Serial Code Optimisations **George Lancaster**
Introduction to High Performance Computing User ID: qv18258

Introduction

This report outlines serial code optimisations applied to the stencil algorithm for assignment one. Speeds of 0.116 seconds, 2.422 seconds and 8.835 seconds were achieved for images of size 1024x1024, 4096x4096 and 8000x8000 respectively.

Optimisations

Compiler Choice and Flags

During optimisation, the program was compiled using both the GNU Compiler Collection version 7.1.0 (gcc), and the Intel Compiler Collection version 16 (icc). It is important to use multiple compilers to produce optimum instructions for the target architecture. In testing, the icc compiler outperformed gcc. This could be due to icc being developed for Intel processors, like the Intel Xeon E5 - 2670 processors used in BlueCrystal nodes.

The fastest time was achieved using the compiler instructions:

icc -std=c99 -O3 -no-prec-div -xAVX -Wall $^o $@

Flag	Description
-03	Aggressively maximise speed. Automatic vectorisation is enabled at this level.
-no-prec-div	Enable optimisations that give less precise results than IEEE division.
-xAVX	Enable processor specific optimisations for SandyBridge processors.

Table 1: Compilation flags used, and descriptions of their functionality.

Removal of Expensive Operations

The inner loop of the stencil function contained five divisions. Unlike multiplication and addition, division is a computationally expensive operation to perform. Division can be computed using fixed-length iterative subtraction. More advanced division algorithms require the use of division-free iteration to approximate the reciprocal of the denominator, which can then be used to calculate a result [1]. To optimise the code, the division in the *stencil* function has been replaced with calculated values ($0.5/5 = 0.1$). This means that less operations are performed in the inner loop. Compiling using the *-O3* flag gave the same results regardless if the divisions were removed as the compiler can guess that the result is the same on each iteration.

Additionally, the number of writes to the *tmp_image* variable has been reduced to one, from five. This reduces the number of times memory is accessed in the inner loop.

Data Types and Layout

The data type of the images has been changed from double to float. A double requires eight bytes in memory, whereas a float uses four bytes. By reducing the amount of memory used, more data can fit into the same sized register, therefore allowing more operations to be processed simultaneously. By doing this, the upper-bound of the peak flop rate is increased, meaning that the it is harder to become compute-bound by the hardware.

Other data types were experimented with, including: unsigned char (one byte), short int (two bytes) and int (four bytes). The float data type was 0.123 seconds faster than the double, whilst maintaining +-1% accuracy.

When summing the values of the surrounding pixels, explicitly casting the multiplier value as a float decreased execution time. By doing this, the compiler no longer has to implicitly assume the type of the variable as float, and therefore no computation is wasted.

Other improvements involved changing the order in which pixels are accessed, from column major to row major. The *stencil* function now loops through the image sequentially, of the form $[0, 1, ..., nx \cdot ny]$. The time taken to retrieve the next item in the array is reduced, as the surrounding values are held in the cache.

Vectorisation

Vectorisation allows multiple operations to be performed simultaneously within a loop by using Single Instruction Multiple Data (SIMD) operations. Adding the *__restrict__* keyword to the *tmp_image* variable in the *stencil* function, and compiling using the *-O2* or *-O3* flags tells the compiler that there is no aliasing between the pointer and any other locations in memory. This means that the compiler can auto-vectorise the inner loop.

Border Addition and Removal of Conditional Statements

The inner loop of the *stencil* function contained four conditional statements checking if the current position was an edge. Due to the frequency of the checks, they add up to a large amount of wasted computation. In addition to this, instructions are slowed down when there are conditional branching statements. Strategies such as branch prediction

[1] Karp, A.H. and Markstein, P., 1997. High-precision division and square root. ACM Transactions on Mathematical Software (TOMS), 23(4), pp.561-589.

and eager execution are used to try to improve the flow in the instruction pipeline [2]. Rather than relying on these automatic compiler techniques, removing the conditional statements brings an improvement to performance.

The conditional statements were removed by increasing the size of the image, and placing a border of zero valued pixels around its perimeter. When looping through the image, iteration starts at position $(x + y * ny) + nx + 3$ to begin on the first row of the true image. There is no longer any need to check if the current pixel is an edge pixel, as edge pixels are summed with the zero-valued border.

Performance Analysis

Intel Advisor rates the arithmetic intensity of the unmodified *stencil* code compiled using gcc at 0.027 FLOP/Byte, and the performance at 0.30 GFLOPS. In comparison, the optimised code compiled using icc achieved an arithmetic intensity of 0.370 FLOP/Byte and a performance of 15.69 GFLOPS. The theoretical maximum performance for the optimised stencil code, operating on single precision floating point numbers is 41.60 GFLOPS. Fig 1 and 2 show these results plotted as a roofline model. Although the code should be memory bound, Intel Advisor suggests that it is compute bound as shown in Fig 2. The reason for this is unknown.

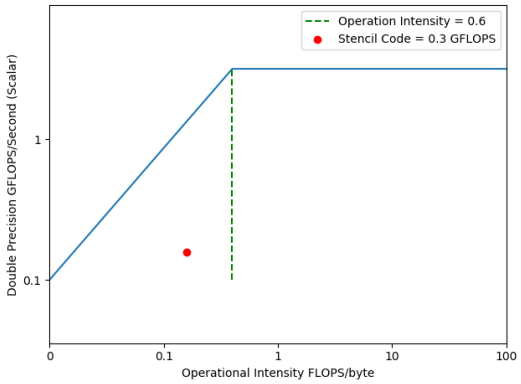

Figure 1: Roofline model for the unmodified stencil code with peak flop rate as double precision scalar.

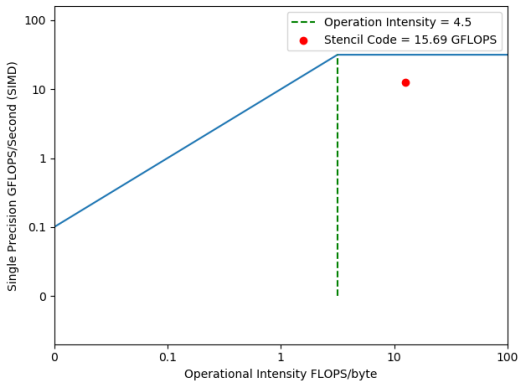

Figure 2: Roofline model for the optimised stencil code with peak flop rate as single precision SIMD.

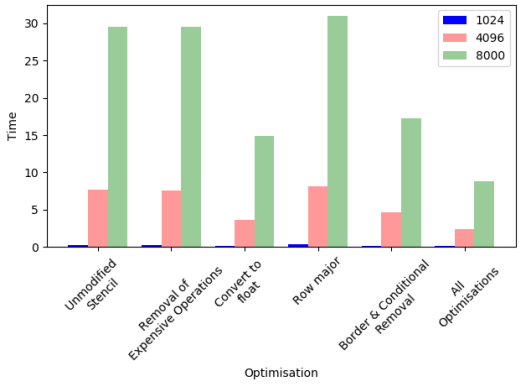

Figure 3: Plots of the optimisations applied.

	Image Size		
Optimisation	1024	4096	8000
Unmodified Stencil Code	0.259	7.631	29.486
Removal of Expensive Operations	0.26	7.623	29.505
Double to Float	0.136	3.6	14.849
Data Ordering Row vs Column	0.314	8.077	30.976
Border Addition & Conditional Removal	0.185	4.585	17.247
All Optimisations	0.116	2.427	8.835

Table 2: Benchmarks of the *stencil* code. All results were gathered using icc with optimisation flags.

The biggest improvements came from vectorisation, removing conditionals and changing the data type from double to float. The benefit of vectorisation is further improved by using float instead of double, as SIMD extensions are targeted for 16-bit operands. A register can fit twice as many 16-bit elements than 32-bit elements.

Not all optimisations were beneficial when applied alone, for example changing the data ordering to row major added 0.055 seconds to the unmodified code. When used with other optimisations, the cumulative effect was greater than if used alone.

Conclusion

Further gains to performance could be seen by the implementation of cache blocking, which could be implemented by traversing the image in smaller chunks rather than sequentially. Cache blocking improves performance by ensuring that each chunk is small enough fit into lower level cache, therefore reducing the number of reads to the higher-level cache.

[2]Lee, J.K. and Smith, A.J., 1984. Branch prediction strategies and branch target buffer design. Computer, (1), pp.6-22.