Introduction to High Performance Computing
Assignment 1 Report

Finn Hobson - fh16413

1 Introduction

This report discusses the attempts I made to optimise the 5-point stencil code to make it run as fast as possible on BlueCrystal Phase 3. I tried a number of different optimisations, some of which improved the performance of the code and some of which did not.

For every runtime used in this report, I ran my code three times on BlueCrystal Phase 3 and took an average of the three times shown in the output files. This ensures that all of my timings are accurate and there are no anomalies.

2 Compilers and Compiler Flags

Before editing the stencil code, I decided to test the performance of the initial code when using different compilers and compiler optimise options. The two compilers that I decided to test during this assignment were the GNU Compiler Collection version 7.1.0 (GCC) and the Intel C++ Compiler version 16.0.2 (ICC). The table below shows the runtime of the stencil code when using the two compilers with each of the available optimisation flags.

Compiler Optimise Flag	GCC Runtime (seconds)	ICC Runtime (seconds)
(None)	8.23	3.40
-O0	7.76	8.50
-O1	6.79	6.79
-O2	6.80	3.40
-O3	6.81	6.81

Table 1: Stencil code runtimes using different compilers and compiler optimise flags (1024x1024 checkerboard).

Interestingly, the best performance is produced when using ICC with the `-O2` optimise option. The reason that ICC improves performance compared to GGC may be because ICC is has a better picture of the machine architecture so can exploit all available registers and minimise memory operations. `-O2` activates a number of optimisations that aim to maximise the speed of the code. When using `-O2`, ICC automatically looks for vectorisation opportunities within the code, which improves the performance of loops.

The reason that `-O2` causes the code to run faster than `-O3` is `-O3` enables all `-O2` optimisations plus more aggressive loop and memory-access optimisations. Because the data that is processed by the stencil code is fairly predictable and only a few floating point operations are performed on each pixel, these more aggressive optimisations are unnecessary and slow down the performance of the stencil code.

The reason that the code runs just as quickly as `-O2` with no optimise option is that ICC automatically optimises the code to run as fast as possible unless you specify no compiler optimisations with the `-O0` compiler flag.

3 Code Optimisations

Using the GNU profiler (GProf), I was able to determine that every time I ran the stencil code, the `stencil` function was taking over 99% of the execution time, making this the critical code. Therefore, I decided to focus my efforts on optimising this function.

3.1 Removing Division

Division is a very expensive operation compared to other arithmetic operations as it takes a processor far more cycles to execute an iterative division algorithm. Therefore, I removed the division from the `stencil` function by converting the pixel weights into single decimal numbers. This greatly improved the performance of the code, decreasing the runtime to 0.327s when using ICC and -O2, and a checkerboard size of 1024x1024 pixels. For the same compiler and compiler options, removing the division operations reduces the 4096x4096 checkerboard runtime from 49.3s to 8.07s and the 8000x8000 checkerboard runtime from 176s to 30.1s. These significant performance improvements show how computationally expensive the division operation is when executing code.

3.2 Changing Data Types

The initial stencil code stores the pixel values in the image arrays as double precision floating point numbers (`double` data type). This means that each pixel value has a storage size of 8 bytes. The `for`-loops in the `stencil` function read and write hundreds of thousands of these values to and from memory, which will be taking a large amount of time during execution. In order to make this data more 'cache-friendly' and make memory access faster, I decided to store every pixel value as a single precision floating point number (`float` data type) in the image arrays so that each pixel value has a storage size of only 4 bytes. This optimisation did improve performance, decreasing the 1024x1024 checkerboard runtime to 0.149s, the 4096x4096 checkerboard runtime to 3.66s and the 8000x8000 checkerboard runtime to 14.3s. The stencil code is memory bandwidth bound, but this optimisation increased the operational intensity of the code, increasing its peak potential performance.

3.3 1D vs. 2D Array

In the initial stencil code, the pixel values were stored in 1-dimensional arrays. I decided convert these arrays into 2-dimensional arrays to see if this would improve the performance of the code. After changing the compiler optimise option to `-O3`, this modification did improve the performance of the code, decreasing the 1024x1024 checkerboard runtime to 0.116s, the 4096x4096 checkerboard runtime to 2.52s and the 8000x8000 checkerboard runtime to 8.84s. This improvement may be because the processor can access the elements of a multi-dimensional array more efficiently from memory than a longer 1-dimensional array.

3.4 Removing Branching within Loops

Branching within loops can be a very expensive operation as it can stall the pipelines in modern processors. Therefore, I decided to try to optimise the code by removing the `if` statements from the inner `for`-loop of the `stencil` function. I did this by separately calculating the new pixel values for the edge rows and columns outside of the main `for`-loops. However, this actually decreased the performance of my stencil code, increasing the 1024x1024 checkerboard runtime to 0.320s. This may be because this caused the processor to jump around the array, preventing it from using the data already stored in the cache. Also, I examined the optimisation report that was generated before I removed the branching and found that ICC has powerful branch prediction that eliminated many conditional branches during compile time. This means that the effect of branching within the loops is significantly reduced before execution time.

3.5 Vectorisation

As mentioned in Section 2 of this report, the `-O2` compiler flag activates ICC's automatic vectorisation functionality. However, the optimisation report showed that the `for`-loops in the `stencil` function could not be vectorised because "vector dependence prevents vectorisation". In order to remove this 'vector dependence', I added the `restrict` keyword in front of the image array pointers. This tells the compiler that the pointer is the only thing accessing that array and there is no overlap between the two image arrays, meaning the compiler can perform more vectorisation. After making this change, the optimisation report shows that the compiler now performs more vectorisation on the `for`-loops in the `stencil` function, but there is very little improvement in the runtime of the code. This is most likely because vectorisation may improve the speed of computation, but the stencil code is bottlenecked by memory bandwidth.

1024x1024 pixels (seconds)	4096x4096 pixels (seconds)	8000x8000 pixels (seconds)
0.116	2.51	8.84

Table 2: Final runtimes of my stencil code.