Name: Jack Zhao

User ID: fz17963

This report presents the serial optimization of a tiny program named *stencil.c* in order to manage trade-offs among performance better in BlueCrystal, which is the supercomputer in the University of Bristol. As known, understanding of the most efficient program and utilizing high performance system is one of the challenging problems in scientific computing industries.

Loop Unrolling

In the first step, it is true that the speed of the running time is faster when you enable the optimization in the compiler but possibly not as fast as before after coding optimisation with the same compiler. To avoid the occurrence of any similar situations, I chose to unroll the loop firstly and kept the same commands in the compiler. To unroll the loop, I have to check the performance of the program so that I can identify which part occupies most of the running time, then I can target the specified part of the program by data collected.

For the stencil game, the original code established a height * width matrix, whereas it iterates through from 1 to the size of actual input columns at first, then goes through the number of rows. But the index of tmp_image works by multiplying the number of columns by the height to access each column and then increment the value of rows to access each element of the row. Because the arrays are flattened out in memory, to avoid the misuse of the indexing work and to decrease the cache references, I expect to get an iteration flow from top to the bottom rather than from the left to the right side in the matrix. To achieve this, I changed the order between column-major and row-major, to verify this is correct and useful, I recorded the running time at the moment which is 4.76s approximately for 1024*1024 input. The run time improved from this to about 3.81s and the check file works as well, which means my image still matched the input size.

Based on the consideration of running time, it's not worthwhile to test 8000*8000 image which still takes a long time to finish at this level of optimisation.

Compilation Flags & Vectorization

After thousands of times experiment, I found Intel compiler always perform a little better than the GCC ones with the default flag. Accordingly, I replace GCC compiler with Intel compiler to compile the program.

For Intel compiler, to make sure that it optimises most reasonably, I compile the program with different flags:

O1 enable optimisations for speed and disables some optimisations that increase code size and affect the speed, it works very well for programs with very large code size and many branches and execution time not dominated by code with loops, apparently it doesn't suit our case. Then I tried O2, which is the default one and included in O3, because the loop and the memory access transformations take place in our stencil method, the speed is highly improved with O3 flag.

With Ofast flag added the program is supposed to be faster than O3, but actually not, there is almost no difference between these two. The following table is an average value by sampling 10 times with different flags.

Icc -Ofast(1024*1024)	0.1510s
Icc -O3(1024*1024)	0.1514
Icc -fast(1024*1024)	0.1500

Table 1. Run time with basic compilation

There is not much difference between these 3 flags on a basis of data from Table 1. Because the content it optimised is almost the same with these 3 flags in our program. In parallel computing, computer program is converted from a scalar implementation to a vector implementation that simultaneously perform the operations. Vectorizing data entails changes in the order of operations within a loop. The intel compiler has helped us to vectorize the loop, which transforms procedural loops by assigning a processing unit to each pair of operands. At the same time, in the loop of stencil

method, I found that I can apply a new variable t to replace i and j, especially for the indexing the image, then I can merge five add operation into one line, which speeds up a little bit as shown on Table 2.

Before (1024*1024)	0.1510s
After (1024*1024)	0.1310s

Table 2. Run time when merge the operations

Since each single instruction with multiple data operates at once (SIMD), the change of the order of operation within a lop might change the result of calculation, there are some data dependencies existed in our loop. I used Pragmas (#pragma ivdep) to provide additional information to the compiler so that any potential data dependencies can be safely removed or ignored.

Data alignment

From the vectorization report, it's mentioned that the reference image has unaligned access. To align the data in the memory, in other words, to allocate memory in aligned manner, Intel compiler provides another set of memory allocation APIs, I can use _mm_malloc and _mm_free to allocate and free aligned blocks of memory, as declared in our program, our cache is 64 bytes based.

After reference image having aligned access, the running time is actually slower than before, which is shown on the following graph:

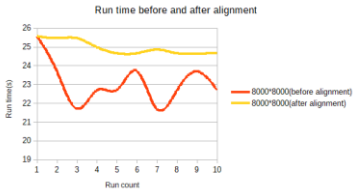

Chart 1. Run time with and without alignment

Performance Analysis

It seems that the run time is always above 20 seconds, to further improve the performance, and based on the consideration of the size of memory stored, although double has twice the precision of float, the precision of float is enough to satisfy our data in the program as required. After converting the data type of our image, the run time is reduced to around 10 to 11s under 8000 * 8000 size of image, the details data is shown as follow:

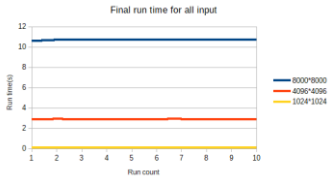

Chart 2. Final run time with different size

Size of image	Average Run time(s)
1024*1024	10.70
4096*4096	2.86
8000*8000	0.09

Table 3. Final run time) with different size.

Cause I only use one node in bcp4, and which only use one core running the program. To check if I have achieved a good fraction of the theoretical value, after calculations, which is 19.14GB/s for 8000*8000 image, the graph of the performance is shown as below:

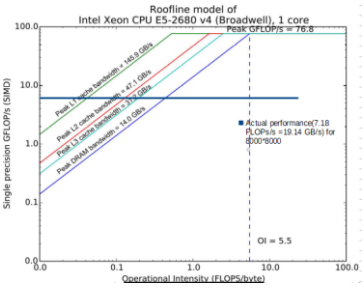

Chart 3. Roofline model and performance

Finally, the comparison of the speed between the optimised one and not is shown as below:

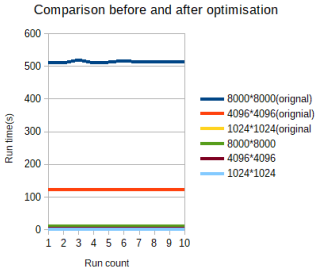

Chart 4. Final comparison.