# OpenCL Optimisation Exercise

In this exercise, we will look at OpenCL which is relavant for C programmers. There is no Fortran equivalent.



## Introduction

This exercise involves optimising a simple image-processing algorithm using OpenCL. It is an iterative reverse-edge-detection code that, given a data file containing the output from running a very simple edge detector on an image, reconstructs the original source image. You will be given OpenCL source files containing all of the necessary support code and a working, but slow, GPU implementation of the algorithm. Your task is to apply various optimisations to the code to improve performance, as described below.

### Algorithm

You will be given a data file that represents the output from a very simple edge-detection algorithm applied to a greyscale image of size $M \times N$. The edge pixel values are constructed from the image using

$$edge_{i,j} = image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - 4\ image_{i,j}$$

If an image pixel has the same value as its four surrounding neighbours (i.e., no edge) then the value of $edge_{i,j}$ will be zero. If the pixel is very different from its four neighbours (i.e., a possible edge) then $edge_{i,j}$ will be large in magnitude. We will always consider $i$ and $j$ to lie in the range $1, 2, \ldots M$ and $1, 2, \ldots N$ respectively. Pixels that lie outside this range (e.g., $image_{i,0}$ or $image_{M+1,j}$) are simply considered to be set to zero.

The exercise is actually to do the reverse operation and construct the initial image given the edges. This is a slightly artificial thing to do, and is only possible given the very simple approach used to detect the edges in the first place. However, it turns out that the reverse calculation is iterative, requiring many successive stencil operations each. This is similar to the edge-detection calculation itself:

$$image_{i,j} = (image_{i-1,j} + image_{i+1,j} + image_{i,j-1} + image_{i,j+1} - edge_{i,j})/4$$

The fact that calculating the image from the edges requires a large amount of computation makes it a much more suitable program than edge detection itself for the purposes of GPU acceleration and performance measurement.

This inverse operation is also very similar to a large number of real scientific HPC calculations that solve partial differential equations using iterative algorithms such as Jacobi or Gauss-Seidel.

### Practical

Navigate to find the template source code in exercises/opencl-optimise/reverse.c. There should be an accompanying kernel template kernels.c. You will need to do this in a separate tab or window.

Please read on for instructions. Where necessary, it may be helpful to refer to the OpenCL reference guide https://www.khronos.org/files/opencl22-reference-guide.pdf

### Optimisation

You are provided with a working OpenCL implementation of the algorithm in the `reverse.c` and `kernels.cl` files. You will find three kernels in the second source file; one of these `inverseEdgeDetect1D_col` is already complete and functional the other two will be completed later on in the optimisation process. The source file `reverse.c` contains the top-level program, some support code for file I/O and timing, and a simple host implementation of the algorithm to check the result.

In the host code, the image data arrays are two-dimensional, which maps intuitively onto the two dimensions of the image. A one pixel wide halo region of zeroes is added to each edge; this simplifies the computation as it allows the edge pixels to be treated in the same manner as other pixels (the `edge` array, which holds the original data, does not have a halo as it does not need one). For the kernel, the data arrays are flattened to a single dimension but are otherwise the same format.

The complete kernel, `inverseEdgeDetect1D_col`, assigns each row of the image to its own OpenCL work item. The kernel contains a loop over the columns of the image, determines the element index that it should work on (getting the column from the loop counter and the row from the OpenCL global ID), and processes that pixel as described above.

As the algorithm is iterative, there is a loop in the top-level program that invokes the kernel a number of times. In between the kernel invocations it copies the output from the previous invocation back to the input buffer ready for the next iteration, via a buffer in host memory.

### Remove unnecessary data transfers

Notice that in the main loop of our code, the data is copied from GPU memory to host memory and then back to GPU memory at each iteration. This is not in fact necessary with the exception of the final iteration when the data must be copied back to the host. Therefore, we can avoid most of the data copying by simply reversing the roles of the input and output buffers in device memory after each iteration (the relevant `cl_mem` entities need to be swapped via a `tmp` variable).

Once you have made these changes, run the code again and take note of the time taken by the GPU version. How does it compare to the previous timing?

```
# Execute this command to compile the code.
$ make

# To submit a batch job
$ qsub submit.sh
```

### Memory access

There is another bottleneck in the implementation of the algorithm as provided. The GPU performs best when work items are reading from adjacent memory locations, allowing some of the memory reads to be combined (coalesced). However, in our code, the threads are reading not from consecutive addresses but from different rows of the image, so no coalescing can take place.

This can be improved by decomposing the problem in a different way. Instead of having each work item process one row of the image data and loop over the pixels within that row, we can have each work item process one column of the image and loop over the rows of that column. This means that the work items will always be reading consecutive memory locations allowing the reads to be coalesced.

This is implemented in the `inverseEdgeDetect1D_row` kernel, which needs two changes to make it complete:

1. determine the column index `col` for the thread using an appropriate OpenCL function call;
2. loop over all the rows of the image.

Try running the code with the new kernel (remember to change the main loop to invoke the new kernel and provide suitable global and local sizes for it). How does the performance compare to the previous version?

### Occupancy

You should have seen a noticeable improvement in performance as a result of the changes you made to reduce data transfers between the host and the device and to enable coalescing. However, the current solution is still sub-optimal as it may not create sufficient work items to utilise all the SMs on the GPU — it may have low occupancy.

OpenCL supports 1-, 2- or 3-dimensional decompositions; here a 2D decomposition maps most naturally onto the pixels of an image so this is what we will use.

A 2D version of the kernel is included: `inverseEdgeDetect2D`. It requires two additional lines of code to make it work:

1. calculate the column index `col` from the first dimension of the OpenCL work item ID information;
2. calculate the row index `row` from the second dimension of the OpenCL work item ID information.

Notice that there is no longer a loop within the kernel, as each work item is now operating on an individual pixel; otherwise it is the same.

### Investigate the work group size

Once you have the 2D kernel working correctly, you can try altering the work group size. This are defined by constants `LOCAL_W` and `LOCAL_H` near the top of the main program. Some suggested sizes to try are 8x8, 16x16 and 32x12. Note that the total number of work items (threads) per work group (block) should not exceed 1024 (the product of the local "width" and "height").

### Keeping a copy of your work

If you wish to keep a copy of your work from this exercise remember to download both the driver source `reverse.c` and the file containing the kernels `kernels.cl`.