# Introduction to GPU programming

Edward Higgins

# What is a GPU?

# What is a GPU

- “Graphics Processing Unit”
  - ...but they can be used for more than just graphics
- Many lightweight computing cores
- Fast onboard memory

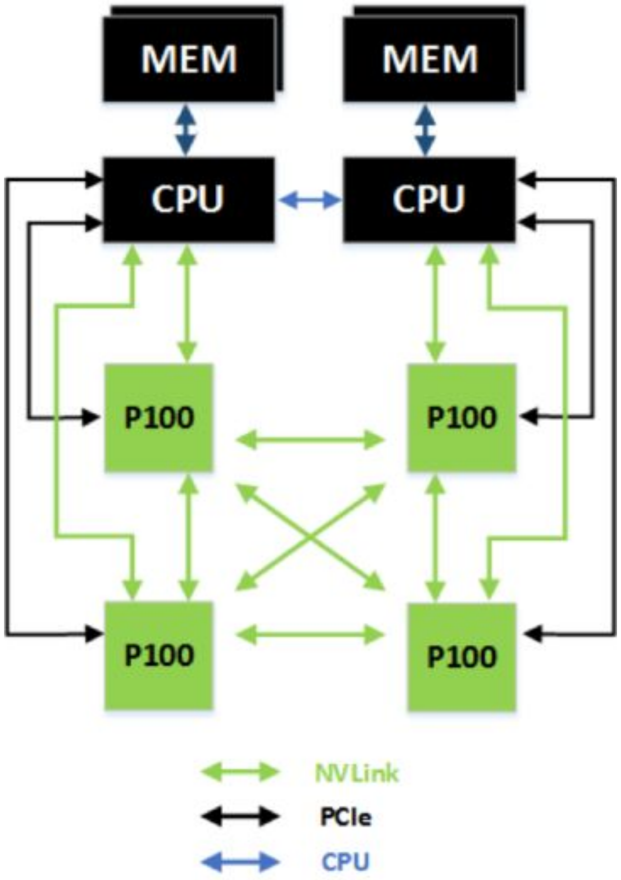
# History of GPUs

| | |
|---|---|
| ‘70s - ‘80s | Video requirements of computers start getting more demanding => dedicated graphics hardware developed |
| ‘90s - 2000s | Discrete GPUs become available |
| Late 2000s | People start trying to use GPUs for computations (CUDA/OpenCL) |
| 2010s | GPU programming gains traction in the HPC market |
| Today | Top 2 supercomputers in the world predominantly GPU based |

# GPUs vs CPUs

| | CPUs | GPUs |
|---|---|---|
| # of Cores | Up to 64 | 100’s to 1000’s |
| Clock speed | 2 to 5 GHz | < 1 to 1.5 GHz |
| Memory Bandwidth | Up to 100 GB/s | 900GB/s (on-board)<br>32 GB/s (over PCIe) |
| Peak performance (single precision) | Up to 2 TFLOPS | Up to 15 TFLOPS |

# Typical node layout



# Basic unit - the Streaming Multiprocessor

# GPU made up of many SMs

# How to program a GPU

# What kind of problems work well on GPUs?

- Many independent parallel tasks
  - Ideally, number of tasks $\gg$ number of GPU cores
- Data parallelism - SIMD
- Small, computationally intensive kernels
- Examples: Linear algebra, Molecular Dynamics, Lattice based/CFD

# GPU basic workflow

1. Copy relevant data onto the GPU

2. Perform some computational *kernel* on that data

3. Copy the results off the GPU

Want to maximise time in (2) and minimise time in (1) and (3)

# Example - the serial algorithm

```
do i = 1, N
    new_grid(i) = grid(i+1) - 2*grid(i) + grid(i-1)
end do

do i = 1, N
    grid(i) = new_grid(i)
end do
```

# OpenACC/OpenMP

- Open standards
- Directives based - you’re giving the compiler “hints”
- CPU code is the same as the GPU code
- Supported in C, C++ and Fortran

# Example - Fortran / OpenACC

```fortran
!$ACC data copy(grid) create(new_grid)
!$ACC parallel vector_length(256)

!$ACC loop
do i = 1, N
    new_grid(i) = grid(i+1) - 2*grid(i) + grid(i-1)
end do

!$ACC loop
do i = 1, N
    grid(i) = new_grid(i)
end do

!$ACC end parallel
!$ACC end data
```

# CUDA

- NVIDIA’s programming platform
- Kernel based - separate code for the GPU
- Explicitly copy data and launch kernels
- Supported in C, C++ and Fortran, with 3rd party wrappers available for other languages

# Example - CUDA C (Compute kernel)

```
// Function for each GPU thread to run
__global__ void do_stuff( float new_grid[N+2],
                          float grid[N+2] )
{
    // Get the element id for this particular thread
    int i = blockIdx.x*blockDim.x + threadIdx.x + 1;

    // Compute the new grid value for this element
    new_grid[i] = grid[i-1] - 2*grid[i] + grid[i+1];
}
```

# Example - CUDA C (Compute kernel)

```
// Specify the problem size
dim3 blocksPerGrid(N/256,1,1);
dim3 threadsPerBlock(256,1,1);

// Allocate memory on the GPU and copy in the grid
cudaMalloc(&new_gpu_grid, grid_size);
cudaMalloc(&gpu_grid, grid_size);

cudaMemcpy(gpu_grid, cpu_grid, grid_size,  cudaMemcpyHostToDevice);

// Run the compute kernel
do_stuff<<<blocksPerGrid,threadsPerBlock>>> ( float new_gpu_grid[N+2],
                                              float gpu_grid[N+2] );

cudaMemcpy(cpu_grid, new_gpu_grid, grid_size,  cudaMemcpyDeviceToHost);
```

# OpenCL

- Another open standard
- Kernel based - separate code for the GPU
- Explicitly copy data and launch kernels
- Supported in C and C++, with 3rd party wrappers available for other languages
- Supports more than just GPUs (embedded graphics, FPGAs etc…)

# Example - OpenCL (Compute kernel)

```
// Function for each GPU thread to run
__kernel void do_stuff(__global float *new_grid,
                       __global float *grid )
{
    // Get the element id for this particular thread
    int i = get_global_id(0) + 1;

    // Compute the new grid value for this element
    new_grid[i] = grid[i-1] - 2*grid[i] + grid[i+1];
}
```

# Example - OpenCL (Compute kernel)

```
char *do_stuff_src = "
    // Function for each GPU thread to run
    __kernel void do_stuff(__global float *new_grid,
                           __global float *grid )
    {
        // Get the element id for this particular thread
        int i = get_global_id(0) + 1;

        // Compute the new grid value for this element
        new_grid[i] = grid[i-1] - 2*grid[i] + grid[i+1];
    }
";
```

# Example - OpenCL (Compiling the kernel)

```
// Create a program with the kernel source from before
program = clCreateProgramWithSource ( context, 1,
                                       (const char **) &do_stuff_src,
                                       NULL, &err
                                     );

// Build the program
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// Create a kernel called "do_stuff" from the program
*do_stuff = clCreateKernel(program, "do_stuff", &err);
```

# Example - OpenCL (Copying in data)

```
// Allocate the device memory
gpu_grid     = clCreateBuffer( context, CL_MEM_READ_WRITE, grid_size,
                               NULL, &err );
new_gpu_grid = clCreateBuffer( context, CL_MEM_READ_WRITE, grid_size,
                               NULL, &err );

// Copy the grid onto the GPU
clEnqueueWriteBuffer( queue, gpu_grid, CL_TRUE, 0, grid_size, cpu_grid,
                      0, NULL, NULL );
```

# Example - OpenCL (Running the kernel)

```
// Specify the arguments for the kernel
clSetKernelArg(do_stuff, 0, sizeof(cl_mem), &new_gpu_grid);
clSetKernelArg(do_stuff, 1, sizeof(cl_mem), &gpu_grid);

// Run the kernel
clEnqueueNDRangeKernel (queue, do_stuff, 1, NULL, N, 256, 0, NULL, NULL);

// Wait for the GPU to finish its work
clFinish(queue);

// Copy the data back off the GPU
clEnqueueReadBuffer( queue, new_gpu_grid, CL_TRUE, 0, grid_size,
cpu_grid,
                                 0, NULL, NULL );
```

# Using external libraries

- Several libraries can make use of GPUs
- Other people put the effort into optimising algorithms so you don’t have to!
- Examples:
    - CuBLAS - LAPACK on the GPU
    - Tensorflow - Machine learning/AI
    - Thrust - Parallel algorithms

# How to get access to GPUs

# Desktop/workstation cards

- NVIDIA
  - GTX gaming series, okay for single precision calculations
  - Quadro workstation series, some have support for double precision
- AMD have similar offerings in Radeon RX and Radeon Pro
- Intel expected to enter the game mid-2020

# Server / Cloud options

- Viking has 8 NVidia V100 server GPUs
- Regional facilities:
  - JADE has 150+ V100 GPUs
  - Next-gen facilities will increase this capacity

- Amazon, Google and Microsoft cloud services provide VMs with GPUs

# Summary

# Summary

- GPUs provide a cost/energy efficient way of tackling certain types of computational problem
- Maximising time computing and minimizing time transferring data will generally give the best performance
- There are a range of technologies available for programming GPUs, at many levels and with different amounts of control
- There are a range of options available for where you can use GPUs

# Useful links

- Hardware information: https://devblogs.nvidia.com/inside-volta/
- Compilers:
  - PGI for OpenACC / CUDA: https://www.pgroup.com
  - CUDA Python: https://developer.nvidia.com/pycuda
- Libraries:
  - https://developer.nvidia.com/gpu-accelerated-libraries
  - https://www.tensorflow.org/
  - http://thrust.github.io/
- Server/Cloud facilities:
  - https://www.jade.ac.uk/
  - https://aws.amazon.com/ec2/
  - https://azure.microsoft.com