# COMS30005

# Serial Optimisation

Louis Heath

lh16421

October 24th, 2018

## 1 Introduction

This report aims to outline my investigation into serial optimisation. In particular it focuses on the improvement of *stencil*, a simple program written in C99 and executed on a single core of BlueCrystal Phase 3. It describes and attempts to explain the approaches taken, and then provides analysis on the resulting performance.

## 2 Algorithm analysis

The first step was to understand the code and determine its algorithmic complexity. The core function *stencil()* receives two square checkerboard images of height and width *n*, and blurs them in a similar fashion to that of a gaussian blur convolution kernel. The function has algorithmic complexity $O(n^2)$ as it loops through every row and column.

The checkerboard is divided into a pattern of 8x8 black and white squares. A potential improvement could be to only blur a 4x4 checkerboard, reducing computation by 75%. The central squares could then be duplicated at the end of iteration to reconstruct the larger 8x8 board. This method would begin to yield incorrect outputs at around *n/8* iterations, as the edge cells are blurred differently to those with four neighbours. These variations propagate towards the center of the image with each iteration, preventing reconstruction to 8x8 if reaching the inner four cells. Similarly, a 6x6 checkerboard could be considered.

## 3 Optimisation

### 3.1 Profiling

Before investing time into optimisation efforts I used a profiler to find the most performance critical regions of code. An initial run of *gprof* revealed that 96% of computation time was being spent on lines 55-59, the double *for*-loop blurring each cell in the image.

### 3.2 Compiler flags

Optimisation flags are inbuilt compiler features that attempt to cut corners and improve performance at execution time. Example tweaks include precomputing hard coded arithmetic, such as replacing *3.0/5.0* with *0.6*, and caching repeat calculations, for example when repeatedly accessing an array index such as *img[j+i]*.

*-O2* triggers a set of individual optimisation flags which aim to improve speed without compromising space. *-O3* calls *-O2*, in addition to more flags which may sacrifice memory performance for sake of speed. *-Ofast* includes a few math-specific optimisations. In *Figure 1* we see that *icc* flags *-O3* and *-Ofast* provide no further benefit over *-O2*, but all three provide a 95% improvement over no flag at all.

| Flag | 1024 | 4096 | 8000 |
|---|---|---|---|
| O0 | 2.2062 | 42.656 | 144.55 |
| O1 | 0.2976 | 4.9544 | 19.444 |
| O2 | 0.0967 | 2.3983 | 8.4033 |
| O3 | 0.0969 | 2.3947 | 8.3926 |
| Ofast | 0.0969 | 2.3953 | 8.4011 |

Figure 1: *Table showing the effect of icc compiler flags on final code*

### 3.3 Division operators

Though some operations such as multiplication and division are quick to compute, others such as division and modulus are more complex and costly.

In some cases we can replace a division with multiplication of its reciprocal. I was able to apply this to *stencil* and achieved a 3X speed-up, suggesting that the *-O3* flag for *gcc 4.7.0* was not precomputing the division.

I was also able to remove four out of five of the operations by scaling them all up by a power of ten, removing the need to multiply by *0.1*. This worked because the pixel values are normalised before output, requiring only the relative proportions to be preserved. I later reverted this change, however, as the now larger pixel values could not fit within a *float*.

### 3.4 Cache usage

Memory access is expensive, and the faster levels of cache are severely limited in size. For this reason we try to maximise the amount of useful data in the cache, and operate on it as much as possible before replacing it.

The array holding the checkerboard is one-dimensional and stored in row-major order, meaning row elements are contiguous in memory. By inverting the order of the loops around the critical section so that the inner loop is the one that covers columns, I significantly increased the number of cache hits. This is because each consecutive iteration operates on nearby data, improving the likelihood of it already being in the cache.

### 3.5 Vectorisation

Compilers are able to utilise *Single Instruction Multiple Data* to vectorise certain *for*-loops, provided they are written simply and include no data dependencies between iterations.

I declared my pointers to be non-aliasing using the *restrict* keyword, this alone dropped execution time by 30%, despite none of the loops actually vectorising. This is because the compiler no longer has to consider the race condition edge cases, reducing branching and excessive machine code.

To simplify the SIMD instruction I removed all branching from the double loop by covering each edge case separately. This, along with the inverting of the loop order, enabled *gcc* to vectorise the main loop and the two 'row' edge cases. The Intel compiler *icc* proved itself to be more sophisticated, as it was able to completely vectorise the loops with the branches still present.

Another advantage to removing branching within the loop is to enable 'pipelining'. As all loop iterations are operated on by exactly the same instruction, the program counter does not need to change address at all during the loop.

### 3.6 Data type

One of the largest performance enhancements I applied was changing my data types from *double* to *float*. This meant suffixing constants with an *f*, for example *0.6f*. Operations on *double* variables are more costly due to their additional size and precision.

This optimisation only took effect when compiled with *icc*, potentially because the compiler is more native to BlueCrystal's Intel SandyBridge processors.

## 4 Performance analysis

The critical region of my final code has 9 floating point operations and 6 memory movements, each of 32 bits, meaning the code has operational intensity 0.375.

Using *icc 16u2* and the *O3* optimisation flag, I recorded final execution times of 0.1, 2.39 and 8.39 seconds for the three respective image sizes. These correspond to speeds of 19.7, 12.6 and 13.7 GFLOP/s.

Plotting this on the roofline model in *Figure 2* we see that the code is memory bound, reaching almost 50% of the peak compute performance for *n=1024*. This is 100X faster than the original 0.2 GFLOP/s code.

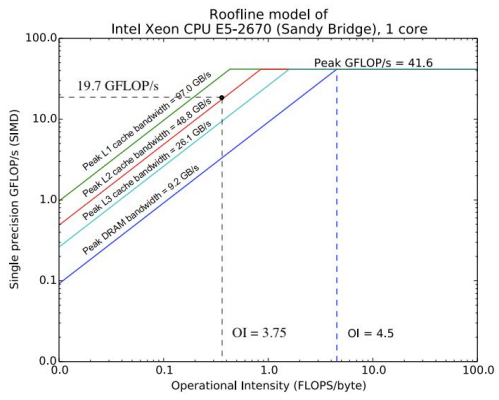


Figure 2: *Optimised code performance plotted on rooftop model*