



CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



CUDA: Introduction and API

Ben Cumming, CSCS
February 18, 2016



CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Introduction

The plan

- learn about the GPU memory model
- implement parallel CUDA kernels for simple linear algebra
- learn how to scale our parallel kernels to utilize all resources on the GPU
- understand which types of workloads can best take advantage of GPU resources
- learn about thread cooperation and synchronization in CUDA
- learn about concurrent task-based parallelism with CUDA

Prerequisites for the course

- no GPU or graphics experience required
- I assume C++ knowledge
 - I will be using C++11 (the bits that make C++ easier!)
 - there is no native CUDA implementation for Fortran
 - there is a CUDA Fortran provided by PGI, however it is not widely used.
 - Fortran users are encouraged to work with a C++ user for the practical exercises
- the generic GPU programming concepts in the CUDA part will be useful for people interested in OpenACC

Source code and slides are online:

github.com/eth-cscs/cineca-cuda

CUDA language is a superset of C++

- write CPU code using C++ (C++11 since CUDA 6.5)
- keywords for writing tasks to be executed by GPU threads (kernels)
- use special syntax for launching tasks/kernels on GPU

CUDA is GPU-specific

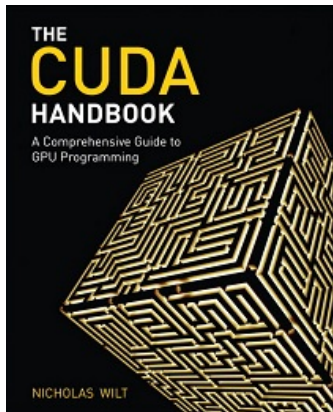
- the CUDA language extensions define the **programming model**
- features map directly to hardware (e.g. shared memory, thread blocks)

CUDA toolkit is more than just a language

- runtime library for managing GPU resources
- tools for profiling and debugging

What about the GPU in my laptop/desktop/cluster?

- the GPUs in Piz Daint are NVIDIA Tesla K20X devices
- Tesla devices are high-end products with features required for high-performance computing
 - high double precision performance (1.2 TFlops)
 - large DRAM (6 GB)
 - ECC memory
- the K20X Tesla cards use the Kepler architecture
 - some features are not supported by older cards
- I focus on features of the K20X devices for this course



recommended reading

CUDA Handbook: A Comprehensive Guide to GPU Programming

- Nicholas Wilt
- released in 2013
- detailed coverage of everything you need to know
- lots of example codes and micro-benchmarks



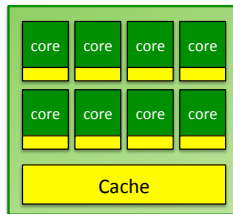
CSCS

Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

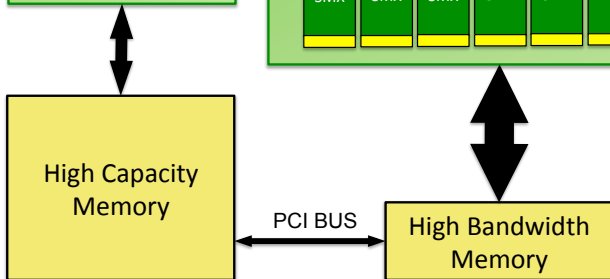
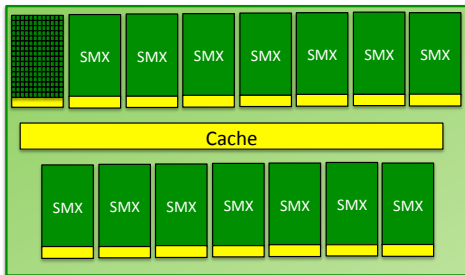
ETH zürich

Working with GPU memory

x86 CPU



K20X GPU



Host and device have separate memory spaces

- data must be copied between host and device memory via PCI
- data must be in device memory for kernels to access
 - not strictly true...
 - but a strict requirement for high performance in vast majority of cases
- ensure data is in the right memory space **before** computation starts
- on Piz Daint the respective bandwidths are:

host ↔ device	6 GB/s	PCIe gen2
host memory	35 GB/s	DDR3
device memory	180 GB/s	GDDR5

CUDA uses C pointers to reference GPU memory

```
double *data = //pass an address to either host or device memory
```

- a pointer can hold an address in **either** device **or** host memory
 - accessing a device pointer in host code, or vice versa, is **undefined behaviour**
 - we have to take care that we know which memory space a pointer is addressing
- The CUDA runtime library provides functions that can be used to allocate, free and copy device memory

Allocating device memory

```
cudaMalloc(void **ptr, size_t size)
```

- `size` number of bytes to allocate
- `ptr` points to allocated memory on exit

Freeing device memory

```
cudaFree(void *ptr)
```

Allocate memory for 100 doubles on device

```
double *v; // C pointer that will point to device memory
auto size_in_bytes = 100*sizeof(double);
cudaMalloc(&v, size_in_bytes); // allocate memory
cudaFree(v);                  // free memory
```

Perform blocking copy (host waits for copy to finish)

```
cudaMemcpy(void *dst, void *src, size_t size, cudaMemcpyKind kind)
```

- `dst` destination pointer
- `src` source pointer
- `size` number of **bytes** to copy to `dst`
- `kind` enumerated type specifying **direction** of copy:
one of `cudaMemcpyHostToDevice`, `cudaMemcpyDeviceToHost`,
`cudaMemcpyDeviceToDevice`, `cudaMemcpyHostToHost`

Copy 100 doubles to device, then back to host

```
auto size = 100*sizeof(double); // size in bytes
double *v_d;
cudaMalloc(&v_d, size);           // allocate on device
double *v_h = (double*)malloc(size); // allocate on host
cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
cudaMemcpy(v_h, v_d, size, cudaMemcpyDeviceToHost);
```

Errors happen...

all API functions return error codes that indicate either:

- success
- an error in the API call
- an error in an earlier asynchronous call

the return value is the enum type `cudaError_t`

- e.g.

```
cudaError_t status = cudaMalloc(&v, 100);
```

 - status is { `cudaSuccess`, `cudaErrorMemoryAllocation` }

Handling errors

```
const char* cudaGetErrorString(status)
```

- returns a string describing status

```
cudaError_t cudaGetLastError()
```

- returns the last error
- resets status to `cudaSuccess`

Copy 100 doubles to device **with error checking**

```
double *v_d;
auto size = sizeof(double)*100;
double *v_host = (double*)malloc(size);
cudaError_t status;

status = cudaMalloc(&v_d, size);
if(status != cudaSuccess) {
    printf("cuda error : %s\n", cudaGetErrorString(status));
    exit(1);
}

status = cudaMemcpy(v_d, v_h, size, cudaMemcpyHostToDevice);
if(status != cudaSuccess) {
    printf("cuda error : %s\n", cudaGetErrorString(status));
    exit(1);
}
```

It is essential to test for errors

But it is tedious and obfuscates our source code if it is done in line for every API and kernel call...

Exercise: Getting Ready

1. to use CUDA we need to set up the environment
 - CUDA uses the gnu compiler to compile the host code
 - A more recent version of GCC (4.8+) required for C++11 support
 - load the cudatoolkit module
2. on EURORA:

```
# get code via git
git clone https://github.com/eth-cscs/cineca-cuda.git

# or via wget
wget https://github.com/eth-cscs/cineca-cuda/archive/
    master.zip
unzip master.zip

# set up environment
module load gnu cuda
```


Exercise: API Basics

Open `cuda/exercises/axpy/util.h`

1. what does `cuda_check_status()` do?
2. look at the template wrappers `malloc_host` & `malloc_device`
 - what do they do?
 - what are the benefits over using `cudaMalloc` and `malloc` directly?
 - do we need corresponding functions for `cudaFree` and `free`?
3. write a wrapper around `cudaMemcpy` for copying data `host→device` & `device→host`
 - see `copy_to_device` & `copy_to_host`
 - remember to check for errors
4. compile the test and run
 - it will pass with no errors on success

```
make axpy_cublas.cuda  
aprun ./axpy_cublas.cuda 8
```