MANCHESTER 1824

The University of Manchester

# Introduction to OpenCL Exercises

June 2012

George Leaver
IT Services
george.leaver@manchester.ac.uk

MANCHESTER 1824

The University of Manchester

# Exercise 0

Log in and "device query"

MANCHESTER 1824

The University of Manchester

# Exercise 0 - Log in to the CSF

1. From your linux desktop

   **`ssh -X username@csf.itservices.manchester.ac.uk`**

   (uppercase **`X`**. Replace *username* with your central username and use you central password. Answer **`yes`** if asked a question).

2. If new to CSF, change your password now!

   **`yppasswd`**

   (follow instructions - enter old p/w then new one twice. **Remember it!**)

3. Download the training exercises (redo to update if already done on other courses)

   **`module load training`**

   **`cd training/ROPENCL/exercises/`**

   if not on CSF do

   ```
   svn checkout http://kato.rcs.manchester.ac.uk/svnpublic/training/ROPENCL
   ```

- You are now logged in to the *login node*. You can compile code here. You cannot run your OpenCL code here – there are no GPUs in the login node. You must submit your programs to the batch queue so that it can run them on a backend node containing a GPU. The backend nodes are much more powerful than the login node and so you only ever do lightweight things on the login node like compiling your code or editing files.
- If you're an Emacs user: Add this to your **`.emacs`** file to syntax-highlight .cl files

  ```
  (setq auto-mode-alist (cons '("\\.cl$" . c-mode) auto-mode-alist))
  ```

Exercise 0 continued on next page...

# Exercise 0 – "device Query"

- Most OpenCL vendors supply a "device query" program to report statistics about the GPU. We'll run the program on one of the backend nodes in CSF containing a GPU.

1. Set up your shell variables for OpenCL

   **`module load libs/cuda/`**

   (loads version 4.0.17 of the CUDA toolkit which includes their OpenCL files)

2. Run the oclDeviceQuery app to get info about the GPUs by submitting a job to the queue.

   **`qsub ocldq.sh`**

   (Your job will run on a backend node containing a GPU soon. Look in the **`ocldq.sh`** file to see exactly which program it is running and how. You can edit it with **`gedit ocldq.sh &`**).

3. Check to see if your job has finished by looking at the job queue

   **`qstat`**

   (there will be a **`qw`** status if it is waiting in the queue or **`r`** if it is running. When finished the queue will be empty)

4. When the job finishes you will have an output file name **`ocldq.o`*`NNNNN`*** - have a look at it.

   **`cat ocldq.o`*`NNNNN`*** or **`gedit ocldq.o`*`NNNNN`***

   (replace *NNNNN* with the number returned for your job by **`qsub`**)

5. Look through the returned info. Each property is named. You should be able to answer the following:
   1. What is the device name?
   2. How many compute units are there?
   3. How many cores are there?
   4. How much global memory is there?
   5. How much local memory is there?

MANCHESTER 1824

The University of Manchester

# Exercise 1

Write a kernel to multiply two vectors (element-wise)

# Exercise 1 – Vector Multiplication (part 1)

- Write a simple kernel to multiply two vectors together (element-wise). For now, the host-code has been written for you – you will run **ex1** after writing the kernel. It will read your kernel source file, compile it and run it. The two input vectors will be initialized with random numbers.

1. Edit the kernel source file **ex1.cl**. Add your kernel, which must be called **ex1kernel**. The **ex1.cl** file contains details of the parameters required by the kernel.

```
gedit ex1.cl &
```

2. Submit your job to the queue and then check the .o*NNNNN* file for output (or for errors).

```
qsub ex1.sh            # submit a job-script to the queue
qstat                  # to check when job has finished
cat ex1.oNNNNN         # to look at the output file
```

3. If there are mistakes in your kernel they'll be reported in the **Build Log** section of the output file.
4. The **ex1** program can also do the computation on the CPU (serially) and compare its results to your kernel's results, reporting if they match. Add the **-c** flag to **ex1** by editing the job script (**gedit ex1.sh &**). Are your results correct?

Exercise 1 continued on next page...

# Exercise 1 (part 2)

- The **`ex1`** program accepts the following command-line flags allowing you to experiment with OpenCL settings (note the flags are not common to all OpenCL programs – we wrote them for the **`ex1`** binary!)

| | |
|---|---|
| -n *number* | Specify a new vector length. Default is 32*1024*1024 (~33million) if flag not given. |
| -g *number* | Specify the global-work-size, i.e., number of work-items (threads) to launch. If flag not given will default to the vector length (i.e., one work-item for each element of the vectors). |
| -l *number* | Specify the local-work-size, i.e., number of work-items in a work-group. Remember, this must divide evenly the global-work-size. If flag not given, or you specify zero, OpenCL will choose a default. |
| -c | Turn on CPU/GPU results comparison by doing the computation serially on the CPU (does not use OpenCL) and then reporting if a difference occurs. |
| -p | Turn on profiling in the OpenCL command queue and report the execution time of the kernel |

1. Edit **`ex1.sh`** to add flags. Try running the kernel with different local-work-sizes (16, 32, 64, 128, 256, 512). What are resulting kernel execution times?
2. If you wrote your kernel correctly first time, introduce a syntax error and have a look at the Build Log section of the **`ex1.o`*`NNNNN`*** output file. It shows any error messages generated by the OpenCL kernel compiler. You'll need to get used to reading these errors!

Exercise 1 continued on next page...

# Exercise 1 (part 3)

- The lecture notes showed how a kernel can process more than one 'data item' (using a loop within the kernel). This means you can specify fewer work-items than there are elements in input vectors (but a work-item does more work). See slides 73,74.

1. Write another kernel in **`ex1.cl`** to handle this situation. You'll need a loop that strides over the elements. What stride length should you use?
   - Rename the first kernel (e.g. to **`ex1kernelpart1`**) and name your new kernel **`ex1kernel`**. Remember, the **`ex1`** program will only run a kernel named **`ex1kernel`** but you can have more than one kernel in the source .cl file. They will all get compiled but you don't need to execute them all.
2. To run the new kernel edit the **`ex1.sh`** file to specify the **`-g number`** flag to set the global-work-size. You want to set it smaller than the number of elements in the vectors so that the work-items have to process more than one element.
3. Run with the same *local-work-sizes* as before. Check that your results are correct. What are the kernel execution times?
4. Question: Does using fewer work-items but asking each work-item to do more work improve performance in this example?

MANCHESTER 1824

The University of Manchester

# Exercise 2

Host code

# Exercise 2 - Host Code

- You are now going to develop the host code for your kernel from Exercise 1.

1. Open **`ex2.c`** in an editor.

```
gedit ex2.c &
```

- Half the code has been written. The variables have been declared and a function (provided by us, not OpenCL) called **`oclHelper()`** will do a lot of the set up for you. It creates the following OpenCL objects as discussed in the lectures: the platform, device, program, compiled program, and kernel. You need to add the code to do the stages 8-13 as discussed in the lectures. They are:
  - allocate OpenCL memory buffers (on the device) for input and output arrays
  - write data from host-to-device
  - set kernel args
  - execute the kernel
  - read data back from the device
  - cleanup your device objects

Exercise 2 continued on next page...

# Exercise 2 - Host Code

- Hint: Look at the lecture notes. Most of the code is there (although you should use the variable names found in **`ex2.c`** – they might be different on the slides.)
- The comments in the host code are there to help. If you simply start coding without reading what has already been provided in **`ex2.c`** you'll find this a much harder exercise!

1. You should copy your kernel from exercise 1 to exercise 2 using the following

   ```
   cp ex1.cl ex2.cl
   ```

2. Then edit the **`ex2.cl`** file and rename you kernel to be **`ex2kernel`**
3. Compile the code using

   ```
   make ex2
   ```

   (it compiles using: `gcc -o ex2 ex2.c -I. -I$CUDA_HOME/include oclHelper.o -L. -lOpenCL`)
4. Run the code using the following qsub command. **`ex2`** won't do very much until you add your own code – check **`ex2.o`*`NNNNN`***

   ```
   qsub ex2.sh
   ```

5. Complete the host code and ensure you get the same results as in exercise 1. The **`ex2.c`** file indicates where you should add your code.

Exercise 2 continued on next page...

# Exercise 2 - Host Code

- Optional extra. If you finish ex2 in good time, try writing the code to do the setup currently performed by the `oclHelper()` function.
  - The first arg of `oclHelper()` tells it which stages of the set up to do. Initially it will do everything (platform, device, program, build program, extract kernel).
  - You can switch off these tasks to allow your own code to do the work. Once you've created an object (e.g., the platform), pass it to `oclHelper()` so it can do the rest. You'll need to modify the first arg's value. For example:

```
cl_platform platform_id;          // We'll set this up, not oclHelper()
SetupFlags  setup_flags;          // See oclHelper.h for valid flags

// Do the platform set up
err = cl...( ..., &platform_id, ...);

// Let oclSetup() do the remaining setup. It will use our platform_id.
setup_flags = SETUP_ALL - SETUP_PLATFORM;
oclHelper( setup_flags, &platform_id, &device_id, &context, ... );
```

  - **Look at the top of the `oclHelper.h` file for the flag values**

MANCHESTER 1824

The University of Manchester

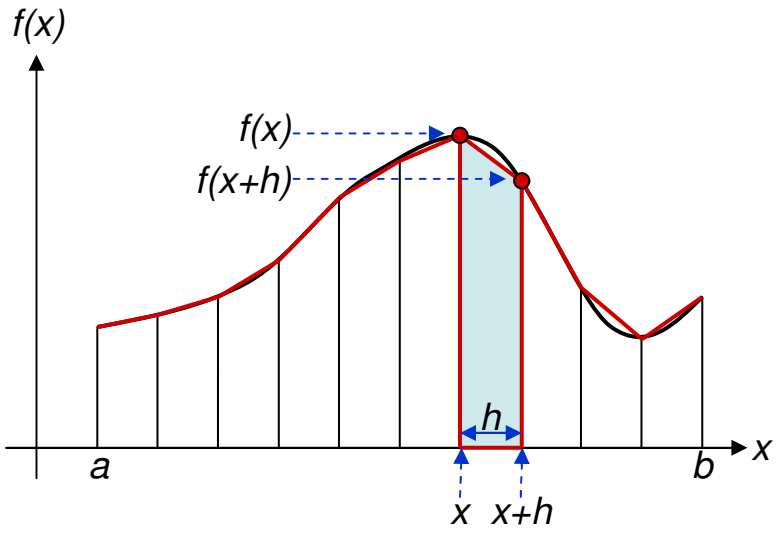
# Exercise 3

Integration using trapezium rule
Double precision

# Exercise 3 – Trapezium Rule Integration

- Approximate the integration of a function using the trapezium rule
  - Divide in to *N* trapezium shaped "panels"
  - Area of trapezium = *½ (b-a)x(f(a)+f(b))*
  - Sum the trapezium areas

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2}(f(a)+f(b))$$
$$\approx \frac{h}{2}\sum_{k=0}^{k=N-1}(f(x_k)+f(x_{k+1}))$$
where $h = \frac{(b-a)}{N}$

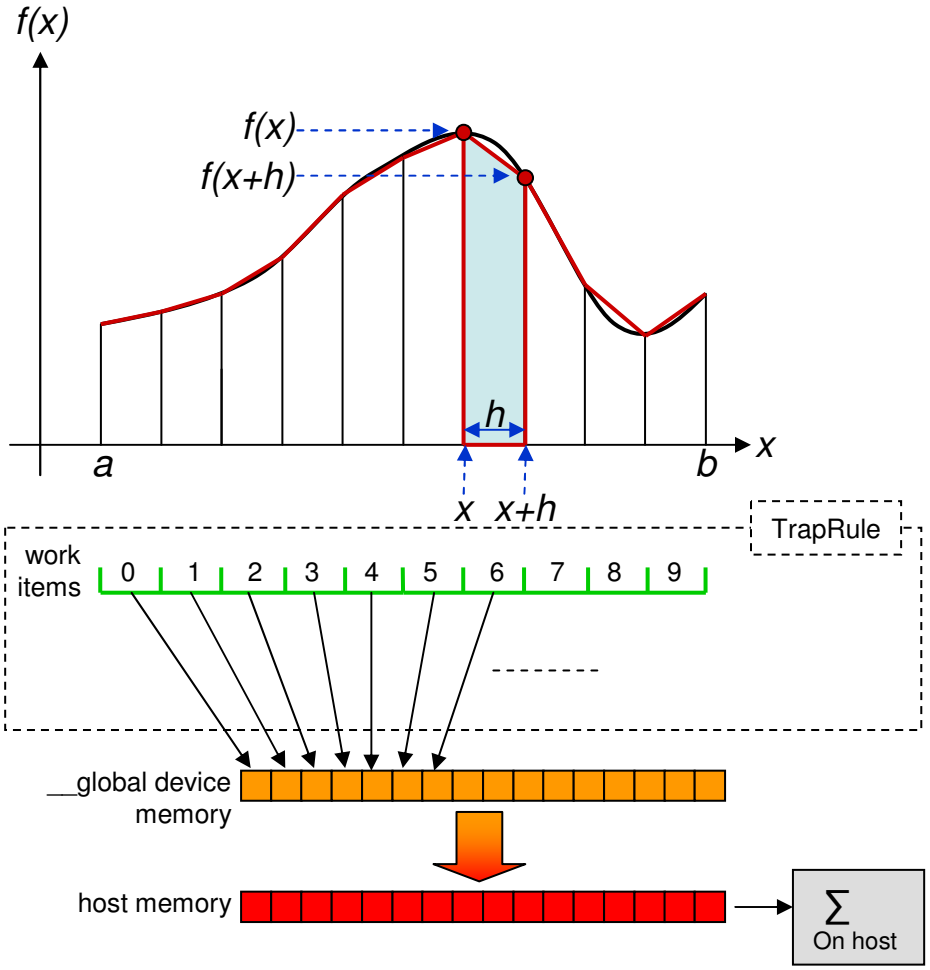



- One work-item for each trapezium
- Question: how do we sum the areas?
  1. Write each area to global memory and sum them on the host (inefficient!)
  2. OR form work-groups of work-items and sum each work group on the device. Write partial sums to global memory and sum them on the host (a little better)
  3. OR as above but sum the partial sums on the device to give the final answer (best)
- 2 & 3 require use of __local memory

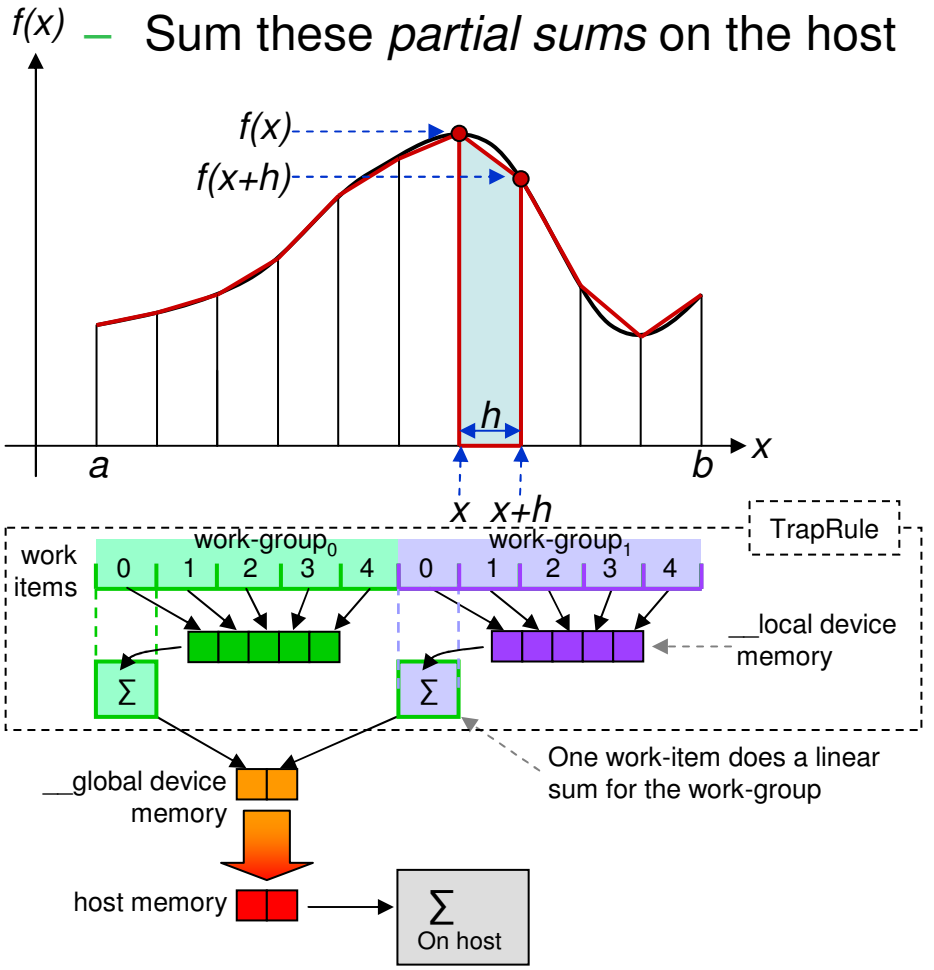
Exercise 3 continued on next page...

# Exercise 3 –Trapezium Rule Integration

- Method 1: Sum all trapezium areas on the host

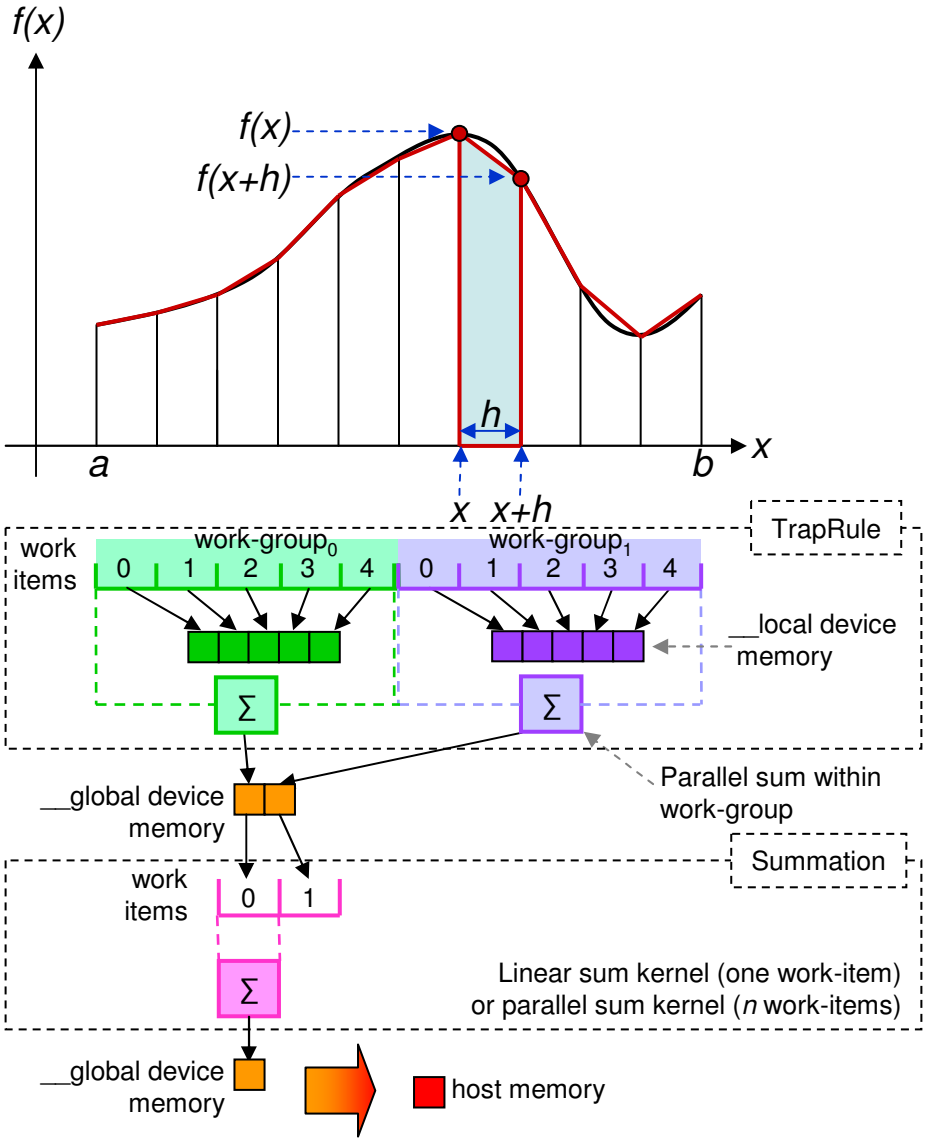



- Method 2: Form partial sums of trapezium areas on the device
  - Sum these *partial sums* on the host




Exercise 3 continued on next page...

# Exercise 3 –Trapezium Rule Integration




- Method 3: Replace the work-group linear sum with a parallel sum (see slides)
- Method 4: Replace the host's linear sum with a kernel (linear/II$^{e}$ sum)
- Other tasks:
  1. Use double-precision in this exercise
  2. Time your kernel using an event (wait for ex4).

Exercise 3 continued on next page...

MANCHESTER 1824

The University of Manchester

# Exercise 3 – Trapezium Rule Integration

- Develop the various trapezium rule methods starting with the simplest (1) and adding to it to form methods (2)-(4). You'll need to complete the host code and the kernel code.
- The comments in the host code are there to help. If you simply start coding without reading what has already been provided in **`ex3.c`** you'll find this a much harder exercise!

1. Open **`ex3.c`** and **`ex3.cl`** in an editor

   ```
   gedit ex3.c ex3.cl &
   ```

2. Write your host and kernel code using double precision. Look in the notes to see how to enable double precision in kernels. Use the **`double`** data type (not **`float`**) in your kernel and (and host) code.
3. Add similar host code from the **`ex2.c`** file. Note there is **no** array to transfer *to* the device, only the results array (e.g., trapezium areas or partial sums) to read back *from* the device. Work out how big the array should be depending on the method. You will also need to specify the **`__local`** memory array size in method 2 onwards.
4. Compile the code using

   ```
   make ex3
   ```

   (it compiles using: **`gcc -o ex3 ex3.c -I. -I$CUDA_HOME/include oclHelper.o -L. -lOpenCL`**)
5. Run the code using the following qsub command. **`ex3`** won't do very much until you add your own code – check **`ex3.o`*`NNNNN`***

   ```
   qsub ex3.sh
   ```

6. Use the **`-n`** ***`number`*** number and **`-l`** ***`number`*** command-line switches on **`ex2`** to change the number of trapeziums (**`-n`**) and the local-work-size (i.e., work-group-size) (**`-l`**) Note: letter 'el' not number 1.
   - With the example integrand function your code should give an answer of 15687500.00

MANCHESTER 1824

The University of Manchester

# Exercise 4

Events for timing and device info

# Exercise 4

- Using the existing exercise codes (no need to rename or copy the files). Add `events` to
  - get the kernel execution times (use Exercise 3 for this)
  - get the data transfer times (use Exercise 2 for this)
    - *Calculate the host-to-device bandwidth (GB/s)*
- Query the device for the maximum allowed local work-group sizes
- Add `"-cl-nv-verbose"` to the build options (pre-processor flags) to see what the Nvidia OpenCL compiler reports.