**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 9.1.0, which is due to the latter being a more recent compiler, hence having more efficient optimisations.

Comparing the GCC compilers with the intel ICC compilers shows a 3.35X improvement on run time. 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.*

Vectorisation is able to provide 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.

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

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Default | 5.906s |  | 5.904s | 5.907s |
| GCC 9.1.0 | 5.875s |  | 5.875s | 5.873s |
| ICC 2017.01 | 1.796s |  | 1.796s | 1.797s |
| ICC 2018-u3 | 1.796s |  | 1.796s | 1.796s |

**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.

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 most optimisations[1] and focuses on reducing compilation time[2], while -O1 will reduce code size and execution time using only optimisations that are quick to perform[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 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.

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

|  |  |  |  |
| --- | --- | --- | --- |
| -O0 | 6.045s | 6.045s | 6.045s |
| -O1 | 2.003s | 2.002s | 2.002s |
| -O2 | 1.795s | 1.796s | 1.796s |
| -O3 | 1.797s | 1.796s | 1.796s |
| -Os | 2.004s | 2.003s | 2.003s |
| -Ofast | 0.251s | 0.244s | 0.245s |
| -fast | 0.196s | 0.186s | 0.186s |

**Code Optimisations**

C

**Final Results and Improvements**

D