Stencil Report
An Introduction to High Performance Computing
COMS30005

Faizaan Sakib
User ID: ss16161

Introduction

This report involves the task of optimising a serial program on the BlueCrystal 3 Supercomputer. The program itself is a 5-point stencil that operates on a large image of a checkerboard. The different methods of optimising this code will be outlined in this report, along with further detail as to why and how it affected the run-time of the program. The table on the right shows the run-times achieved with the default Makefile and unchanged code using GCC 7.1.0 with no flags. The report will make comparisons with these image sizes throughout. This report can be seen as a chronological timeline of the different optimisations applied to the program over time.

Dimensions	Run-Time (s)
1024x1024	8.31
4096x4096	312.53
8000x8000	>600

Table 1: Initial Run-Times

Loop Calculations

To start off, there seemed to be a few trivial inefficiencies within the stencil function loop. A calculation for the array subscript accessing the images was performing 10 times per iteration (4 of those times with an offset). To compute it that many times, for an image up to 8000x8000 pixels is extremely inefficient. So the expression is instead stored in a variable. However, since the expression is dependent on two of the loop counters, it cannot be removed out of the loop. It will be calculated $x*y$ (x being the row length and y being the column length) times, once in each iteration but it is a vast reduction than before. Similarly, there were 5 constant weightings being computed every loop. These used division, which is a costly computation due to its complex iterative nature making it hard to parallelise. The reciprocal of one the weight multipliers (0.1 to 10) was taken and used to divide the value on the image by it rather than multiply. As suggested, this simply increased run-time and the code was reverted. The constants happen to equate to a precise number so they could be stored explicitly without division. These simple, yet effective optimisations managed to reduce run-times by approximately 50%. Furthermore, using the `register` keyword, variables can be stored in register for quick access, reducing run-time even further.

Compilers and Optimisation Flags

A compiler is essential for optimisation as it includes many features that abstract away a lot of the work under flags. Using the `-O` flags on both the GCC and Intel Compiler (ICC) heavily reduced run-times by around 50-60%. The flags increase in optimisation from `-O1` up to `-O3`. The first two introduce loop optimisations, instruction scheduling, among others. The `-O3` flag introduces auto-vectorisation making it suitable to process large data sets and floating-point calculations. Using ICC v13 up to v16, the performance gradually got better with each newer version. It is important to note that ICC had faster run-times, before the next optimisations take place, compared to GCC. This is due to ICC being tailored towards Intel chipsets, which is what the Blue Crystal runs on. The flag `-xhost` enables processor-specific optimisations, in this case it was for Sandybridge chips.

Row Major Order

To convert the code from column major to row major order, the position of the loops had to be swapped so that the loop counters were also swapped. This ensured that the array offset values that were being calculated were in row major. This proved a good optimisation, especially for the larger images, reducing run-time by 1.4s, ~103s and ~120s (in increasing image size). Accessing the array like this means it has a contiguous access pattern, as the next iteration of the loop will perform on the memory item next to the current one. This prevents cache thrashing, where cache lines that are retrieved are not compatible with the process occurring. Row major therefore avoids cache misses and unwanted accesses to higher levels of memory such as L3 or disk memory, which can add a lot to the run-time.

Vectorisation

Vectorisation is an important step in optimising loops, and so it is especially relevant as this program fully depends on loops. Although there is auto-vectorisation taking place due to the `-O3` flag, much more can be done with the code to fully take advantage of it.

Firstly, the presence of conditional if statements creates delays in the pipeline with branches, as it will be often waiting for the conditional branch to execute before moving on to the next instruction. To get rid of the if statements, the grid was processed differently. This involved hard coding the corners, then top and bottom rows, and then left and right columns, then finally the main grid in the middle. The conditionals were gone, but the run-times had dramatically increased up to 3.65s for 1024x1024 and more for the larger images.

By contrast, the second attempt involved using three main loops. The first computes the top row, the second computes the main block (from rows 1 to n - 1) and the third the bottom row. The edge cases would be processed just before and after the inner loop. Here, memory access is done with spatial and temporal locality in mind. This way, compared to the first solution, memory access was way more efficient due to good use and reuse of cache.

Then, to allow the use of SIMD, all the calculations for each point in the image is put into one statement. `#pragma ivdep` is used before each loop to explicitly state the absence of vector dependencies to the compiler and the `restrict` keyword implied the memory pointers to the images are not aliased, and so the memory cannot be accessed by any other way but that pointer itself. SIMD can now take place, as the loop can now be unrolled and performed in parallel.

After vectorisation, the run-time seemed to be performing better on GCC with the inclusion of the `-Ofast` flag which includes harsher optimisation which has less portability, but is sufficient for this program. The run-times were now 0.182s, 4.607s and 17.121s.

Data Types

For this program, double is used as the data type for the images. There are an extremely large number of multiplications in every iteration, and there is no need for the high precision the double delivers. The amount of memory used can be halved simply by using floats of 32 bit size instead. Also, vectorised code can process twice as many floats than doubles in one pass. The constants are also changed to floats so there are no type conversions. The constants are then factorised out instead of being multiplied with each neighbour point to reduce computation. This proved to be one of the best optimisations, reducing run-time by ~50% while at already such a low time (0.182s to 0.091).

Memory Alignment

Since memory is accessed in blocks of 2^x bytes, if the CPU tries to access data that is not a power of two, it is not aligned. This means two or more blocks of data have to be read before shifting out the unwanted bytes to access what it's looking for. `_mm_malloc` and `_mm_free` were used to allocate aligned memory space for the array. Then `__builtin_assume_aligned` is used to let the compiler know which arrays are aligned and by how many bytes. In this case, it is aligned by 16 bytes as SSE defines 128-bit (16 byte) registers. This did bring about small performance enhancement by about ~2% to ~4%.

Dimensions	1024x1024	4096x4096	8000x8000
Run-Time (s)	0.0873	2.397	8.426

Table 2: Final run-times on GCC 7.1.0 including -Ofast and -march-native (processor specific optimisation)