University of Leeds

Lecture 15: GPU threads and k

Previous lecture

In the last lecture we started looking at General Purpose GPU prog

-
- Thread scheduling is performed **in hardware**.
- Programmable using **OpenCL** (th

  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:

-
-
- **Work items** are the basic unit of concurr
- Arranged into **work groups** for s
- 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:

In seri

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


where vectors **a**, **b** and **c** all have N el

mathematical and computer indexing differ by o

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

Assu

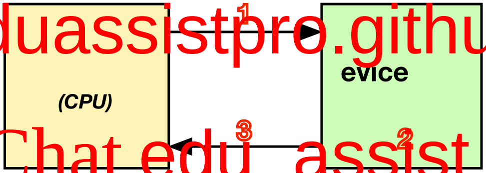
If the initial data is only accessible to the CPU, must to the GPU to perform the calculations, then to the CPU.

- 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
t
2. 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. **eue**.

The r s:


```
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));
```

Simil

Can a

host a

```
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 flag CL_MEM_COPY_HOST_PTR existing **host** array (the 4^{th} argu

- For device_c, where no host data (yet) ex 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.

Use st


```
__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`.
-

- __
    -
- `get_global_id()` returns the (globa
    - 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.

Requ[illegible]

1. Start with the program as a `char*` **string** (typically read from file ending in `.cl`).
2. Create the **program** for the context w[illegible] `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 th

```
cl_k
  "v
  "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()`.


```
statu  = c
  ke                     // The kernel obje
  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_   i
index
workG

// 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 lat

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(
  qu                    // The command queue
  de                    // Devic
  CL                    // Blockin
  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():

```
statu   = c

stat    = c
```


- Copies **from** host **to** device.
- CL_FALSE used for **non-blocking**
- The device memory **always** come
  argument list.

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

Work items

Definition

The **work item** is the unit of concurrent execution. It usually

As thr
is (essentially) **no overhead** in launchin

- No problem **oversubscribing**, *i.e.*
  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.

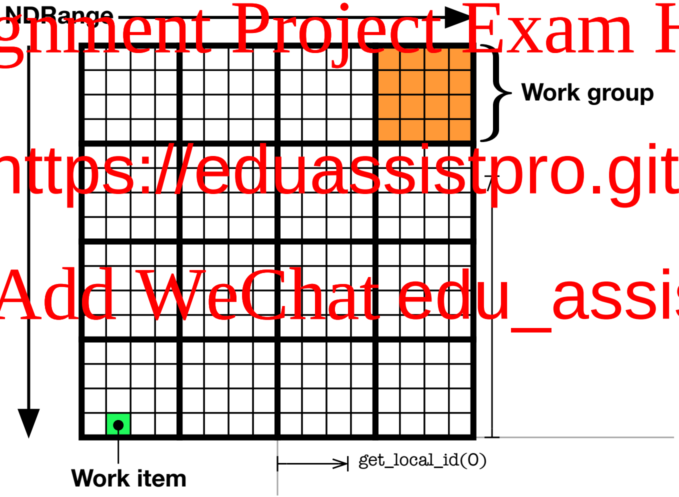
Inste

- Communication (including synchronisation) only possible **within** a work group.

The full range of all threads is called *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_   g
size    w

statu   = c
    globalSize,workGroupSize,0,NULL,NULL);
```


- Launches X*Y kernels in total (one per w
- 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.
```

Simil
get_


```
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 th 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;
clGet
```


Note t

Other factors may suggest using work group size I 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:

-
-
-
- Group into **work groups**, within whi
possible.

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