**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 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 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.

**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 bigger benefit in execution time by enabling an optimisation flag.

From Table2, we can see that by progressing through the various iterations of optimisation flags, we were able to achieve a further 9.49X improvement on the run time of the code.   
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], hence the slight increase in execution time.

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 used by the compiler in the for loops inside our Stencil(…) function has doubled from two to four. This increase in vector size will decrease the number of operations performed within our program by approximately half.

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. There was some variability to the times produced, however these can be attributed to noise on the data rather than actual speedup. This is due to the compiler, and selected optimisation flags, having already carried out these code changes as part of its optimisation process.

Removed the devisions

Removed 4 tmp-image accesses

Calculate memory index in outer for loop

Remove multiplications by adding terms then \* by 0.1

Replaced all doubles with floats

Only saw speed increase if additions were certain way around

**Final Results and Possible Changes**

Final times for all images

Graph showing increase over optimisations

What could be done to get further improvements