COMS30005

Harry Goode - hg15150

***Abstract*—The following report documents the optimisation techniques used to optimise the *stencil* code provided. The techniques outlined are each dedicated their own section in Method and, as far as possible, the techniques have been kept in chronological order. The optimisation was concluded once it was determined that the code was memory-bound, discussed in the conclusion.**

I. Introduction

The objective of this project was to optimise the C code, *stencil*. To assist judgement of progress, targets were given for each image size. For example, the target for image of size 1024x1024 was to achieve a sub **0.3s** run-time.

Before any optimisation had been attempted, the 1024x1024 image ran in time **8.17s** on the default GNU compiler, *gcc 4.7.7*. To optimise the code, several different methods were attempted. These are described in the Method section of the report.

II. Method

A. Compiler

The first method considered for optimisation was the choice of compiler. Initially, the program was compiled with the supplied GNU compiler, *gcc*. After loading the latest GNU compiler, *gcc 7.1.0*, into the environment, and experimenting with a few flags, the optimal time was 6.8s with a *-O5 -ffast-math* flag.

After this, experimentation with the Intel compiler, *icc 16.0.2*, was undertaken. Flags *-Ofast -xHOST* were seen to be substantial, resulting in the optimal time of **3.39**s.

For fairness, both compilers were run with the *-O0* flag. This resulted in a better time for the Intel compiler, and so for the time being that was the compiler of choice.

The reasons for these improvements of time is down to the ability of the Intel compiler to optimise loops with heavy float-point calculations and memory access for large data structures, as well more aggressive vectorisation, but this will be talked about later in the report.

The *-xHOST* flag detects the architecture and tailors compilations as appropriate, further optimising performance.

B. Profiler

Rather than guessing which lines of code were the limiting factors, the gprof profiler was used to determine exactly where to begin the optimisation. The initial results showed that 99% of the time spent was in the loop of the *stencil* function. This technique was used throughout, ensuring that the focus of the work was spent on the limiting factor of the code.

C. Arithmetic Operations

Division is an inefficient operation. This was such that, upon first glance, it became apparent to replace the division operations with multiplication operations of the reciprocals. This produced an improvement to **0.328s**, a speedup of factor 10.

This method produced optimisation as division is an iterative algorithm, whereas multiplication and addition can be optimised into a series of bit-wise operations, therefore a much more desirable operation to perform. As these operations were applied a large number of times, this resulted in the largest speedup of all the optimisation techniques used.

D. Memory Access

Accessing recently loaded values meant that they were stored in higher cache memory. This results in quicker memory access. Therefore, the loop was altered such to be row major rather than column major. This reduced time for memory access, resulting in a time of **0.299s**. While not huge, significant enough to include, particularly as the project scales.

E. Data type

Reading and writing to memory was the next optimisation to be performed. The reason for this simply was that it was the easiest improvement to make. The *Double* data type is 8 bytes in size. This means that every read or write operation, without optimisation, is 8 operations. Compare this to the *Float* data type, of size 4 bytes, it is twice the size. Therefore, by converting all relevant data to *Float* data type significantly reduces the amount of operations required and as such resulted in a time of **0.256s**, and a speedup of 0.22X.

One reason for this, seen using that Godbolt tool, was that a lot of the memory access was already optimised. For instance, if a loop continually wrote to the same space of memory, then the compiler optimised this by only writing to memory after the loop iteration was

completed. This reduced the number of memory access operations performed, which explains why the speedup was minimal.

Another reason for this small speedup is that the operations being performed on the data were still double precision floating point numbers. This meant that, whilst the number of memory access operations were reduced, the data was being converted back into double precision to perform the operation, and then back into single precision. By making all arithmetic operations single precision floating-point operations, the resulting time was **0.221s**. This results in a total speedup of 0.32X.

Another reason that changing from *double* to *float* improved performance is that by halving the data type, this doubles the amount of information being held in each level of cache. This results in faster memory access and so improves performance.

This technique was also attempted with the *unsigned char* data type, as the end values ranged from 0 to 255. However, this data type did not have high enough precision so was quickly abandoned.

F. Vectorisation

After running a new profiler, *Tau*, it still showed the majority of the time was being spent in the stencil loop. Therefore, one final optimisation technique would be attempted, vectorisation.

Vectorisation is the process of running a single instruction over multiple data (SIMD). Due to writing to a temporary image, rather that overwriting the image itself, our loop was prime for this optimisation.

After checking that the compiler was not already vectorising this loop, by generating a vectorisation report, there were two areas to address. The *restrict* keyword was used to tell the compiler that the data structures were running dependency free, and so vectorisation would be safe. This resulted in a time of **0.148s**. The vectorisation report indicated that the loop was not fully vectorised however, and so fully optimisation was required.

The branches seen in the code needed to be removed, as the compiler was not vectorising fully because the branching could have lead to dependencies. Therefore the code was restructured to handle each edge case with no conditionals. After running a vectorisation report to confirm that the loop was fully vectorised, the resulting time was **0.0895s**.

Vectorisation improves performance by simultaneously running single operations on data that does not depend on previous operations in the loop. By not fully optimising, each operation was being calculated one by one, rather than simultaneously. This process is achieved by saving the data in a vectorisation register to perform simultaneous operation. By halving the data size, from *double* to *float*, it doubles the amount of data stored in this vectorisation register and so aid vectorisation.

III. Results

Table I shows compound optimisation run-times on the 1024x1024 image.

TABLE I
DEVELOPED TIMES

Change	Time (s)
No update	8.173803
gcc flags	6.807103
icc compiler	3.398744
Arithmetic operations	0.328454
Memory access	0.298996
Data type	0.256847
Single precision operations	0221548
Restrict (vectorisation)	0.147921
Branch removal (vectorisation)	0.089491

Table II shows the final run-times on the three sizes of image.

TABLE II
FINAL TIMES

Image size	Time (s)
1024x1024	0.089491
8000x8000	8.371515
4096x4096	2.441066

IV. Discussion and Conclusion

The report so far has shown improved performance, relative to itself. However, it is necessary to compare the performance with the potential performance of the hardware. The algorithmic complexity of the code was calculated to be O(n^2). The operational intensity at the limiting section of the *stencil* code was 0.25 FLOPS/byte. Blue Crystal Phase 3 has operational intensity of 4.5 FLOPS/byte. This shows that *stencil* is memory bandwidth bound. As the images increase in size, the data cannot be completely stored in cache and so is stored in DRAM. As no improvements to this can be deduced, it is possible conclude sufficient optimisation has been achieved. This therefor concludes the report