Parallel Computing Systems and Applications
GPU General Purpose Programming

Salvatore Filippone, PhD

DICII
salvatore.filippone@uniroma2.it

Course Objectives

- Understand the CUDA programming model (threading, memory, synchronisation)
- Understand the most important factors affecting GPU codes performance
- Be able to develop GPGPU applications using the CUDA framework
- Apply CUDA optimisation techniques to applications using knowledge of the programming model and hardware architecture
- Use existing GPU development tools and libraries;

What Should You Already Know?

Multi-threading

- Thread execution is non-deterministic;
- Need to find sweet spot for number of threads, not too few, not too many;

Shared Memory Models

- Process nodes access the same data structures;
- Multiple threads may attempt to access the same data at the same time

Synchronisation

- Data dependencies must be dealt with by communication between threads;
- Resequencing code (very, very often) reduces performance;

SIMD Revision

Flynn classification

Single Instruction, Multiple Data

- Each processing node runs a kernel program
- Each processing node is assigned input data
- Each processing node executes the same instruction at the same time

GPUs: SIMD approach on steroids!

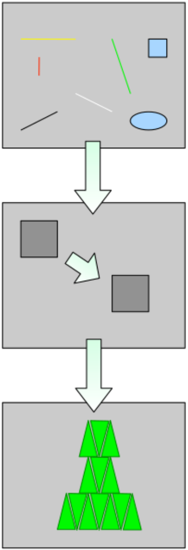
In the MPI world, most popular approach is SPMD Single Program Multiple Data, which is close to SIMD, although MPI is a lot more flexible. SIMD maps naturally to many application fields, including graphics.

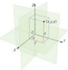
What are GPUs?

"GPU" name introduced by NVIDIA Co.

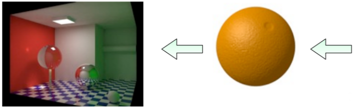
- Specialised hardware used for rendering images on a computer screen
- Hardware architecture designed specifically for that purpose
- How do they differ from CPUs?
  - Different purposes
  - Hardware specific to graphics rendering
  - Different design goals

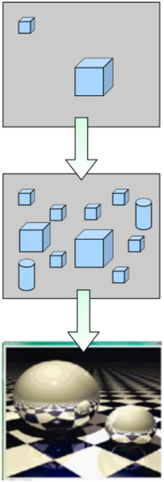
- Originally, graphics handled by CPU
- Blitters — move graphic surface very quickly
- 2D Acceleration — shape primitives



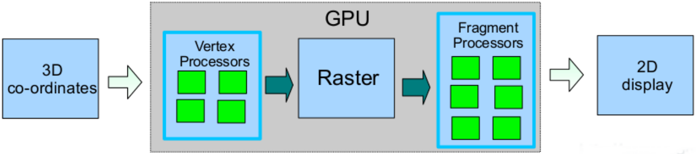


- Early 3D acceleration performed by CPU — Hardware 3D acceleration introduced
- Increased acceleration of graphical effects *Transform & Lighting*
- Introduction of user programmable shaders — Allowed more freedom in producing graphics effects





- The graphics pipeline has evolved as the capabilities evolved
- The introduction of shaders allows portions of the pipeline to be programmed
  - Previously all stages were fixed
  - The pipeline itself has limited flexibility in terms of data workflow
  - But shaders introduce significant flexibility to the processing capabilities



Nvidia Shaders

- Shaders originally programmed using machine level instructions — Often still done this way for high performance graphics in games
- Nvidia introduced Cg as a more convenient way of programming the graphics hardware
  - Targeted to graphics programmers, but also useful for GPGPU
  - Still required detailed knowledge of the processing pipeline
  - Required the computational problem to be mapped to the hardware
  - Not simple to produce good results without significant effort
- CUDA then introduced to abstract the hardware architecture and simplify the programming tasks

Originally CUDA stood for *Compute Unified Device Architecture*, but is now used as a noun in its own right.

Shader Implementation

Initially shaders were dedicated to a particular task

- Vertex Shaders – manipulated 3D coordinates
- Fragment Shaders – manipulated pixel values

Later GPU architectures

- Single type of "unified" shader
- Move from vector to scalar processors
- A large increase in the number of processing units – 10's to 100's

Games still the primary motivator for GPU innovation

- But manufacturers recognise the potential for performance computation;
- Introducing specific innovations (ie. double precision, large memory);
- Specific product lines (NVIDIA Tesla);

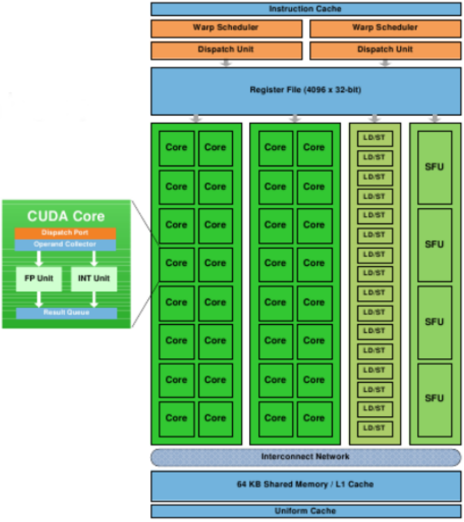
Many of the mathematical operations underpinning graphics rendering also serve other numerical intensive tasks

- This is an SIMD architecture
- The increased flexibility of shaders allows them to be used to perform non-graphical processing
- General Purpose GPU computation was born

Computational scientists (always on the lookout) quickly grasped the potential; as a result, see current Top500 list where accelerators dominate.

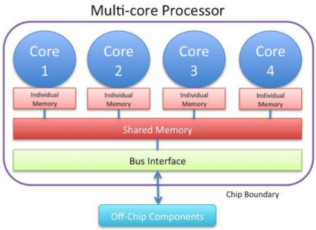
What are GPUs?

A bunch of these $\Rightarrow$



Fermi Streaming Multiprocessor (SM)

Instead of that $\Downarrow$



GPU is a massively multi-threaded co-processor

- Contains hundreds of simple processing nodes
- Complex general purpose circuitry is replaced by simple ALUs

Needs many threads to achieve peak performance

- Capable of managing thousands of threads
- Can scale between orders of magnitudes of threads

Computational problem must be decomposed to expose the parallel opportunities.

It is not always faster than a CPU!

$\Rightarrow$ In fact, a single ALU is slower than a normal CPU.

CUDA (Compute Unified Device Architecture)

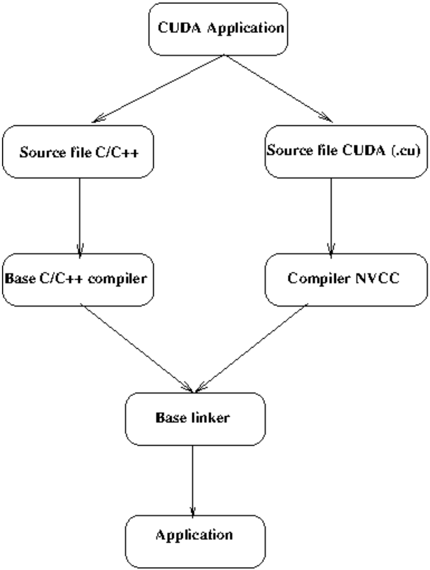
A framework for transferring general purpose computational problems onto graphics hardware

- Extended version of the C programming language
- Separate toolkit including specialist compiler:
- Meta compiler / wrapper - nvcc
- Splits compilation duties between native CPU compiler and CUDA compiler

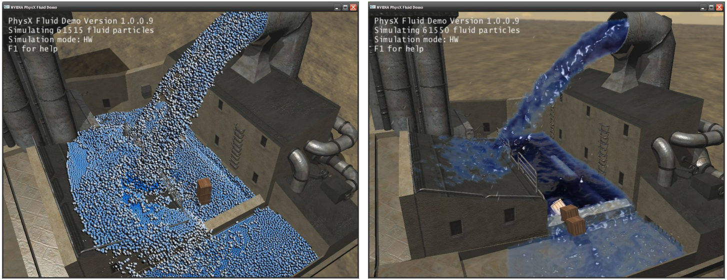
Also available from Fortran in the PGI and GNU compilers.

CUDA Toolchain

- Additional rules guide compilation
- C/C++ source can be saved in .cu files for convenience
- nvcc will work out which compiler will be called



CUDA SDK



SDK available with each version of CUDA; Contains helper routines, tools, and many code samples (great for learning GPGPU!)

CUDA

- The most mature technology
- Very popular in academic/scientific areas
- Only runs on NVIDIA hardware

OpenCL

- Cross platform standard
- Runs on CPU and GPUs
- Implemented on many devices

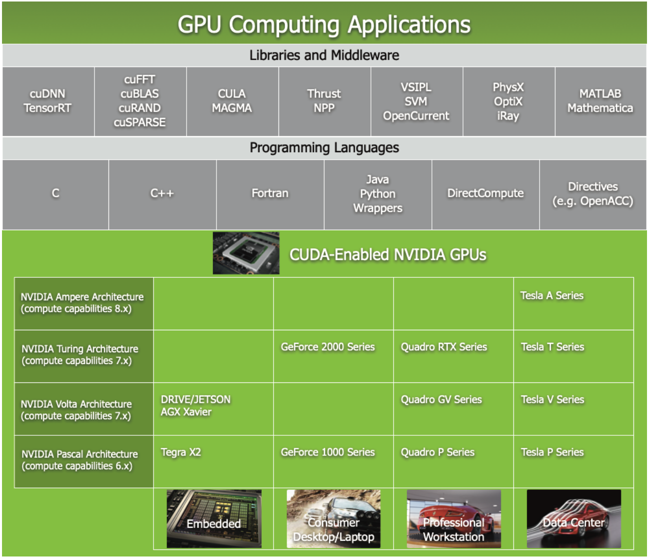
OpenACC

- Directive extensions to the base language;
- Available for Fortran and C;
- Multiple compiler and hardware vendors (including GNU);

DirectCompute

- Microsoft API (DirectX 11)
- Works on Windows

CUDA Landscape



GTX 285

K80

Useful references

- David B. Kirk & Wen-mei W. Hwu: Programming Massively Parallel Processors, Morgan Kaufmann, 2010;
- Jason Sanders & Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010;
- Wen-mei W. Hwu (ed): GPU Computing Gems, Morkan-Kaufmann, 2011
- Gerassimois Barlas: Multicore and GPU Programming, an integrated approach, Morgan-Kaufmann, 2015
- Norm Matloff: Programming on Parallel Machines `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`

Useful references

NVIDIA developer websites:

- To get CUDA Toolkit: https://developer.nvidia.com/cuda-toolkit
- https://developer.nvidia.com/category/zone/cuda-zone
- https://developer.nvidia.com/cuda-education
- CUDA Programming Guide: http://docs.nvidia.com/cuda/cuda-c-programming-guide/
- CUDA API Reference: http://docs.nvidia.com/cuda/index.html

Countless online courses and blogs:

- Udacity course https://www.udacity.com/course/cs344
- Parallel Forall blog http://devblogs.nvidia.com/parallelforall/
- Presentations by V. Volkov http://www.cs.berkeley.edu/~volkov/