EBU7501: Cloud Computing
Week 2, Day 2: GPU Programming using CUDA

Dr. Gokop Goteng

Queen Mary
University of London

Lecture Aim and Outcome

- Aim
  - The aim of this lecture is to introduce students to graphics processing unit (GPU) programming using compute unified device architecture (CUDA) model
- Outcome
  - At the end of this lecture students should be able to:
    - Know the architecture of CUDA model
    - Write simple “Hello World” CUDA program in C language
    - Know the differences between standard C program compilation and CUDA C program compilation

Outline

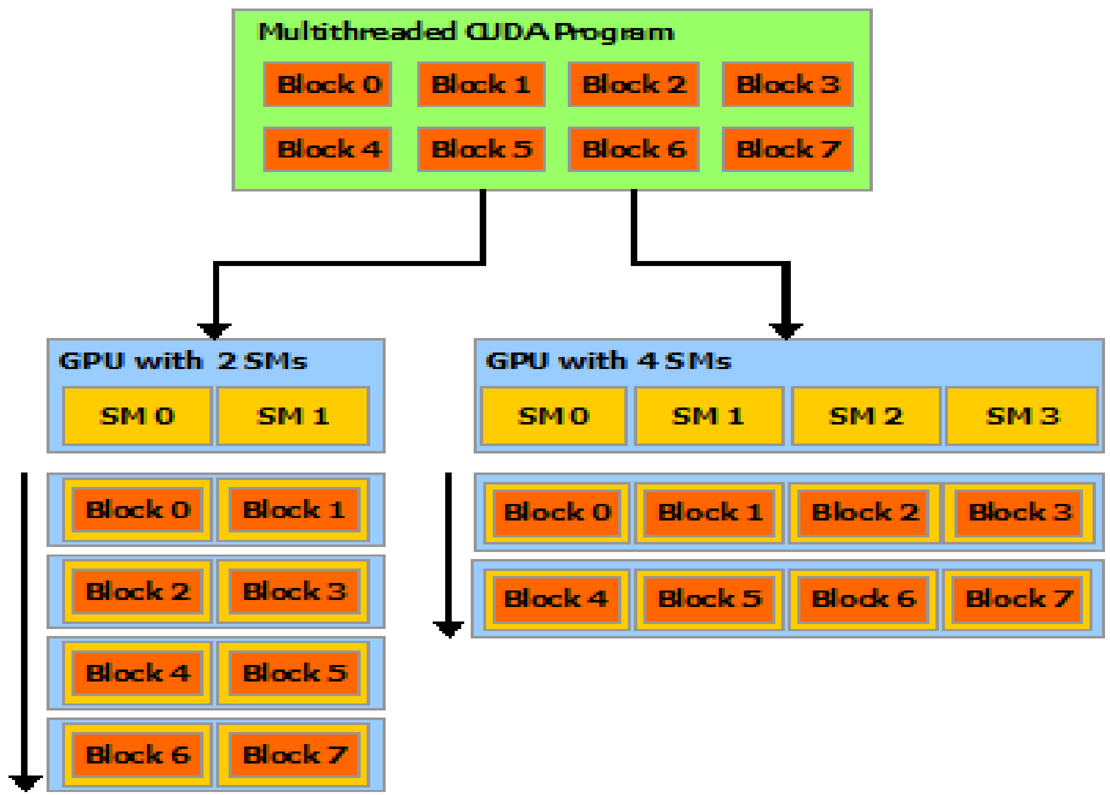
- CUDA
- CUDA Model
- CUDA Scalability
- Automatic Scalability in CUDA and GPU
- CUDA Programming Structure
- Thread Hierarchy
- Memory Hierarchy
- Hello World! In Standard C Program
- Hello World! With CUDA Code
- Class Task

CUDA

- CUDA stands for Compute Unified Device Architecture
- NVIDIA introduced CUDA in November 2006 as a model to programme its GPU-enabled devices
- CUDA is a general-purpose parallel programming platform and programming model
- The CUDA model uses the parallel compute engine in NVIDIA's GPUs to solve computationally and data intensive problems in a much faster and efficient way than CPUs
- CUDA model has a software development environment that allows programmers to use C/C++ high level languages
  - Other high level languages APIs are also supported by CUDA
    - FORTRAN, OpenACC and DirectiveCompute APIs are supported by CUDA

Automatic Scalability in CUDA and GPU

- A GPU is implemented around an array of Streaming Multiprocessors (SMs)
- A multithreaded program is partitioned into blocks of threads that execute independently from each other
  - This allows a GPU with more multiprocessors to automatically execute the program in less time than a GPU with fewer multiprocessors



CUDA Programming Structure

- Kernel
  - CUDA C extends C by allowing the programmer to define C functions
  - These C functions are called *kernels*
  - When these kernels are called, they are executed N times in parallel by N different *CUDA threads*
    - This is as opposed to only once like in regular C functions
  - A kernel is defined in programming with the following declaration specifier:

    `__global__`

CUDA Programming Structure

- Kernel
  - The number of CUDA threads that will execute the kernel for a given kernel call is defined using the following execution configuration syntax

    <<< . . . >>>

  - To invoke a kernel, you call the kernel as:

    kernel_function_name<<<number_of_blocks, number_of_threads>>>(param1, param2, …, paramn);

  - Each thread that executes the kernel is given a unique *thread ID* that is accessible within the kernel through the built-in "threadIdx" variable

CUDA Programming Structure

- Kernel
  - Example of Kernel definition

```
#define N 2
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C) {
	int i = threadIdx.x;
	C[i] = A[i] + B[i];
}

int main() {
	 ...
	// Kernel invocation with N threads
	VecAdd<<<1, N>>>(A, B, C);
	...
}
```

CUDA Programming Structure

- The code above adds two vectors *A* and *B* of size *N* and stores the result into vector *C*
- Each of the *N* threads that execute VecAdd() performs one pair-wise addition

Hello World! With Host Code Only

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

- Standard C that runs on the host
- NVIDIA compiler (nvcc) can be used to compile programs with no *device* code
- Compiling and getting output:

```
$ nvcc hello_world.cu
```

Output:

```
$ ./a.out
Hello World!
$
```

Hello World! with Device Code

```
__global__ void mykernel(void) {
}

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

- Two new syntactic elements…

Hello World! with Device Code

```
__global__ void mykernel(void) {
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
  - Runs on the device
  - Is called from host code

- `nvcc` separates source code into host and device components
  - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
  - Host functions (e.g. `main()`) processed by standard host compiler
    - **`gcc`, `cl.exe`**

Hello World! with Device COde

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
  - Also called a "kernel launch"
  - We'll return to the parameters (1,1) in a moment
- That's all that is required to execute a function on the GPU!

Hello World! with Device Code

```
__global__ void mykernel(void){
}

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

Compile as:

```
$ nvcc hello.cu
$ ./a.out
```

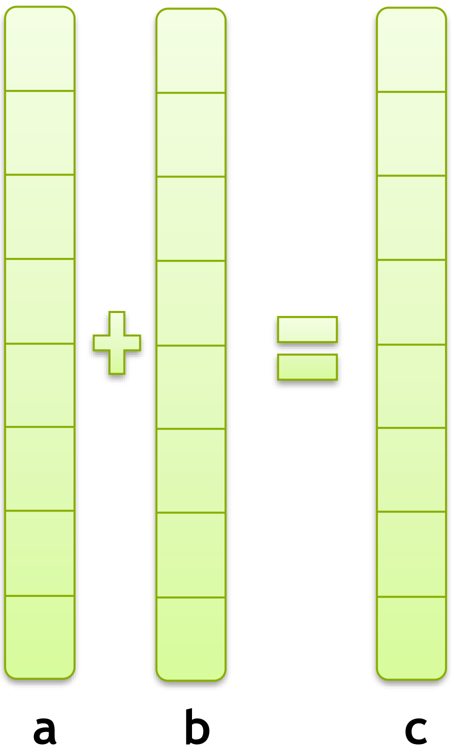
Output:

```
Hello World!
$
```

Queen Mary University of London

Parallel Programming in CUDA C/C++

- This program is too simple to employ the use of GPU
- GPU computing is about massive parallelism
- We need a more interesting example that justifies using GPU as they are expensive
- We'll start by adding two integers and build up to vector addition



Queen Mary
University of London

Addition on the Device

- A simple kernel to add two integers

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

- As before `__global__` is a CUDA C/C++ keyword meaning
  - `add()` will execute on the device
  - `add()` will be called from the host

Addition on the Device

- Note that we use pointers for the variables

```
__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 device memory

- We need to allocate memory on the GPU

Thread Hierarchy

- The "threadIdx" variable is a 3 component vector
- This enables threads to be identified as one-dimensional, 2-dimensional or 3-dimensional thread index
- This again enables them to form one-dimensional, two-dimensional or three-dimensional thread block.
- The index of a thread and its thread ID relate to each other in almost similar ways to most data structures

Thread Hierarchy

- Example of code that adds two matrices *A* and *B* of size *NxN* and stores the result into matrix *C*

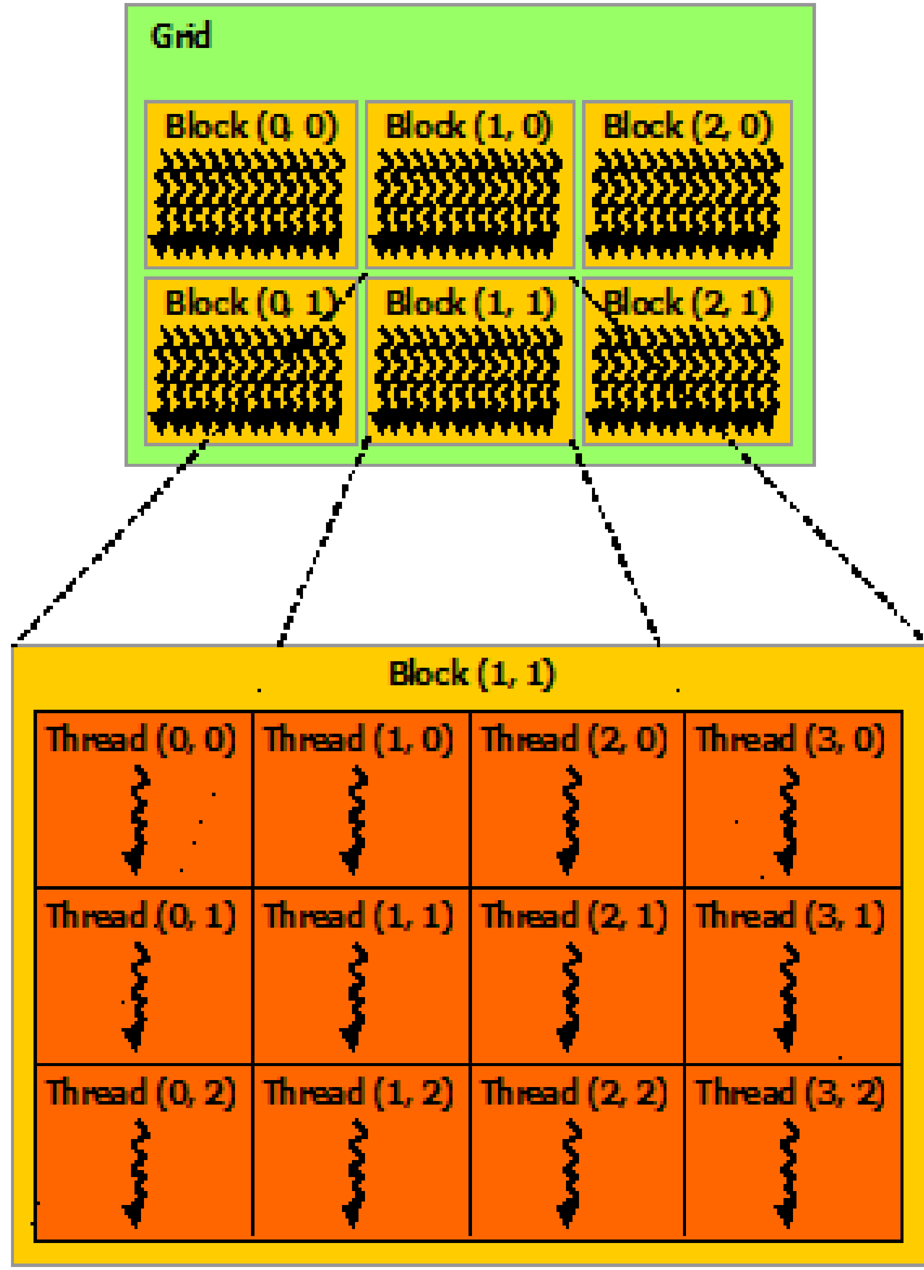
```
#define N 10
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- There is a maximum of threads per each block
  - This is because all threads in a block reside on the same processor core and share the limited memory resources in that core
  - Modern GPUs can have up to 1024 threads per thread block
- One kernel can be executed by multiple equally-shaped thread blocks
  - The total number of threads is equal to the number of threads per block times the number of blocks
- Blocks can be represented as one-dimensional, two-dimensional or three-dimensional *“Grid”* of thread blocks
- The number of thread blocks in a grid depends on the size of the data being processed or number of processors

Thread Hierarchy

- Thread blocks in a "Grid"
  - The values in <<<…>>> are number of threads per block and number of block per grid
  - The data types that in <<<…>>> can only be "int" or "dim3"
  - This diagram shows 2-dimensional blocks or grids
  - Each block in the grid can be identified by a 1-dimensional, 2-dimensional or 3-dimensional index
  - This index can be accessed within the kernel using the built-in "blockIdx" variable
  - The dimension of the thread block is accessible within the kernel through the "blockDim" variable



Thread Hierarchy

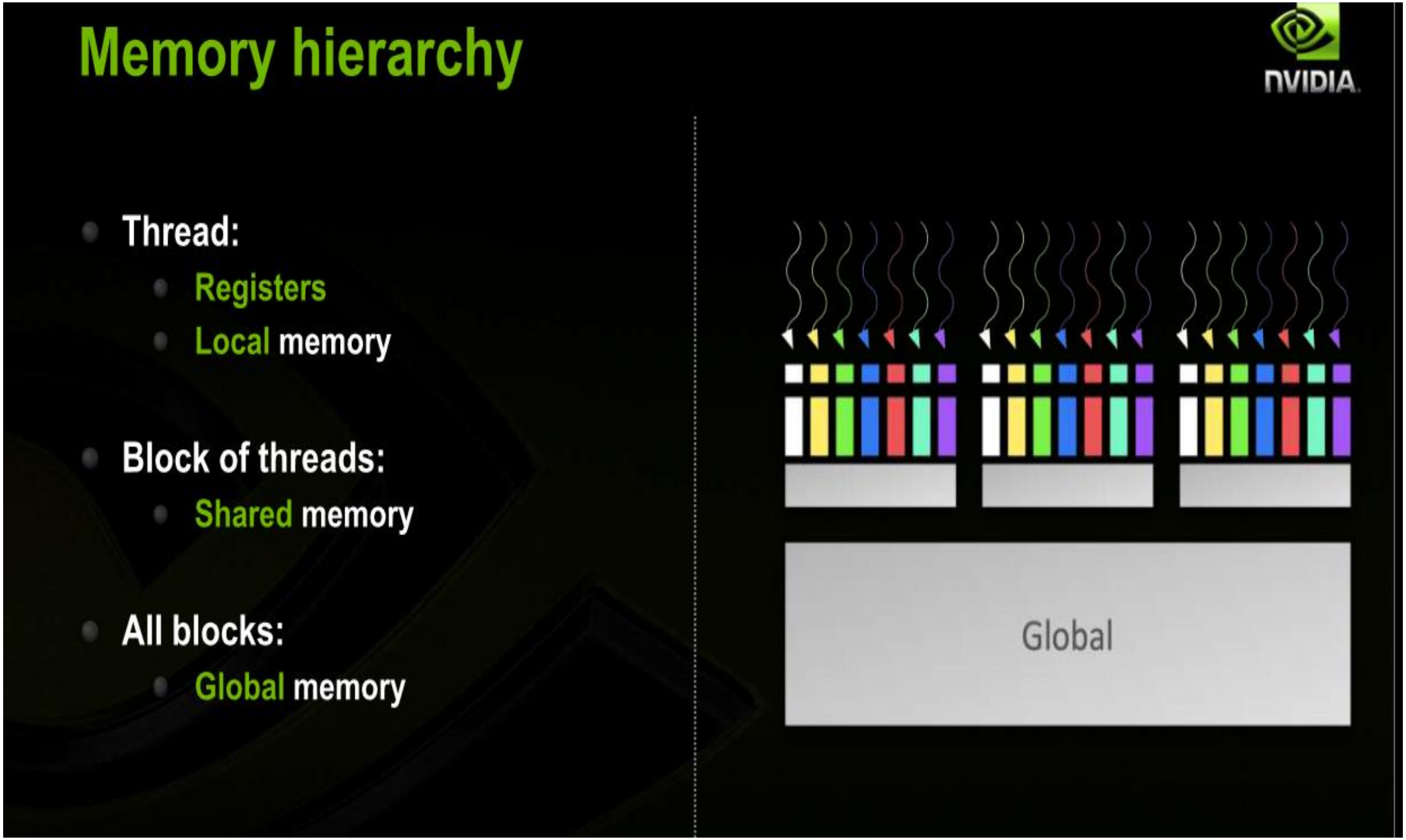
- Multi-block example code:

```
#define N 16
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N]) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i < N && j < N)
        C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    ...
}
```

Thread Hierarchy

- An arbitrary thread block size of 16x16 (256 threads) is used to describe how to use multi-block CUDA programming

Memory Hierarchy



Source: NVIDIA

Hello World! In Standard C Program

- The NVIDIA compiler to compile CUDA C codes is known as NVIDIA C Compile (nvcc)
  - "nvcc" is the command line tool to compile the code
- NVCC tool can be used to compile standard C programs just as using the "cc" or "gcc" command line tool to compile a C/C++ program without the "device" (GPU)
- Example of a file helloworld.cu:

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

Hello World! In Standard C Program

- Now compile the code using the command

nvcc helloworld.cu

- As usual the executable named “a.out” will be created. Run the executable

a.out

- The output will be:

Hello World!

Hello World! With CUDA Code

```
__global__ void myKernel(void) {
}

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

Hello World! With CUDA Code

- The CUDA C/C++ __global__ indicates a function (kernel) that
  - Runs on a device (GPU)
  - It is called from a host code
- nvcc tool separates source code into host and device components
  - The device functions, which in this example is the myKernel() and is processed by the NVIDIA compiler
  - The host functions, which in this case is the main() which is processed by standard host compiler such as gcc, cc, cl.exe, etc

Hello World! With CUDA Code

- The myKernel<<<1,1>>>();
  - The triple brackets delimiters (<<< and >>>) mark the call from host code to device code
  - This is also known as the “kernel launch”
  - The first parameter within the triple delimiters is the number of blocks
  - The second parameter is the number of threads in each block
  - Parameters can only be scalars (int) or multidimensional (dim3)
- That is all that you need to know to execute a simple program in a GPU! Simple?

Hello World! With CUDA Code

- Now compile the CUDA code as we did for standard C code
- Name the file helloworlddevice.cu

```
__global__ void myKernel(void) {
}

int main(void) {
    myKernel<<<1,1>>>();
    printf(“Hello Workd!\n”);
    return 0;
}
```

Hello World! With CUDA Code

- Now compile the code using the command

nvcc helloworlddevice.cu

- As usual the executable named “a.out” will be created. Run the executable

a.out

- The output will be:

Hello World!

- Output is same as the previous standard C code

Class Task

- What is the limitation of the CUDA C Hello World program?
- Describe other hierarchies in CUDA programming