Introduction

In this report I will be outlining the optimisations I have made to the *stencil.c* program. I will be giving explanations to the changes I have made, along with my understanding as to why said changes yielded a speed change in execution time.

Times shown in all tables are for the image size 1024x1024 unless otherwise stated.

Compiler Type and Version

Initially I stared to improve the speed of *stencil.c* by changing which compiler I was using, and the compiler versions. As seen from Table1, there was a marginal increase from the default Blue Crystal compiler (GCC 4.8.5) to GCC 9.1.0, which is due to the latter being a more recent compiler, hence having more efficient optimisations built in.

Table1 – showing the taken times for each compiler version used.

Compiler	Run 1	Run 2	Run 3	Average
GCC 4.8.5	5.906s	5.904s	5.907s	5.906s
GCC 9.1.0	5.875s	5.875s	5.873s	5.874s
ICC 2016-u3	2.004s	2.004s	2.004s	2.004s
ICC 2017.01	1.796s	1.796s	1.797s	1.976s
ICC 2018-u3	1.796s	1.796s	1.796s	1.976s

Comparing the GCC compilers with the intel ICC compilers shows a maximum 3.35X speedup. From the compiler reports of both GCC and ICC it was shown that the Intel compiler was able to vectorise the loops inside *stencil.c* for ICC 2017 and 2018, whilst GCC was not.

Vectorisation provides such a drastic decrease in run time as it utilises vector operations. This is enabled via Single Instruction, Multiple Data (SIMD) hardware in order to replace multiple operations with a single operation, which is applied to multiple data items at once.

The earlier Intel compiler ICC 2016 did not perform vectorisation as shown by the report. However, there was still a significant speed increase over the GNU compilers. This is due to the Intel compilers having the `-O2` optimisation flag enabled by default[1], where as the GNU compilers do not.

The Intel compiler reports also showed that the execution order of the two loops inside the `Stencil(..)` function are switched around. This causes the memory to be accessed in a row-major fashion rather than column-major, which matches the convention that the C language uses to write data to memory. Henceforth, a speedup occurs as the values in the array are being fetched more sequentially than before, reducing the number of jumps being made between cache lines.

Furthermore, as Blue Crystal is an Intel based platform, the ICC compiler is better tuned to the system architecture and so performance gains can be seen from this.

Ultimately, I chose to use the Intel ICC 2018-u3 compiler.

Compiler flags

Enabling compiler flags lets the compiler optimise the code for us automatically. Although this can, in some cases, increase compilation time, there can be a substantial benefit in execution time

From Table2, we can see that by progressing through the various iterations of optimisation flags, I was able to achieve a further 9.49X speedup when using the `-fast` flag with ICC 2018-u3. From this table we can also see varying degrees of improvement and in certain cases regress in our execution times.

Table2 – showing the taken times for each compiler flag that was tested.

Compiler Flag	Run 1	Run 2	Run 3	Average
-O0	6.045s	6.045s	6.045s	6.045s
-O1	2.003s	2.002s	2.002s	2.002s
-O2	1.795s	1.796s	1.796s	1.976s
-O3	1.797s	1.796s	1.796s	1.976s
-Os	2.004s	2.003s	2.003s	2.003s
-Ofast	0.251s	0.244s	0.245s	0.247s
-fast	0.196s	0.186s	0.186s	0.189s

Enabling the `-O0` or `-O1` flags sees a decrease in execution times, since they turn off certain default flags that the compiler will have enabled. For example, `-O0` disables optimisations entirely[1] and focuses on reducing compilation time[2], while `-O1` prevents loop unrolling and favours reducing code size to speeding up execution[1]. Similarly, `-Os` will only keep enabled optimisations that do not increase code size[1].

Optimisation flag `-Ofast` sees a 7.27X speedup over using the default `-O2` flag due to it disregarding the strict standard compliances, plus enabling optimisations that are not compatible with all standard-compliant programs[1].

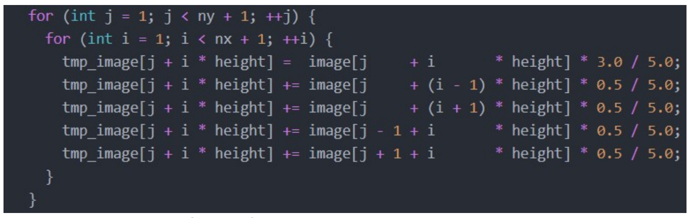
We see a further 1.3X decrease in execution time from the `-Ofast` to `-fast` flag. From looking at the compiler report, we can see that the vector length has doubled from two to four. This increase in vector size will decrease the number of operations performed within our program by approximately half. Additionally, the `-fast` flag enables the `-xHost` flag, which utilises the best suited SIMD instructions for Blue Crystal's architecture[2].

[1] - https://software.intel.com/sites/default/files/m/d/4/1/d/8/icc.txt
[2] - http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/intel-compiler-flags/
[3] - https://software.intel.com/en-us/articles/fdiag15134

However, the compiler report also shows that this change in flag means that inside our critical section of the code (`Stencil(…)`), most array accesses are unaligned meaning that time is being spent using the less efficient unaligned memory access instructions[3]. Despite this, increasing the vector size, and therefore decreasing the amount of operations being carried out, outweighs the time difference between aligned and unaligned load and store instructions.

Code Optimisations

Many optimisations to the code that I carried out saw little to no speedup. This is due to the compiler, and the selected optimisation flag, having already carried out these code changes as part of its optimisation process. There was some variability to the times produced, however these can be attributed to noise on the data rather than actual speedup.



```
for (int j = 1; j < ny + 1; ++j) {
  for (int i = 1; i < nx + 1; ++i) {
    tmp_image[j + i * height] =  image[j     + i       * height] * 3.0 / 5.0;
    tmp_image[j + i * height] += image[j     + (i - 1) * height] * 0.5 / 5.0;
    tmp_image[j + i * height] += image[j     + (i + 1) * height] * 0.5 / 5.0;
    tmp_image[j + i * height] += image[j - 1 + i       * height] * 0.5 / 5.0;
    tmp_image[j + i * height] += image[j + 1 + i       * height] * 0.5 / 5.0;
  }
}
```

Image1 – The original Stencil function in stencil.c

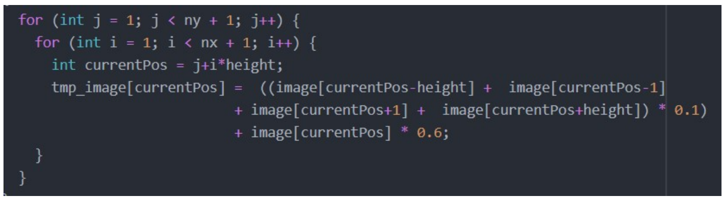
Initially I removed the divisions from each line and replaced these values with their appropriate constants.
`* 3.0 / 5.0;` became `* 0.6;`
`* 0.5 / 5.0;` became `* 0.1;`
Despite there now being less division operations (which can take over 20 clock cycles to compute compared with 2 for multiplications) there was no speed up seen, due to the ICC compiler and `-fast` flag doing this optimisation already.

Removing four memory accesses of `tmp_image[j+i*height]` did see a slight speedup of 1.05X. Despite running the code multiple times, due to variability between runs that can occur on Blue Crystal, this speedup could be due to noise.

Pre-computing the index values of the data cell's with `int currentPos = j+i*height;` also saw no decrease in execution time, due to similar compiler optimisation factors.

Similar to before, removing some of the multiplications, by adding together all terms that are multiplied by 0.1 before multiplying them by the constant, didn't provide speedup. Again, this can be attributed to the ICC compiler and `-fast` flag already optimising the code in this way.

The last change made also saw a more substantial speedup of 1.22X. This occurred when I replaced all variables of type `double` to `float`. This speedup can be deduced from the compiler report where the vector length has again been doubled from four to eight. This vector length increase is due to the fact that a `float`'s size (32bits) is half the size of a `double` (64bits), meaning more data can be fit into a vector of the same size. This reduced size also means that we can fit more data onto a single cache line, reducing the number of cache line jumps.



```
for (int j = 1; j < ny + 1; j++) {
  for (int i = 1; i < nx + 1; i++) {
    int currentPos = j+i*height;
    tmp_image[currentPos] =  ((image[currentPos-height] +  image[currentPos-1]
                           + image[currentPos+1] +  image[currentPos+height]) * 0.1)
                           + image[currentPos] * 0.6;
  }
}
```

Image2 – The optimised Stencil function in stencil.c

Final Results and Limitations

Overall, I saw a speedup of 38.0X utilising the optimisations mentioned in this report. Much of this speed up was seen when changing between which compiler was used, and what optimisation flag was to be used along with the chosen compiler.

Many changes to the code saw little speedup due to the effectiveness of the compiler's optimisations. Table4 shows the original and final average time for each image size provided.

Table4 – Shows optimised and unoptimized execution times

Image Size	Un-optimised	Optimised
1024 x 1024	5.906s	0.154s
4096 x 4096	130.194s	3.221s
8000 x 8000	561.116s	11.653s

By calculating the operational intensity of our un-optimised code, we get a value of 0.1875 FLOPS/byte. Looking at Image3, we can see that this places Stencil.c as memory bandwidth bound.

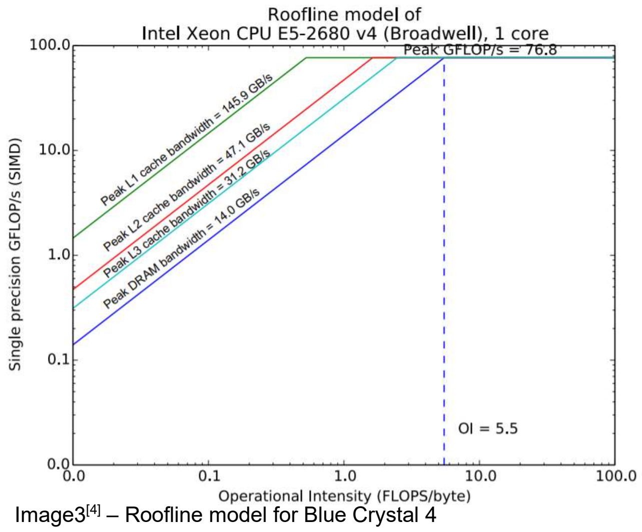


Image3[4] – Roofline model for Blue Crystal 4

The operational intensity after the optimisations rose to 0.2857 FLOPS/byte, which still places Stencil.c as memory bandwidth bound. Therefore, to decrease our execution time even further we would need to utilise the STREAM bandwidth more efficiently.

[4]- https://www.ole.bris.ac.uk/bbcswebdav/pid-3923027-dt-content-rid-12621741_2/courses/COMS30005_2019_TB-1/Open%20Access%20for%20CS/lectures/hpc_performance_analysis.pdf