Programming Massively Parallel Processors

Chapter01 Introduction
&
Chapter02 Heterogeneous data parallel computing

1944023 JiYeong Yi

Chapter01 Introduction

Table of contents

1.1 Heterogeneous parallel computing

Single core CPU

- Energy consumption and heat dissipation issues limit the increase of the clock frequency and the productive activities

Multicore CPU

- Seek to maintain the execution speed of sequential programs while moving into multiple cores

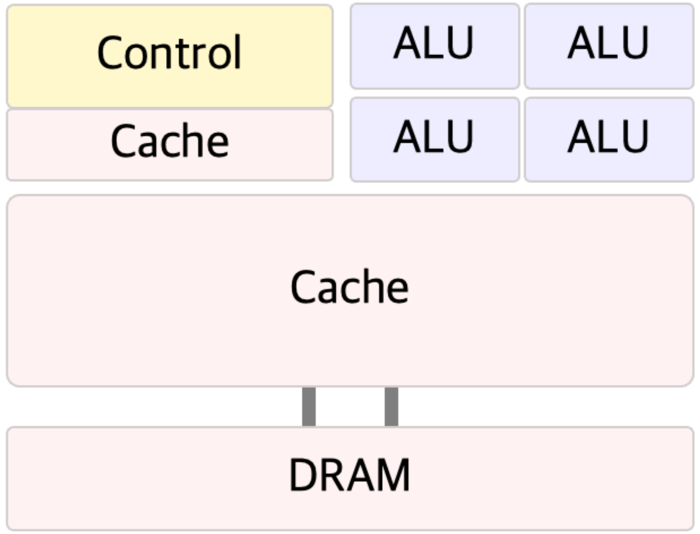
Many-thread GPU

- Focus more on the execution throughput of parallel application

1.1 Heterogeneous parallel computing

Core	• A core is a physical component of the microprocessor that can execute instruction
Thread	• A thread is a virtual sequence of instruction that can be executed by a core

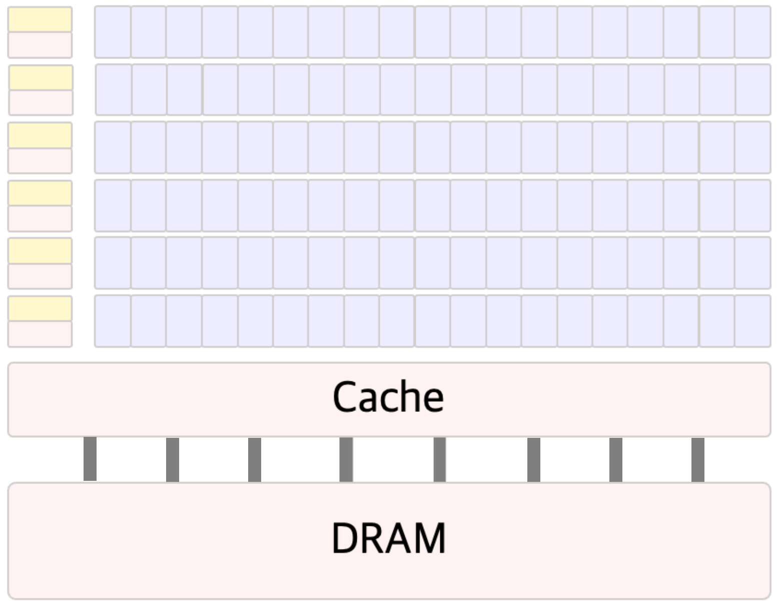
1.1 Heterogeneous parallel computing



CPU
(latency-oriented design)

- The design of CPU is optimized for sequential code performance
- A smaller number of larger arithmetic units and a smaller number of memory channels in CPU design
- The arithmetic units and operand data delivery logic are designed to minimize the effective latency of arithmetic operands at the cost of increased use of chop area and power per unit

1.1 Heterogeneous parallel computing



GPU
(throughput-oriented design)

- The design of GPU is optimized for **the execution throughput of massive numbers of threads** rather than reducing the latency of individual threads

- A larger number of smaller arithmetic units and a larger number of memory channels in GPU design

- The reduction in area and power of the memory access hardware and arithmetic units allows increasing the total execution throughput

1.1 Heterogeneous parallel computing

<Heterogeneous parallel computing system>

Multicore CPU + Many-thread GPU

- For programs that have one or very few threads, CPUs with lower operation latencies can achieve much higher performance
- The **sequential parts** on the CPU

- When a program has a large number of threads, GPUs with higher execution throughput can achieve much higher performance
- The **numerically intensive parts** on the GPUs

1.2 Why more speed or parallelism?

- The main motivation for massively parallel programming is for applications to enjoy continued speed increase in future hardware generations

Computational model

HDTV

User Interface

Electronic game

Deep learning

many applications require to high computation power and speed

↓

effective management of data delivery can have a major impact on the achievable speed of a parallel application

1.3 Speeding up real applications

Speedup

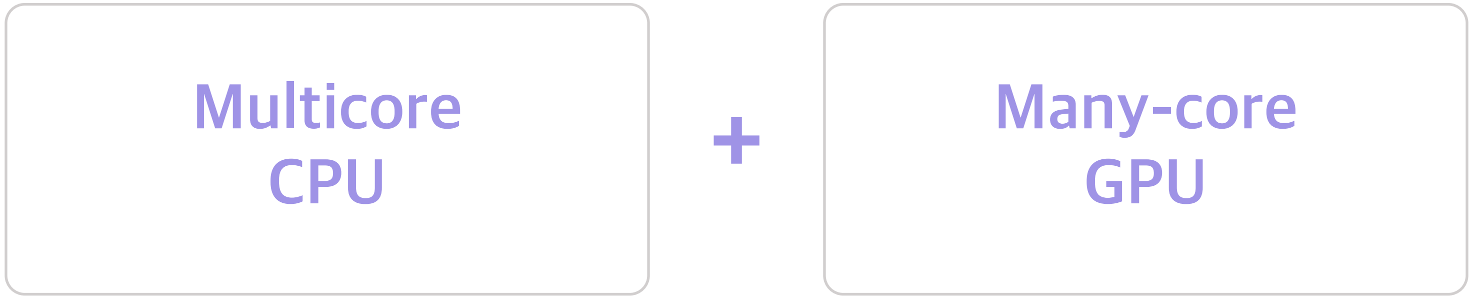
- The ratio of the time used to execute the application in system B over the time used to execute the same application in system A

- Important factor for the achievable level of speedup for application (parallel computing system over a serial computing system)

  1. The portion of the application that can be parallelized
  2. How fast data can be accessed from and written to the memory

1.3 Speeding up real applications

- In some applications, CPUs perform very well, making it harder to speed up performance using a GPU and most applications have portions that can be much better executed by the CPU



<Heterogeneous parallel computing system>

1.4 Challenges in parallel programming

- Challenges in achieving high performance

1. Challenging to design parallel algorithms with the same level of algorithmic (computational) complexity as that of sequential algorithms
2. The execution speed of many applications is limited by memory access latency and/or throughput
3. The execution speed of parallel programs is often more sensitive to the input data characteristics than is the case for their sequential counterparts
4. Synchronization operations such as barriers or atomic operations impose overhead on the application

1.5 Related parallel programming interfaces

OpenMP
(for shared memory multiprocessor)

- OpenMP provides compiler automation and runtime support for abstracting away many parallel programming details
- Programmer need to understand all the detailed parallel programming concepts

↓

CUDA

- CUDA gives programmers explicit control of parallel programming details

1.5 Related parallel programming interfaces

MPI
(for computing nodes in a cluster do not share memory)

- All data sharing and interaction must be done through explicit message passing
- The amount of effort that is needed to port an application into MPI can be quite high, owing to the lack of shared memory across computing nodes

↓

CUDA

- CUDA provides shared memory for parallel execution in the GPU

1.5 Related parallel programming interfaces

OpenCL
(standardized
programming model)

- (Similar to CUDA)
  It defines language extensions and runtime APIs to allow programmers to manage parallelism and data delivery

- (Comparison to CUDA)
  OpenCL relies more on APIs and less on language extensions

1.6 Overarching goals

01 how to program massively parallel processors to achieve high performance

02 how to parallel programming for correct functionality and reliability

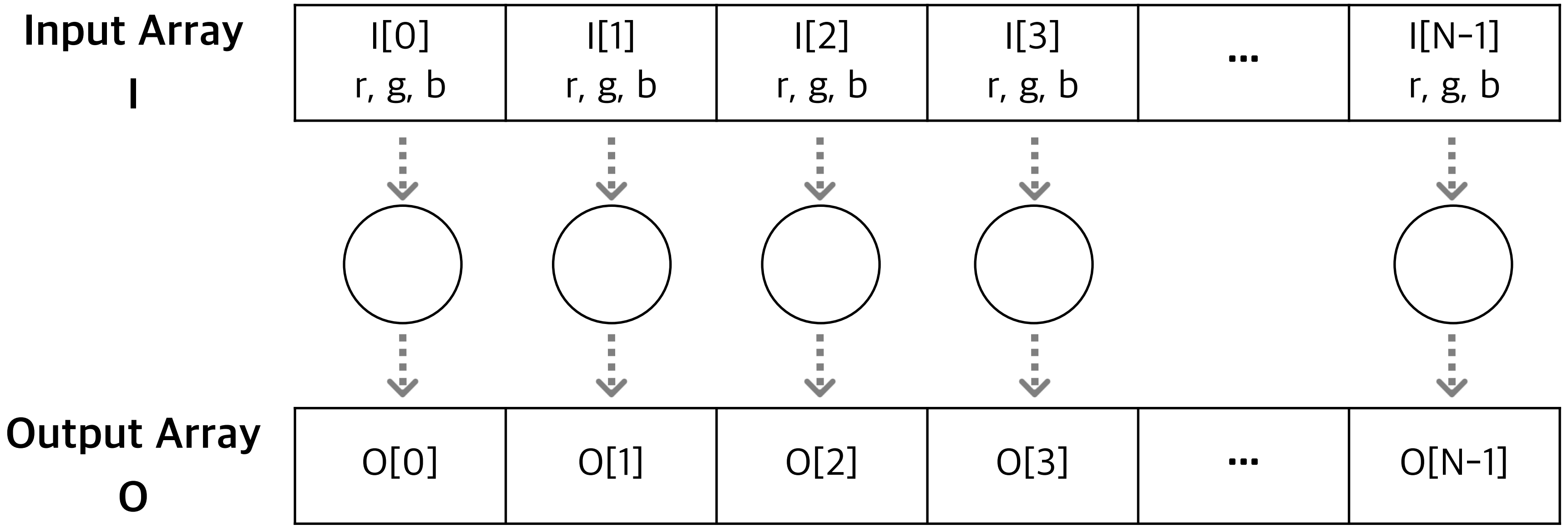
03 Scalability across future hardware generations by exploring approaches to parallel programming

Chapter02 Heterogeneous parallel computing

Table of contents

2.1 Data parallelism

- Data parallelism refers to the phenomenon in which the computation work to be performed on different parts of the dataset can be done **independently** of each other and thus can be done in parallel with each other



2.1 Data parallelism

Type of parallelism used in parallel programming

1. Task parallelism

- Different operations are performed on the same or different data
- Task parallelism is typically exposed through task decomposition of application

2. Data parallelism

- Same operations are performed on different subsets of same data
- With large datasets, one can often find abundant data parallelism to be able to utilize massively parallel processors and allow application performance to grow with each generation of hardware that has more execution resource

2.2 CUDA C program structure

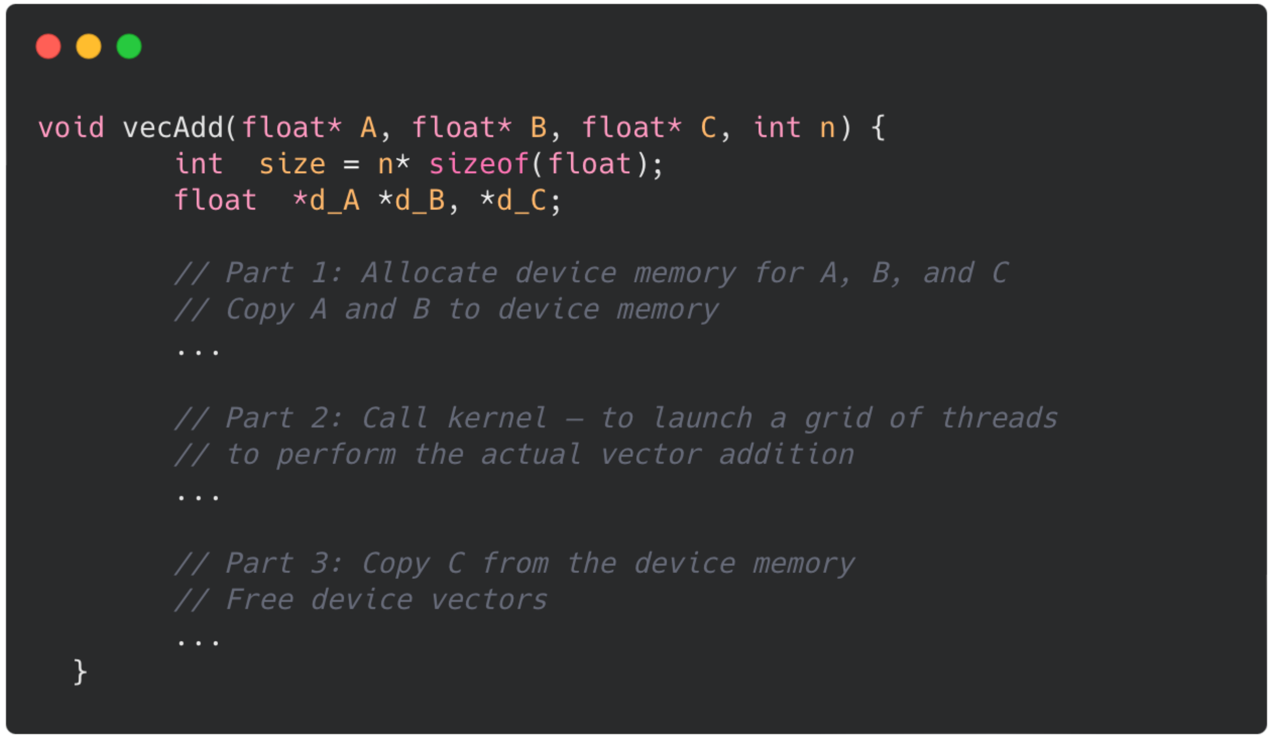
- The structure of a CUDA C program reflects the coexistence of a host (CPU) and one or more devices (GPUs) in the computer.

- **Execution of a CUDA program**
  - The Execution starts with host code (CPU serial code)
  - When Kernel function is called, a large number of threads (grid) are launched on a device to execute the kernel
  - When all threads of a grid have completed their execution, the grid terminates, and the execution continues on the host until another grid is launched

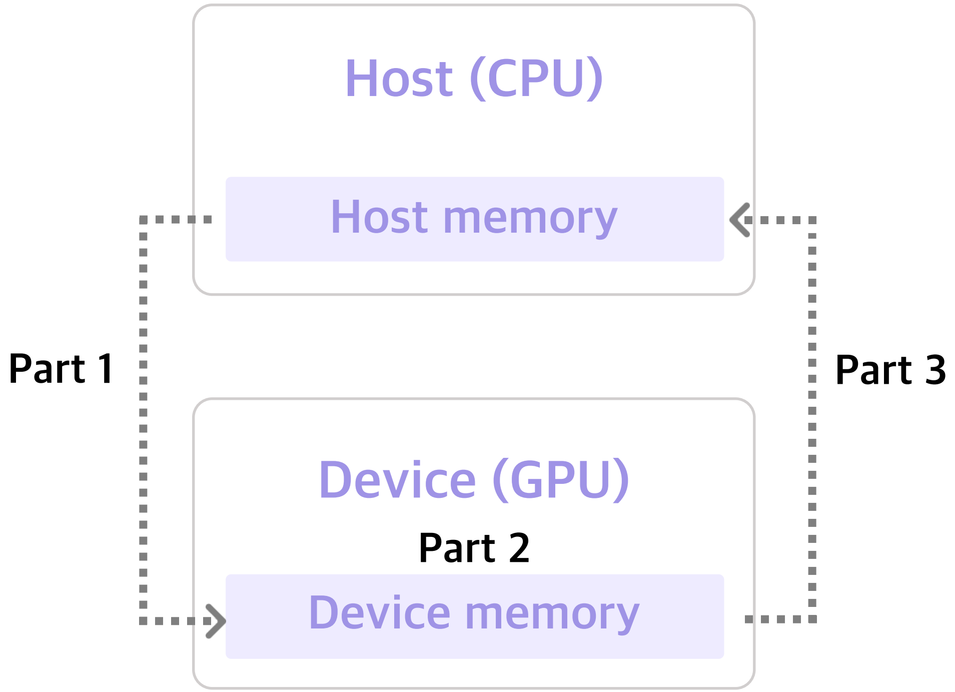
Thread

- Consist of the code of program and the values of its variables and data structure
- The execution of each thread is sequential

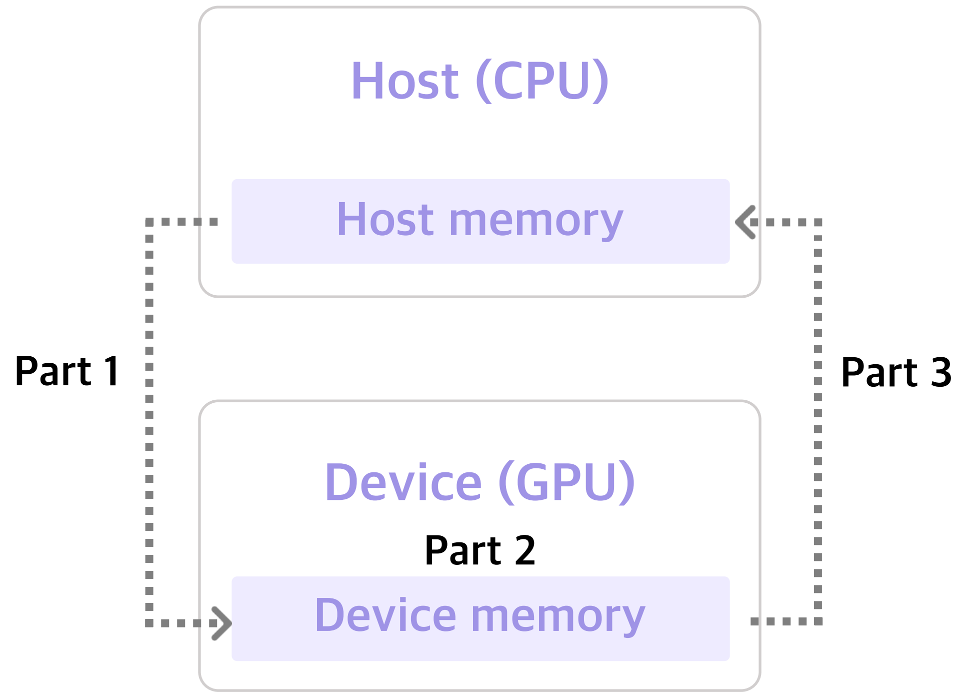
2.3 A vector addition kernel



```
void vecAdd(float* A, float* B, float* C, int n) {
        int  size = n* sizeof(float);
        float  *d_A *d_B, *d_C;

        // Part 1: Allocate device memory for A, B, and C
        // Copy A and B to device memory
        ...

        // Part 2: Call kernel – to launch a grid of threads
        // to perform the actual vector addition
        ...

        // Part 3: Copy C from the device memory
        // Free device vectors
        ...
   }
```



2.4 Device global memory and data transfer



Part 1

- Allocating device memory -> cudaMalloc()
- Copying data from Host to Device -> cudaMemcpy()

Part 3

- Copying data from Device to Host -> cudaMemcpy()
- Deallocating device memory -> cudaFree()

2.4 Device global memory and data transfer

cudaMalloc()

- Allocates object in the device global memory
- Two parameters
  - Address of a pointer to the allocated object
  - Size of allocated object in terms of bytes

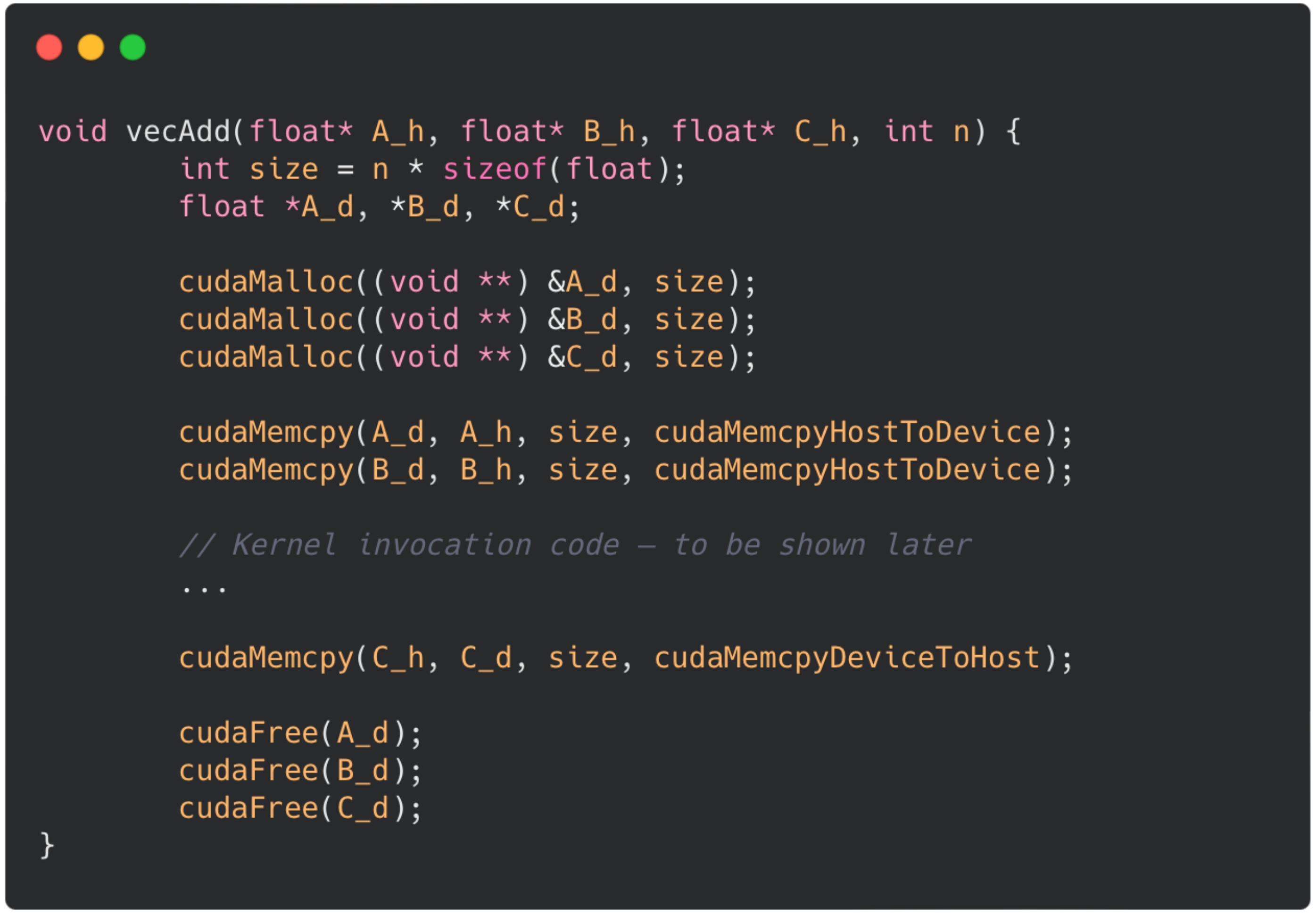
cudaFree()

- Frees object from device global memory
  - Pointer to freed object

cudaMemcpy()

- Memory data transfer
- Requires four parameters
  - Pointer to destination
  - Pointer to source
  - Number of bytes copied
  - Type/Direction of transfer

2.4 Device global memory and data transfer

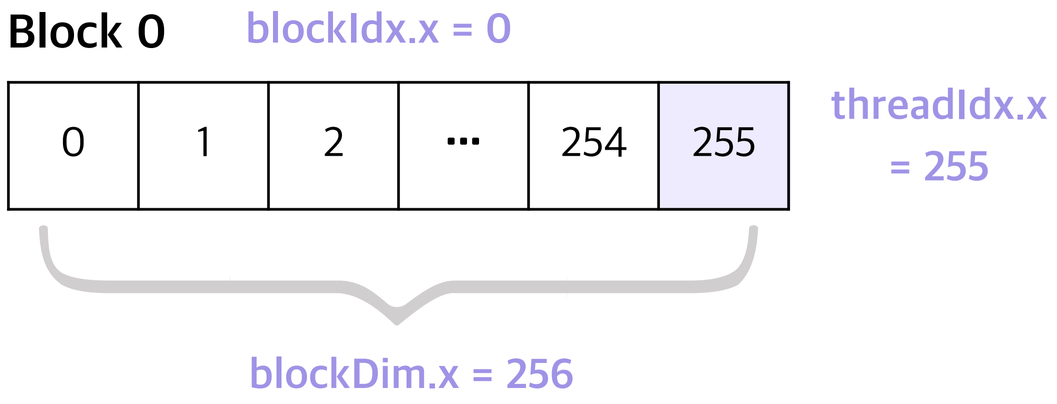


```
void vecAdd(float* A_h, float* B_h, float* C_h, int n) {
        int size = n * sizeof(float);
        float *A_d, *B_d, *C_d;

        cudaMalloc((void **) &A_d, size);
        cudaMalloc((void **) &B_d, size);
        cudaMalloc((void **) &C_d, size);

        cudaMemcpy(A_d, A_h, size, cudaMemcpyHostToDevice);
        cudaMemcpy(B_d, B_h, size, cudaMemcpyHostToDevice);

        // Kernel invocation code – to be shown later
        ...

        cudaMemcpy(C_h, C_d, size, cudaMemcpyDeviceToHost);

        cudaFree(A_d);
        cudaFree(B_d);
        cudaFree(C_d);
}
```

2.5 Kernel functions and threading

- CUDA Kernels have access to **three built-in variables** that allow threads to **distinguish** themselves from each other and to determine the area of data each thread is to work on

  - **blockDim**: the numer of threads in a block
  - **threadIdx**: giving each thread a unique coordinate within a block
  - **blockIdx**: giving all threads in a block a common block coordinate



- Unique global index i

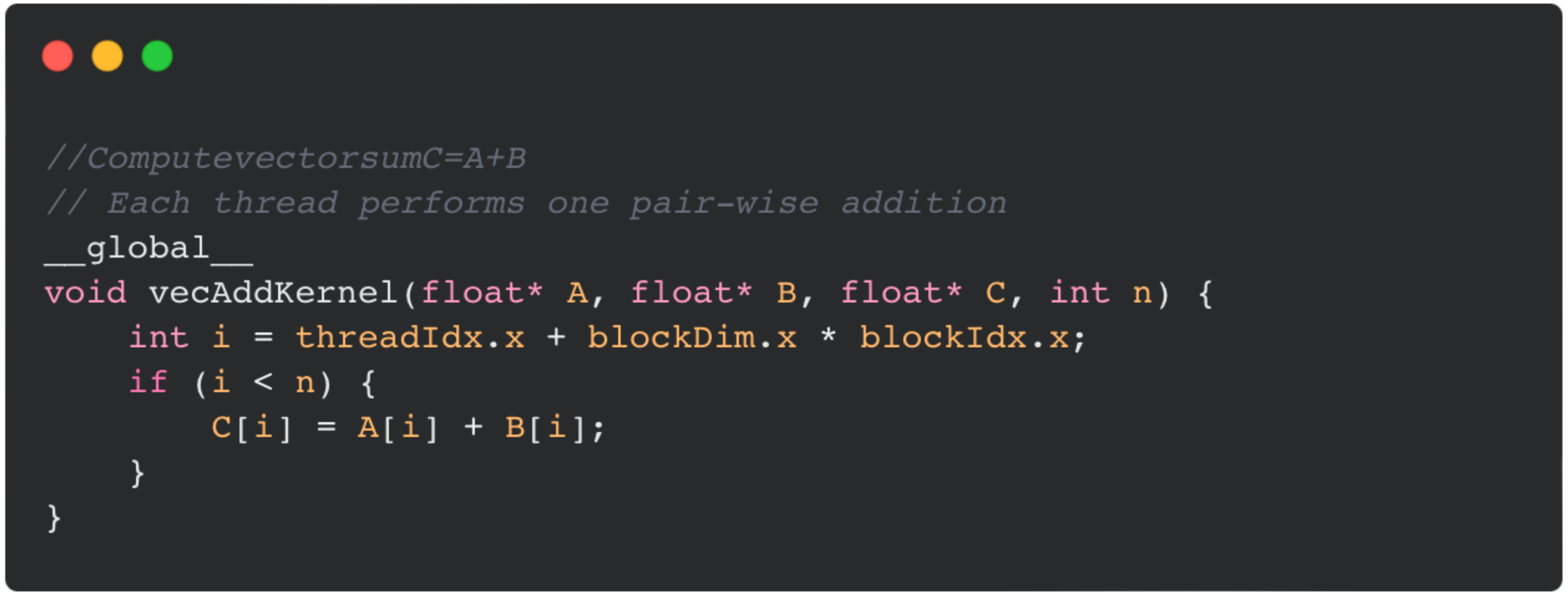
  i = blockIdx.x * blockDim.x + threadIdx.x

2.5 Kernel functions and threading

- In general, CUDA C extends the C language with three qualifier keywords that can be used in **function declaration**

	Qualifier Keyword	**Callable From**	**Executed On**	**Executed By**
Host Function	__host__ (default)	Host	Host	Caller host thread
Kernel Function	__global__	Host or Device	Device	New grid of device threads
Device Function	__device__	Device	Device	Caller device thread

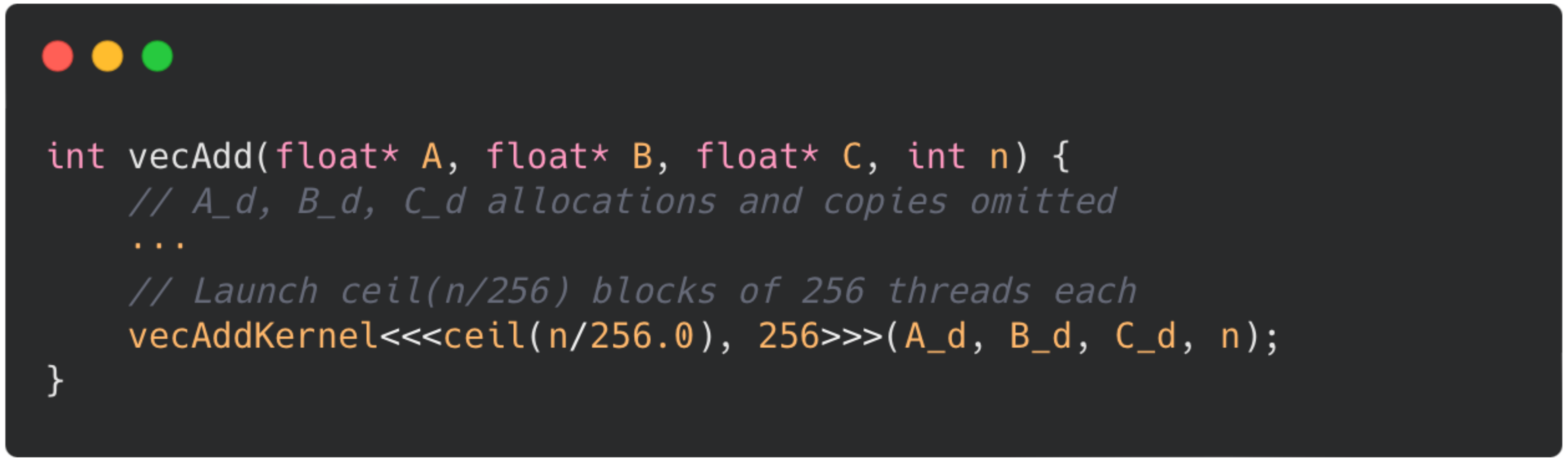
2.5 Kernel functions and threading



```
//ComputevectorsumC=A+B
// Each thread performs one pair-wise addition
__global__
void vecAddKernel(float* A, float* B, float* C, int n) {
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    if (i < n) {
        C[i] = A[i] + B[i];
    }
}
```

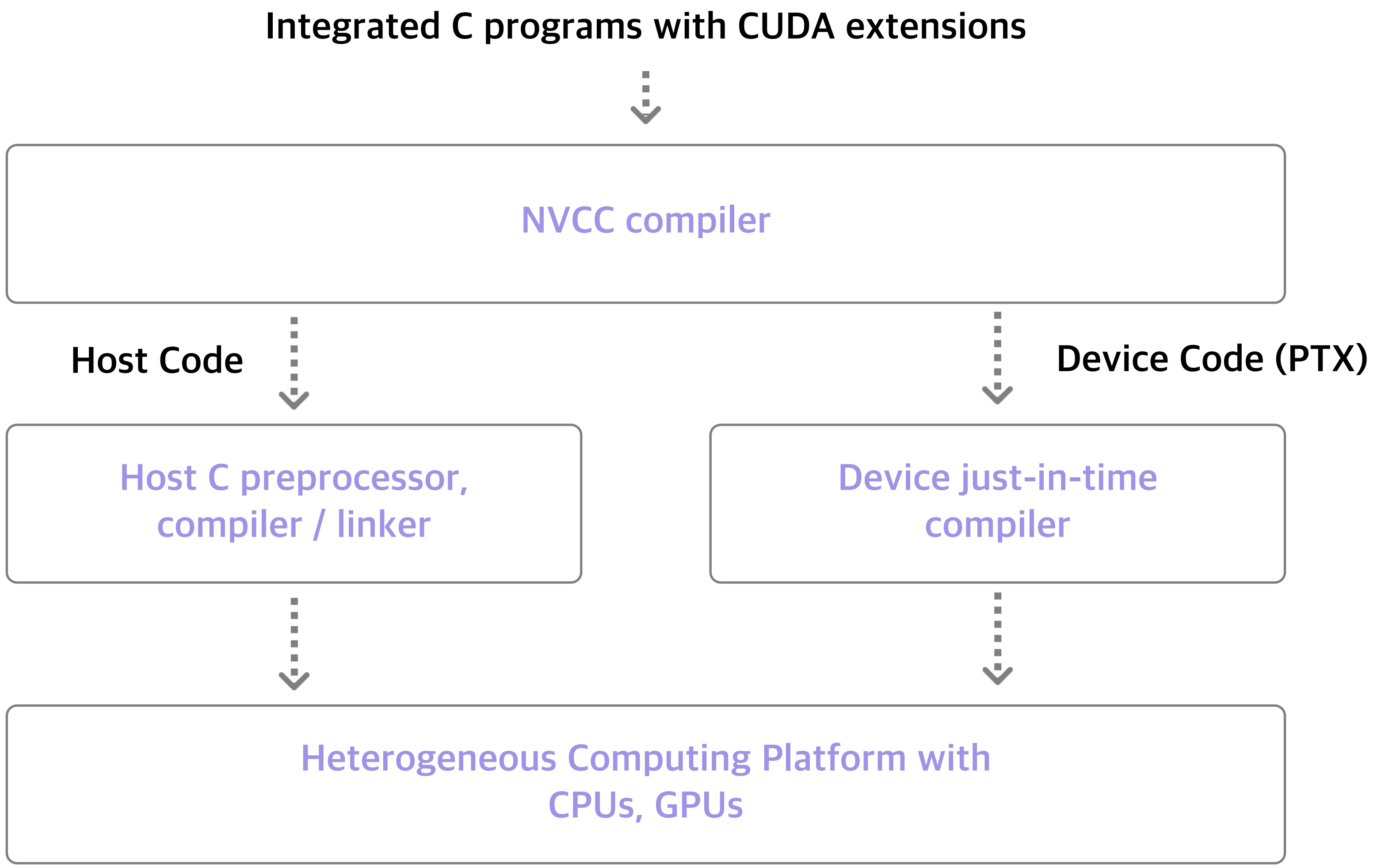
2.6 Calling Kernel functions

- When the host code calls a kernel, it sets the grid and thread block dimensions via execution configuration parameters
- <<<parameter1, parameter2>>>
  - Parameter1 : the number of blocks in the grid
  - Parameter2: the number of threads in each block

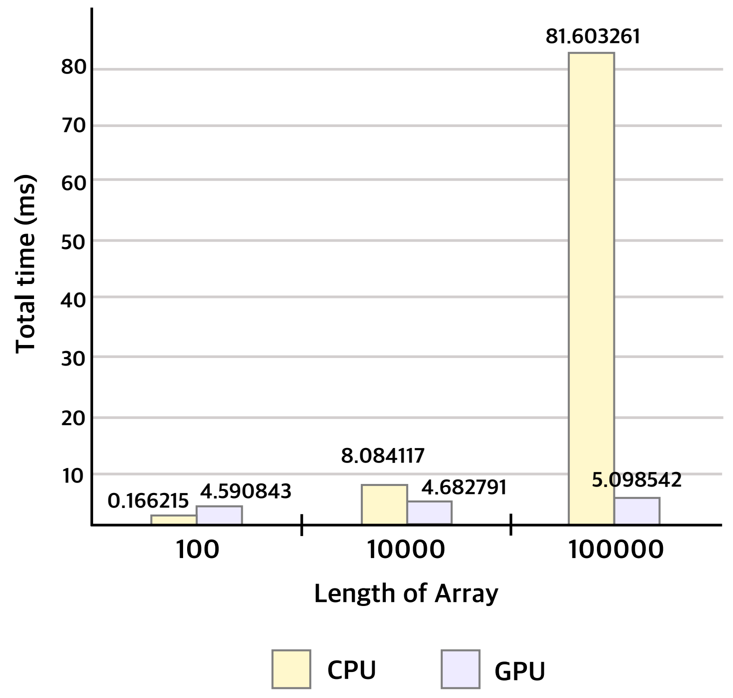


```
int vecAdd(float* A, float* B, float* C, int n) {
    // A_d, B_d, C_d allocations and copies omitted
    ...
    // Launch ceil(n/256) blocks of 256 threads each
    vecAddKernel<<<ceil(n/256.0), 256>>>(A_d, B_d, C_d, n);
}
```

2.7 Compilation


Integrated C programs with CUDA extensions
NVCC compiler
Host Code
Device Code (PTX)
Host C preprocessor, compiler / linker
Device just-in-time compiler
Heterogeneous Computing Platform with CPUs, GPUs

2.8 Summary



Case 1) Length of Array = 100

- Total time of CPU = 0.166215 (ms)
- Total time of GPU = 4.590843 (ms)

Case 2) Length of Array = 10000

- Total time of CPU = 8.084117 (ms)
- Total time of GPU = 4.682791 (ms)

Case 3) Length of Array = 100000

- Total time of CPU = 81.603261 (ms)
- Total time of GPU = 5.098542 (ms)