# Serial Optimisations

**Chanthru Uthaya-Shankar (cu17780)**

## Introduction

This report outlines my findings when applying serial optimisations to code. The aim was to take the *stencil.c* code and make it run as quickly as possible.

## Optimisation Flags

Compilers offer many optional flags that can be added whilst compiling the code. One of these options that the compiler offers are optimization flags which make the compiler attempt to improve the performance of the code. Presented below is a table of different optimization flags and their associated times to process the image using the compiled code. These times are from compiling using gcc version 4.8.5.

| Optimization flag | Time (s) for each image size (width x height) | | |
|---|---|---|---|
| | 1024x1024 | 4096x4096 | 8000x8000 |
| O0 | 5.91 | 129.81 | 561.12 |
| O1 | 5.90 | 38.75 | 167.16 |
| O2 | 2.00 | 37.07 | 153.57 |
| O3 | 2.00 | 37.02 | 153.21 |
| Ofast | 1.18 | 35.72 | 140.27 |

These different flags turn on different optimisations, with O0 turning on no optimisations. O1 turns on basic optimisations causing in a decrease in time. Each optimisation flag higher up the hierarchy turns on more optimisations resulting in more of a decrease in time.

## Compiler Choice

Intel also offers a compiler ICC, which can be compared to GCC. Presented below are the times taken for the ICC compiled code to run using the -fast optimizing flag.

| Image Size (width x height) | Time (s) |
|---|---|
| 1024x1024 | 0.19 |
| 4096x4096 | 5.75 |
| 8000x8000 | 23.2 |

This data compared to the what was shown in the previous table shows that using the ICC compiler results in much quicker times. This is due to the way the compiler presents the instructions to the CPU. ICC presents the instructions in a way that enables the CPU to take more advantage of data movements and calculations on data. This includes making more vector operations on data. When the code is compiled, assembly language is generated which will lay the foundation to telling the processor what to load. Certain characteristics of the data that is loaded will decide the layer in memory the data is loaded into, characteristics such as the type of the data. Ideally, we want as much of our data that we are working with to be stored in caches and registers that are close to the processor, the location and structure of these caches mean that the access times are significantly less than other memories. But a trade-off is that they are much smaller in size so can store much less data at a given moment.

The pixels of the image that we are processing on consist of double precision floating point values which are 64 bits in length. This precision is too high and values in the image can be represented as single precision values by changing the data type to float. Below shows a table comparing times to process the image using double precision floating point values and single precision floating point values.

| Precision | Time (s) for each image size (width x height) | | |
|---|---|---|---|
| | 1024x1024 | 4096x4096 | 8000x8000 |
| Double | 0.19 | 5.75 | 23.2 |
| Single | 0.10 | 2.93 | 11.06 |

By using single precision, we are now representing all our values using 32 bits. This means we can now fit twice the amount of data in a single cache line, so more of the data we need to operate on are in storage mediums higher up the memory hierarchy meaning quicker access times. This results in a quicker overall time to process the image with a speed up of almost double.

## Vectorization

Compilers have the option to do loop-level auto-vectorization on data, which allows the CPU to make use of vector operations by taking this data into vector registers as a vector of operands and computing many calculations at once. For the gcc compiler auto-vectorization is turned on when using the -O2 flag, the effects of auto-vectorization can again be shown by the first table where there was a speed up in performance.

Although the gcc compiler auto-vectorize code, it does not vectorize all the possible code that can be vectorized. Code can be changed so that it is presented in a way that can be vectorized, and the use of restrict keywords on pointers will instruct the compiler that it will not alias and that there are no memory overlaps. #pragma simd can be used to indicate to the compiler that multiple iterations of the for loop can be executed at once using SIMD instructions. There is an increased amount of vectorization when switching from GCC to ICC which is indicated by the decrease in time. Whilst compiling using ICC I found that including the pragmas had no effect on the speed of the code.

## Memory Layout

How the array is accessed and how it is laid out in memory will greatly affect the time taken. Here I change the code so that the array is accessed one contiguous element (row-major) at a time instead of the array being accessing in multiples of the width of the image (column-major)

Using gcc the below table shows the difference when the array is accessed in column-major fashion and when it is accessed in a row-major fashion

| Image size (widthxheight) | Time to process image (s) | |
|---|---|---|
| | Column Major | Row Major |
| 1024x1024 | 1.18 | 0.76 |
| 4096x4096 | 35.72 | 12.56 |
| 8000x800 | 140.27 | 45.72 |

This data shows that changing the way the array was accessed increases the performance. This is due to how data is loaded and the layout of the array in cache. The array is flattened out in memory, elements contiguous in the array will be contiguous in memory, where as elements on separate rows will be far away in memory. This means there will be a difference in access time. When conducting the same test using intel's ICC compiler I found there was no decrease in time. This indicates that ICC does this automatically.

## Operational Intensity

Operational intensity is affected by the amount of read and writes to and from main memory. I made a change to the code to minimize the amount reads and writes to 5 reads and 1 write.

## The Roofline Model

The roofline model is a way of visualising and comparing the operational intensity and the performance of a system. From this we can deduce whether the problem is compute bound or memory bound..

| Picture size (WidthxHeight) | Operational Intensity (FLOPS/byte) | Performance (GFLOPS/s) |
|---|---|---|
| 1024x1024 | 0.375 | 18.874 |
| 4096x4096 | 0.375 | 10.306 |
| 8000x8000 | 0.375 | 10.405 |

Figure 1 shows the roofline model for Blue Crystal Phase 4.

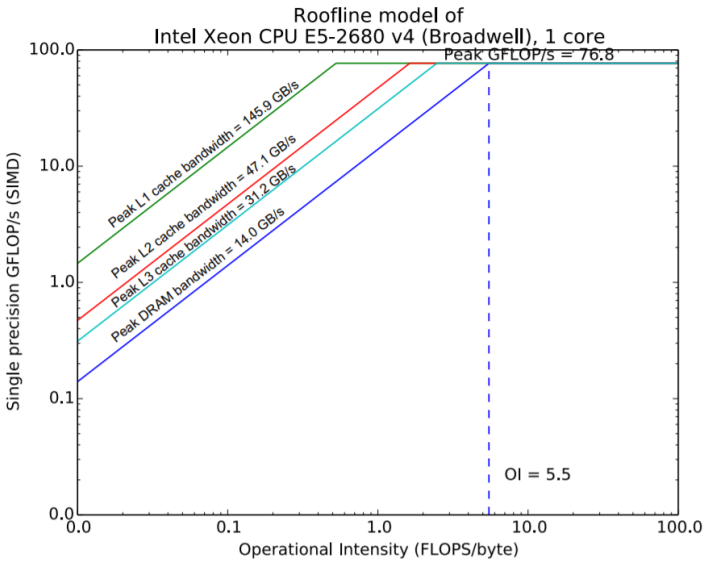


Fig 1.

## Conclusion

This report shows how all the changes made affect the overall performance of the code. Shown in the table above are the performance calculations after all optimisations mentioned in the report are applied. It shows that we are memory bound and not compute bound as the calculation for operational intensity lies to the right-hand side of the dotted line that corresponds to the operational intensity at peak performance. We aren't always using the level 1 cache to load and store data otherwise our calculation for performance would be closer to the L1 cache line shown in figure 1. This indicates during runtime the processor is reading and writing to and from other storage mediums.

Going from the 1024x1024 images to the 4096x4096 and 8000x8000 there is a big drop off in performance with the performance almost halving. This indicates that a lot of the image was not stored in L1 cache due to the image size being too big, this would mean longer access times to retrieve the data required and also explains why the time doesn't scale up in square fashion as the image dimensions are increased.

My final times are shown below running on ICC version 2018-u3

| Image size (Width x Height) | Time to process image (s) |
|---|---|
| 1024x1024 | 0.10 |
| 4096x4096 | 2.93 |
| 8000x8000 | 11.06 |

**References**

Figure 1. Performance Analysis Techniques – Simon Mcintosh