Stencil Single Node Serial Optimisation

Mark Ergus Nicholl (mn16660)

October 26, 2018

1 Introduction

The task of this assignment was to optimise the given Stencil code and to understand the underlying factors that causes a change in performance. Understanding the principles of high performance computing optimisation techniques and the target CPU architecture was crucial in this task. In this report, I will explain the optimisation that has been made to my code, state the results from each change and explain why each result was achieved. As the data was gathered sequentially from the beginning, comparing the initial run time and the run time after a certain optimisation will not give a fair result. Hence, the results given will be the difference between the time with a feature and the time without said feature, all other things being equal.

Optimisation	Time Before(s)	Time After(s)	Time Reduction (%)
2.1 Rearranging Loop Structure	6.798507	2.104552	69
2.2 Column Major To Row Major	2.104552	0.359228	83
2.3 SIMD Vectorisation w/ Floats	0.359228	0.123626	66

Table 1: Optimisation and corresponding times on 1024x1024 problem

2 Effective Optimisation

2.1 Rearranging Loop Structure

After profiling and determining the stencil function as the bottleneck, the initial course of action was to optimise the structure of the loops in the function to improve performance. By analysing the initial code, it was clear that the 'if' statements were a potential cause of poor performance. On a machine code level, 'if' statements require the program counter to carry out conditional branches. The processor does branch predictions and decides whether it branches or not. However, if a branch misprediction occurs, the processor needs to flush its pipelines and load new instructions, which can be a very expensive operation. By rearranging the loops to process the edge cases and the main array content seperately, we can reduce this uncertainty and allow the processor to carry out its operation in a streamline manner. After rearranging, profiling results show that the edge cases took comparatively negligible time to run, hence now we can identify the main array content loop as the bottleneck and narrow down our focus there.
Other optimisation of the loop include pre-computing multipliers such as (0.5/5.0) to (0.1) and factorising out common multipliers in the equation. These minor optimisation may only produce little effect in the beginning, but they are key for SIMD vectorisation further down the line.

2.2 Column Major To Row Major

The original code provides a 'for' loop that operates on the array in non-contiguous access pattern. This was inefficient because in C, 2D arrays are stored in row major order. After each iteration, the processor would probably need to go up the memory hierarchy to fetch new data. Conversely, in a contiguous pattern, the processor needs to go up the memory hierarchy less often. The processor and cache make use of spatial locality of data, hence it pre-fetches data close to each other. By switching the order of the loop, we now have a contiguous access pattern. The processor now has data it needs 'closer' to it more often than before. This affected the performance dramatically. More importantly, this affected the larger problems to perform significantly faster. The 4096x4096 problem went from 99.5s to 6.3s with this change, 15.8 times faster than before. This is due to the fact that the entirety of the 1024x1024 data can be stored in at least the L3 cache, whereas in the larger problems, some data is stored in the memory which is very expensive to access. Hence, data locality carries a larger significance in problems with larger data due to memory access overhead.

2.3 SIMD Vectorization With Floats

At this point, fine tuning was needed to make the code run even faster. The next logical step was to implement SIMD vectorisation. After studying the architecture of the Sandy Bridge processor on Blue Crystal 3, I decided to take advantage of the Advanced Vector Extensions(AVX). In order to 'signal' the compiler to carry out vectorisation, the array pointers are declared as 'float * restrict' to tell the compiler that there will be no loop dependencies from iteration to iteration. The compiler then knows that loops can be unrolled and multiple values can be calculated as vectors. In terms of compilers, I decided to use the Intel compiler with the -xAVX flag to signal the kind of architecture that will be running the code, as well as -Ofast that allows aggresive optimisations. The Intel compiler also has a very detailed vectorisation report that was useful to identify why certain code did not vectorise. As double precision was not needed, the data type was changed to floats instead. In theory, this would allow twice the number of data to be processed at the same time. However, I found that this optimisation only worked with factorised equations and the specific AVX compiler flag. With research, I found out that some architectures have specific mechanisms for double precision values that work better, particularly in 64-bit machines. This further emphasises the notion that trial and error is highly important in this field.

3 Non-Effective Optimisation

Other methods were attempted to push the code to the next level. However, few gave any significant improvement. One method attempted was cache blocking, where we step through the array in blocks to increase the usage of spatial locality of data. This made performance of the code worse in my case. This could be due to the fact that unlike programs that benefit from this, we do not reuse the data needed that many times. Also, the compiler probably already utilises the cache in an efficient way.
Besides, memory arrangement was tried by using Intel's '_mm_alloc' instead of 'malloc' while adding '#pragma vector aligned' to the crucial loop. This did not produce a significant improvement in my case. It could be that the processor is proficient enough in using scattered memory and the vectorisation without the pragma is already well optimised.
Although these optimisations did not work, attempting them and analysing the performance while looking for reasons as to why they do not work were very insightful. Again, it emphasises the importance of trial and error, and how these complex architectures are very peculiar in their own way.

4 Analysis and Conclusions

To analyse the data, we can use the roofline model to estimate where we are at in terms of performance. Calculating the operational intensity of the bottleneck main array content loop, it comes out at 0.3 FLOP/byte. Base on the graph given during the performance analysis lecture, this gives us about 5 GFLOP/s and puts us in the memory bandwidth bound region. Also, by calculating 1024x1024(array)x100(rounds)x6(FLOP) divided by 0.123626s, we know that the program performs roughly 5 GFLOP/s. This is quite a number away from the peak GFLOP/s of 41.6. Hence, if future development is in mind, knowing that we are memory bandwidth bound, an emphasis of focus on managing cache and memory would be logical.

Problem Size	1024x1024	4096x4096	8000x8000
Time(s)	0.123626	2.559609	9.189007

Table 2: Final obtainable times

The table above shows the final obtainable times on all 3 problems. These numbers are vastly superior to the original program runtimes.
We can conclude that optimisation and smart programming plays a big role in harnessing the hardware capabilities of a supercomputer. A powerful machine is not enough to run codes quickly, it is merely a tool for programmers to run efficient code. Besides, efficient memory usage plays a large roll in efficient code. From the results in table 1, we can see that the biggest difference came from accessing data in the right order and taking advantage of spatial locality of data. Last but not least, taking advantage of key cpu features such as vectorisation and letting the compiler intepret them in efficient machine code is important in high performance computing.