School of Computing, University of Leeds

COMP3221 Parallel computation

Worksheet 3: GPGPU programming with OpenCL

This worksheet gives examples to explore general purpose GPU programming OpenCL. The lecture most relevant to each question is given next to the question number.

Compiling and running OpenCL programs

For school machines, please read the instructions in Lecture 14 about how to install CUDA.

Once the module has been loaded, compile using Nvidia's `nvcc` compiler with the `-lOpenCL` option:

```
nvcc -lOpenCL -o <executable> <source>.c
```

Note that `nvcc` does not support the usual C-compiler options (`-Wall` *etc.*).

On a Mac use the `OpenCL` framework:

```
gcc -framework OpenCL -Wall -o <executable> <source>.c
```

For both cases, execute as normal:

```
./<executable> [any command line arguments]
```

If you have problems when executing, (a) try a different version of CUDA (`module avail` to see what versions are available), and/or (b) try another machine – those at the back of Bragg 2.05 seem to work. If the problem persists, post a message on the `Teams` page for this module, explaining what the problem is and what you have tried to do to resolve it.

Questions

1. *[Lecture 14]* Download `displayDevices.c` from the Minerva page for Lecture 14 and compile and execute as per the instructions above. Do you understand what is output? Note that only OpenCL-compliant devices are reported. You may also look at the source, but don't worry too much about platforms, contexts *etc.* as code will be provided that handles this for you for subsequent questions.

2. *[Lecture 15]* Download the codes `vectorAddition.c`, `vectorAddition.cl` and `helper.h` from the Minerva page for Lecture 15, compile for your system (instructions given above and in Lecture 14) and execute. Now change the kernel in `vectorAddition.cl` to perform **subtraction** instead, $\mathbf{c} = \mathbf{a} - \mathbf{b}$, and check the answer is correct. You should also change the part of the code that checks the answer.

Look closely at where the work group size is defined in the code. Confirm that replacing the work group size argument with `NULL` when enqueueing the kernel still results in correct operation. Now use the maximum group size supported by your device, and determined at runtime using `clGetDeviceInfo()` as explained towards the end of the lecture. Again ensure the code functions correctly.

3. *[Lecture 16]* Download the codes `registerOverflow.c`, `registerOverflow.c` and `helper.h` from the Minerva page for Lecture 16, and compile as normal. This example uses a 2-dimensional NDRange to perform some arbitrary calculations on an array. The complexity of the calculation, and therefore the amount of private memory required, is specified by `N` which is defined near the start of `registerOverflow.cl`. The code also times the kernel execution.

   Try varying `N` to see if there is a single value at which the performance *suddenly* worsens. The effect is device-dependent and not always clear - for instance, on one School machine I found a slight worsening of performance between $N = 15$ and $N = 16$, but a much more sudden jump in execution times between $N = 32$ and $N = 33$ on my laptop. Try and see what you find for your device. Is what you see consistent with exceeding the available register memory for your device?

4. *[Lecture 17]* Download the codes `workGroupReduction.c`, `workGroupReduction.cl` and `helper.h` from the Minerva page for Lecture 17. As provided, this calculates the vector product with no synchronisation during the reduction stage. Compile and execute using the instructions given at the head of the `.c` file. Can you generate answers that differ from the correct answer shown? It is more likely to fail for large vector size `N`, but this is limited by your device. Is the error the same every time you execute the code?

   In the C code, change the kernel name from `reduceNoSync` to `reduceWithSync` (both are given in `workGroupReduction.cl`). Does the code now always work as expected?

5. *[Lecture 18]* Download the codes `histogram.c`, `histogram.cl` and `helper.h` from the Minerva page for Lecture 18. As provided, this constructs a histogram from procedurally-generated data and checks the result against a serial calculation; however, the kernel does not use atomics, and so undercounts the correct values. Change the kernel to instead use atomic instructions as covered in the lecture, and confirm that it now gives the correct answer.

   Note that the code also times the kernel. Change the kernel again to use a local histogram per work group, again following the lecture notes, and see if it does indeed improve the performance. Note that you will also need to add a fourth kernel argument to specify the local memory to use in the kernel; see Lectures 16 and 17 for examples on how to do this.

6. *[Lecture 19]* Look at the task graph in Fig. 1, in which each task takes equal time to complete. Identify the critical path, and calculate its work and span. Estimate the maximum speedup according to the work-span model.

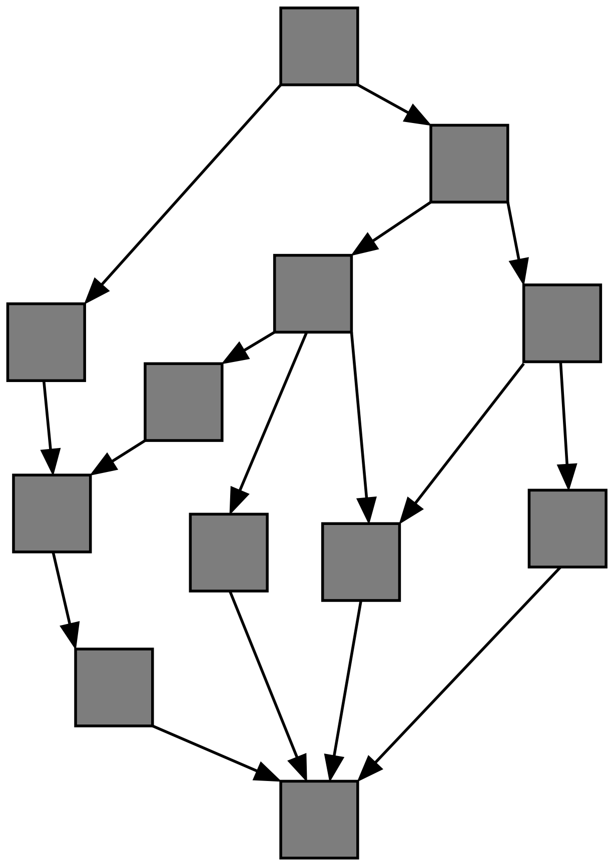

Figure 1: Task graph for Question 19. Grey squares refer to tasks, arrows to dependencies.