HPC - Serial optimisation

Martin Dimitrov
md16024 1644531

1 Introduction

The objective of this report is to outline my research on serial optimisation. In particular, it focuses on the improvements I have made for *stencil* - a program written in C that runs on a single CPU core. With the help of profilers and reports, I will describe my choices for compilers, flags used and code optimisations.

2 Compiler choice and Optimisation flags

Optimisation flags are inbuilt compiler features. They attempt to improve performance and/or code size at the cost of compilation time, memory usage and in some cases the ability to debug[1]. Optimisation levels like *-O0,1,2,3* and *-Ofast* are no more than just groups of flags.

-O2 tries to increase code performance without sacrificing memory. *-O3* is a super-set of *-O2* which trades memory for speed and enables *ftree-vectorize* for loop vectorisations. *-Ofast* is *-O3* with the addition of some math-specific flags [2]. Some of the optimisations relevant to *stencil* include caching repeated calculations like $(j + i * height)$, combining multiple commands on the same variable into one (in our case calculating the $tmp\_image[x]$ element in one command instead of 5 different ones) and precomputing hardcoded arithmetic.

Different compilers can also affect run time. I experimented with the 2018 Intel C Compiler (ICC) and the GNU Compiler Collection 9.1.0v (GCC). Note that ICC uses *-O2* as a default, unlike GCC which defaults to *-O0*. Comparing them with no optimisations shows no significant difference across all image sizes. However, increasing the optimisation levels reveals that ICC tends to be faster especially with *-O3* and *-Ofast*.

In *Table 1* I have compared run times using ICC and the flags above. At first glance, there is little to no difference between *-O2* and *-O3*, however, looking at the compiler report after running *-O3* reveals that the code could not take advantage over vectorisation. *-Ofast* proves to be the fastest - 13x faster than no optimisations at all (*ICC -O0*) and up to 23x for the smaller 1024 image.

Table 1: Run times using ICC and different optimisation levels. Times are in seconds rounded to the first decimal.

size/flag	O0	O1	O2	O3	Ofast
1024	4.7	2.0	1.7	1.7	0.2
4096	77.1	32.0	28.7	28.7	5.8
8000	291.7	122.7	109.5	109.6	23.2

The flag *-xHost* forces the compiler to use the highest instruction set available on the CPU [3]. However, in our case I was compiling the code on a Blue Crystal's login node and executing it on a compute node which may lead to a mismatch. To fix this issue I used *-xBROADWELL* flag, which uses the Broadwell architecture used in the compute nodes.

3 Code optimisations

Most of the code optimisations I have done become obsolete as they are automatically added via optimisation flags.

3.1 Static arithmetic

One of the easier optimisations is precomputing hard-coded arithmetic. In our case replacing $3.0/5.0$ with 0.6. Not only it reduces the number of operations but also removes division, which is more computationally heavy than addition/multiplication. Given the *-O3* flag is not being used already, applying this to *stencil* gives a 3x speedup (using *ICC -O2*).

Reducing the number of operations also decreases the Operational Intensity (IO). Factoring out the multiplication (** 0.1*) reduced the total amount operations per cycle from 9 down to 6 therefore reducing the OI from 0.375 to 0.25.

3.2 Caching

A good way to increase performance is to cache regularly used data. The image in *stencil* is a one-dimensional array initialised in row-major order, meaning that row elements are contiguous in memory. However, the double loop manipulating that array is accessing it in a column-major manner. By simply inverting the loops order ensures that each consecutive iteration operates on a neighbouring data address in memory which is normally already cached. That leads to an increase of cache hits and ultimately to a 2x speedup (tested with *ICC -O0*). Using *-Ofast*, however, makes this change obsolete as it's already being optimised automatically in compile time.

3.3 Vectorisation

Vectorisation is the idea of executing a *Single Instruction* on *Multiple Data* (*SIMD*), given there are no data dependencies.

Reading Intel's compiler report unveils that *ICC* automatically vectorises the main loop in *stencil* with a vector length of 4. Adding the *restrict* keyword to the image pointers makes them *non-alias* which further speeds up run times. As a note, using *restrict* with *Intel's 2017 C compiler* breaks the integrity of the image, hence, *check.py* failing the check. The 2018 version fixes this.

3.4 Data type

At first, the data type of the image array was *double*. Changing it to *float*, however, reduces the size of each element from 8 bytes down to 4. For this to take affect, another small change needs to be made: adding the suffix f to the multiplication scalars (e.g. 0.6 becomes $0.6f$). This ensures that all types are *float*, allowing the compiler to skip unnecessary type casting commands. Not only that but reducing the data size allows vectorisation with a vector length of 8 instead of 4. After making these tweaks, there was a drastic improvement in run time: 2.5x speed improvement across all image sizes (tested with *ICC -Ofast -xBROADWELL*).

4 Conclusion

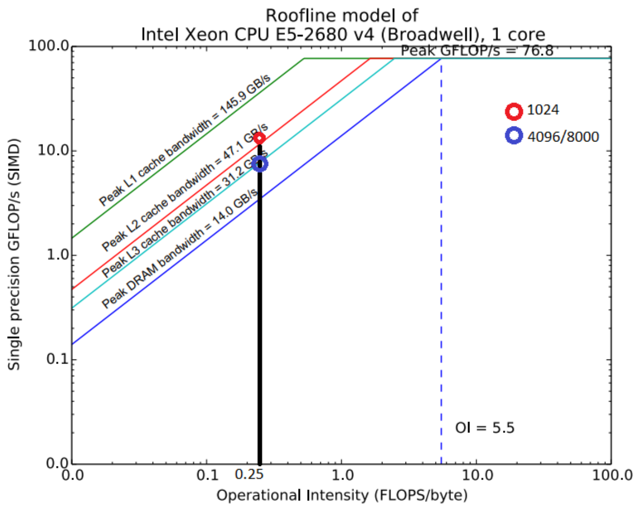
Finally, to see how everything comes together, I compiled the optimised code using *ICC 2018* with *-Ofast -xBROADWELL* flags. The comparisons between the original run time (*GCC* and no optimisation flags) and the optimised one are displayed in *Table 2*.

Table 2: Final vs Original run times in seconds

size/code	Original	Optimised
1024	5.9	0.09
4096	130.1	2.9
8000	561.1	10.8

The original code computes the 1024 image in 5.9 seconds - 0.3 GFLOP/s. The optimised version which does it for 0.09 - 18 GFLOP/s (with an OI of 0.37) or in other words a 60x increase.

Table 3: Optimised code performance plotted on a roofline model



References

[1] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] https://wiki.gentoo.org/wiki/GCC_optimization

[3] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-xhost-qxhost