XJCO3221 Parallel Computation

Peter Jimack

University of Leeds

Lecture 15: GPU threads and kernels

Previous lecture

In the last lecture we started looking at General Purpose GPU programming, or GPGPU:

- **Device** contains a number of **SIMD processors**, each containing some number of **cores**.
- Thread scheduling is performed **in hardware**.
- Programmable using **OpenCL** (this course), CUDA, and others.

- Device discovery performed at run time (*cf.* the `displayDevices.c` example).

Today's lecture

Today we will see how to perform vector addition on a GPU:

- **Communicating** data between the device (GPU) and the host (CPU) using the **command queue**.
- Compiling and executing **kernels** on the device.
- **Work items** are the basic unit of concurrency.
- Arranged into **work groups** for **scalability**.
- How to set the work group size.

Vector addition

Code on Minerva: `vectorAddition.c`, `vectorAddition.cl` and `helper.h`

Once again use **vector addition** as our first worked example:

$$\mathbf{c} = \mathbf{a} + \mathbf{b} \quad \text{or} \quad c_i = a_i + b_i \,,\; i = 1 \ldots N.$$

In serial code:

```
for( i=0; i<N; i++ )
  c[i] = a[i] + b[i];
```


where vectors **a**, **b** and **c** all have N elements (*as before, mathematical and computer indexing differ by one*).

This is a **map**/**data parallel** problem with no **data dependencies**.

Host and device

The CPU is the **host**, and the GPU is the **device**:

Host	CPU
Device	GPU, accelerator, FPGA, . . .

Assume that CPU and GPU memory are **separate**[1].

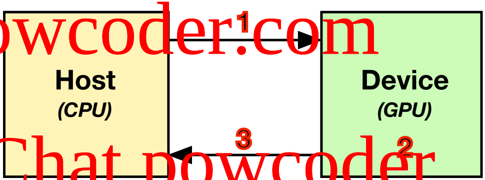
If the initial data is only accessible to the CPU, must **transfer** to the GPU to perform the calculations, then **transfer** the result back to the CPU.

- This requires **explicit communication**, somewhat similar to the distributed memory model.

[1] Some modern GPUs support **unified memory** — see next lecture.

Typical program structure

1. **Send** problem data from host to the device.
2. **Perform** calculations on the device.
3. **Return** results from device to the host.



Contexts and command queues

Recall from last lecture that to use OpenCL we first need to:

1. Identify the **platform** and a suitable **device**.
2. For each device, initialise a **context** and **command queue**.

The routine `simpleOpenContext_GPU()` in `helper.h` helps:

```
cl_device_id device;
cl_context context = simpleOpenContext_GPU(&device);

cl_int status;
cl_command_queue queue = clCreateCommandQueue(context,
    device,0,&status);
...     // Use the GPU.
clReleaseCommandQueue(queue);
clReleaseContext(context);
```

Device memory allocation

Suppose arrays a, b and c initialised on the **host**:

```
float *host_a = (float*) malloc(N*sizeof(float));
```

Similar for `host_b`, `host_c`.

Can allocate **device memory for this array and copy from the host array** using `clCreateBuffer`:


```
cl_mem device_a = clCreateBuffer(
  context,
  CL_MEM_READ_ONLY|CL_MEM_COPY_HOST_PTR, // Flags.
  N*sizeof(float),           // Size in bytes.
  host_a,                    // Copy from this host array.
  &status                    // Error status.
);
```

Similar for `device_b`, `device_c`.

clCreateBuffer() usage

- The context has been initialised for the GPU.
- The flag CL_MEM_READ_ONLY refers to how the **device** accesses the memory.
  - Specifying 'read only' allows the runtime system to optimise execution — see next lecture.
- The flag CL_MEM_COPY_HOST_PTR automatically copies from an existing **host** array (the 4^{th} argument).
- For device_c, where no host data (yet) exists, the flag is just CL_MEM_WRITE_ONLY and the 4^{th} argument is NULL.
- status is set to CL_SUCCESS if the operation was successful, otherwise some other error code.

GPU kernel

Definition

Kernels are functions that execute on the device.

Each thread within the SIMD cores executes the kernel.

Use standard C syntax:

```
__kernel
void vectorAdd(__global float *a, __global float *b,
    __global float *c)
{
  int gid = get_global_id(0);
  c[gid] = a[gid] + b[gid];
}
```

OpenCL kernels

- All kernels[1] preceded with `__kernel`.
- Must return `void` — otherwise which thread's return value would be returned to the host?

- `__global` refers to the device memory we have just allocated.
  - More on this next lecture.
- `get_global_id()` returns the (global) index for 'this' thread.
  - For this problem it is the index of the vector.
  - See later.

[1] CUDA kernels are preceded `__global__` (if they are callable by the host).

Building a kernel

OpenCL kernels are compiled **at run time** (of the C code).

- Allows optimisation for the device that executes it.

Requires a series of API calls. Typically:

1. Start with the program as a `char*` **string** (typically read from file ending in `.cl`).
2. Create the **program** for the context with `clCreateProgramWithSource()`.
3. **Build** (compile and link) using `clBuildProgram()`.
4. Create a **kernel** using `clCreateKernel()`.

Building a kernel with `helper.h`

To simplify this process, the file `helper.h` contains the routine `compileKernelFromFile()`.

For this vector addition example:


```
cl_kernel kernel = compileKernelFromFile(
  "vectorAddition.cl",     // File with kernel code.
  "vectorAdd",             // Name of function.
  context,                 // Same as before.
  device                   // Same as before.
);

... // Use kernel.

clReleaseKernel(kernel);
```


It also includes some basic error handling.

Setting kernel arguments

Each kernel argument must be set by using clSetKernelArg().


```
status = clSetKernelArg(
  kernel,            // The kernel object.
  0,                 // The argument number.
  sizeof(cl_mem),    // The size of the argument.
  &device_a          // The value.
);
```


This is repeated for argument 1 (→device_b) and argument 2 (→device_c) for the vector addition example.

Starting a kernel in OpenCL[1]

To start a kernel, you place it on the **command queue** using `clEnqueueNDRangeKernel()`:


```
// Will cover this later.
size_t indexSpaceSize[1], workGroupSize[1];
indexSpaceSize[0] = N;
workGroupSize [0] = 128;

// Place the kernel onto the command queue.
status = clEnqueueNDRangeKernel(queue,kernel,1,NULL,
    indexSpaceSize,workGroupSize,0,NULL,NULL);
```


There are many arguments; we will cover some later.

Note that `size_t` is an **unsigned integer**.

[1]In CUDA: `kernel<<<workGroupSize,indexSpaceSize>>>(...)`.

Copying data between device and host[1]

To get the result (device_c) back to the host (host_c), enqueue a **read buffer** command:

```
status = clEnqueueReadBuffer(
  queue,              // The command queue.
  device_c,           // Device memory.
  CL_TRUE,            // Blocking
  0,                  // Offset; must be zero.
  N*sizeof(float),    // Data size.
  host_c,             // Host memory.
  0, NULL, NULL       // Events; ignore for now.
);
```

Note this is a **blocking** communication call - **it will not return until the copy has finished** — like `MPI_Send()`/`MPI_Recv()`.

[1] In CUDA: `cudaMemcpy(...,cudaMemcpyDeviceToHost)`.

Copying data from host to device[1]

If we had *not* used CL_MEM_COPY_HOST_PTR earlier, we would need two calls to clEnqueueWriteBuffer():

```
status = clEnqueueWriteBuffer(queue,device_a,CL_FALSE,0,N*sizeof(float),host_a,0,NULL,NULL);
status = clEnqueueWriteBuffer(queue,device_b,CL_FALSE,0,N*sizeof(float),host_b,0,NULL,NULL);
```


- Copies **from** host **to** device.
- CL_FALSE used for **non-blocking** communication.
- The device memory **always** comes before host memory in the argument list.

[1] In CUDA: `cudaMemcpy(...,cudaMemcpyHostToDevice)`.

Work items

Definition

The **work item** is the unit of concurrent execution. It usually maps onto a single **hardware thread**.

As thread scheduling on a GPU is implemented in hardware, there is (essentially) **no overhead** in launching/destroying threads.

- No problem **oversubscribing**, *i.e.* issuing more threads than there are physical cores.

Normally issue as many threads as the problem requires.

Work item hierarchy

To remain **scalable**, the hardware does not allow communication (including synchronisation) between *all* threads at once.

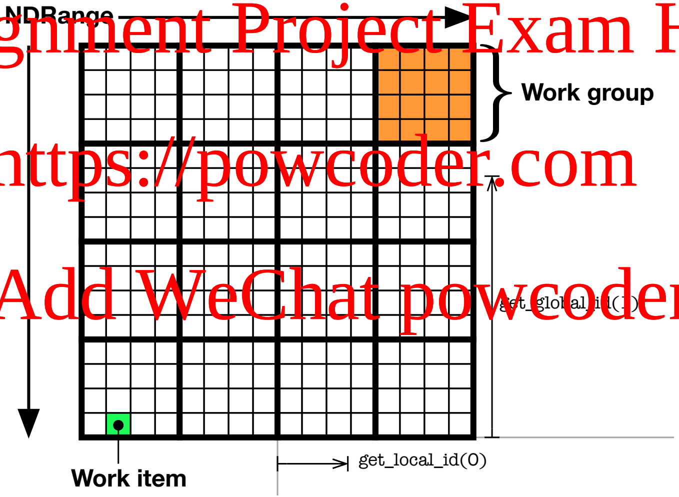
Instead employs a **hierarchy**:

- **Work items** belong to **work groups**[1].
- Communication (including synchronisation) only possible **within** a work group.

The full range of all threads is called **NDRange** in OpenCL, for *n*-dimensional range[2].

[1] *Threads* and *thread blocks* in CUDA.
[2] *Grid* in CUDA.

Hierarchy of work items: 2D example



Specifying the *n*-dimensional range NDRange

The NDRange must be 1, 2 or 3 dimensions.

A 2-dimensional example:


```
size_t globalSize    [2] = {X,Y };
size_t workGroupSize [2] = {8,16};

status = clEnqueueNDRangeKernel(queue,kernel,2,0,
    globalSize,workGroupSize,0,NULL,NULL);
```


- Launches X*Y kernels in total (one per work item).
- In work groups of 8*16.

OpenCL 2.0 allows X and Y to be arbitrary, but in earlier versions they must be multiples of the work group size (8 and 16 here).

Once in a kernel, can get the **global** indices using get_global_id(). For this 2D example:


```
get_global_id(0);  // Varies from 0 to X-1 inc.
get_global_id(1);  // Varies from 0 to Y-1 inc.
```

Similarly can get the indices **within** the work group using get_local_id():


```
get_local_id(0);  // Varies from 0 to 7 inc.
get_local_id(1);  // Varies from 0 to 15 inc.
```


Can also get the number of work items in a group or in the NDRange using get_local_size() and get_global_size():

```
get_local_size (1);  // Returns 16.
get_global_size(0);  // Returns X.
```

What group size to use?

Devices have a **maximum work group size** they can support. This can be determined at run time as follows:

```
size_t maxWorkItems;
clGetDeviceInfo(device,CL_DEVICE_MAX_WORK_GROUP_SIZE,
    sizeof(size_t),&maxWorkItems,NULL);
```


Note this refers to **all** items in a group (*i.e.* 8*16=128).

Other factors may suggest using work group sizes less than this maximum.

- We will look at one of these next time.

Passing NULL as the work group argument lets OpenCL try to determine a suitable size **automatically**.

Summary and next lecture

Today we have looked at a complete GPGPU solution:

- **Communication** between **host** and **device**.
- **Kernels** that execute on the device.
- Basic unit of concurrency is the **work item**.
- Group into **work groups**, within which communication is possible.

Next time we will look at the different memory types on a GPU.