An Application with OpenACC

In this exercise you will, working with (a slightly different version of) the same application you used for the CUDA optimisation session, learn how to use the alternative OpenACC model.

Introduction

This exercise involves porting an image reconstruction application to the GPU using OpenACC.

You start with an image that looks like this:

Input image

Which has been generated from the original:

Original image

On the left is an image of Edinburgh Castle, processed such that the edges between light and dark areas replace the original picture. Your job is to reconstruct the initial image. This is an artificial thing to do, but it mimics many scientific applications (e.g. that solve systems of PDEs) since the algorithm is iterative, requiring many successive stencil operations. Each pixel of the new image is generated based on its neighboring pixel values and the original edge data by repeatedly performing the following update:

imagei,j = (imagei-1,j + imagei+1,j + imagei,j-1 + imagei,j+1 - edgei,j)/4

The more iterations, the better the reconstruction (although for simplicity we work in greyscale rather than colour).

You are provided with a version that works corectly, but only using the CPU. First of all, let's compile and run the code. Choose to work with either C or Fortran.

To build and run the code in the each case:

```
$ make
```

```
$ qsub submit.sh
```

View the resulting image:

```
# To view the resulting image image2048x2048.pgm
display image2048x2048.pgm
```

You should be able to see that the picture is starting to become clearer. As the algorithm is iterative, there is a loop in the main routine that invokes the kernel N times where N is set to 100. Increasing N will increase the quality of the reconstruction, but please don't do this during the lab to avoid hogging the resources. If you were to run for 10 million iterations, the resulting image would look like this:

Now it's time to GPU-enable and optimise the code, to improve the timing, by editing the source code. Note that a one pixel wide halo region of zeroes is added to each edge of the various image-data arrays; 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 edge data, does not have require a halo.)

Using the correct directory for the source code you want (src_c or src_fortran) and edit the source file.

Offloading to GPU with parallel loop construct

Edit the file reconstruct.f90 (fortran) or reconstruct.c (C). For each iteration of the main loop (with index iter), we want to offload the loops with index i,j to the GPU. Inside the main iteration loop, apply the combined parallel loop directive to each of the two loops labelled "perform stencil operation" and "copy output back to input buffer". For this and each stage below: compile, run, check correctness (by viewing the image), and compare the time to the previous run.

Combining parallel regions

You will see that the above "accelerated" code takes much longer than when run on the CPU. The reason for this is unnecessary data CPU to GPU copies associated with each parallel region. Now replace the combined parallel loop directives with a single parallel region containing two loop directives, and test.

Using a data construct

The code is a bit quicker but still inefficient because data is unnecessarily being copied at every iteration. Now use a data construct which encompasses the whole main loop to avoid all unnecessary copies, and test. If time is still much slower that the CPU, it is due to the data copies at the very start and very end of the main loop. A more realistic simulation will have many, many more iterations, so these copy times will become insignificant. Edit the code to ensure that the timing statements are inside the data region, i.e. these transfers are excluded from the timings, and test again. You will find that the GPU code is now much faster than the CPU.

Varying the vector length

You will see, from the compiler output, that the default value for vector length (i.e. the number of threads per block in CUDA language) being used is 256. Use the vector_length clause to the parallel region to experiment with other values and see how the time varies.

Optional Further work

Try refactoring the body of the main loop using functions/subroutines and the present directive.

Keeping a copy of your work

Finally, if you want a copy of the template and the solution, files are available from the course repository.

If you want to save a copy of your particular solution, please remember to to copy the relevant files to your own file space.