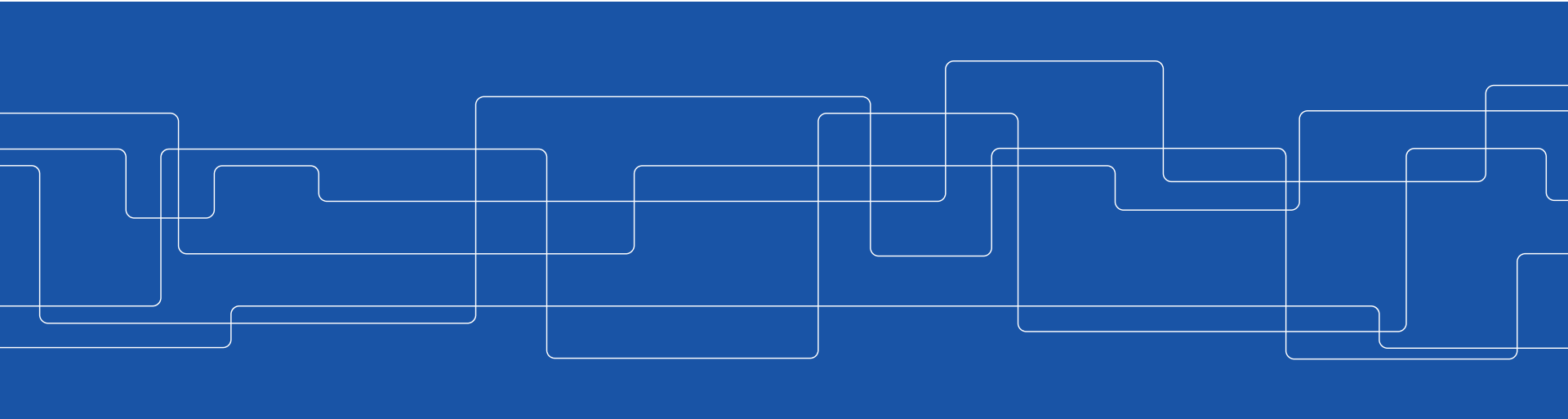
CUDA – Essentials

Stefano Markidis

KTH Royal Institute of Technology



CUDA

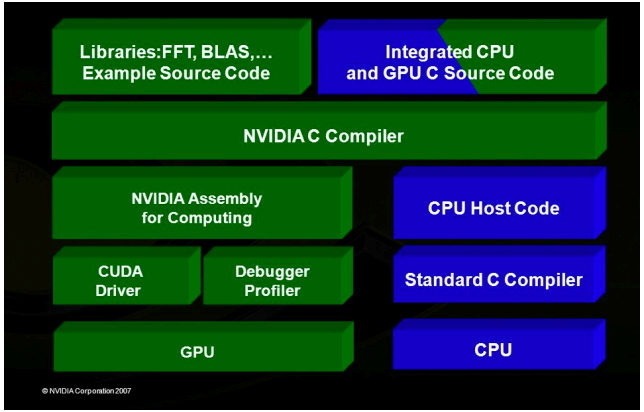
CUDA (Compute Unified Device Architecture) is **NVIDIA**'s program development environment:

- based on **C/C++** with some extensions
  - FORTRAN support provided by compiler from PGI (Something about this later in the lab)
- **Indexing math** and **synchronization** are the main conceptual difficulties

CUDA Components

Installing CUDA on a system, there are **3 components**:

1. **Driver** low-level software that controls the graphics card
2. **Toolkit**
   - **`nvcc` CUDA compiler**
   - Nsight IDE plugin for Eclipse or Visual Studio
   - profiling and debugging tools
   - several libraries for math
3. **SDK**
   - lots of demonstration examples
   - some error-checking utilities



CUDA Programming

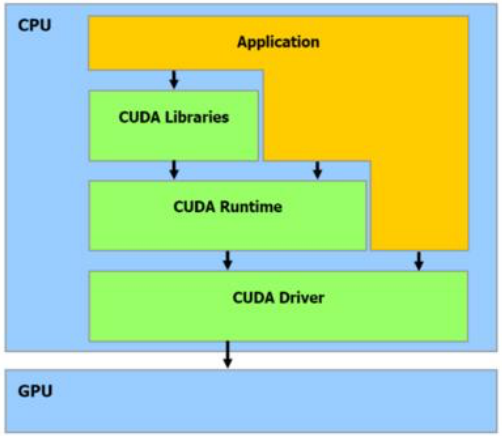
CUDA terminology:

- **host** = CPU and its memory
- **device** = GPU and its memory

At the host level, there is a choice of 2 APIs:

- **runtime** simpler, more convenient
- **driver** much more verbose, more flexible (e.g. allows run-time compilation), closer to OpenCL

We will only use the **runtime API**



CUDA Parallelism Model

CUDA employs the **Single Instruction Multiple Thread** (**SIMT**) model of parallelization.

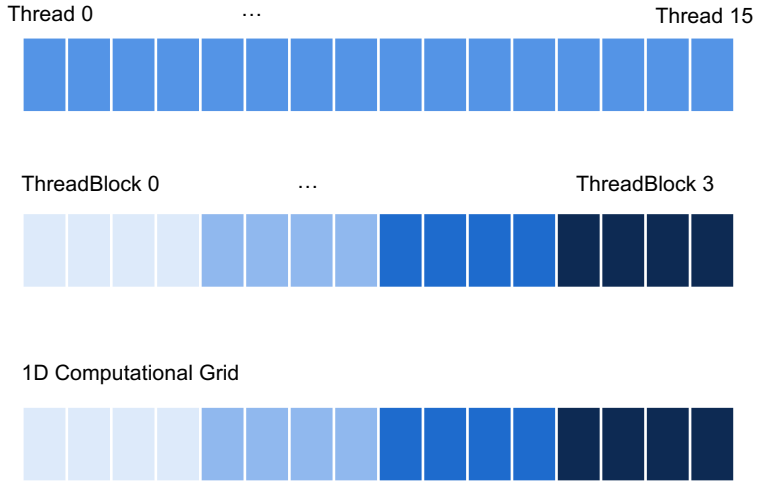
- Each thread executes the **same code** but operates different data (**Data parallelism**)

Parallelization with CUDA

As in OpenMP, we parallelize with **threads** but now organized into a **three-level hierarchy**:

- **Thread**
- **Threadblock** (might be 1D, 2D or 3D)
- Threadblock **Grid** (might be 1D, 2D or 3D)

Three-levels hierarchy (1D)



The Whole CUDA in one Sentence!

Launching a kernel on the GPU from the CPU to create a computational **grid** composed of **threadblocks**

Launch a Kernel in CUDA

Kernel is a kind of **special function executed on the GPU**
Kernel **launch** ≅ regular function call with addition of number of threads

```
aKernel<<<BPG, TPB>>>(arg1, arg2, …)
```

To specify a kernel launch, we start with kernel name (`aKernel`) and end with argument list between `()`

Now for the CUDA extension: we specify the dimensional of the computational grid, the **grid dimensions** and **block dimension** between triple angle brackets (`<<<BPG, TPB>>>`).

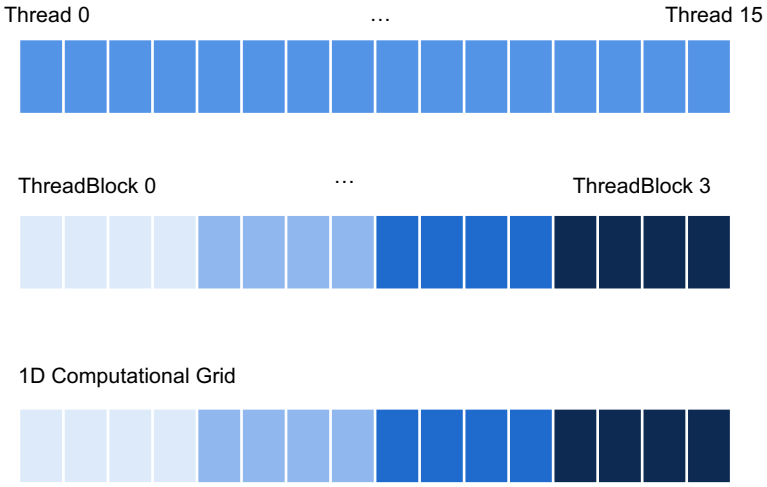
Execution Configuration

`BPG` = number of blocks in the grid

`TPG` =number of threads in the block

Together they constitute the **execution configuration** and specify the **dimensions of the kernel launch**

What it is BPG and TPB in this case?



If we operate on a vector of length `N`, we set `TPB` to a number that is some multiple 32 and `BPG = N/TPB`.

Question: What is the total number of threads?

How to declare a function called by host but executed on device?

CUDA makes this distinction by prepending one of the following function type qualifiers:

- `__global__` is the **qualifier for kernels** (which can be **called by the host and executed on device**)
- `__host__` functions called from the host and executed on the host (default qualifier, often omitted)
- `__device__` functions are called from **the device and execute on the device** (a function that is called from a kernel needs the `__device__` qualifier)

Question: which qualifier do you have before the function you call **from the GPU** and you want to run **on GPU**:

- __global__
- __host__
- __device__

?

Question: which qualifier do you have before the function you call **from the CPU** and you want to run on **GPU**:

- __global__
- __host__
- __device__

?

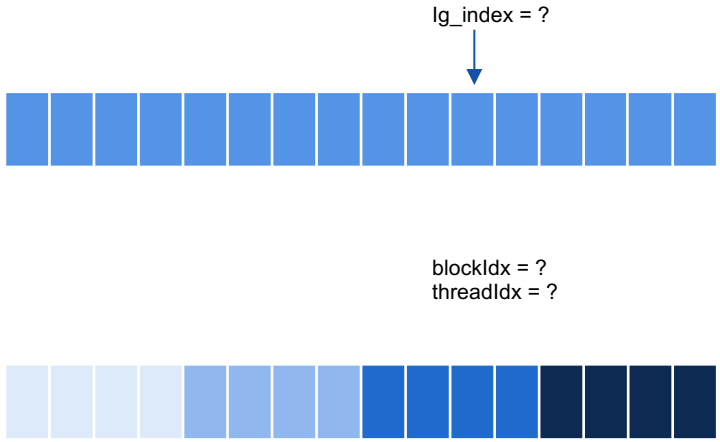
Built-in Variables

CUDA provides build-in dimension and index variables when in the kernel

- **Dimension variables**
  - `gridDim` = number of blocks in the grid
  - `blockDim` = number of threads in each block
- **Index variables**
  - `blockIdx` = index of the block in the grid
  - `threadIdx` = index of the thread within the block

Question: How do I calculate my global thread ID (1D grid)?

Using `threadIdx`, `blockIdx`, and what do I need also?



Why Kernels are special functions?

- Kernels execute on the GPU and **do not**, *in general*, **have access to data stored on the host side**
- **Kernels cannot return a value**, so the return type is always void, and kernel declarations starts as

```
__global__ void aKernel(arg1, arg2, …)
```

  - **How do I get the results from my kernel ??**

Transferring Data from/to Device

The CUDA runtime API provides these functions for transferring input data to the device and transferring results back to the host:

- `cudaMalloc()` allocates device memory
- `cudaMemcpy()` transfers data to or from a device
  - `cudaMemcpy(void* dest, void* src, size_t size,cudaMemcpyHostToDevice)` **host mem → GPU mem**
  - `cudaMemcpy(void* dest, void* src, size_t size, cudaMemcpyDeviceToHost)` **GPU mem → host mem**
- `cudaFree()` frees device memory that is no longer in use

Data Transfers are Synchronous

By default, **data transfers are synchronous (the function does not return until the data transfer is complete)**, so `cudaMemcpy()` finishes execution before the GPU can move to other operations.

Kernel Launching is Asynchronous

As soon as the kernel is launched, **the CPU returns from the call of kernel without waiting for the completion of the kernel**.

In practice, the CPU launches the kernel and right away executes what is after the kernel launch without waiting for the kernel to finish

Asynchronicity might create problems …

Example: a code that launches a kernel (=GPU) to print to screen and then ends.

In such situation, **after starting the GPU threads, control returns to the application and the application exits**.

At application exit, it's ability to send output to the standard output is terminated by the OS → the output generated by the kernel has nowhere to go!

Today Lab Problem!

Thread Synchronization

Kernels enable multiple computations in parallel but **they don't ensure order of execution** (asynchronous). CUDA provides functions to synchronize :

- **`cudaDeviceSynchronize()`** effectively synchronizes **all threads** in a grid → waits for all the threads in the kernel to complete before proceed.
- `__synchThreads()` synchronizes **threads within a block**

Question: how can we solve the problem of `printf` ?

CUDA Vector types

Vector types CUDA extends the standard C data types of length up to 4.

```
float4 f = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
```

Individual components are accessed with the **suffixes `.x`, `.y`, `.z`, and `.w`.** Accessing components beyond those declared for the vector type is an error.

```
float3 pos;
pos.z = 1.0f; // is legal
pos.w = 1.0f; // is illegal
```

Data Types for Index Variables?

CUDA uses the vector type `uint3` for the index variables, `blockIdx` and `threadIdx`.

A `uint3` variable is a vector with three unsigned integer components.

Question: How do I get component of `threadIdx` in a 1D grid in the `x` direction?

CUDA dim3 type for Dimension Variables

The `dim3` type is equivalent to `uint3` with unspecified entries set to 1.

CUDA uses the vector type `dim3` for the dimension variables, `gridDim` and `blockDim`. **We will use `dim3` variables for specifying execution configuration.**

Let’s write now our first CUDA program