COMS30005 - Serial Optimisation

Karthik Sridhar
ks17226
October, 2019

1 Introduction

This report explains my approach and analysis of *serially optimising* a code in C that implements a weighted *5-point stencil* on a rectangular grid. The code is compiled using *GCC 9.1.0* and *ICC 18.0.3*, executed via a simple job script on a single node on BlueCrystal Phase 4 and tested via a python script.

2 Compiler Flags Optimisation

Optimisation flags are a part of the compiler that aim at improving the program's performance and execution time. *GCC* and *ICC* come with their own set of flags, each having their own improvements. Without optimisations (*gcc -O0*), I recorded initial run times of 5.88, 130.18 and 561.2 seconds for the three respective images. From the results as shown in *Table 1*, we can see

Compiler Flags	**1024**	**4096**	**8000**
gcc -O1	2.00	37.67	156.34
gcc -O2	2.00	37.23	156.02
gcc -O3	1.98	37.08	155.97
gcc -Ofast	1.18	36.35	139.48
icc -O1	2.00	37.33	156.07
icc -O2	1.79	28.76	109.60
icc -O3	1.79	28.76	109.60
icc -Ofast	0.24	5.85	23.35
icc -fast	0.18	5.75	23.28

Table 1: Runtimes (in seconds) with various compiler flags on BCp4.

that *gcc* -O1 significantly improves the run time from the original as it reduces the code size, leading to faster execution time. However, this comes at the cost of compilation time. *gcc* -O2 calls -O1 and -O3 calls -O2, both aiming to further improve execution time compromising for memory, but do not provide any noticeable improvement. *gcc* -Ofast calls -O3 and also enables mathematical optimisations through *-ffast-math*, improving the time of the former.

On the other hand, compiling with *icc* had quicker run times. *icc* -O2 and -O3 had similar run times, but effectively quicker than *icc* -O1 due to *vectorisation* being enabled. Vectorisation allows the program to make use of additional registers, thereby making it more efficient. *icc* -Ofast and -fast add further optimisations in terms of *precision of division* while the latter also maximises the speed of the entire program.

Based on these results, I decided to continue with *icc -fast* as it returned the best result with times 32X, 22X and 24X faster than the initial times of the three respective images.

3 Code Changes

This section describes my incremental changes to the code.

3.1 First Change - Division Operations

Unlike Addition and Multiplication, Division has a more complex computation and thus is more costly. I replaced the divisions *3.0/5.0* and *0.5/5.0* with constants of 0.6 and 0.1 respectively. This did not significantly improve the run time with *icc -fast*, but achieved an approximate 2X speed up when compiled using *gcc -O1, -O2 and -O3* suggesting that the latter do not pre-compute division.

3.2 Second Change - Using Cache Memory

Initially, the *stencil()* function iterates through the image columns then rows, making the process *column major*. This does not fully utilise the cache line, causing unnecessary fetching of elements. I swapped the order of the for-loops to ensure the image is stored in a one-dimensional row major order. This is because C is a row major language and uses *Spatial Locality*. This allows us to use cache memory effectively as the cache line will have the next elements already loaded when needed, allowing the pre-fetcher to bring in the neighbouring elements in advance.

3.3 Third Change - Changing Datatypes

My third major change in code involved converting the image pointers from *Double* to *Float*. This also involved changing the Double constants to Floats. This is because floats are single precision and have a smaller size of 4 bytes as compared to double (8 bytes) on a regular 64-bit hardware. This makes operations *less* costly, leading to faster access and better cache performance.

Code Changes	1024	4096	8000
Original Code	0.18	5.75	23.28
First Change	0.17	5.73	23.21
Second Change	0.17	5.73	23.20
Third Change	0.10	2.92	11.24

Table 2: Run times (in seconds) for all images with *icc -fast* on BCp4 for each stage of code change

From *Table 2*, we can see that the run time for all images when compiled with *icc -fast* drops by almost 50% after using floats. This is because ICC is a native compiler on BlueCrystal and when optimised with *-fast*, enables aggressive optimisations on floating-point data.

4 Further Optimisations

In the critical section of my program, I stored the central cell position in a variable and used that as a reference to get the neighbouring cells. Furthermore, I added all the neighbouring cells first, then multiplied with a common factor of 0.1f. This helped me reduce the number of floating point operations from 9 to 6. I used *godbolt*'s compiler explorer to view the assembly of my code before and after the changes. When compiled with *ICC*, the updated version took fewer instructions to achieve the same result.

5 Analysis

As my *stencil()* function runs in a row major order, the memory is contiguous. This allows optimisation in terms of vectorising the for-loops. I initially added the *vector align* pragma above the function to ensure the ICC compiler always vectorises the loop, but removed it as the *-fast* optimisation flag auto vectorises certain for-loops in the entire program. I confirmed this by studying the vectorisation report generated by adding the *-qopt-report=1 -qopt-report-phase=vec* flags in my Makefile.

To understand more about the bottlenecks related to the vectorisation, I performed a roofline survey using Intel Advisor 2018. I proceeded to generate a Cache Aware Roofline model to understand the upper bound performance of my optimised program.

I have 6 floating point operations and 6 memory loads (4 bytes each) in the *stencil()* function which correspond to an arithmetic intensity of 0.25 FLOPS/Byte. My final run times for the three respective images were 0.1, 2.92 and 11.24 seconds when compiled with *icc -fast* version 18.0.3. With these run times, I recorded performances of 12.5, 6.9 and 6.8 GFLOP/s for the three respective images.

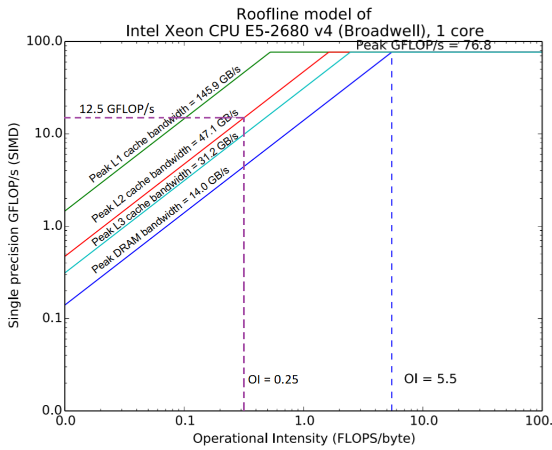

Figure 1: Cache Aware Roofline Model for 1024x1024

From *Figure 1*, we can see that the program is memory bound. With fewer floating point operations, I was able to achieve a 1.2X performance speed up over the original code (10.4 GFLOP/s) for 1024x1024 when compiled with *icc -fast*.

6 Potential Improvements

To further improve the performance of my code, I tried to incorporate *tiling* in my stencil() function so that the data fetched can be reused effectively. However, this caused my run times to increase potentially because of optimisation flag constraints.