Intro to HPC Assignment 1: Serial Optimisation

Kheeran Naidu (kn16063)

***Abstract*—Optimisation is a key aspect to computational prowess. Before the necessary introduction of multi-core processing, computational speed-ups came from hardware improvements and serial optimisation. Put simply, hardware improvements came from an increase in the number of transistors on a chip. Serial optimisation came from many different techniques, which I will describe and analyse here.**

INTRODUCTION

THIS report is based around the stencil.c programme which is an unoptimised piece of code. Generally, the programme creates a black and white checkerboard image and a white temp image, and then iteratively compares and edits the images in order to average each pixel with it's surrounding pixels. The final image produced is a black and white checkerboard with the black to white edges blurred.

TABLE I
INITIAL TIMINGS

Image size	stencil runtime	Mem Bandwidth
1024*1024	8.506736s	0.592GFLOPS
4096*4096	342.353997s	0.235GFLOPS
8000*8000	595.251455s	0.516GFLOPS

In the serial optimisation of this code, the task was to optimise the stencil function. Profiling using gprof confirms that the stencil function takes up >99% of the run-time. In order to optimise, I used 3 main techniques; Memory accessing, division and vectorisation.

In order to do a performance analysis I compared the images sizes, and their associated run-time and memory bandwidth, after each optimisation technique. The performance before any optimisation is in TABLE I.

I used the following equation to calculate the memory bandwidth:

$$\frac{nx * ny * (4 * float\_ops) * (8 * dbl\_ops) * iter * func\_calls}{runtime} \tag{1}$$

(I've approximated the long latency operation, division, to be about 3[1] times slower than floating point operations)

I. MEMORY ACCESSING

In the FOR loop of the stencil function, the image arrays of both tmp_image and image were being accessed in a non-continguous manner. This is because C uses a row major order for 2D arrays and the array was being accessed by column major order, which was least optimal. This caused much more frequent cache trashing and didn't allow the full use of the memory in the L1, L2 and L3 caches.

A simple swap of the i^{th} and j^{th} terms allowed for the memory to be accessed contiguously and decreased the runtime by approximately 35% for the largest stencil and approx 20% for the smallest stencil (TABLE II). Naturally as the size of the stencil increases, the size of the for loop increases and so the affect of this optimisation increases.

TABLE II
OPTIMISED MEMORY ACCESSING (OMA)

Image size	stencil runtime	Mem Bandwidth
1024*1024	6.981878s	0.721GFLOPS
4096*4096	107.461833s	0.749GFLOPS
8000*8000	386.683197s	0.794GFLOPS

II. DIVISION

The division operation is a long latency operation and is approximately 3 times slower than a floating point operation. In the stencil function's FOR loop there were 5 division operations which calculated independent constants. This could therefore be changed to the floating points of the constants which it calculated. This improved the overall performance (TABLE III); runtime decreasing by approx 40% in all stencils, and memory bandwidth increasing by 50%. This shows that the division optimisation had a more significant affect on code than the memory accessing.

TABLE III
OMA + DIVISION REMOVAL (DR)

Image size	stencil runtime	Mem Bandwidth
1024*1024	4.388271s	1.147GFLOPS
4096*4096	71.050412s	1.133GFLOPS
8000*8000	242.059310s	1.269GFLOPS

III. VECTORISATION

Vectorisation is the process of using multiple vector lanes in order to execute thhe same instruction on multiple pieces of data simultaneously.

Loop vectorisation works very well on the FOR loop of stencil function as there are no data dependencies.

Using intel's compiler with the flag 'fast', the loop is vectorised. The process gives one of the pointers (either tmp_image or image) aligned access to the memory, however the other is still unalligned.

AVX offers 256bits of vector width. since double is 8 bytes (64 bits), we would be able to have 4 vector lanes. However, changing double to single precision float, which is half of double, allows us to have 8 vector lanes. This increases the optimisation of the vectorisation flag 'fast' and significantly increases the memory bandwidth to much closer to that of Intel's Sandy Bridge (83 GFLOPS [3])

TABLE IV
OMA + DR + VECTORISATION (V)

Image size	stencil runtime	Mem Bandwidth
1024*1024	0.381265s	26.402GFLOPS
4096*4096	6.262072s	25.720GFLOPS
8000*8000	23.967969s	25.634GFLOPS

In order to prevent pointer aliasing which also affects the optimisation of the code by the compiler, I made all the float pointers restricted. This means that the memory allocated to the image and tmp_image is fixed.

TABLE V
OMA + DR + V WITH FLOAT

Image size	stencil runtime	Mem Bandwidth
1024*1024	0.089678s	56.125GFLOPS
4096*4096	2.450277s	32.866GFLOPS
8000*8000	8.603345s	35.707GFLOPS

Overall the use of vectorisation increased the memory bandwidth approx 2300% and decreased the runtime by >90% in all stencil sizes. This has the most affect on optimisation compared to the other 2 techniques.

IV. COMPILER FLAGS

In order to find the optimum compiler flag and compiler, I used all GCC's optimisation flags and all of ICC's optimisation flags and compared all optimisations made with the timings for the different compilers.

GNU's C compiler, GCC, was very similar to Intel's C compiler, ICC, however ICC vectorisation was much better especialy since there were specific flags such as '-xAVX' which is specific to the type of processors produced by Intel.

All my timings shown before vectorisation was run using no optimisation flags and vectorisation was run using the '-fast' optimisation flag.

V. IMPROVEMENTS

Though not necessary, I could have removed the conditional statements which trigger conditional branching. This would have sped up the runtime becuase Intel's Sandy bridge uses deep pipelining to run it's operations and therefore when a conditional branch is met, it must discard all operations in the pipeline inorder to branch to to another part of the programme. Since we have 4 if statements per iteration of the FOR loop in stencil, this causes a significant slow down in the overall speed of the programme as it would be discarding the pipeline on every condition.

CONCLUSION

Overall the optimisations tackled significant portions and areas of the code, however vectorisation had the most significant decrease in runtime and increase in memory bandwidth and is therefore the best optimisation used.

REFERENCES

[1] Nicolas LIMARE - Integer and Floating-Point Arithmetic Speed vs Precision http://nicolas.limare.net/pro/notes/2014/12/12_arit_speed/

[2] Boston University Info Services - Intel Compiler Flags http://www.bu.edu/tech/support/research/software-and-programming/programming/compilers/intel-compiler-flags/

[3] Wikipedia - Sandy Bridge https://en.wikipedia.org/wiki/Sandy_Bridge