Chapter 4

GPU Computing and Accelerators: Part II

Compute Unified Device Architecture (CUDA)

What is CUDA?

CUDA is two things at the same time:

1. **platform model**
   for the hardware implementation of general purpose graphics processing units made by the NVIDIA® Corporation.

2. **programming model**
   realizing the software implementation and scheduling of tasks of the parallel programs on the above hardware.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (thread)

A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (thread)

A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (thread)

A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.

Definition (warp)

The CUDA hardware consists of streaming multi-processors that are executing several threads simultaneously. GPU-threads are therefore grouped in so called *warps* of threads.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (thread)

A *thread*, or more precisely *GPU-thread* is the smallest unit of data and instructions to be executed in a parallel CUDA program.

In contrast to CPU-threads a task switch between GPU-threads is usually almost for free due to the special CUDA architecture.

Definition (warp)

The CUDA hardware consists of streaming multi-processors that are executing several threads simultaneously. GPU-threads are therefore grouped in so called *warps* of threads.

The number of threads in a warp may depend on the hardware. They are mostly 32 threads per warp which in turn is the smallest number of tasks executed in SIMD style.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (block)

A *block* is larger group of threads that can contain 64-512 threads.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (block)

A *block* is larger group of threads that can contain 64-512 threads.

Ideally it contains a multiple of 32 threads so it can best be split into warps by the CUDA environment for scheduling.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (block)

A *block* is larger group of threads that can contain 64-512 threads.

Ideally it contains a multiple of 32 threads so it can best be split into warps by the CUDA environment for scheduling.

Definition (grid)

The actual work to be performed by a program or algorithm is distributed to one or two dimensional *grid* of blocks.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (block)

A *block* is larger group of threads that can contain 64-512 threads.

Ideally it contains a multiple of 32 threads so it can best be split into warps by the CUDA environment for scheduling.

Definition (grid)

The actual work to be performed by a program or algorithm is distributed to one or two dimensional *grid* of blocks.

The grid represents the largest freedom in design that the developer has.

Compute Unified Device Architecture (CUDA)

Basic Definitions

The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

Compute Unified Device Architecture (CUDA)

Basic Definitions

The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

In case multiple GPUs are installed on a single host system with multiple CPUs, each GPU is connected to a single CPU representing a single NUMA node of the host system.

Compute Unified Device Architecture (CUDA)

Basic Definitions

The central notions to understand data management in a CUDA program are those of *host* and *device*. Here *host* refers to the computer that hosts the GPU. Especially the CPU and memory of the host are relevant. The *device* then is the GPU installed on the host system.

In case multiple GPUs are installed on a single host system with multiple CPUs, each GPU is connected to a single CPU representing a single NUMA node of the host system.

The host CPU controls the execution of the program. However host and device may execute their tasks asynchronously. When not specified differently data transfers between them serve as implicit synchronization points.

Compute Unified Device Architecture (CUDA)

Basic Definitions

Definition (kernel)

The *kernel* is the core element of a CUDA parallel program. It represents the function that specifies the work a certain thread in a block on a grid has to execute.

We will see in the course of this Chapter how we the kernel knows what it has to do.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension

We will next introduce the most basic elements of the CUDA C language extension. These consist of two very important things.

1. **qualifiers** that apply to functions and specify where the function should be executed,
2. **launch size specifiers** that control the grid and block sizes that are used to run a kernel.

An extensive API, defining C-style functions and data types to be used in CUDA programs, together with a handful of libraries for several kinds of tasks (e.g., a BLAS implementation) complete the picture.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension

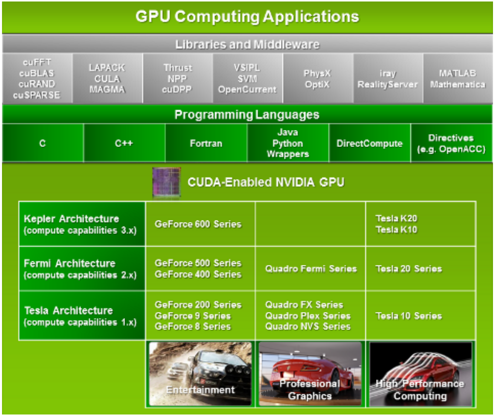

Figure: The CUDA GPU computing applications framework (taken from CUDA C programming guide)

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- `__host__` Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- `__host__` Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- `__shared__`applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- `__host__` Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- `__shared__` applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor
- `__constant__` applies to a variable specifying the residence in the constant memory.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Qualifiers

- `__global__` This qualifier applies to a function and is used to indicate that it in fact represents a kernel.
- `__device__` The qualifier that specifies functions that should be run on the device, but are not kernels. It can be useful for subtasks called in a kernel. It also applies to variables determining them to reside on the device.
- `__host__` Being basically redundant this qualifier can be used to explicitly state that a function is to be executed on the host. It is therefore optional.
- `__shared__`applies to a variable declaring that it should reside in the shared memory of a streaming multiprocessor
- `__constant__`applies to a variable specifying the residence in the constant memory.

Note that `__global__` and `__device__` functions are not allowed to be recursive.

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Launch size specifiers

The basic launch size specification for a kernel takes the form

```
<<< grid , block size >>>
```

where `grid` specifies the block distribution and `block size` indicates the number of threads per block in the grid.

Example

- `<<<1,1>>>` launches 1 block with 1 thread
- `<<<N,1>>>` launches N blocks with 1 thread each
- `<<<1,N>>>` launches 1 block with N threads
- `<<<N,M>>>` launches a 1d grid of N block running M threads each

Compute Unified Device Architecture (CUDA)

Most Basic Syntax of the CUDA C Extension: Launch size specifiers

Both the arguments can be two dimensional distributions. CUDA defines special tuple hiding types for these declarations. Using

```
dim3 grid(3,2)
dim3 threads(16,16)
```

one defines a 3×2 grid of blocks for running 256 threads arranged in a 16×16 local grid. These are then used in the launch specification as

```
<<< grid, threads>>>
```

Launch size specifications are simply appended to the kernel function name upon calling it.

Compute Unified Device Architecture (CUDA)

Intodoctory Examples

The following examples are taken from the “CUDA by Example” book.

Example

```
#include "../common/book.h"

__global__ void kernel( void ) {
}

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

Compute Unified Device Architecture (CUDA)

Intodunctory Examples

Example

```
#include "../common/book.h"

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

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );

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

    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );

    return 0;
}
```

Compute Unified Device Architecture (CUDA)

Intodoctory Examples

Example

```
#include "../common/book.h"

__device__ int addem( int a, int b ) {
    return a + b;
}

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

int main( void ) {
    int c;
    int *dev_c;
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, sizeof(int) ) );
    add<<<1,1>>>( 2, 7, dev_c );
    HANDLE_ERROR( cudaMemcpy( &c, dev_c, sizeof(int),
                              cudaMemcpyDeviceToHost ) );
    printf( "2 + 7 = %d\n", c );
    HANDLE_ERROR( cudaFree( dev_c ) );
    return 0;
}
```

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Before we can rush of and compile the previous examples, we need to check a few prerequisites:

- NVIDIA® device drivers and hardware,
- NVIDIA® CUDA toolkit installation,
- compiler for the host code.

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Basic information on CUDA in general can be found at
`http://www.nvidia.com/cuda`.

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Basic information on CUDA in general can be found at `http://www.nvidia.com/cuda`. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at `https://developer.nvidia.com/cuda-toolkit`.

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Basic information on CUDA in general can be found at `http://www.nvidia.com/cuda`. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at `https://developer.nvidia.com/cuda-toolkit`.

As for the hardware, basically every NVIDIA® GPU released after the appearance of the GeForce 8800 GTX in 2006 is CUDA enabled. However, one needs to make sure that the OS version, the device driver and CUDA Toolkit version are fitting. Working combinations should be available in the toolkits documentation.

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Basic information on CUDA in general can be found at `http://www.nvidia.com/cuda`. The Toolkit and all the information on the included accelerated libraries and developer tools can be found at `https://developer.nvidia.com/cuda-toolkit`.

As for the hardware, basically every NVIDIA® GPU released after the appearance of the GeForce 8800 GTX in 2006 is CUDA enabled. However, one needs to make sure that the OS version, the device driver and CUDA Toolkit version are fitting. Working combinations should be available in the toolkits documentation.

Regarding the compilers NVIDIA® recommends the following

- **Microsoft Windows** Visual Studio
- **Linux** Gnu Compiler Collection (GCC)
- **MacOS** GCC as well via Apple's Xcode

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

We will in the following restrict ourselves to the Linux world again.

Consider our basic "Hello World!" example is stored in a text file called `hello_world.cu`. Using the `nvcc` compiler provided in the CUDA Toolkit we can compile it by

```
nvcc hello_world.cu
```

Since on Linux `nvcc` uses `gcc` to compile the host code this will also generate a binary called `a.out`. As for `gcc` we can specify the output filename, i.e. name of the resulting executable via

```
nvcc hello_world.cu -o hello_world
```

The file extension `.cu` is used to indicate that we have a C file with CUDA C extensions.

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Among the further compiler options we meet many old friends:

- `-c` for generating object files of single `.c` or `.cu` files
- `-g` for generating debug information in the host code
- `-pg` the same for profiling information
- `-O` for specifying the optimization level for the host code
- `-m` specify 32 vs 64bit host architecture

Compute Unified Device Architecture (CUDA)

Compiling CUDA Programs

Among the further compiler options we meet many old friends:

- `-c` for generating object files of single `.c` or `.cu` files
- `-g` for generating debug information in the host code
- `-pg` the same for profiling information
- `-O` for specifying the optimization level for the host code
- `-m` specify 32 vs 64bit host architecture

And we have a few more for the device code, e.g.

- `-G` generates debug information for the device code
- `-arch` specifies the GPU architecture to be assumed, i.e. the compute capabilities of the device (e.g. `-arch=sm_20`)