HPC Workshops

Exercise 1

2019

Christian Arnold

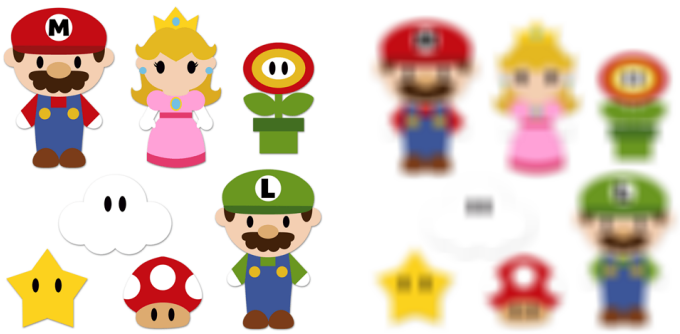
Vectorisation

Answers will be posted on DUO at the end of the week.
Have a look at the Hamilton primer on DUO before starting the exercises. It tells you how to login, how to copy the code to Hamilton/Cosma and how to load different compiler modules.

1) Blurring an image

The directory `blur_image_vec` contains a program which can be used to blur an image. It can be compiled by typing `make`. This runs the Makefile. The Makefile tells the compiler how to compile the code; it can be edited with a text editor of your choice. You will need to load the appropriate compiler module; ideally an intel compiler (on Hamilton: `module load intel`, on Cosma `module load intel_comp`). Make sure you always remove the old binaries by typing `make clean` before recompiling the code with a different setting. In the default setting, the code compiles explicitly without Vectorisation. You can run the program calling
`./mycode input_file.ppm output_file.ppm`,
where the input and output files should be replaced with the appropriate path (you find a couple of feasible input pictures in the `assets` subdirectory).

1. Compile the code and run it on an input picture of your choice. Have a look at the output picture. How long does the smoothing take (you might have to run your code ≈ 3 times to get a realistic timing)?



2. Now allow the compiler to vectorise your code by changing the appropriate flags in the Makefile, recompile and run it again. Does the runtime improve?
   The runtime does not improve with vectorisation.

3. Now add the appropriate flags to create a vectorisation report. Recompile and have a look at the reports to find out why your runtime does not improve. In case the reports are empty, try a different setting for the report output.
   The main loop is found in filters.c. Investigating the optimisation report filters.optrpt for this file (created by adding `-ftree-vectorize -qopt-report=1 -qopt-report-phase=vec`

to the `CFLAGS` in the Makefile), one finds that the main loop was not vectorised:

```
LOOP BEGIN at filters.c(41,3)
remark #25460:  No loop optimizations reported
LOOP END.
```

4. Have a look at the file `filters.c`. Why is the compiler unable to vectorise the loops?
   The main loop contains several sub-loops with conditional statements.

2) Performing expensive calculations

The directory `add_numbers_serial` contains a code which reads random numbers from a file (`numbers.dat`), performs some (no particularly useful but computationally expensive) calculation for each number in the file and adds all results. The actual sum suffers from floating point roundoff errors, so don't be surprised if the result differs depending how you add the numbers. You can compile the code in the same way as for problem 1). The default version of the Makefile does again not allow for vectorisation.

1. Run the code in serial. It will tell you the time it needs to perform the calculations in serial.

2. Now switch on vectorisation and also create vectorisation reports. Is this more successful than for the smoothing code? Are there any loops which the compiler was able to vectorise? If yes, what speedup do you get?
   For this simpler loop, the vectorisation works for the main loop in numbers.c. The speedup (≈ 2.3) is nevertheless smaller than the expected factor of 4 as part of the calculation (i.e. adding the intermediate to the final result) has to be carried out in serial.

3. Now edit the main loop in `add_numbers.c` such that `result_i` is only added to the main `result` if it is larger than 0. Can the compiler still vectorise the loop?
   Yes, the code can still be vectorised.

4. Edit the main loop further and stop the calculations as soon as `result` exceeds 10^{20} (you can do this by adding a conditional `break` statement). Can this still be vectorised? Why?
   No, vectorisation is not possible anymore for this version of the code. The reason is, that the loop might terminate after any iteration (by hitting the break statement). There is thus no way to carry out four consecutive iterations at the same time.