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 Compiler and flags

To undersand how much of an order of magnitude I would be trying to decrease my runtimes, I done a base run for all the pgm sizes.

1024	4096	8000
8s 219ms	6m 9s 793ms	10m 7s

2.1 GNU Compiler Collection

After seeing these times, it was very important to see what part of code was being the bottleneck. This was important to save time, we can always optimise small parts of code, yet generally there is always one or multiple limiting factors in particular parts of code. By using the pgrof profiler, I was that the the stencil function was the limiting factor. Yet in particular, it was the 5 lines inside stencil.c:

$$\begin{aligned} tmp\_image[j+i*ny] &= image[j+i*ny]*3.0/5.0;\\ if(i\ 0)tmp\_image[j+i*ny]+ &= image[j+(i-1)*ny]*0.5/5.0;\\ if(i<nx-1)tmp\_image[j+i*ny]+ &= image[j+(i+1)*ny]*0.5/5.0;\\ if(j>0)tmp\_image[j+i*ny]+ &= image[j-1+i*ny]*0.5/5.0;\\ if(j<ny-1)tmp\_image[j+i*ny]+ &= image[j+1+i*ny]*0.5/5.0; \end{aligned} \tag{1}$$

2.2 GNU Compiler Collection

2.2.1 Ofast

The easiest thing to increase optimisation was to tell the compiler to optimise as much as it could . There were many optimisation flags to chose 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 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.

	1024	4096	8000
O3	6s 801ms	1m 55s 488ms	5m 58s 116ms
Ofast	2s 294ms	1m 50s 233ms	2m 11s 280ms

2.3 Intel Compiler

2.4 Base

Just because gcc was the main compiler out there, it didn't mean that that it would be the most optimised. Therefore I continued to empirically check other compilers. Using the intel-compiler-16-u2, I achieved these results.

1024	4096	8000
2s 132ms	1m 44s 650ms	1m 48s 289ms

xHOST Flag

I then explored different flags available for the intel compiler to see if i could get even more optimal results. Coming accross the xHOST flag, it mentioned that it optimised your code according to your local device. Testing it empirically gave slightly better results than the base intel compiler.

3 Arithmetic Optimisation's

4 Memory Hierarchies

4.1 Row vs Column Major Order

4.1.1 Cache Misses

4.1.2 Data-types

5 Analysis

5.1 Operational Intensity

6 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].