Introduction to High Performance Computing
Assignment 1: Serial Optimisation

user id: ya17227 name: Yash Agarwal

Introduction

For this assignment we have a code that implements a weighted 5-point stencil code on the rectangular grid. The goal is to discover and experiment with a myriad of optimisation techniques available. With all these optimisation we must run our code as fast as we can with minimising the energy resources utilised and maximising the performance.

After logging-in to Blue Crystal via ssh key we try to build the *stencil.c* file using
gcc -std=c99 -Wall stencil.c
gcc is the GNU c compiler, *-std=c99* means we are using the C version 99 and *-Wall* enables all warnings.

Compiler choices & flags

The default compiler for Blue Crystal phase 4 is *gcc version 4.8.5*. The latest gcc version available is *gcc-9.1.0*. The code was found out to having similar runtimes for both. The intel compiler *icc* made the code use significantly less time to run as shown in the figure below. I am using the *intel/2018-u3*.

The First step should always be to build the code without using any optimisations to check for the correctness of the code. I did this by using *-O0* flag which turns off all optimisations if enabled by default. After the successful build, I began testing with the general optimisation flags *-O1*, *-O2*, *-O3*, *-Ofast*, *-xHOST*.

Flag	Runtime(s)	Speed
-O0	6.03	1x
-O1	2.02	3x
-O2	1.79	3.3x
-O3	1.79	3.3x
-Ofast	0.24	25.1x
-Ofast -xHOST	0.19	31.7x

As we can see in the figure the last flag has exponentially speeded up the performance by 31.7x times. The *-O1* flag optimises for size which tries to create the smallest optimised code. The *-O2* flag maximises for speed which enable optimisation like vectorisation. The *-O3* flag enable all *-O2* and further optimisation like loop and memory access optimisations like scalar replacement, loop unrolling, loop blocking to all more efficient use of cache and additional data prefetching. The *-Ofast* optimises for speed across entire program. The *-xHOST* optimisation targets the architecture. It detects what processor it is running and optimises for that processor in particular which explains the boost in speed.

Reducing Data Footprint

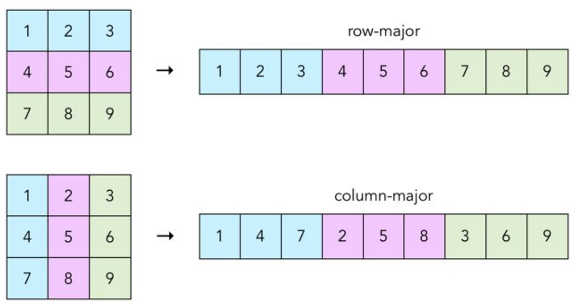
Arithmetic is performed on the data in registers, which requires data to frequently flow from memory into and out of the registers. The program does floating point operations inside the for loops of *stencil()* function. Currently we are using *double* data type for the image variables. The *double* data type is a double precision (64-bit) floating point data type . Each *double* variable occupies 64-bits or 8-bytes. The Cache lines are 64-bytes long which means that only 8 variables are stored in one Cache line.

The *float* data type is a single precision (32-bit) floating point data type whose each variable occupies only 32 bit. By changing the data type in the stencil function from *double* to *float* we can store 16 variables in one Cache line, i.e. exactly double. By having *float* data types we will lose precision but it will not damage our compute. It reduces data footprint, i.e. less data needs to be moved around. Since our program is memory bandwidth bound this helps us a lot in optimising our program. It increases the speed by approximately 50%.

Runtime using double(s)	Runtime using float(s)
0.19	0.10

Memory Access patterns

We can implement the memory access pattern by implementing the Row Major layout in loops.



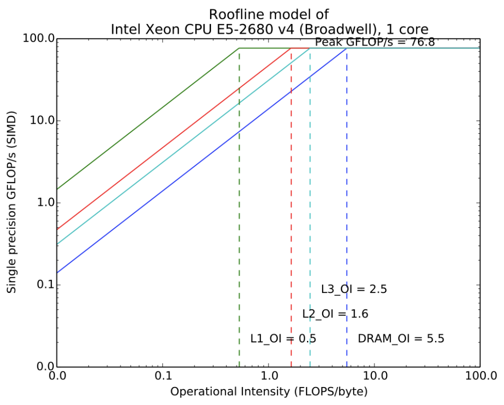
All the data in memory are stored in a 2d layout. So it is very important to follow a Row-Major layout and not Column Major layout. In the above figure the first matrix is stored as Row Major layout. This means the elements in the first row are stored left-to-right then the 2nd row elements left-to-right and so on. Whereas in 2nd matrix, the elements in 1st column are stored up-to-down then the 2nd row elements up-to-down and so on.

Now let's say we need to read the array in increasing order of their indexes, i.e. like 1,2,3,4,….,9. For the first matrix, we can easily do that since it is stored in that order. But, for the Second matrix we'll have to access 1,2,3,…,9. After reading 1 we'll have to go to the 2 which is stored three slots away from 1. Sometimes when large matrices are stored, the subsequent values can be stored in different place in the memory far away. This decreases the memory bandwidth and impairs the performance. Hence, it is vital that we use Row Major layout. This could make us get more cache hits and less cache misses.

Improve Performance by observing Roofline Model

We can use the knowledge of the data types and cpu type to predict how much resources we are using. A flop is a floating point operation. A single calculation the cpu does. We are either caped but the memory band width bound or compute bound. If the program has higher FLOPS than the cpu, it's a cpu speed problem. Otherwise it is probably a memory access problem that is making the program slow down.

I have increased the performance by making 1 store instead of 5 originally, 10 floating point operations instead of 15 originally in the *stencil()* function. The 32 bit float data type also helps in having more flops since each task requires less memory to run now and can do more on one line. This decreases the operational intensity which in turn improves performance.



Conclusion

By engaging in this coursework, I have encountered a lot to optimisation techniques which make the program store more data in cache, optimise code in accordance with the target platform. Find out if a program is memory bandwidth bound or compute bound. Storing data in the correct order can be very vital in some cases. How to get more cache hits. All these optimisations are not limited to the stencil code we used in our coursework and are universally applicable.

Image size	Runtime (s)
1024 x 1024	0.10
4096 x 4096	2.93
8000 x 8000	11.10

references:
https://software.intel.com/sites/default/files/managed/c1/61/compiler-essentials.1.pdf

http://www.physics.udel.edu/~bnikolic/QTTG/shared/docs/quick_reference_optimize_intel_compilers.pdf