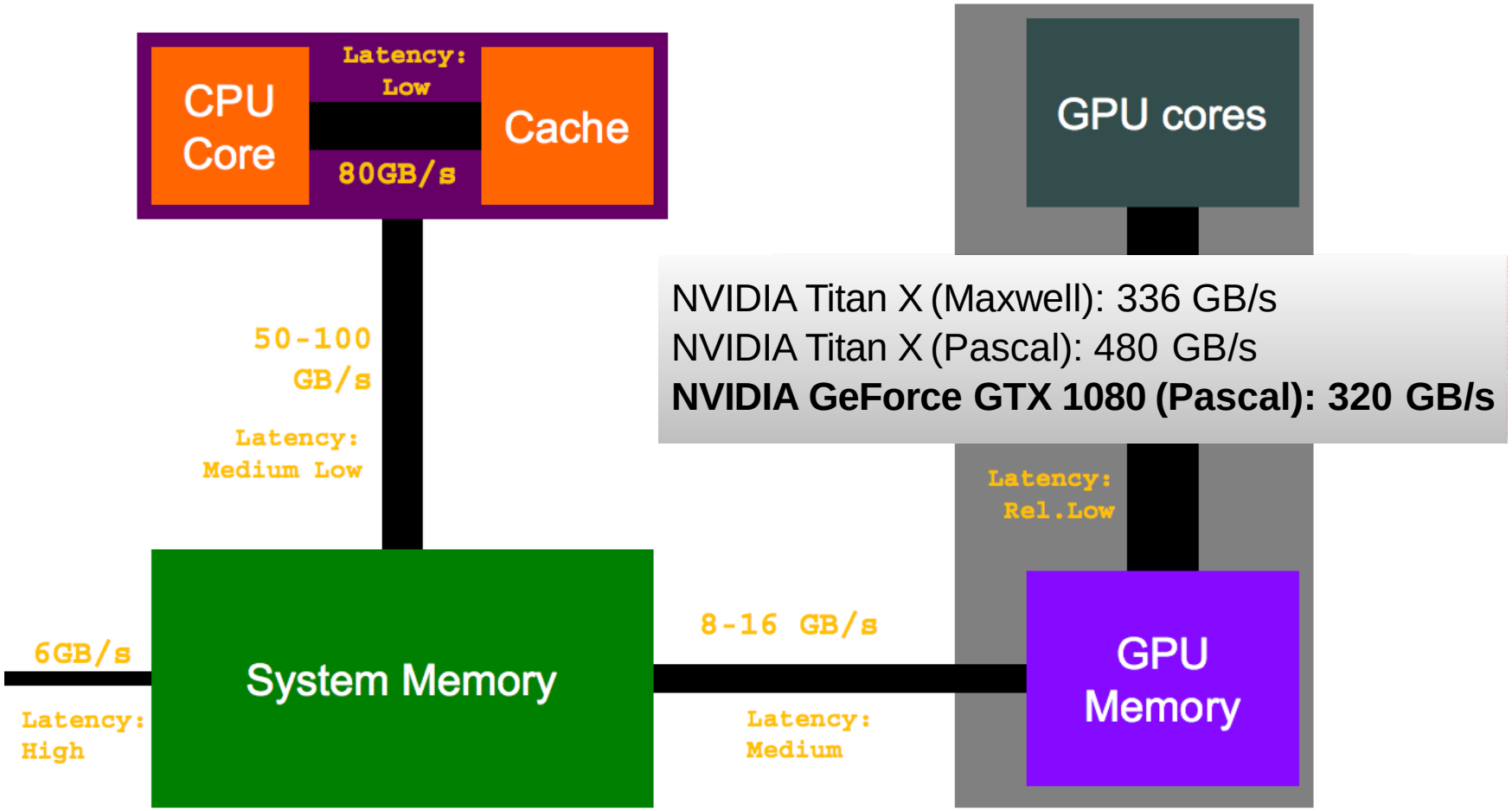
Heterogeneous Computing

- CPU computing is good for control-intensive tasks, and GPU computing is good for data-parallel computation-intensive tasks.
- The CPU is optimized for dynamic workloads marked by short sequences of computational operations and unpredictable control flow;
- GPUs aim at workloads that are dominated by computational tasks with simple control flow.

Heterogeneous Computing



Bandwidth in a CPU-GPU System



Heterogeneous Computing

- **A heterogeneous application consists of two parts:**
  - Host code
  - Device code
- **Host code runs on CPUs and device code runs on GPUs.**

Threads

- **Threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off CPU execution channels to provide multithreading capability. Context switches are slow and expensive.**

- **We deal with a few tens of threads per CPU, depending on Hyper-Threading.**

- **Threads on GPUs are extremely lightweight. In a typical system, thousands of threads are queued up for work. If the GPU must wait on one group of threads, it simply begins executing work on another.**

- **We deal with tens of thousands of threads per GPU.**

SIMT

GPU is a SIMD (Single Instruction, Multiple Data) device → it works on "streams" of data

- Each "GPU thread" executes one general instruction on the stream of data that the GPU is assigned to process
- NVIDIA calls this model SIMT (single instruction multiple thread)

The SIMT architecture is similar to SIMD. Both implement parallelism by broadcasting the same instruction to multiple execution units.

A key difference is that SIMD requires that all vector elements in a vector execute together in a unified synchronous group, whereas SIMT allows multiple threads in the same group to execute independently.

SIMT

- The SIMT model includes three key features that SIMD does not:
- Each thread has its own instruction address counter.
- Each thread has its own register state.
- Each thread can have an independent execution path.

Introduction to CUDA

High Performance Computing

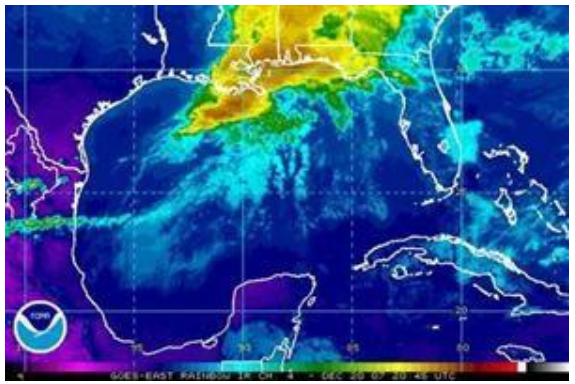
- Speed: Many problems that are interesting to scientists and engineers
  would take a long time to execute on a PC or laptop: months, years, "never".

- Size: Many problems that are interesting to scientists and engineers
  can't fit on a PC or laptop with a few GB of RAM or a few 100s of GB of disk space.

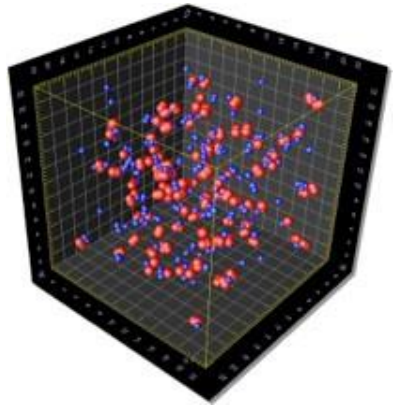
- Supercomputers or clusters of computers can make these problems practically numerically solvable.

Scientific and Engineering Problems

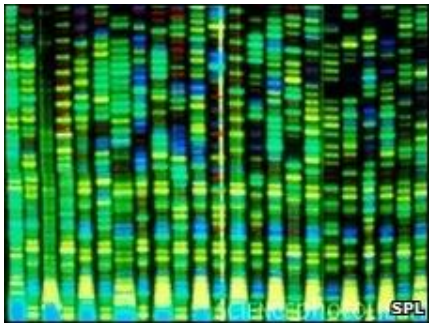
- Simulations of physical phenomena such as:
  - Weather forecasting
  - Earthquake forecasting
  - Galaxy formation
  - Oil reservoir management
  - Molecular dynamics



- Data Mining: Finding needles of critical information in a haystack of data such as:
  - Bioinformatics
  - Signal processing
  - Detecting storms that might turn into hurricanes



- Visualization: turning a vast sea of data into pictures that scientists can understand.



- At its most basic level, all of these problems involve many, many **floating point operations**.

General Purpose GPU Designs

- High performance pipelines call for high-speed floating point operations.
- Known as GPGPU (General-purpose computing on graphics processing units) -- Difficult to do with specialized graphics pipelines, but possible.)
- By mid 2000's, recognized that individual stages of graphics pipeline could be implemented by more general purpose processor cores (although with a data-parallel paradigm)
- 2006 -- First GPU for general high performance computing as well as graphics processing, NVIDIA GT 80 chip/GeForce 8800 card.
- Unified processors that could perform vertex, geometry, pixel, and **general computing operations**
- Could now write programs in C rather than graphics APIs.
- Single-instruction multiple thread (SIMT) programming model

CUDA Execution Model Overview

- **Architecture and programming model, introduced in NVIDIA in 2007.**
- **Enables GPUs to execute programs written in C in an integrated host (CPU) + device (GPU) app C program.**
  - Serial or modestly parallel parts in host C code.
  - Highly parallel parts in device SIMT codes kernel code.
- **Differences between GPU and CPU threads:**
  - GPU threads are extremely lightweight with very little creation overhead.
  - GPU needs 1000's of threads for full efficiency (multi-core CPU needs only a few).

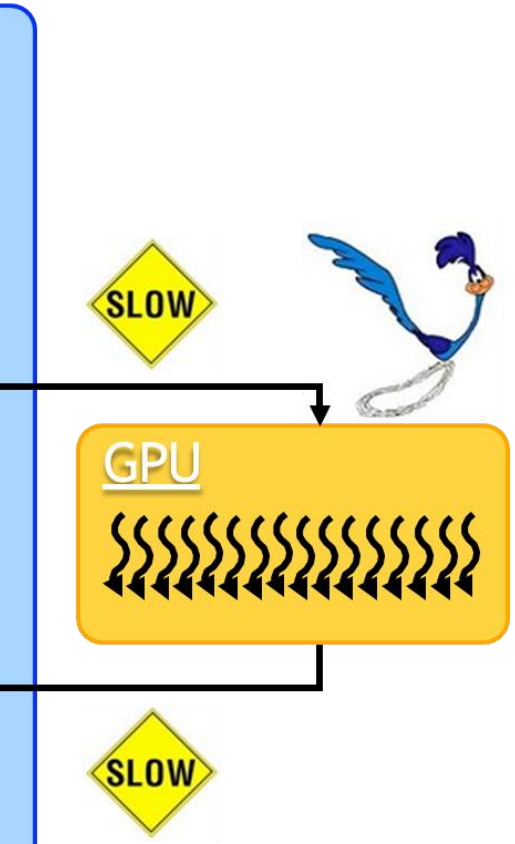
CUDA Execution Model Overview

C/CUDA Code

```
// serial code
int main() {
 printf("Hello world!"\n);
// allocate data
 cudaMalloc(...);
// copy data
 cudaMemcpy(...);
// execute kernel

 cudaRun<<<..>>>(...);

   ...

 cudaThreadSynchronize();

// serial code
 printf("Running.."\n);

   ...

 exit (0);
}
```

CUDA Kernel Code

```
// kernel
___global___
void cudaRun(...) {
   ...
}
```



CUDA Programming Basic

Hello World v.1.0: Basic C Program

```
#include <stdio.h>

int main() {
 printf("Hello world!"\n);
 exit (0);
}
```

```
nvcc -o hello hello.cu
```

```
OUTPUT:

Hello World!
```

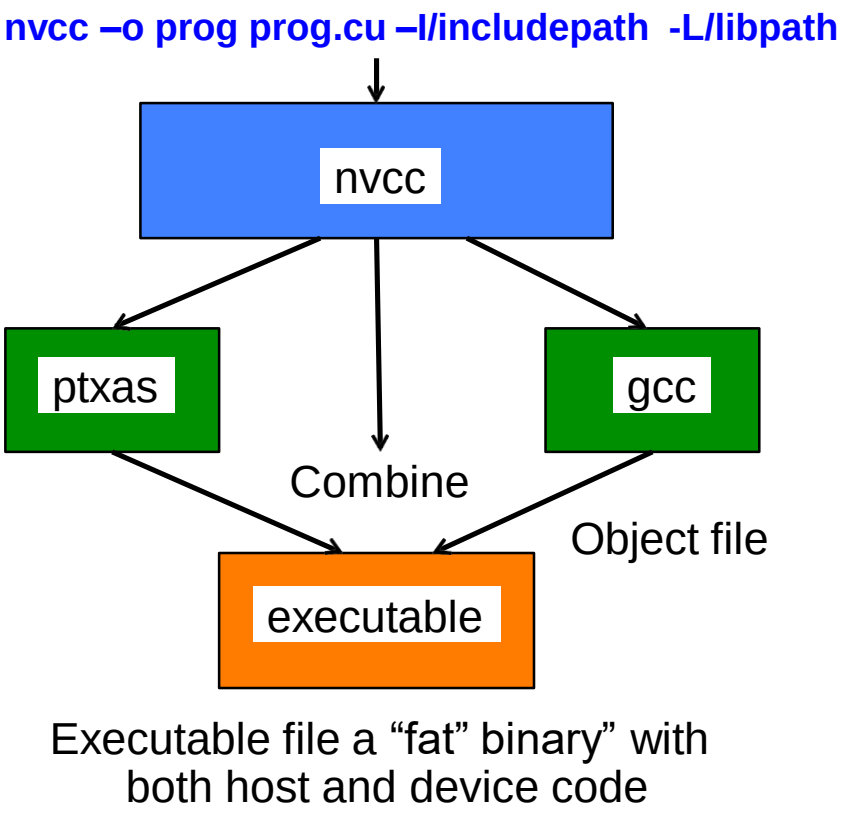
Executing a CUDA Program

- **./hello(Execution File Name)**
- Host code starts running.
- When first encounter device kernel, GPU code physic ally sent to GPU and function launched on GPU.
- Hence first launch will be slow!!

Compilation Process

ptxas =PTX assembler =p
arallel thread execution

- nvcc "wrapper" divides code into host and device parts.
- Host part compiled by regular C compiler.
- Device part compiled by the NVIDIA "ptxas" assembler.
- Two compiled parts combined into one executable.



Executable file a "fat" binary" with both host and device code

Hello World v.2.0: Kernel Calls

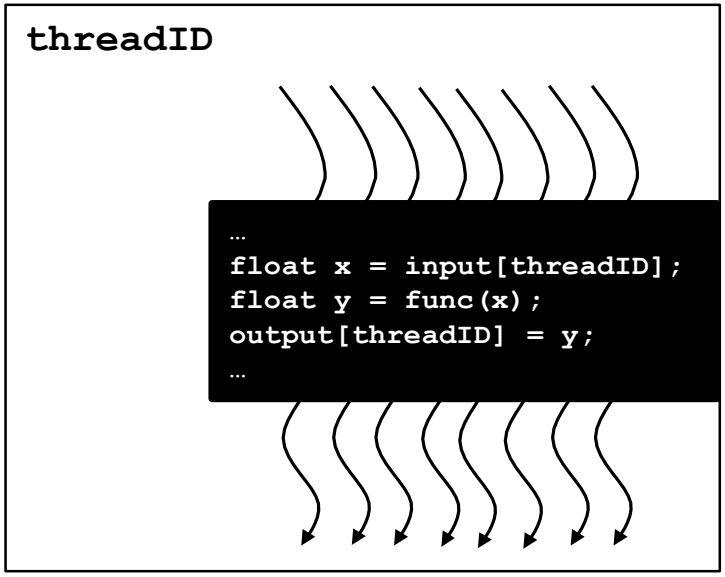
```
#include <stdio.h>


int main() {
 kernel<<<1,1>>>();
 printf("Hello world!\n");
 exit (0);
}

__global void kernel () {
  // does nothing
}
```

- An empty function named "kernel" qualified with the specifier __global__ (there are **two** underscores on each side)
  - Indicates to the compiler that the code should be run on the device, not on the host.
- A call to the empty device function with "<<<1,1>>>"
  - Within the triple brackets <<< >>> are **memory arguments** (for the blocks and threads) and within the parentheses () are the **parameter arguments** (that you normally use in C/C++).

```
OUTPUT:

Hello World!
```

CUDA Kernel Routines

- A kernel routine is executed on the device. This one set of instructions is executed by each allocated thread (SIMT).
- The kernel code is mostly the same as C. One exception is that each kernel code has an associated thread ID so that it can access different data.



CUDA Function Declarations

Function Declarations	Executed on the:	Only callable from the
__device__ float myDeviceFunc()	Device	Device
__global__ void myKernelFunc()	Device	Host
__host__ float myHostFunc()	Host	Host

- **`__global__` defines a kernel function, launched by host, executed on the device**
  - Must return `void`

- **By default, all functions in a CUDA program are `__host__` functions if they do not have any of the CUDA keywords in their declaration.**

Hello World v.3.0: Parameter Passing

```
#include <stdio.h>

__global__ void add (int a, int b, int *c);

int main() {
 int c;
 int *dev_c;

 cudaMalloc((void**)&dev_c, sizeof(int));

 add<<<1,1>>>(2,7,dev_c);

 cudaMemcpy(&c, dev_c, sizeof(int),
            cudaMemcpyDeviceToHost);

 printf("Hello world!\n");
 printf("2 + 7 = %d\n", c);

 cudaFree(dev_c);

 exit (0);
}

__global__ void add (int a, int b, int *c) {
 c[0] = a + b;
}
```

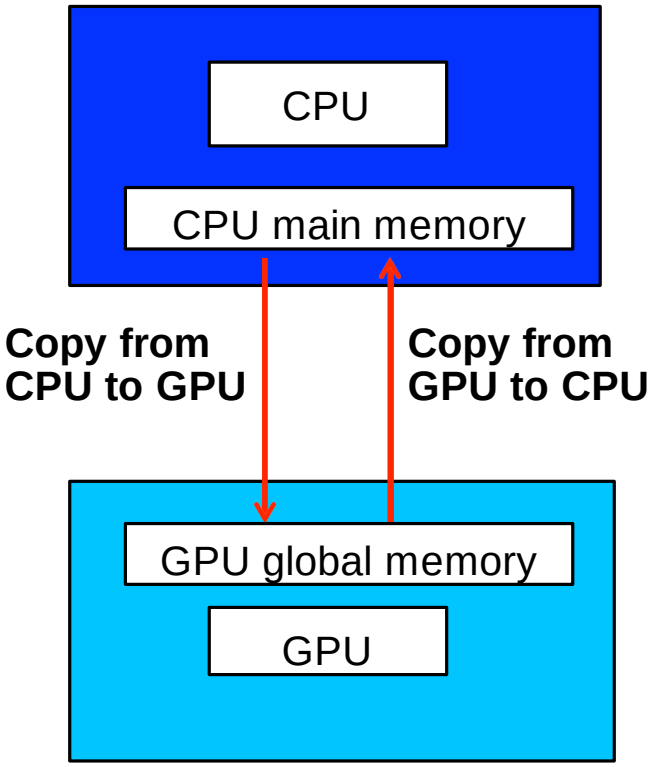
- Parameter passing is similar to C.
- There exists a separate set of host + device memory.
- We need to allocate memory to use it on the device.
- We need to copy memory from the host to the device and/or vice versa via cudaMemcpy().
- CudaMalloc (similar to malloc) allocates global memory on the *device*.
- CudaFree (similar to free) deallocates global memory on the *device*.

```
OUTPUT:

Hello World!
2 + 7 = 9
```

CPU and GPU Memory

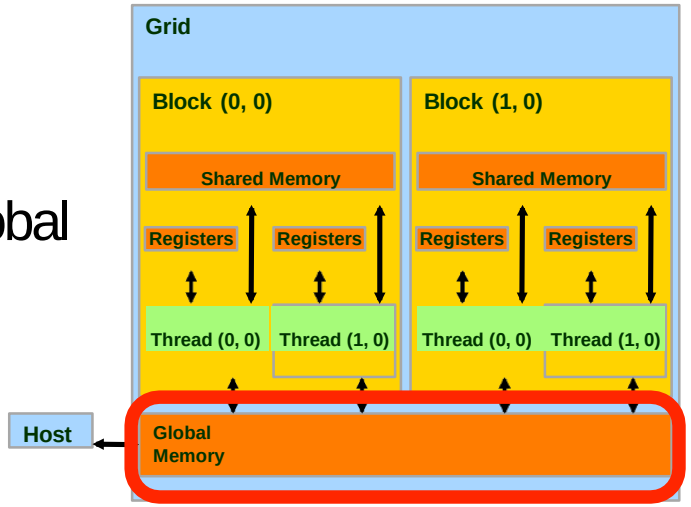
- A compiled CUDA program has:
  - 1) code executed on the CPU
  - 2) (kernel) code executed on the GPU

- The CPU and GPU memories are distinct and separate. Therefore, we need to:
  - 1) allocate a set of host data and device data and
  - 2) explicitly transfer data from the CPU to the GPU and vice versa.



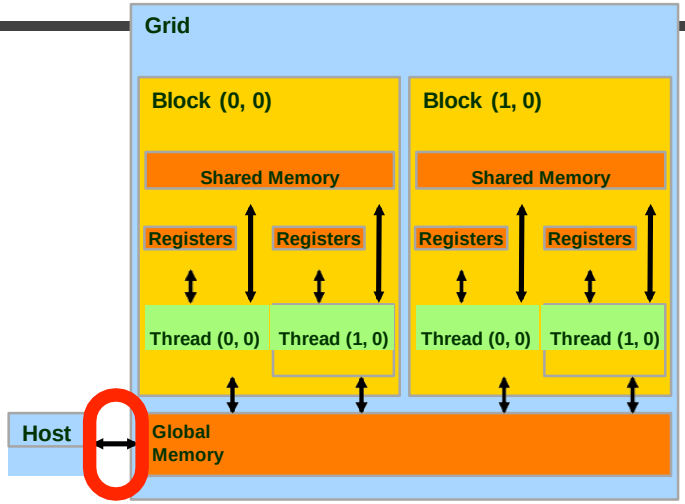
Allocating and Deallocating Device Memory Space

- Allocating Memory: allocates object in device global memory and returns pointer to it.
  - `int *dev_C;`
  - `int size = N *sizeof(int);`
  - `cudaMalloc((void**)&dev_C, size);`



- Deallocating Memory: free object from device global memory
  - `cudaFree(dev_C)`

Transferring Data via cudaMemcpy



Destination Source

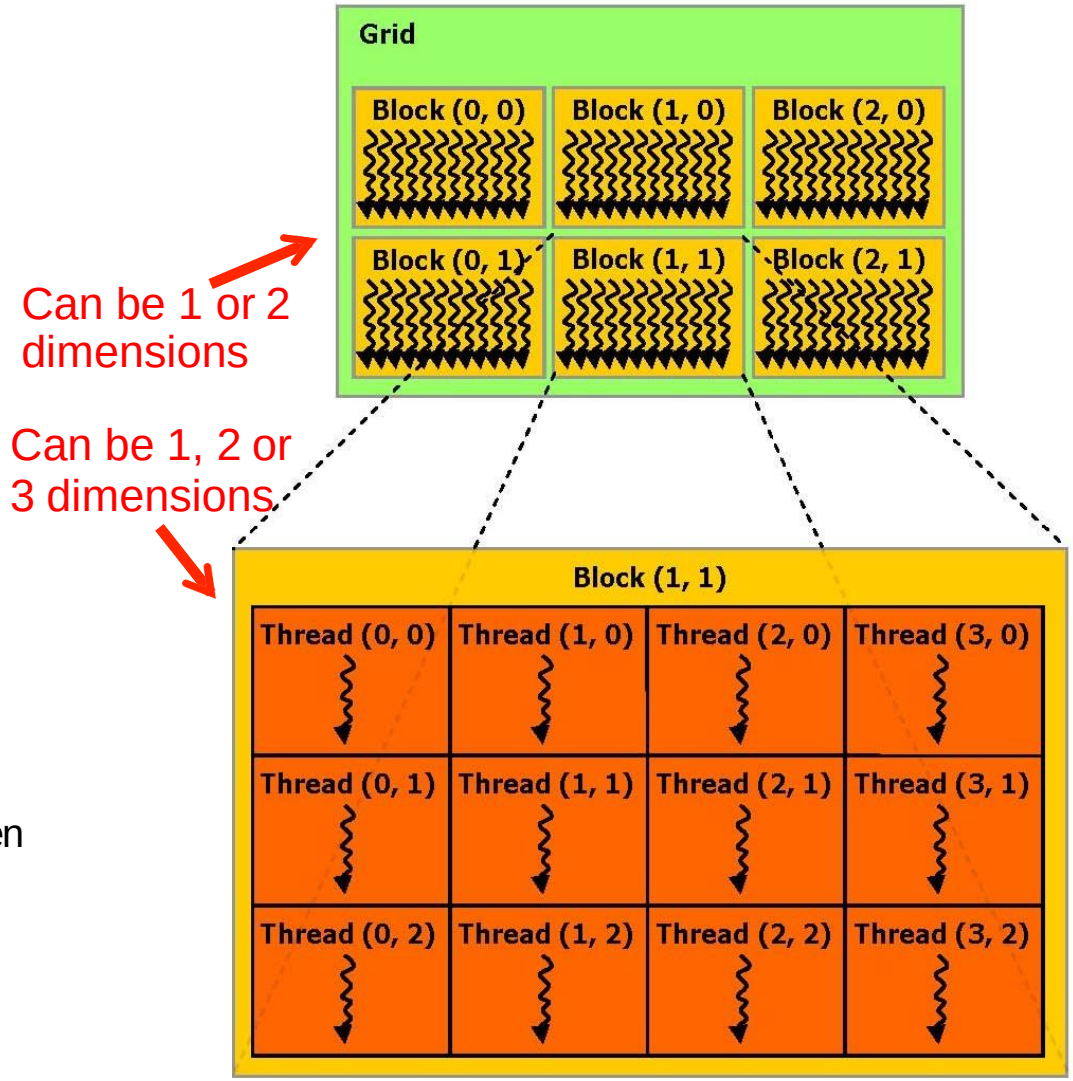
- `cudaMemcpy( dev_a, a, size, cudaMemcpyHostToDevice);`

- `cudaMemcpy( c, dev_c, size, cudaMemcpyDeviceToHost);`

- "dev_a" and "dev_c" are pointers to device data
- "a" and "c" are pointers to host data
- "size" is the size of the data
- "cudaMemcpyHostToDevice" and "cudaMemcpyDeviceToHost" tells cudaMemcpy the source and estination of the operation.

GPU Thread Structure

- A CUDA kernel is executed on an **`array or matrix`** of threads.
  - All threads run the same code (SIMT)
  - Each threads has an ID that is uses to compute memory addresses and make control decisions.
    - Block ID → 1D or 2D
    - Thread ID → 1D, 2D, or 3D
  - Simplifies memory addressing when processing multidimensional data



Practice: Vector Addition

- **Copy Sample Skeleton Code**
  - cp –r /home/share/16_VectorAdd ./[FolderName]
  - cd [FolderName]
- **Notepad: VectorAdd.cu 코드 완성**
- **Compile & run program**
  - make
  - ./EXE

Example: Vector Addition

```
#include "cuda_runtime.h"
#include <stdio.h>
#define N 3

__global__ void addKernel( int *a, int *b, int *c ){
        //Declare thread ID
        //Addition command
}
int main( void ){
        int h_a[N], h_b[N], h_c[N];
        int *d_a, *d_b, *d_c;
        int size = N * sizeof(int);

        //Allocate device memory for data

        h_a[0] = 1;         h_a[1] = 2;         h_a[2] = 3;
        h_b[0] = 4;         h_b[1] = 5;         h_b[2] = 6;

        //Copy the host memory to device memory
        //Kernel call
        //Copy the device memory to host memory

        for( int i=0 ; i<N ; i++ ) {
                printf( "[%d] %d + %d = %d₩n" , i, h_a[i] , h_b[i] , h_c[i] );
        }

        //Unbind device memory
        return 0;
}
```

Following Lectures

- 1강: Basic
- 2강: Thread & Block
- 3강: Advanced Thread Coding
- 4강: Optimization with Memory Model
- 5강: Atomic Operation
- 6강: Streaming & Multi-GPU Coding

END