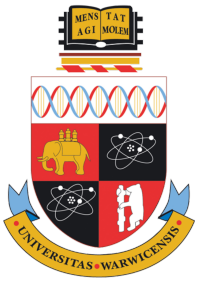


University of Warwick

CS257 Advanced Computer Architecture Coursework Assignment Report

Andrei-Daniel Nicolae
u1516780

March 15, 2017

Contents

1. Introduction

This coursework assignment is concerned with increasig the performance of a given program that computes scientific data, such that performance indicators like *time/s* and *GFLOPs/s* are improved as much as possible, while maintaining the precision of *Answer* as close as possible to the one achieved by the original code. This report shows the results achieved by incrementally applying code-refactoring optimisations and programming techniques such as Single Instruction Multiple Data(SIMD) Vectorisation Intrinsics, specifically Streaming SIMD Extensions(SSE), and multi-core threading to allow parallel execution of operations. The test results presented in this report were achieved by calculating the average result of three runs using a DCS machine that has an Intel® Core™ i5-4590 CPU @ 3.30 GHz processor with peak GFLOP/s specified in Equation 1 below.

$$\begin{aligned} \text{Peak GFLOP/s} &= 4 \text{ cores} \times 4 \text{ SSE units} \times 2 \text{ instructions per cycle} \times 3.30 \text{ GHz} \\ &= 105.6 \end{aligned} \tag{1}$$

2. Loop 1

Running the program for the first time using the suggested simulation (*stars = 1000*, *timesteps = 1000*), it became apparent that Loop 1 is the bottleneck, accounting for more 99% of the running time. Thus, Loop 1 was the first to be optimised.

2.1 Optimisation through vectorisation

Vectorisation was approached manually using Intrinsics in order to gain greater control than the compiler auto-vectorisation.

The inner loop was unrolled by a factor of 4 to help align data into cache-lines. The single-precision float variables have been converted to 128-bit single-precision vector units. SSE have a register width of 16 bytes, therefore a vector operation can consist of up to 4 packed single-precision (32-bit) floating-point elements [3]. Using 4 registers of 16 bytes leads to utilising 64 bytes, which are perfectly aligned on the cache-line, as the processor used has a 64 bytes cache line (checked using the linux *getconf LEVEL1_DCACHE_LINESIZE*).

Vectorisation allows for 4 operations to be computed at once on 4 different values, compared to the unvectorised code which can only do operations on single array elements. The *_mm_set1_ps()* function was used in order to broadcast the floating point values of x[i], y[i], z[i] to vector units. In order to improve locality, the broadcasting is done outside the inner loop as the values are only bound to i, being unnecessary to do the operation for each j.

Vectorisation allowed for the elimination of Loop 0, using *_mm_setzero_()* to obtain three vectors with all elements set to zero while increasing speed.

Four elements of the arrays x, y, z(j,j+1,j+2,j+3) are loaded using *_mm_load_ps()* into a vector, to be used into further arithmetic computations, such as subtracting x, y, z(i, i+1, i+2, i+3) from it.

The next step in vectorising was to make use of Intrinsics functions for arithmetic operations, such as *_mm_sub_ps()*, *_mm_add_ps()* and *_mm_mul_ps()* for subtraction, addition and multiplication respectively. These functions take two vectors as inputs and return a dst vector containing the results[1].

2.1.1 Precision

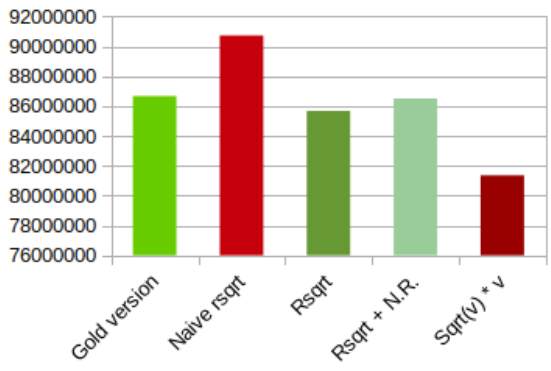


Figure 1: Precision of different rsqrt methods.

In optimising SIMD instructions available, one of the operations that affect performance and precision is the reciprocal square root. For full accuracy but slow performance, the division and square root operations are preferred [2]. The computational latency is different for each processor architecture, but it is worth mentioning that the *_mm_sqrt_()* latency for the Haswell architecture is 13, whilst the latency for Knights Landing is 38 [1]. According to [2], *_mm_rsqrt_()* (latency of 5 [1]) is recommended when reduced accuracy is acceptable. "If near full accuracy is needed" [2], *_mm_rsqrt_()* and a Newton Rhapson iteration is recommended. Furthermore, *r6inv* can be reduced as Eq. 2:

$$\text{r6inv} = \frac{1.0f}{\sqrt{\text{r2}}} \times \frac{1.0f}{\sqrt{\text{r2}}} \times \frac{1.0f}{\sqrt{\text{r2}}} = \text{r2}^{\frac{3}{2}} = \sqrt{2}^3 = \sqrt{\text{r2}} \times \text{r2} \tag{2}$$

Each of the aforementioned were tested on 1000 stars and 1000 timesteps. The best in terms of speed and loss of precision was *_mm_rsqrt_()*, as shown in in Figure 1 above.

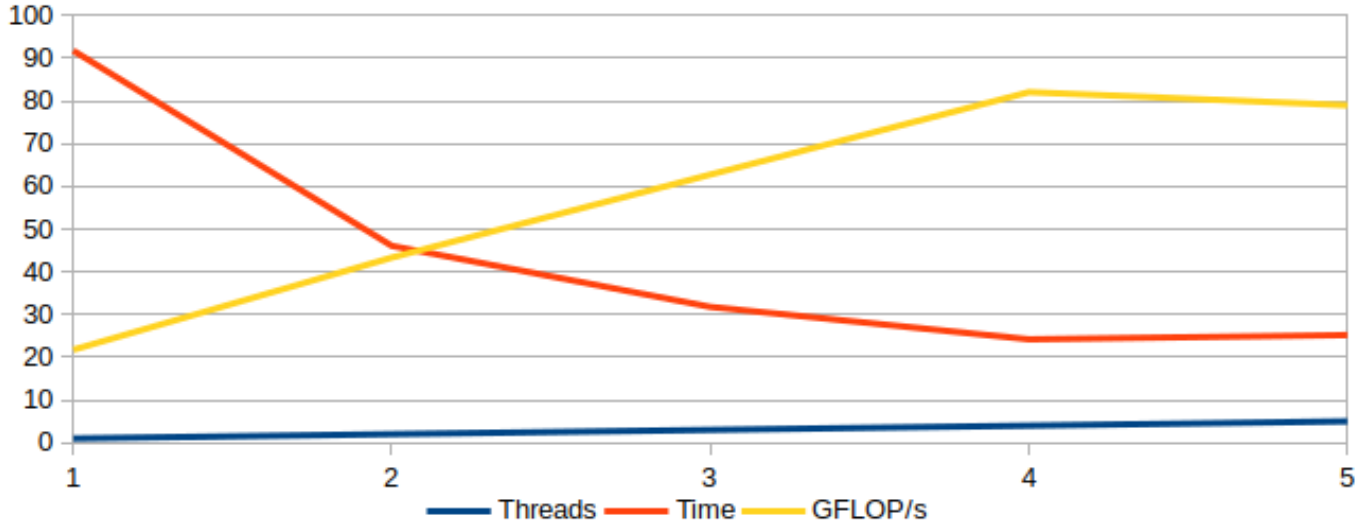
2.2 Optimisation through multi-core threading

Using multiple threads improves the overall performance of the program because the workload is split between all cores available, making possible for computations to be run in parallel.

The OpenMP API was used in order to achieve multi-threading. The following pragma extension was used: *#pragma*

omp parallel for num_threads(4) schedule(dynamic, 64) and the following was used to avoid race conditions caused by variables that would be affected if multiple threads would perform computations on them: *default(shared) private(rx_v, ry_v, rz_v, r2_v, r2inv, r6inv,s)*. The reason for the choice of schedule is to divide the loop into chunks of 64 iterations and assign new blocks to threads upon completion. This makes perfect use of the cache line (which is 64 bytes as specified above) and avoids false sharing as threads don't access memory locations that share cache-lines with memory locations accessed by other threads. By adjusting the number of threads, the following running times and GFLOP/s are achieved:

Figure 2: Threading performance based on number of threads.



As shown in Figure 2, the peak performance in terms of speed and GFLOP/s increase was achieved using 4 threads. Trying to use more than 4 decreases performance due to the limitations of the quad-core processor used, which is capable of supporting only one thread per core and does not support Hyper-threading.

2.2.1 Further improvements

The following could have been attempted: *reduction(+:ax[:i], ay[:i], az[:i])* to improve performance by updating the array[i] value shared across all threads. However, the ability to reduce arrays has been introduced in OpenMP 4.5, while the machine used only supports OpenMP 3.1 specification [4] (based on the version and gcc and cited source).

2.3 Testing

	Loop 0	Loop 1	Loop 2	Loop 3	Total	Answer	GFLOPs
Gold version (1000 x 1000)	0.002346s	10.15s	0.002550s	0.003331s	10.16s	86672752	1.96
Gold version (7000 x 500)	0.005720s	244.6s	0.007434s	0.011828s	244.68s	2829120000	2.00
Gold version (10000 x 1000)	0.015882s	995.6s	0.021111s	0.038470s	995.67s	1098840320	2.00
Vectorised version (1000 x 1000)	0.000000s	1.04s	0.000988s	0.001422s	1.04s	85672464	19.1
Vectorised version (7000 x 500)	0.000000s	22.51s	0.002056s	0.003960s	22.52s	2902899200	21.7
Vectorised version (10000 x 1000)	0.000000s	91.9s	0.006808s	0.011401s	91.91s	1140634368	21.7
Final version (1000 x 1000)	0.000000s	0.29s	0.001244s	0.001743s	0.29s	85672464	64.7
Final version (7000 x 500)	0.000000s	6.05s	0.003549s	0.005498s	6.04s	2902899200	81.7
Final version (10000 x 1000)	0.000000s	24.20s	0.007378s	0.016064s	24.2	1140634368	82.1

Table 1: Results on different simulations (stated in parentheses) and versions.

The peak performance in terms of GFLOP/s and time was obtained while simulating 10000 stars and 1000 timesteps, as shown in Table 1. Multiple simulations were run in order to test the consistency of the optimisations. As observed, It can be observed that the following percentage differences are maintained between simulations: Answer (∼3%), Gold version compared to vectorised version (∼160%), Gold version and Final version (∼188%).

3 Loops 0, 2 and 3

3.1 Optimisation

To begin with, the loops showed poor temporal locality as they contained unrelated operations. Thus, in attempting to improve performance, all three loops were fissioned, then they were unrolled and vectorised. Both (fissioned and

vectorised) and (original and vectorised) versions were tested for each loop. Loop 0 was fissioned into three loops as it contained three unrelated operations, attempting to improve locality. An attempt to integrate Loop 2 into Loop 1 slowed down the overall performance due to reduced spatial locality. Furthermore, loop-invariant code motion was applied. $dmp \times dt$ was moved outside the loops since it was loop invariant.

3.2 Testing

3.2.1 Loop 0

As shown in Table 2, fission and vectorisation is slower than the vectorised gold version code.

3.2.2 Loop 2

Though moving code outside the loop improved performance by 10%, it affected the answer precision. Subsequently, loop fission did not improve performance. As shown in Table 2, vectorising the original code being the most efficient approach.

	Original	Loop fission	Fission & vectorisation	Original & Vectorisation
Loop 0	0.002346s	0.001884s	0.001369s	0.001067s
Loop 2	0.002550s	0.002531s	0.001127s	0.000864s
Loop 3	0.003331s	0.003135s	0.001345s	0.001586s

Table 2: Loops 0, 2, 3 optimisation results for 1000 stars and 1000 steps.

3.2.3 Loop 3

Vectorisation and loop fission improved performance by 84.94% as shown in Table 2. In vectorising the *if* statements, firstly *_mm_cmplt_ps(array[i], 1.0f)* and *_mm_cmpgt_ps(array[i], -1.0f)* are computed as masks, where array stands for ax, ay, az, in order to check if they are outside the interval [-1.0f, 1.0f] . Bitwise AND is computed using *_mm_and_ps()* (which returns zero or not a number (NaN)) to check if any of the condition evaluates to 0 (false). If both conditions are false, or one is false, the initial if condition evaluates to true. Applying the min function *_mm_min_ps()* on the previous result and 2.0f yields either 0 (in the case of the if statement evaluating to true) or 2 (in the case of AND returning NaN). The final operation is subtracting 1.0f from the previous result, which gives -1.0f in the case of the if statement evaluating to true and 1 in the case of it being false. The result will then be multiplied by array[i] and stored in array[i]. Thus, false evaluation multiplies the array by 1, not modifying it.

4 Difficulties

Accuracy and consistency of results were the main issues that occured during optimising the code and collection of data, which complicated spotting small improvements. This behaviour is especially observed if the code is run on Bash on Windows, as well as some Linux kernels. To deal with this, the optimisations were made on a DCS machine running Atom and a terminal, and the data results presented in the report were obtained by calculating the average result of three runs. An additional issue was incorrect code that appeared efficient. That is, more GFLOP/s were obtained with a code that was incorrect, as assessed by running the visualisation version of the code and observing it's behaviour.

5 Conclusion

This report has sought to show the possible optimisations that can be applied to a given code in order to achieve the most efficient version of the program without highly affecting it's functionality and precision. The four main techniques used were loop fission, loop unrolling, vectorisation and multi-core threading. Chosing the techniques that were most time efficient lead to a peak of 82.1 GFLOP/s, a Total time of 24.2 seconds and an Answer of 1140634368 for 10000 stars and 1000 timesteps. The GFLOP/s obtained are 77.75% of the Peak GFLOP/s for the system, as computed in Equation 1 and the answer is 3.73% inaccurate. Similarly, on an i5-6500 @ 3.20 GHz, the running time is 21.55 seconds and GFLOP/s achieved are 92.7. This shows that performance is also highly dependent on the machine.

Bibliography

[1] I. Corporation. Intel intrinsics guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[2] I. Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[3] M. Leeke and G. Martin. Topic 6: Compiler optimisations and parallelism. http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs257/6_compiler_optimisations_and_parallelism_-_4up.pdf.

[4] T. Schwinge. Gcc wiki. https://gcc.gnu.org/wiki/openmp.