GPU Computing: Introduction to CUDA

Dr Paul Richmond
http://paulrichmond.shef.ac.uk
This lecture

- CUDA Programming Model
- CUDA Device Code
- CUDA Host Code and Memory Management
- CUDA Compilation
Programming a GPU with CUDA

Use GPU to Parallelize Compute-Intensive Functions

Rest of Sequential CPU Code
Simple processing flow

1. Copy input data from CPU memory to GPU memory

Images: Mark Harris (NVIDIA)
Simple processing flow

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
Stream Computing

- Data set decomposed into a **stream** of elements
- A single computational function (**kernel**) operates on each element
  - A **thread** is the execution of a kernel on one data element
- Multiple Streaming Multiprocessor cores can operate on multiple elements in parallel
  - Many parallel threads
- Suitable for **Data Parallel** problems
How does the stream competing principle map to the hardware model?
CUDA Software Model

- Hardware abstracted as a **Grid of Thread Blocks**
  - Blocks map to SMPs
  - Each thread maps onto a CUDA core

- Don’t need to know the hardware characteristics
  - Code is portable across different GPU architectures
CUDA Vector Types

- CUDA Introduces a new dim types. E.g. dim2, dim3, dim4
  - dim3 contains a collection of three integers (X, Y, Z)

    dim3 my_xyz (x_value, y_value, z_value);

- Values are accessed as members

    int x = my_xyz.x;
Special dim3 Vectors

- `threadIdx`
  - The location of a thread within a block. E.g. (2,1,0)

- `blockIdx`
  - The location of a block within a grid. E.g. (1,0,0)

- `blockDim`
  - The dimensions of the blocks. E.g. (3,9,1)

- `gridDim`
  - The dimensions of the grid. E.g. (3,2,1)

*Idx values use zero indices, Dim values represent a size*
Analogy

- Students arrive at halls of residence to check in
  - Rooms allocated in order
- Unfortunately admission rates are down!
  - Only half as many students as rooms
  - Each student can be moved from room $i$ to room $2i$ so that no-one has a neighbour
Serial Solution

- Receptionist performs the following tasks
  1. Asks each student their assigned room number
  2. Works out their new room number
  3. Informs them of their new room number
Parallel Solution

“Everybody check your room number. Multiply it by 2 and go to that room”
CUDA Programming Model
CUDA Device Code
CUDA Host Code and Memory Management
CUDA Compilation
A First CUDA Example

- Serial solution

```c
for (i=0; i<N; i++) {
    result[i] = 2*i;
}
```

- We can parallelise this by assigning each iteration to a CUDA thread!
__global__ void myKernel(int *result) {
    int i = threadIdx.x;
    result[i] = 2*i;
}

- Replace loop with a “kernel”
  - Use __global__ specifier to indicate it is GPU code
- Use threadIdx dim variable to get a unique index
  - Assuming for simplicity we have only one block
  - Equivalent to your door number at CUDA Halls of Residence
CUDA C Example: Host

- Call the kernel by using the CUDA kernel launch syntax
  - `kernel<<<GRID OF BLOCKS, BLOCK OF THREADS>>>(arguments);`

```c
dim3 blocksPerGrid(1,1,1); // use only one block
dim3 threadsPerBlock(N,1,1); // use N threads in the block

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```

![Diagram of GPU with threads and blocks](image-url)
Vector Addition Example

Consider a more interesting example

Vector addition: e.g. $a + b = c$
Vector Addition Example

//Kernel Code
__global__ void vectorAdd(float *a, float *b, float *c)
{
    int i = threadIdx.x;
    c[i] = a[i] + b[i];
}

//Host Code
...
dim3 blocksPerGrid(1,1,1);
dim3 threadsPerBlock(N,1,1); //single block of threads

vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
Only one block will give poor performance

- A block gets allocated to a single SMP!
- Solution: Use multiple blocks

```
dim3 blocksPerGrid(N/8,1,1);  // assumes 8 divides N exactly
dim3 threadsPerBlock(8,1,1);   // 8 threads in the block

myKernel<<<blocksPerGrid, threadsPerBlock>>>(result);
```
Vector Addition Example

The integer \( i \) gives a unique thread index used to access a unique value from the vectors \( a, b \) and \( c \).
A note on block sizes

- Thread block sizes cannot be larger than 1024.
- Max grid size is 2,147,483,647 for 1D.
  - Grid y and z dimensions are limited to 6,5535.

- Block size should ideally be divisible by 32.
  - This is the warp size in which threads are scheduled.
  - Not less than 32 as in our trivial example!

- Varying the block size will result in different performance characteristics.
  - Try incrementing by values of 32 and benchmark.

- Calling a kernel with scalar parameters assumes a 1D grid of thread blocks.
  - E.g. `my_kernel<<<8, 128>>>(arguments);`
Device functions

- Kernels are always prefixed with `__global__`
- To call a function from a kernel the function must be a device function (i.e. it must be compiled for the GPU device)
  - A device function must be prefixed with `__device__`
- A device function is not available from the host
  - Unless it is also prefixed with `__host__`

```c
int increment(int a){ return a + 1; }

__device__ int increment(int a){ return a + 1; }

__device__ __host__ int increment(int a){ return a + 1; }
```

- Host only
- Device only
- Host and device
- CUDA Programming Model
- CUDA Device Code
- CUDA Host Code and Memory Management
- CUDA Compilation
Memory Management

- GPU has separate dedicated memory from the host CPU
- Data accessed in kernels must be on GPU memory
  - Data must be explicitly copied and transferred
- `cudaMalloc()` is used to allocate memory on the GPU
- `cudaFree()` releases memory

```c
float *a;
cudaMalloc(&a, N*sizeof(float));
...
cudaFree(a);
```
Memory Copying

- Once memory has been allocated we need to copy data to it and from it.
- `cudaMemcpy()` transfers memory from the host to device to host and vice versa

```c
cudaMemcpy(array_device, array_host, N*sizeof(float), cudaMemcpyHostToDevice);
```

```c
cudaMemcpy(array_host, array_device, N*sizeof(float), cudaMemcpyDeviceToHost);
```

- First argument is always the destination of transfer
- Transfers are relatively slow and should be minimised where possible
```c
#define N 2048
#define THREADS_PER_BLOCK 128

__global__ void vectorAdd(float *a, float *b, float *c) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    c[i] = a[i] + b[i];
}

int main(void) {
    float *a, *b, *c;          // host copies of a, b, c
    float *d_a, *d_b, *d_c;    // device copies of a, b, c
    int size = N * sizeof(float);

    cudaMalloc((void **)&d_a, size);
    cudaMalloc((void **)&d_b, size);
    cudaMalloc((void **)&d_c, size);

    a = (float *)malloc(size); random_floats(a, N);
    b = (float *)malloc(size); random_floats(b, N);
    c = (float *)malloc(size);

    cudaMemcpy(d_a, a, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, size, cudaMemcpyHostToDevice);

    vectorAdd <<<N / THREADS_PER_BLOCK, THREADS_PER_BLOCK >>>(d_a, d_b, d_c);

    cudaMemcpy(c, d_c, size, cudaMemcpyDeviceToHost);

    free(a); free(b); free(c);
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);
    return 0;
}
```
Device Synchronisation

Kernel calls are non-blocking
  - Host continues after kernel launch
  - Overlaps CPU and GPU execution

`cudaDeviceSynchronise()` call be called from the host to block until GPU kernels have completed

```c
vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(a, b, c);
//do work on host (that doesn’t depend on c)
cudaDeviceSynchronise(); //wait for kernel to finish
```

Standard `cudaMemcpy` calls are blocking
  - Non-blocking variants exist
CUDA Programming Model
CUDA Device Code
CUDA Host Code and Memory Management
CUDA Compilation
Compiling a CUDA program

- CUDA C Code is compiled using `nvcc` e.g.
- Will compile host AND device code to produce an executable

```
nvcc -o example example.cu
```
**Compilation**

- CUDA source file (*.cu) are compiled by `nvcc`
- An existing `cuda.rules` file creates property page for CUDA source files
  - Configures `nvcc` in the same way as configuring the C compiler
  - Options such as optimisation and include directories can be inherited from project defaults
- C and C++ files are compiled with `gcc`

```
.c / .cpp
```
```
.cu
```
```
Host functions
```
```
CUDA kernels
```
```
Host .obj
```
```
Host .obj
```
```
CUDA .obj
```
```
l linker
```
```
Executable
```
Device Versions

- Different generations of NVIDIA hardware have different compatibility
  - These are classified by **CUDA compute versions**
- Compilation normally builds for CUDA compute version 2
  - This can be changed by passing `--arch` to `nvcc`
  - Default value is “compute_20,sm_20”
  - E.g. `nvcc source.cu --arch=compute_20,sm_20`
  - Any hardware with greater than the compiled compute version can execute the code (backwards compatibility)
- You can build for multiple versions using separator
  - E.g. “compute_20,sm_20;compute_30,sm_30;compute_35,sm_35”
  - This will increase build time and execution file size
  - Runtime will select the best version for your hardware

[https://en.wikipedia.org/wiki/CUDA#Supported_GPUs](https://en.wikipedia.org/wiki/CUDA#Supported_GPUs)
Summary

- CUDA is a C like programming language
- Programming a GPU requires moving data to and from the device
- Parallel regions are executed using Kernel
- Kernels require high levels of parallelism
  - Exposed as many threads grouped into blocks
  - Thread blocks are mapped to SMs
- Host and device code are compiled separately and linked into a single executable
Hands on Session

- Navigate to [https://nvlabs.qwiklab.com](https://nvlabs.qwiklab.com)
  - Same account as for Alison Lowndes labs
  - Class name: **Deep Learning, Big Data and Big Compute Camp (Rabat)**
  - SSH is blocked by the proxy!
    - Use the link for the Web-Based SSH Client.

- Get the lab handout from
  - [http://paulrichmond.shef.ac.uk/teaching/NVIDIA/rabat/](http://paulrichmond.shef.ac.uk/teaching/NVIDIA/rabat/)
  - or