XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 14: Introduction to GPGPU programming

Previous lectures

So far we have looked at CPU programming.

- **Shared memory systems**, where lightweight **threads** are mapped to cores (scheduled by the OS) *[Lectures 2-7]*.
- **Distributed memory systems**, with explicit communication for whole **processes** *[Lectures 8-13]*.
- Many **common** parallelism issues (scaling, load balancing, synchronisation, binary tree reduction).
- Also some **unique** to each type (locks and data races for shared memory; explicit communication for distributed memory).

Today's lecture

Today's lecture is the first of 6 on programming **GPU**s (Graphics Processing Units) for **general purpose calculations**.

- Sometimes referred to as **GPGPU** programming for General Pupose Graphics Processing Unit programming.
- GPU **devices** contain multiple **SIMD** units.
- Different **memory types**, some 'shared' and some that can be interpreted as 'distributed.'
- Programmable using a variety of C/C++-based languages, notably **OpenCL** and **CUDA**.

Development of GPUs[1]

Early **accelerators** were driven by graphical operating systems and high-end applications (defense, science and engineering *etc.*).

- Commercial 2D accelerators from early 1990s.
- OpenGL released in 1992 by Silicon Graphics.

Consumer applications employing 3D dominated by **video games**.

- First person shooters in mid-90s (Doom, Quake *etc.*)
- 3D graphics accelerators by Nvidia, ATI Technologies, 3dfx.
- Initially as external **graphics cards**.

[1] Sanders and Kandrot, *CUDA By Example* (Addison-Wesley, 2011).

Programmable GPUs

The first **programmable** graphics cards were Nvidia's GeForce3 series (2001).

- Supported DirectX 8.0, which includes **programmable** vertex and pixel shading stages of the graphics pipeline.
- Increased programming support in later versions.

Early **general purpose** applications 'disguised' problems as being **graphical**.

1. Input data converted to pixel **colours**.
2. Pixel shaders performed calculations on this data.
3. Final 'colours' converted back to **numerical data**.

GPGPUs

In 2006 Nvidia released its first GPU with CUDA.

- General calculations *without* converting to/from colours.

Now have GPUs that are *not intended* to generate graphics.

- Modern HPC clusters often include GPUs.
- *e.g.* Summit has multiple Nvidia Volta GPUs per node.
- Vendors include Nvidia, AMD and Intel.

Originally designed for **data parallel** graphics rendering.

- Increasing use of GPUs for *e.g.* **machine learning**[1] and **cryptocurrencies**.

[1] Now also have **neural processing units** (NPUs) for machine learning.

Overview of GPU architectures

Design and terminology of GPU hardware differs between vendors.

- Nvidia *different to* AMD *different to* Intel *different to* ...

Typically will have 'a few' **SIMD processors**:

- **SIMD**: Single Instruction Multiple Data.
- **Streaming multiprocessors** in Nvidia devices.

SIMD processors contain **SIMD function units** or **SIMD cores**:

- Each SIMD core contains multiple **threads**.
- Executes **the same** instruction on multiple data.

Hierarchy:

$$\text{Threads} \in \text{SIMD Cores} \in \text{SIMD Processors} \in \text{GPU}$$

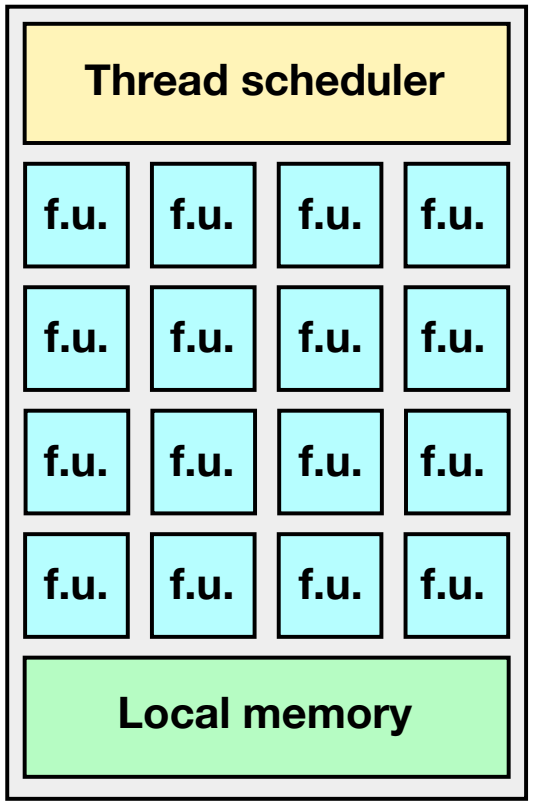
SIMD processor

A typical SIMD processor has:

- A thread **scheduler**.
- Multiple **SIMD function units** (**'f.u.'**) or **SIMD cores**, each with 32/64/*etc.* **threads**.
- **Local memory**

Not shown but usually present:

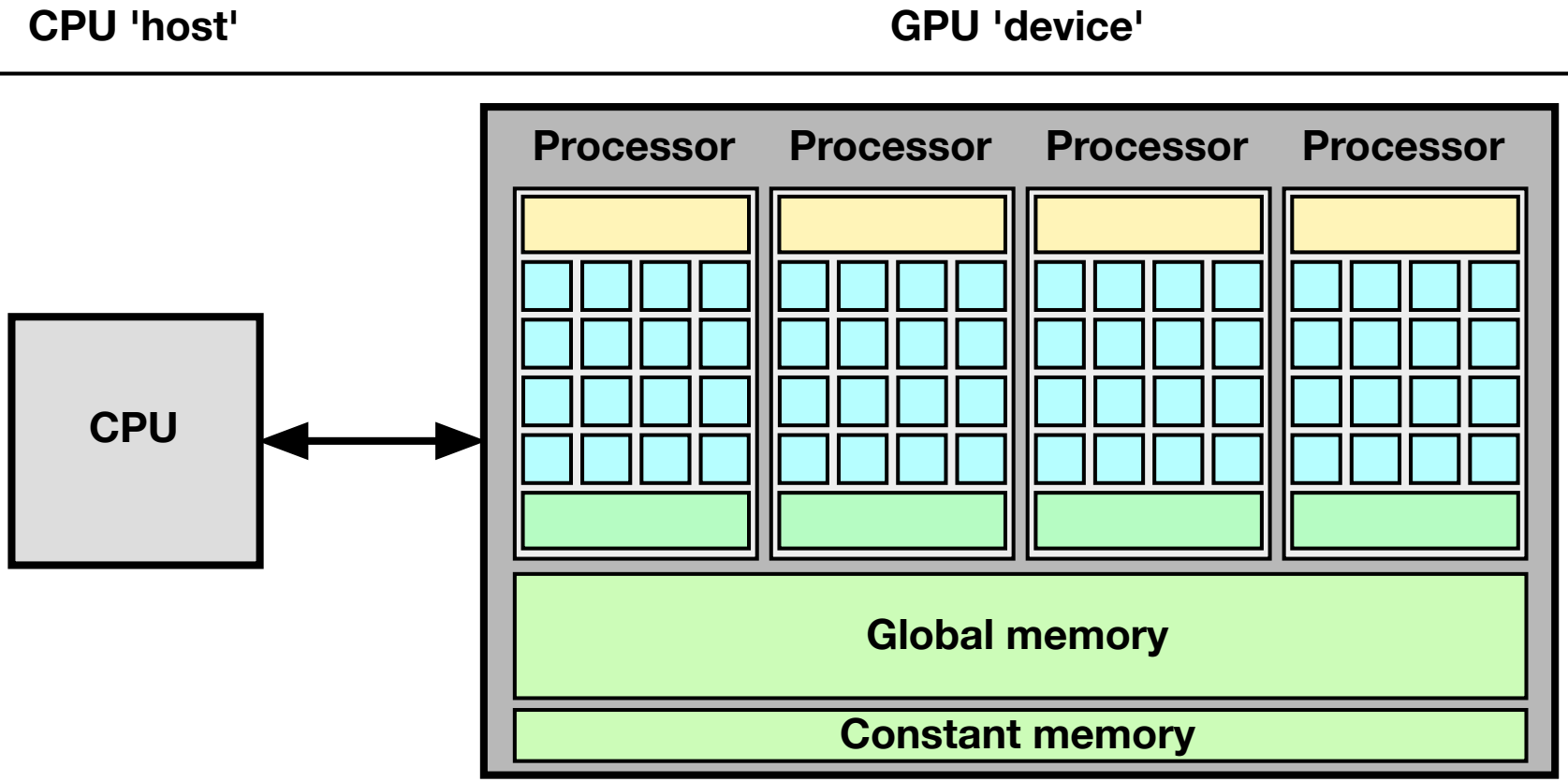
- Registers, special floating point units, ...



Note:

Thread scheduling is performed **in hardware**.

CPU with a single GPU



- The **data bus** between CPU and GPU is **very slow**.
- Faster for **integrated GPU**s.

SIMD *versus* SIMT

Nvidia refer to their architectures as **SIMT** rather than SIMD.

- Single Instruction Multiple **Threads**.
- **Conditionals** can result in **different** operations being performed by **different** threads.
- However, cannot perform different instructions **simultaneously**.
- Therefore 'in between' SIMD and MIMD.

Will look at this more closely in Lecture 17, where we will see how it can be detrimental to performance.

Books

McCool *et al. [Lecture 1]* includes *some* OpenCL, but does not address GPUs specifically. Books for GPU programming include:

- **Heterogeneous computing with OpenCL 2.0**, *Kaeli, Mistry, Schaa and Zhang* (Morgan-Kauffman, 2015).
  - Quite detailed and practical, not too technical.

- **CUDA by example**, *Sanders and Kandrot* (Addison-Wesley, 2011).
  - Slightly old, but a gentle introduction.
  - Only considers CUDA, whereas we will use OpenCL, but may still be useful.

You do not need any of these books for this module!

GPU programming languages 1. CUDA

The first language for GPGPU programming was Nvidia's **CUDA**.

- Stands for **Common Unified Device Architecture**.
- C/C++-based (a FORTRAN version also exists).
- First released in 2006.
- **Only works on CUDA-enabled devices**, *i.e.* Nvidia GPUs.

As the first GPGPU language it has much documentation online. Therefore we will reference CUDA concepts and terminology quite frequently, often in footnotes.

GPU programming languages 2. OpenCL

Currently the main alternative to CUDA is **OpenCL** (2008).

- Stands for **Open Computing Language**.
- Maintained by the **Khronos** group after proposal by members of Apple, AMD, IBM, Intel, Nvidia and others.
- Runs on **any** (modern) GPU, not just Nvidia's.
- Can also run on CPUs, FPGAs (=Field-Programmable Gate Arrays), ...
- C/C++ based.
- Similar programming model to CUDA.
- OpenCL 3.0 released Sept. 2020.

Directive based programming abstractions

OpenACC (2011):

- Open ACCelerator, originally intended for **accelerators**.
- Uses `#pragma acc` directives.
- Limited (but growing) compiler support — *e.g.* gcc 7+.

OpenMP:

- GPU support from version 4.0 onwards, *esp.* 4.5 (gcc 6+).
- Usual `#pragma omp` directives, with `target` to denote GPU.

Both give **portable** code, but both require some understanding of the hierarchical nature of GPU hardware to produce reasonable performance.

Installing OpenCL

Already installed on `cloud-hpc1.leeds.ac.uk` (and most Macs).

Otherwise, download drivers and runtime for your GPU architecture:

Nvidia: `https://developer.nvidia.com/opencl`

Intel: `https://software.intel.com/en-us/intel-opencl/download`

AMD: `https://www.amd.com/en` and search for OpenCL.

OpenCL header file

All OpenCL programs need to include a header file.

Since the name and location is different between Apple and other UNIX systems, most of the example code for this module here will have the following near the start:

```
1 #ifdef __APPLE__
2 #include <OpenCL/opencl.h>
3 #else
4 #include <CL/cl.h>
5 #endif
```

Note that the coursework will be marked on a system similar to `cloud-hpc1.leeds.ac.uk`, so it **must** run on that system.

Compiling and running

We use the CUDA `nvcc` compiler on `cloud-hpc1.leeds.ac.uk`:

```
nvcc -lOpenCL -o <executable> <source>.c
```

Note there is no '`-Wall`' option for `nvcc`.

Executing:

To execute on a GPU it will be necessary to use the batch queue (see next slide). However, it is also possible to run an OpenCL code on the login node's CPU by launching as any normal executable:

```
./<executable> [any command line arguments]
```

Running on GPU via batch jobs

The batch node of `cloud-hpc1.leeds.ac.uk` may be configured with a Tesla T4 GPU.

Hence GPU jobs should be executed via the batch queue using the following approach:

- Compile your code as described in previous slide;
- Create a job submission script as outlined below;
- Submit ot the batch queue using `sbatch` in the usual manner.

Here is a typcal batch script to run "`gpu-example`":

```
#!/bin/bash
#SBATCH --partition=gpu --gres=gpu:t4:1
./gpu-example
```

Compiling and running: Macs

Compiling:

Use the `OpenCL` framework:

```
gcc -Wall -framework OpenCL -o <executable> <source>.c
```

- If you see deprecation **warnings**, drop `-Wall`, or add `-DCL_SILENCE_DEPRECATION` or `-Wno-deprecated`.
- If you see deprecation **errors**, try `clang` or another version of `gcc`.

Executing:

Launch as any normal executable

```
./<executable> [any command line arguments]
```

Platforms, devices and contexts

Since OpenCL runs on many different devices by many different vendors, it can be quite laborious to initialise.

Need to **determine**:

Platform	Common interface between **host** (CPU) and **vendor-specific** OpenCL runtimes.
Device	Belongs to a platform; may be more than 1.

Need to **initialise**:

Context	Coordinates interaction between host and a **device** (*e.g.* a GPU). One per device.
Command queue	To request action by a device. Normally one per device, but can have more *[Lecture 19]*.

Initialisation code

Most code for this module will come with `helper.h`, which contains two useful routines:

`simpleOpenContext_GPU()`

- Finds the **first** GPU on the **first** platform. Prints an error message and `exit()`s if one could not be found.

`compileKernelFromFile()`

- Compiles an OpenCL **kernel** to be executed on the device. Will cover this next lecture.

You don't need to understand how these routines work, but are welcome to take a look.

Using simpleOpenContext_GPU()

```
#include "helper.h"   // Also includes OpenCL.
int main() {
  // Get context and device for a GPU.
  cl_device_id device;
  cl_context context = simpleOpenContext_GPU(&device);

  // Open a command queue to it.
  cl_int status;
  cl_command_queue queue = clCreateCommandQueue(
    context,device,0,&status);

  ...  // Use the GPU through 'queue'.

  // At end of program.
  clReleaseCommandQueue(queue);
  clReleaseContext(context);
}
```

'Hello world' in OpenCL

Code on Minerva: `displayDevices.c`

Since most GPU's cannot print in the normal sense, there is no simple 'Hello World' program.

Instead, try the code `displayDevices.c` (which *doesn't* use `helper.h`).

- Loops through **all** platforms and devices.
- Lists all **OpenCL-compatible** devices.
- Also a list of extensions; *e.g.* `cl_khr_fp64` means that device supports double precision floating point arithmetic.
- In the output, a **compute unit** is a SIMD processor or streaming multiprocessor.

Summary and next lecture

Today we have started looking at GPU programming:

- Overview of GPU architectures.
- Options for programming: **OpenCL**, CUDA, ...
- How to install, compile and run an OpenCL program.
- `displayDevices.c`, which lists all OpenCL-enabled devices using the functions:
  - `clGetPlatformIDs`
  - `clGetDeviceIDs`

Next time we will implement a "real" program in OpenCL: **vector addition**.