Getting started with CUDA
Part 1 - CUDA overview

Edwin Carlinet, Joseph Chazalon {firstname.lastname@lrde.epita.fr}

April 2020

EPITA Research & Development Laboratory (LRDE)

CUDA overview

What is CUDA?

A product

- It enables to use NVidia GPUs for computation

A C/C++ variant

- Mostly C++14-compatible, with extensions
- and also some restrictions!

A SDK

- A set of compilers and toolchains for various architectures
- Performance analysis tools

A runtime

- An assembly specification
- Computation libraries (linear algebra, etc.)

A new industry standard

- Used by every major deep learning framework
- Replacing OpenCL as Vulkan is replacing OpenGL

The CUDA ecosystem

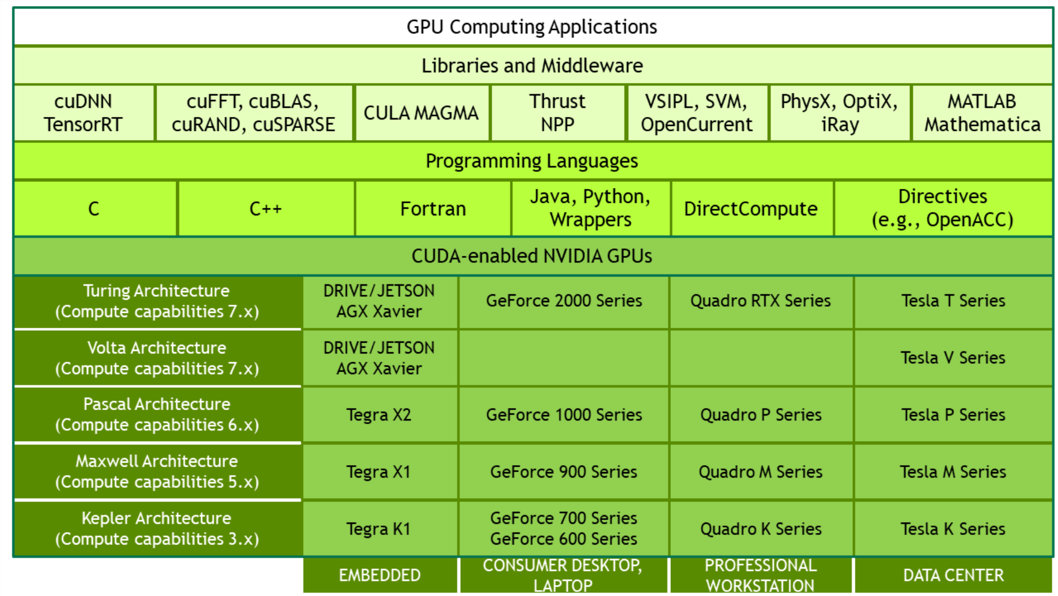

Figure 1: The CUDA ecosystem

Libraries *or* Compiler Directives *or* Programming Language?

CUDA is mostly based on a "new" **programming language**: CUDA C (or C++, or Fortran).
This grants much flexibility and performance

But is also exposes much of GPU goodness through **libraries**.

And it supports a few **compiler directives** to facilitate some constructs.

```
#pragma unroll
for(int i = 0; i < WORK_PER_THREAD; ++i)
  // Some thread work
```

The big idea: Kernels instead of loops

Without CUDA (vector addition)

```
// compute vector sum C = A + B
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  for (int i = 0; i < n; ++i)
    h_C[i] = h_A[i] + h_B[i];
}

int main()
{
  // Allocation for A, B and C
  // I/O to read n elements of A and B
  vecAdd(h_A, h_B, h_C);
}
```

With CUDA (1/2): move work to the separate compute device

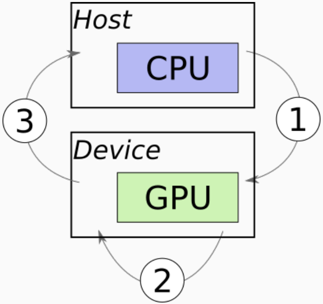

Figure 2: Computation on separate device

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size = n * sizeof(float);
  float *d_A, *d_B, *d_C;

  // 1.1 Allocate device memory for A, B and C
  // 1.2 Copy A and B to device memory

  // 2. Launch kernel code - computation done on device

  // 3. Copy C (result) from device memory
  // Free device vectors
}

int main() { /* Unchanged */ }
```

With CUDA (2/2): Kernel sample code

```
// kernel
__global__ void kvecAdd(float *d_A, float *d_B, float *d_C, int n)
{
  int i = blockDim.x * blockID.x + threadId.x;
  if (i >= n) return;
  d_C[i] = d_A[i] + d_B[i];
}
```

No more `for` loop!

Arrays of parallel threads

A CUDA kernel is executed by a **grid** (array) of threads

- All threads in a grid run the same kernel code (Single Program Multiple Data)
- Each thread has indexes that is uses to compute memory addresses and make control decisions

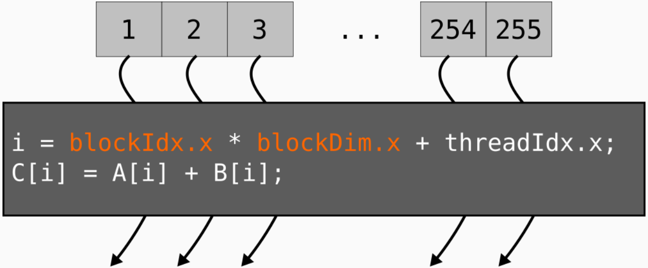

Figure 3: A thread block

Thread blocks

Threads are grouped into thread blocks

- Threads within a block cooperate via
  - **shared memory**
  - **atomic operations**
  - **barrier synchronization**
- Threads in different blocks do not interact[1]

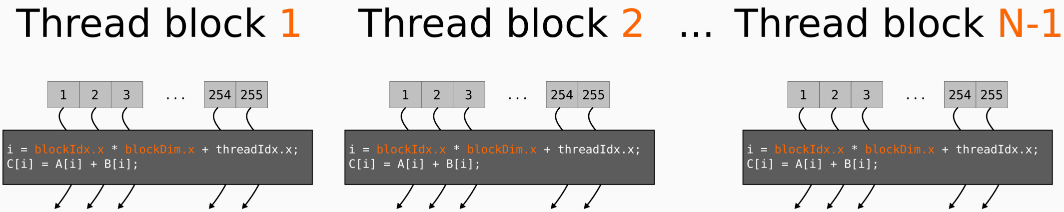

Figure 4: Independent thread blocks

[1] Not in this course, though there are techniques for that.

A multidimensional grid of computation threads (1/2)

Each thread uses indices to decide what data to work on:

- `blockIdx` (0 → `gridDim`): 1D, 2D or 3D
- `threadIdx` (0 → `blockDim`): 1D, 2D or 3D

Each index has `x`, `y` and `z` attributes to get the actual index in each dimension.

```
int i = threadIdx.x;
int j = threadIdx.y;
int k = threadIdx.z;
```

Simplifies memory addressing when processing multidimensional data:

- *image processing*
- *solving PDE on volumes*
- ...

A multidimensional grid of computation threads (2/2)

Grid and blocks can have different dimensions,
but they usually are two levels of the same work decomposition.

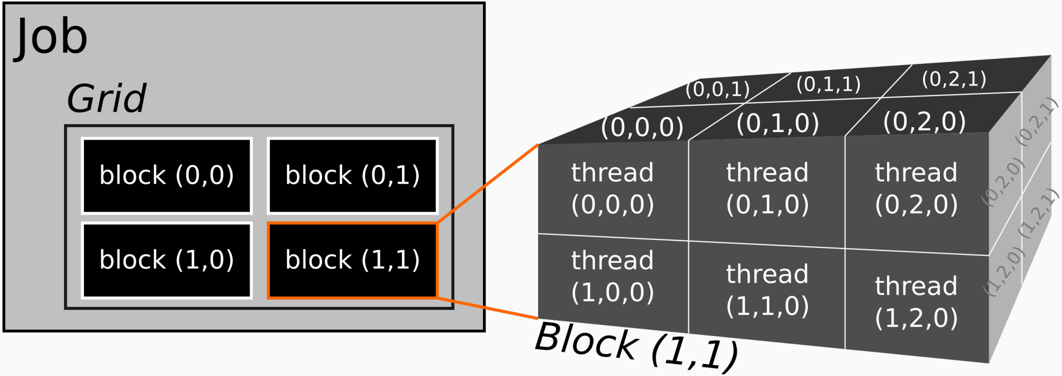

Figure 5: An example of 2D grid with 3D blocks

Grid & block examples (1/2)

Vector addition (N elements)

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];  /* Missing boundary check. */
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);  // <-- So this is how we launch CUDA kernels!
    ...
}
```

Grid & block examples (2/2)

Matrix addition (N×N elements)

```
// 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]; /* Missing boundary check. */
}

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);
    ...
}
```

Block decomposition enable automatic scalability

Because the work is divided into **independent blocs** which can be **run in parallel** on each *streaming multiprocessor* (SM), the same code can be **automatically scaled** to architectures with more or less SMs...

as long as SMs architectures are compatibles (100% compatible with the same Compute Capabilities version — a family of devices, careful otherwise).

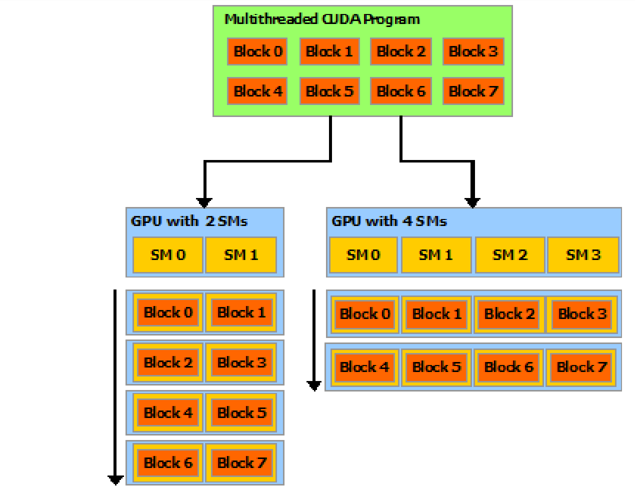

Figure 6: Automatic scaling

Building and running a simple program

CUDA Hello world (hello.cu)

```
#include <stdio.h>

__global__ void print_kernel() {
    printf(
      "Hello from block %d, thread %d\n",
      blockIdx.x, threadIdx.x);
}

int main() {
    print_kernel<<<2, 3>>>();
    cudaDeviceSynchronize();
}
```

Compile

```
$ nvcc hello.cu -o hello
```

Run

```
$ ./hello
Hello from block 1, thread 0
Hello from block 1, thread 1
Hello from block 1, thread 2
Hello from block 0, thread 0
Hello from block 0, thread 1
Hello from block 0, thread 2
```

What you need to get started

NVidia GPU hardware

NVidia GPU drivers, properly loaded

modprobe nvidia ...

CUDA runtime libraries

libcuda.so, libnvidia-fatbinaryloader.so, ...

CUDA SDK (NVCC compiler in particular)

relies on a standard C/C++ compiler and toolchain

docs.nvidia.com/cuda/cuda-installation-guide-linux

Basic C/C++ knowledge

Summary

Host vs Device $\leftrightarrow$ Separate memory

GPUs are computation units which require explicit usage, as opposed to a CPU

Need to load data to and fetch result from device

Replace loops with kernels

Kernel = Function computed in relative isolation on small chunks of data, on the GPU

Divide the work

Problem $\rightarrow$ Grid $\rightarrow$ Blocks $\rightarrow$ Threads

Compile and run using CUDA SDK

`nvcc`, `libcuda.so`, ...