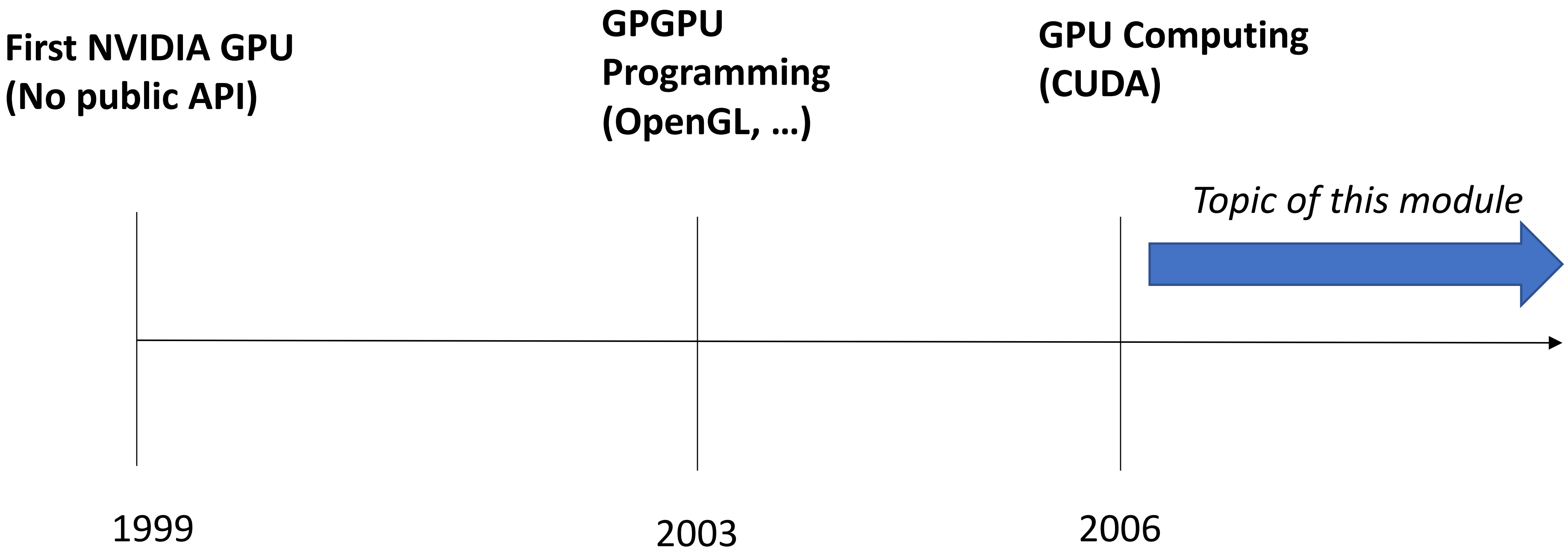
Computing with GPUs & CUDA

Stefano Markidis and Sergio Rivas-Gomez

Four Key-Points

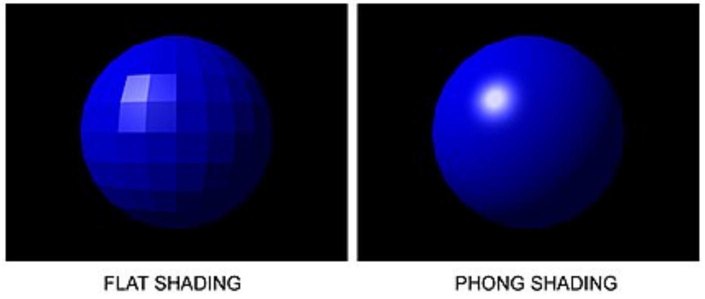
- The first general-purpose computing on GPU (GPGPU) required reformulating computational problems in terms of graphics primitives and using graphics APIs, such as OpenGL.
- GPU Computing frameworks, like CUDA, allow to bypass graphics API and ignore the underlying graphical concepts.
- Three major approaches for programming GPU: low-level, compiler directives-based and library approaches.
- CUDA is a framework for parallel computing on NVIDIA GPUs, based on extending C/C++ for programming GPUs.

Timeline of GPU Computing



GPGPU Computing (2003)

- General-purpose computing on graphics processing units (**GPGPU**) is the use of GPU to perform computation in applications traditionally handled by the CPU.
- GPUs can operate on pictures and graphical data much faster than a traditional CPU.
  - Migrate data into graphical form and then use the GPU to scan and analyze it.
- GPGPU emerged after 2003, with the advent of **programmable shaders** and support for **floating-point** on graphics processors.



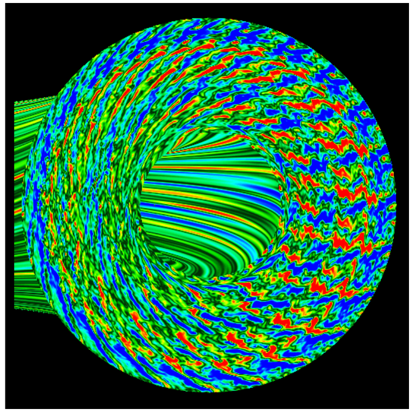
A shader is a program to
do shading: light, darkness, color, …

GPUGPU Computing – Do Computing Like It were Graphics …

- A GPGPU provided parallel processing that analyzes data as if it were in image or other graphic form.
  - Problems involving matrices and/or vectors were easy to translate to a GPU.
- These required **reformulating computational problems in terms of graphics primitives**, as supported by the two major APIs for graphics processors, OpenGL and DirectX.

Graphics Techniques, APIs

Computer Simulation
Fusion Reactor



GPU Computing – Computing without Graphics APIs (2006)

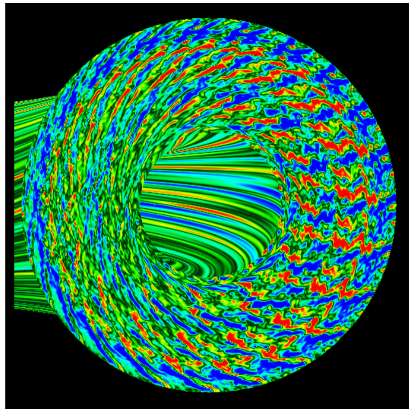
GPU computing is the use of GPU for computing without using the traditional graphics API and graphics pipeline mode:

- NVIDIA introduced in 2006 **CUDA** to bypass graphics API and ignore the underlying graphical concepts and simply program in C or C++
    - No need for full and explicit conversion of the data to a graphical form.
- Newer, hardware vendor-independent offerings include MS DirectCompute and Apple/Khronos Group's **OpenCL**.

~~Graphics Techniques APIs~~

HPC Approach

Computer Simulation
Fusion Reactor



GPU Computing: 3 Major Approaches

Control of the hardware

More lines of code

High-Level Approaches

Libraries
Thrust and ArrayFire

Compiler Directives
OpenACC/OpenMP

Low-Level Approaches
CUDA, OpenCL

Low-Level Approaches: CUDA and OpenCL

- NVIDIA **CUDA** is a parallel platform created by NVIDIA.
  - Proprietary framework targeting only NVIDIA computing platforms ☹
- **OpenCL** specifies a programming language, based on C99 and C++11, for programming:
  - NVIDIA GPUs, AMD GPUs, FPGA, DSP, CPUs, integrated GPUs, …

They have many concepts common (but different names for things!) so you can apply the same CUDA concepts to OpenCL.

CUDA term	OpenCL term
GPU	Device
Multiprocessor	Compute Unit
Scalar core	Processing element
Global memory	Global memory
Shared (per-block) memory	Local memory
Local memory (automatic, or __local__)	Private memory
kernel	program
block	work-group
thread	work item

From "Experiences porting from CUDA to OpenCL" by Matt Harvey

Compiler Directives: OpenACC and OpenMP

The programmer annotates C/C++ and Fortran source to identify the areas that should be executed on GPU using compiler directives (or pragmas) and additional functions

- **OpenACC** (for *open accelerators*) is a programming standard developed by Cray, CAPS, NVIDIA and PGI from 2012
  - It supports only NVIDIA GPUs
- **OpenMP** (> 4.0 supports GPUs, 2014) programming standard for supporting thread parallelism
  - It supports CPU, GPU and other accelerators

OpenACC OpenMP

```
parallelRand(A);
#pragma acc parallel loop
for (i=0; i<N; i++)
{
  A[i] *= 2;
}
```

Libraries: Thrust and ArrayFire

- Thrust is a **C++ library**, strongly resembling C++ **STL**, of parallel algorithms and data structures: good for sort, scan, transform, … operations on GPU.
- ArrayFire is an open-source **function library** with interfaces for C, C++, Java, R and Fortran. It integrates with any CUDA applications.

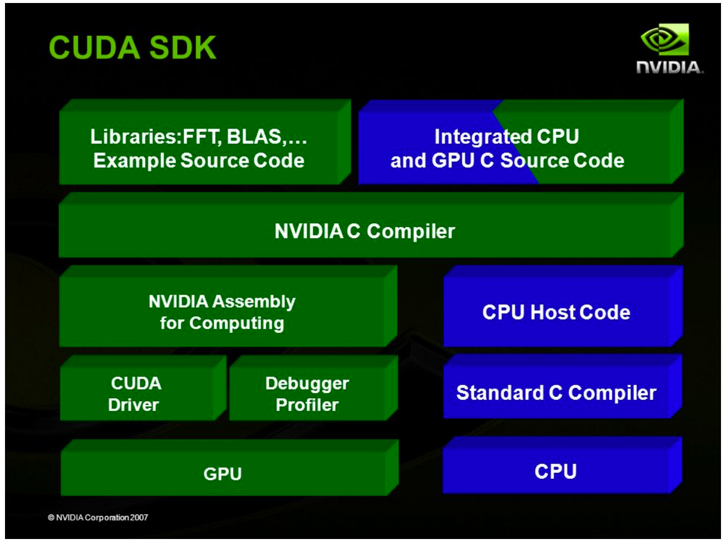
Why we are going to use CUDA?

- It is a low-level interface that allows for understanding all the basic concepts and foundations you need for basic GPU programming
  - You can move later on to higher-level interfaces but you will still need to understand basic CUDA concepts to use effectively GPUs
  - You can move to OpenCL as they share the same concepts
- It provides better performance than OpenCL on NVIDIA GPUs - dah

CUDA Framework

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
3. SDK
   - lots of demonstration examples
   - some error-checking utilities



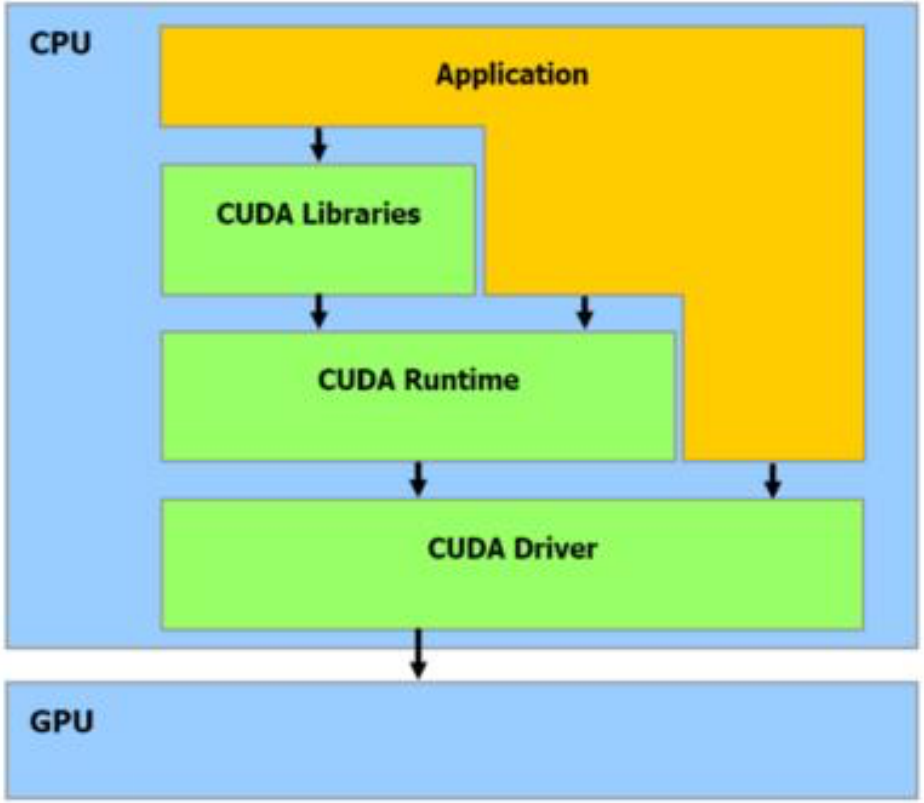
CUDA APIs

CUDA provide the choice of two APIs:

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

These APIs are mutually exclusive: an application should use either one or the other.

We will only use the **runtime API**



CUDA Virtual Processr as `nvcc` Flag

- When you compile your CUDA code, you provide `nvcc` with the `–arch=…` (or `--gpu-architecture=`) flag followed by
  - `compute_xx` or equivalent `sm_xx`
- Via this flag, you provide the compiler with the *virtual processor architecture* of your GPU
  - A set of CUDA programming features your GPU supports

5.5. Virtual Architecture Feature List

`compute_30` and `compute_32`	Basic features + Kepler support + Unified memory programming
`compute_35`	+ Dynamic parallelism support
`compute_50`, `compute_52`, and `compute_53`	+ Maxwell support
`compute_60`, `compute_61`, and `compute_62`	+ Pascal support
`compute_70`	+ Volta support

http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html#virtual-architecture-feature-list

CUDA C/C++ - CUDA Fortran - CUDA Python/PyCUDA

- Extensions to C/C++ language to use GPUs. C/C++ programmers use the CUDA C/C++, compiled with *nvcc*, that is the NVIDIA LLVM-based C/C++ compiler.
- Fortran programmers can use **CUDA Fortran**, compiled with the PGI CUDA Fortran compiler.
- Continuum Analytics provides **Numba Python** CUDA compiler that is now part of Anaconda Distribution. **PyCUDA** is a Python Wrapper to CUDA.

To Summarize

- General-purpose computing on GPU (GPGPU) requires reformulating computational problems in terms of graphics primitives and using graphics APIs, such as OpenGL.
- GPU Computing frameworks, like CUDA, allow to bypass graphics API and ignore the underlying graphical concepts.
- Three major approaches for programming GPU: low-level, compiler directives-based and library approaches.
- CUDA is a framework for parallel computing on NVIDIA GPUs, based on extending C/C++ for GPUs.