Introduction to GPGPU Programming

CPS343

Parallel and High Performance Computing

Spring 2013

Outline

1. Introduction
   - Hardware Overview
   - Software Overview

Acknowledgements

Material used in creating these slides comes from

- Course on CUDA Programming by Mike Giles, Oxford University Mathematical Institute
- NVIDIA's CUDA C Programming Guide

GPGPU and Accelerators

- GPGPU stands for *General Purpose Graphics Processing Unit*
- Very fast at certain floating point calculations.
- By the early 2000s graphics hardware was starting to support user-programmable features such as lighting and shading calculations, not to mention texture mapping.
- By the mid 2000s some were using OpenGL and other programming tools to do numeric computations on GPU devices.
- In effect, one had to map a mathematical computation onto a graphics computation, do the graphics (probably ignoring the display), and extract the result from the graphics memory.

History

- During the late 2000 NVIDIA introduced CUDA as a general purpose tool to make GPUs available for numerical computation.
- Originally CUDA was an acronym for "Compute Unified Device Architecture"
- It is proprietary, but gained a wide following
- Soon after the OpenCL standard was introduced by a consortium of vendors.
- OpenCL is an open standard, using it makes it easy to run GPU-accelerated programs on different vender's hardware.
- In 2012 Intel released it's Phi accelerator, similar to GPU hardware but without the graphics heritage.

Other Software

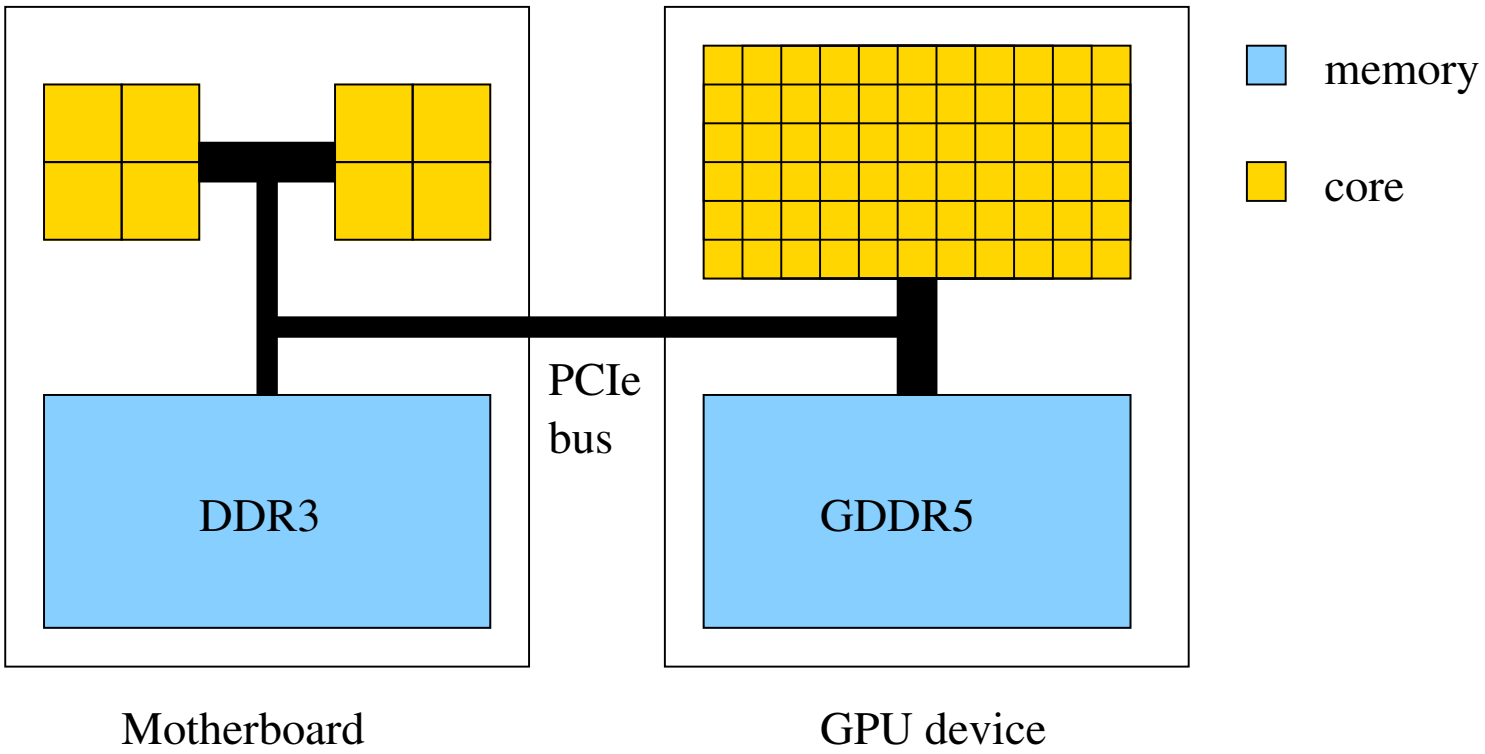
- Although we'll focus on CUDA in this course, be aware that other systems are in use and under development.
- Primary of these is OpenCL
- Another approach is to use compiler directives (as is done in OpenMP) to take advantage of accelerators. The OpenACC project does this. Currently not available in GNU compiler tool chain.

Outline

1. Introduction
   - Hardware Overview
   - Software Overview

Top level hardware view

Typical arrangement is motherboard with mutlicore CPU(s) and a graphics card or general purpose GPU device with many cores attached via a PCIe bus.



NVIDIA CUDA generations

NVIDIA has released three generations of CUDA-enabled GPU devices

1. Tesla - supported only single-precision floating point
2. Fermi - supports double-precision floating point
3. Kepler - double precision

The Tesla generation name should not be confused with the current line of dedicated non-graphics GPU-based HPC cards, also called Tesla.

Current NVIDIA graphics cards

Here is a brief list of Kepler generation devices:

GeForce (consumer graphics cards)

- GTX650 Ti: 768 cores, 1 or 2GB GDDR5 RAM
- GTX660 Ti: 1344 cores, 2GB
- GTX680: 1536 cores, 2 or 4GB
- GTX690: 2×1536 cores, 2×2GB

Tesla (HPC cards)

- K10: 2×1536 cores, 2×4GB
- K20: 2496 cores, 5GB
- K20X: 2688 cores, 6GB

Streaming multiprocessor

The "building block" is a "streaming multiprocessor" (SMX):

- 192 cores and 64K registers
- 64KB of shared memory / L1 cache (user configurable)
- 8KB of cache for constants
- 64KB of texture cache for read-only arrays
- up to 2K threads per SMX

Multithreading

Cores in SMX are SIMT (Single Instruction, Multiple Thread).

- all cores execute the same instructions simultaneously, but with different data
- minimum of 32 threads all doing same thing at the same time
- no "context switching;" each thread has its own registers (limits the number of active threads)
- threads in each SMX are grouped into "warps" of 32 threads
- each thread in a warp executes exactly the same instruction, or waits while others in the warp execute instructions
- note - want to avoid branches within a warp as the branches are executed sequentially

Outline

1. Introduction
   - Hardware Overview
   - Software Overview

High-level execution profile

The computer with CPU is called the "host" and the GPU card is called the "device."

1. host and device are initialized when program starts
2. separate memory for data is allocated on host and on device
3. host data is initialized
4. data copied from host to device
5. host launches multiple instances of execution "kernel" on device
6. data copied from device to host
7. steps 4–6 repeated as needed
8. host deallocates memory on device and host and terminates

Lower-level view

Within the GPU device

- each instance of the execution kernel executes on an SMX
- multiple execution kernels may be scheduled on an SMX if there are sufficient registers and shared memory; otherwise they wait in a queue
- all threads in one instance can access local shared memory but but not memory or registers local to the other threads
- the execution kernel instances may execute in any order