Getting started with CUDA
Part 1 - CUDA overview

Edwin Carlinet, Joseph Chazalon {`firstname.lastname@epita.fr`}
Fall 2023

EPITA Research Laboratory (LRE)

Slides generated on September 8, 2023

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

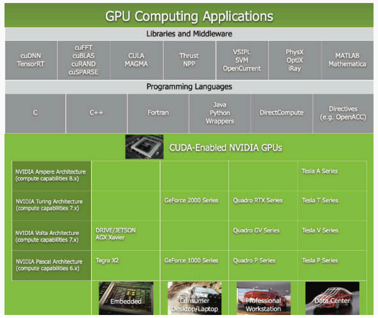
A runtime

- 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 (missing L and H series)



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT | cuFFT cuBLAS cuRAND cuSPARSE | CULA MAGMA | Thrust NPP | VSIPL SVM OpenCurrent | PhysX OptiX iRay | MATLAB Mathematica

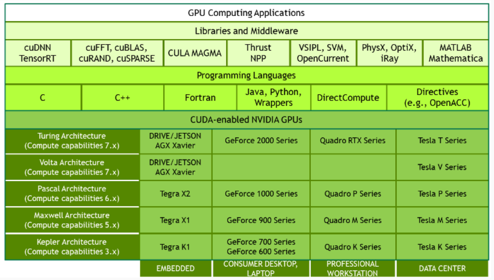
Programming Languages

C | C++ | Fortran | Java Python Wrappers | DirectCompute | Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

The CUDA ecosystem (not so long ago)



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT | cuFFT, cuBLAS, cuRAND, cuSPARSE | CULA MAGMA | Thrust NPP | VSIPL, SVM, OpenCurrent | PhysX, OptiX, iRay | MATLAB Mathematica

Programming Languages

C | C++ | Fortran | Java, Python, Wrappers | DirectCompute | Directives (e.g., OpenACC)

CUDA-enabled NVIDIA GPUs

Turing Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier	GeForce 2000 Series	Quadro RTX Series	Tesla T Series
Volta Architecture (Compute capabilities 7.x)	DRIVE/JETSON AGX Xavier			Tesla V Series
Pascal Architecture (Compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
Maxwell Architecture (Compute capabilities 5.x)	Tegra X1	GeForce 900 Series	Quadro M Series	Tesla M Series
Kepler Architecture (Compute capabilities 3.x)	Tegra K1	GeForce 700 Series GeForce 600 Series	Quadro K Series	Tesla K Series
	EMBEDDED	CONSUMER DESKTOP, LAPTOP	PROFESSIONAL WORKSTATION	DATA CENTER

Figure 2: The CUDA ecosystem

Libraries *and* Compiler Directives *and* 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()
{
  // MISSING: Allocation for A, B and C
  // MISSING: I/O to read n elements of A and B
  vecAdd(h_A, h_B, h_C, n);
}
```

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

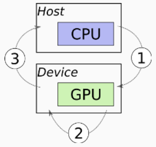


Figure 3: Computation on separate device

```
#include <cuda.h>
void vecAdd(float *h_A, float *h_B, float *h_C, int n)
{
  int size_bytes = 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

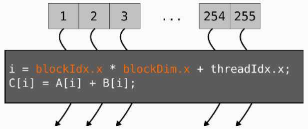
```
// kernel
__global__ void kvecAdd(float *d_A, float *d_B, float *d_C, int n)
{
  int i = blockDim.x * blockIDx.x + threadIdx.x;
  if (i >= n) { return; }
  d_C[i] = d_A[i] + d_B[i];
}
```

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



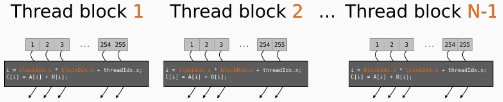
```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Figure 4: 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]



```
i = blockIdx.x * blockDim.x + threadIdx.x;
C[i] = A[i] + B[i];
```

Figure 5: 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 (added by the compiler) 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.

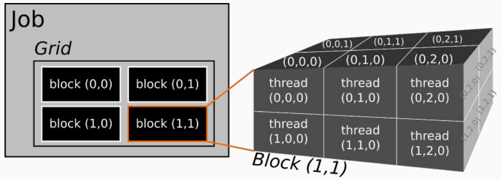


Figure 6: An example of 2D grid with 3D blocks

Grid & block examples (1/2)

Vector addition (N elements)

```
// Kernel definition
__global__ void VecAdd(float* d_A, float* d_B, float* d_C, int sz)
{
    int i = threadIdx.x;  // /!\ Assuming 1 block here
    if (i >= sz) { return ; }
    d_C[i] = d_A[i] + d_B[i];
}

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

Grid & block examples (2/2)

Matrix addition (N×N elements)

```
// Kernel definition
__global__ void MatAdd(float d_A[N][N], float d_B[N][N], float d_C[N][N], int sz)
{
    int i = threadIdx.x;  // /!\ Assuming 1 block here
    int j = threadIdx.y;  // /!\ Assuming 1 block here
    if (i >= sz || j >= sz) { return; }
    d_C[i][j] = d_A[i][j] + d_B[i][j];
}

int main()
{
    ...
    int numBlocks = 1;  // grid size: 1 * 1 * 1 blocks
    dim3 threadsPerBlock(N, N);  // block size: N * N * 1 threads
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C, sz);
    ...
}
```

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).

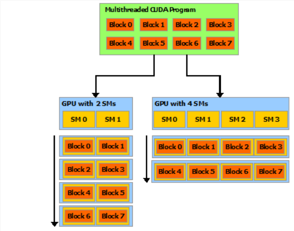


Figure 7: Automatic scaling

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 ↔ 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 → Grid → Blocks → Threads

Compile and run using CUDA SDK
nvcc, libcuda.so, ...