COMS30005: Report on Serial Optimisation

Yi-Ching Chen - yc16011

I Introduction

In this coursework, we are given serial code which performs convolution on images. Our aim is to optimise the stencil function and reduce the runtime down to certain time considering factors such as data types, structure, compiler choices and flags. The runtime mentioned and shown throughout the report will be of the 1024*1024 image, unless specified.

II Compiler flags

There are many different versions of gcc compilers available on Blue Crystal. I loaded different gcc modules - version 4.6.4, 4.7.0 and 7.1.0 - and the runtime of the original function on 1024*1024 image is 8.08s, 8.17s and 8.20s respectively. Although the difference is not significant and older versions are marginally faster, I chose gcc-7.1.0 because through later alterations in the code, it appears to be faster and supports some flags that older versions do not.

To make more use of the compiler functions, I tested some optimization flags, such as -O1, -O2, -O3, and -Ofast. The first three flags have fairly similar runtime of around 6.80s, but -Ofast reduces the runtime to 2.35s, which is almost a quarter of the initial time. The -Ofast flag turns on all the flags that the others have and even more, giving an additional performance increase. The fact that -Ofast has non-standard compliance it might cause errors in calculations, however it passed the python check in this case.

III Implementation on the code

The optimisation on the code will be compiled using gcc-7.1.0 with -Ofast flag.

The first simple optimisation I performed was to remove divisions. The divisions in the function can be simply replaced with multiplications by taking the reciprocals. Division only output few bits per cycle, meaning it will take more cycles to go through one calculation, resulting in a longer runtime. The runtime suggested that the -Ofast flag might have done the conversion already.

Data Layout

The pixels of the 2D image are stored in a 1D array where the position is defined as $j+i*ny$, where i is the current row number and j is the current column number, with nx being the number of rows and ny the number of columns. It means that each row of pixels is stored in the memory after another. As the for-loop is trying to iterate through column first, meaning it is jumping through the array. I swapped the two for-loop lines around, so that it will traverse the array row first. This makes the program faster because when it starts reading an array, it automatically caches the few elements close by, since they are very likely to be used next. When we want to process the next cell, we can get a quicker response. But if it wants the element which is further away, like from the next row, it may need to fetch the data from the memory every loop which can be time consuming. This change reduced the time from 2.34s to 0.87s.

Before I tried to eliminate all the if statements in the function, I write out all the possible computations on the pixels into one big if statement. This is to make sure that the combinations of the computations on each cases do actually output the correct image. And by changing 4 ifs into one, it also makes the process on smaller image faster bar the 8000. When the processor sees if statements, it will perform branch prediction and calculations in the conditions. The reason the if statements might be costly would be branch misprediction, meaning it then has to reload and go through the correct path.

After checking the comparison passed, I removed all the if statements and separated everything into 4 statements for corner cases and 3 loops; one for edge rows, one for edge columns, and one nested *for* loop for the rest.

I tried to store the calculations for the position into separate variables, but it does not significantly change the runtime. The compiler most likely did the calculation once and stored it in case it will be used again. So, the runtime remained at around 0.558s.

Then I changed the data types of *image* and *tmp_image* from double to float. The differences between them are that double has double precision with 64 bits where float has single with 32 bits and we do not need the accuracy that double is offering us. Therefore, float numbers will take up half the memory space, and therefore half the time

to process and hence are used. Moreover, stencil is memory bandwidth bound, the operational intensity then doubled allowing greater potential in performance. This change brought the runtime down to 0.451s.

Considering that running two loops may take up twice the time of running one, I combined the two small loops that compute the edge cases together, as *nx* and *ny* are equal. However, it did not make a notable change to the runtime.

However we can decrease the runtime even more by using the restrict pointer. When we specify the function parameters to be restrict, it means that the memory block of the parameters are exclusively pointed by them only. This eliminates the possibility of pointer aliasing, where the same object is accessed using different pointers. Using restrict pointer also means that updating one pointer would not affect the other, so the compiler is able to optimize the code, possibly through vectorisation. The data is loaded from those memory locations, and computed without being stored them back and reloaded for the next calculation definitely increase the efficiency of the program. The runtime then went down to 0.118s.

Vectorisation

Last change on the code would be trying to vectorise the loop, as the program is performing the same function multiple times. I put *#pragma omp simd* at the beginning of the *for* loop, instructing the compiler to ignore any dependency and execute the computations in the following loop concurrently using SIMD (single instruction on multiple data) instruction. In the SIMD operations, we can perform calculations on several float variables all at once. In order to vectorise the loop via that, I need to also add -fopenmp-simd flag when compiling. I believed that setting the pointers into float has already allowed the compiler to perform such optimisation, and so have minimal effect on the runtime.

IV Other exploration

After all the changes I believed have optimized the code the most, I compiled it using the intel compiler version 16 with -Ofast, which brought the runtime down to 0.100 second. Using other compiler flags -O1, -O2, and -O3, the runtime of -O2 and -O3 is similar as they both optimize to increase speed, whereas the -O1 gives a much slower runtime. For intel compiler, -O1 has -Os turned on, where it will then focus on optimization that do not increase the code size, also vectorization is turned off in this scenario.

When the inital unchanged code was compiled with intel compiler without any optimization flags, it only ran for around 3.40 seconds, which is much faster than the gcc compiler. Then testing out the compiler flags, -O2 had the greatest speedup, as it managed to perform vectorization by removing the branching itself. Even though the vectorization was only performed on the nested *for* loop, because of its runtime being *nx*ny* which is much more significant comparing to the other loop, that alone would reduce the runtime a lot. As for -O3, it is relatively slower since it might transform the loop, and will not be optimized without the loop and memory access being done more efficient.

More on GCC flags, -march=native and -mtune improved the performance time to 0.093s. The tells the compiler to produce code to the specific architecture of the processor it is on, and tune it. However, in practical it will require recompilations on other machines that have different architectures and capabilities.

V Conclusion

Overall, in this task, it can be recognized that the compiler flags and the different compiler choices contribute to the speed up of the runtime. They allow us to write our code and not needing to consider that much on the computations we are performing on the input data, however, paying more attention to the data type and its layout, it can even boost the speed even more.

Image/Runtime	init	+ -Ofast	+div removed	+scan row	+no ifs	+float	+restrict	final (II&III)
1024*1024	8.20s	2.35s	2.35s	0.868s	0.558s	0.451s	0.118s	0.117s
4096*4096	315s	111s	105s	14.3s	9.25s	4.67s	2.61s	2.60s
8000*8000	636s	125s	123s	47.4s	33.7s	17.1s	9.22s	9.22s

Table 1: Runtime with gcc-7.1.0 on all 3 images