Serial Optimisations Report **Oliver Ryan-George (or16131)**
Introduction to High Performance Computing (COMS30005) 26/10/2018

Introduction

This report is a discussion of the optimisation of the function stencil in the given program stencil.c. I will begin by presenting the speeds of the fastest optimisation and then continue by explaining how different optimisations affect the speed and why. I will then finish by reflecting on how fast the final program could have been and what could be done to further optimise it.

Results

The results for the final program submitted on SAFE were taken as an average of five runs and are presented below in Table 1. The effects of different alterations to the final code are shown in Figure 1 and are referenced throughout.

Table 1: Results

Image Size	Initial runtime (s)	Final runtime (s)
1024x1024	8.153425	0.088110
4096x4096	313.894160	2.463002
8000x8000	611.285008	8.371530

1 Vectorisation

The most impactful changes on the efficiency of the function involved optimising it for vectorisation. Vectorisation is where the compiler parallelises code by performing an operation, lying within a for loop, on a vector containing several elements of a subject array simultaneously. This is as opposed to performing the operation on sequentially on each element of the array.

In order for the compiler to vectorise code effectively, for loops and data accesses need to take a certain form. One important rule is to make sure that array elements are accessed incrementally according to their index. This is so the compiler can put elements positioned next to each other in memory in vectors. This was not the case in the original code as the variable i was on the inside of a nested loop which was being multiplied in the index of arrays. By changing the loops in the final program we see a decrease in the time taken by a factor of 11 for an 8000x8000 image.

Another rule is to make sure it is clear to the compiler that there will be no pointer aliasing in the program using the *restrict* keyword. Without it, the vectorisation report states that there is an assumed dependence between arrays. Putting *restrict* in sees a decrease in the time taken by a factor of 5 for an 8000x8000 image.

A third rule is to make sure operations are separated to make it simple for the compiler to perform them. This is a compromise since I initially hypothesised that it would be faster to do all the operations at once so the subject array element would only have to be loaded and stored once. This ended up mattering little, probably since the cost of going to a low-level cache is very small. It would also have the benefit of allowing factorisation so less costly multiplication would need to be done. The benefits of vectorisation outweigh this, however, as the vectorised version is marginally faster.

2 Data types

The choice of data type has a significant impact on the efficiency of the program. The original code used doubles, each consisting of eight bytes which is unnecessarily precise for the task being performed. Floats consist of four bytes each and are sufficient to pass the test. Changing from doubles to floats doubles the operational intensity of the function which is especially beneficial as the program is memory bandwidth bound so we want to minimise the number of bytes loaded and stored per operation. Doing this approximately halves the time taken to run the function for all image sizes.

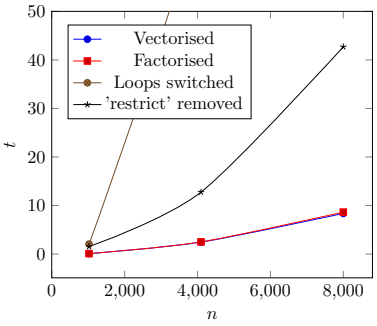
The floating point operation, division, takes far more clock cycles to compute than multiplication because the Intel compiler used divides by first finding the reciprocal of the denominator and then multiplying it. This is why replacing all division operations with multiplication reduces the time taken to run the function by a factor of 20.

3 Storing variables

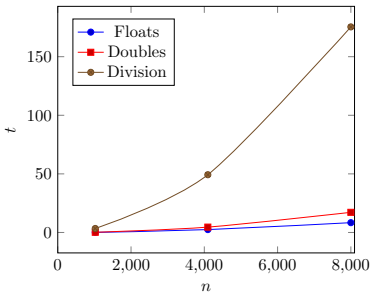
Aware that memory bandwidth is often the bottleneck for programs, I originally approached the task with the attitude that no variables should be stored and all repeated values should be recomputed. After some trial and error, this was the slowest approach. Recomputing multiplications turned out to be slightly slower than retrieving data from a low-level cache. It was fastest, albeit only just, to take a mixed approach with values that were computed with multiplication and that were used several times stored as variables with simple, less used, values recalculated every time.

4 Compilers

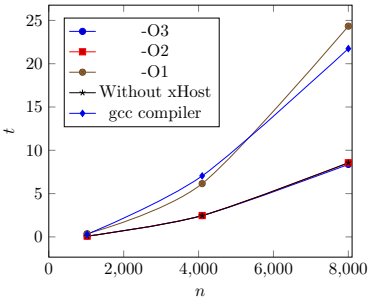
My final program is compiled using the command: "icc -O3 -xHOST -std=c99 -Wall stencil.c -o stencil". I chose to use the icc compiler because of its automatic vectorisation which enabled all the speed increases mentioned in the first section. As can be seen below in Graph 1c, the gcc compiler performs similarly to using the icc compiler with the -O1 flag. This is because vectorisation is disabled with the -O1 flag. The -O3 flag performs best because it is designed for loop-heavy programs that operate on large datasets. xHost marginally increases the efficiency of the program by telling the compiler to generate the highest level instruction set available on the compilation host processor.



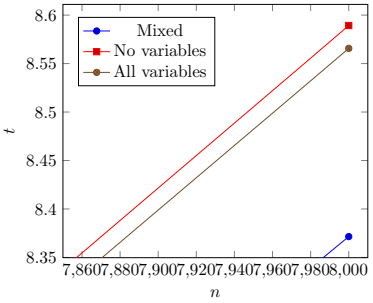
(a) Efficiency using varying levels of vectorisation



(b) Efficiency of different data types



(c) Efficiency of different compilers



(d) Efficiency of using variables against recomputing

Figure 1: Results of different optimisations

Evaluation

Analysing for an 8000x8000 image, I found that the program performed $1.162 GFLOPS$ in $8.372s$. This gives it a performance of $0.139 GFLOPS/s$. Meanwhile, with $3.10 \times 10^9 bytes$ read and written, this gives an operational intensity of $0.374 FLOPS/byte$ This puts the model some way off the roof of Blue Crystal's peak DRAM bandwidth. The function would have to be 32 times faster at that operational intensity to hit Blue Crystal's peak DRAM bandwidth.

This raises the question of how could the program be made faster to reach that peak performance. A profiler shows that 85% of time spent in the program is within the inner loop. Therefore, it would be most appropriate to look to make improvements here. Since it is a loop, it would be best to optimise further for vectorisation, possibly using techniques such as data alignment. It can also be seen from vectorisation reports that the other two loops do not vectorise because of supposed data dependencies but this is less pressing because there are far fewer iterations of these loops so not much time is spent on them.