Introduction To High Performance Computing

Serial

Adam Sadiq - as16165

1 Introduction

In this paper, I will describe and analyse the methods and techniques utilised to improve the efficiency, and therefore runtime of the stencil.c code.

2 Optimisations

2.1 GNU Compiler Collection

The easiest thing to increase optimisation was to tell the compiler to optimise as much as it could. There were many optimisation flags to choose from in the GNU. The natural one to use was 'O3', which seemed to be the highest flag for optimisation. 'O3' increases both compilation time and the performance of the generated code, to ensure runtimes are as fast as possible. Yet there was another flag option called 'Ofast'. Ofast does all of the O3 optimisations, yet also enables optimizations that may not be valid for some programs [1]. Testing it on the stencil.c code, I saw that it did compile, and the difference between O3 and Ofast was considerable.

2.2 Intel Compiler

To verify if my base compilation would be the fastest, I continued to empirically check other compilers. I started using the Intel-compiler-16-u2, exploring different flags available to see if I could get even more optimal results. Coming across the xHOST flag, it mentioned that generates instructions for the highest instruction set available on the compilation host processor [2]. Testing it empirically gave better results than the GNU Compiler.

	1024	4096	8000
base	8s 219ms	6m 9s 793ms	Overflow
O3	6s 801ms	1m 55s 488ms	5m 58s 116ms
Ofast	2s 294ms	1m 50s 233ms	2m 11s 280ms
xHOST	2s 126ms	1m 33s 759ms	1m 45s 240ms

2.2.1 Profiling

After seeing these times, it was very important to see what part of the code was a bottleneck. By using the gprof profiler, it was evident that the stencil function was the limiting factor; in particular, the 5 lines inside stencil.c:

2.2.2 Arithmetic Simplification

The arithmetic operations used were generally inefficient. This is because divisions use more clock cycles than other operations. I then simplified the (3.0/5.0) to 0.6, and (0.5/5.0) to 0.1. I saw that the difference made however was marginal. This is because the compiler will usually simplify these results before runtime.

2.3 Row vs Column Major Order

Processor Speed is much faster than memory speed. Therefore, redoing calculations is minimal compared to getting data from memory. Inside the C language, as data is loaded into the cache, other elements across the row are also called - following a Contiguous Access pattern. If this step is not followed, Cache Thrashing occurs. This is where you attempt to find data in memory where it is not present, making further ventures to the larger caches and memory hierarchy. The original code was not following a contiguous access pattern, following a column-major order, alike to Fortran's convention: I swapped the $i's$ and $j's$ in the for-loop to reduce cash thrashing.

1024	4096	8000
659ms	10s 784ms	40s 944ms

2.4 Conditional Loops & Data-Types

Branching in loops can stall the pipelines in modern processors. This is because it is hard to predict if the certain statements in the loop will execute or not depending on the varying conditions. Therefore, I knew that I had to cover all the cases that the if conditions would do. I divided the loop into 6 cases: Corner, top, left, right, bottom and middle. Though it resulted in 'more code', removal of the conditions provided faster results:

Because you can store the same information in the cache using less space, I queried the precision values of a float compared to what was needed. Realising it would be sufficient, using floats halved my runtimes. It is also important to state that with optimisation in relation to branching in loops, it varies depending on the compiler you use. I then decided to revert back to my original and see if it could be faster than it currently was. Somehow, the GNU Complier gave me my fastest results!

1024	4096	8000
99ms	2s 506ms	8s 661ms

3 Conclusion

Evaluating my program, using operational intensity, defined: as operations per byte of memory movement - My program was memory bandwidth bound, rather compute bound. However, from the initial operational intensity being: $OI_{fp64} = 9/((5\text{r} + 1\text{w})*8) = 0.1875$: from my double $\rightarrow$ $float$ conversion, $OI_{fp32} = 9/((5\text{r} + 1\text{w})*4) = 0.375$, my operational has intensity doubled.
In conclusion, I have learned how to create optimised programs, not only through algorithmic complexity, but also understanding how to help the compiler optimise effectively, through Memory Hierarchical Techniques, avoiding Disk Thrashing, following convention, yet most of all, empirical experimentation!

4 Sources used

[1] Gcc.gnu.org. (2018). Optimize Options - Using the GNU Compiler Collection (GCC). [online] Available at: https://gcc.gnu.org/onlinedocs/gcc-4.9.1/gcc/Optimize-Options.html [Accessed 24 Oct. 2018].
[2] Software.intel.com. (2018). xHost, QxHost. [online] Available at: https://software.intel.com/en-us/node/522846 [Accessed 25 Oct. 2018].