AN INTRODUCTION TO HIGH PERFORMANCE COMPUTING
ASSIGNMENT 1: SERIAL OPTIMISATION OF A 5-POINT STENCIL CODE

ANGELOS CHARITIDIS
1864704 – vi18347

This report analyzes the optimizations implemented on the stencil code. In order to evaluate the effectiveness of an optimization, there needs to be a comparison between the running time of the code before and after that optimization. All run-times stated on this report are in seconds and were recorded as the code run on 1 core of the Blue Crystal Phase 3 supercomputer.

The running time of the code before any optimization is shown in the following table:

Grid Size	Run-Time
1024x1024	8.248
4096x4096	357.600
8000x8000	639.696

1st OPTIMIZATION: COMPILER FLAGS

The first optimization implemented on the stencil code was the compiler flags. The flags **-O3** and **-march=native** were used and added to the Makefile. The **-O3** flag (Optimization Level 3) is the 3rd level of Optimization compiler flag and turns on various Optimization flags in order to decrease the code size and its run-time *[1]*. The default Optimization flag in GCC (which was used before using **-O3)** is **-O0** (Optimization Level 0) and turns off all optimizations. The **-march=native** flag makes the compiler detect what processor it is running on and optimize for the specific processor in particular *[2]*. It is very useful since different processors require different method and depth of optimization. The table below contains the run-time before and after adding the two flags and the speed-up.

Grid Size	Previous Run-Time	Run-Time	SpeedUp
1024x1024	8.248	6.801	**1.21x**
4096x4096	357.600	120.666	**2.96x**
8000x8000	639.696	370.544	**1.73x**

2nd OPTIMIZATION: COMPILER CHOICE

After adding the compilation flags, I tried different compilers. The compiler used before was GNU's **GCC 4.8.4**. First, I upgraded the compiler to the latest version of GCC installed on Blue Crystal Phase 3, which is **GCC 7.1.0** *[3]*. Then, I tried the latest version of Intel's ICC compiler, which is **ICC 16.0.2** *[4]*. In order to change from GCC to ICC, I had to change the ***-march=native*** flag to the equivalent of ICC, which is ***-xHOST***. As can be seen in the table below, there wasn't a significant speed-up by changing the compiler from **GCC 4.8.4** to **GCC 7.1.0** and from **GCC 4.8.4** to **ICC 16.0.2**. However, after implementing various other optimizations described later, the fully optimized code run at half the run-time (SpeedUp ≈ **2x**) when compiled by **ICC 16.0.2** compared to **GCC 4.8.4**. As a result, I decided to choose **ICC 16.0.2** and thus, the run-times for the rest of the report were recorded when the code was compiled by **ICC 16.0.2**.

Grid Size	Previous Run-Time (GCC 4.8.4)	Run-Time (GCC 7.1.0)	Run-Time (ICC 16.0.2)	SpeedUp (GCC 7.1.0)	SpeedUp (ICC 16.0.2)
1024x1024	6.801	6.800	6.806	**1.00**	**1.00x**
4096x4096	120.666	122.542	110.411	**0.98**	**1.09x**
8000x8000	370.544	361.339	356.073	**1.03**	**1.04x**

3rd OPTIMIZATION: DIVISIONS REMOVAL

As said during the "Performance Analysis" lecture, floating points and integer operations are generally very cheap. However, divisions are an exception to the rule as they are not. Examining the stencil code, I realized that there are three to five divisions (depending on the elements position) which are repeated for every iteration in the double for loop. As a result, I decided to hard-code their result to speed up the code. So, I replaced **3.0/5.0** by **0.6** and **0.5/5.0** by **0.1**, which resulted in the speed-up displayed below.

Grid Size	Previous Run-Time	Run-Time	SpeedUp
1024x1024	8.248	2.174	**3.79x**
4096x4096	357.600	93.690	**3.82x**
8000x8000	639.696	106.479	**6.01x**

4th OPTIMIZATION: STORING/LOADING OPTIMIZATION

On the 15th iteration of the double for loop (for j=0 and i=14), the CPU fetches the elements image[13312], image[15360] and image[14337]. On the 16th iteration of the double for loop (for j=0 and i=15), the CPU fetches the elements image[14336], image[16384] and image[15361]. Depending on various factors, such as bus size and bandwidth, the CPU will fetch 64 bytes or more. Considering that each variable is a double (so 8 bytes of memory), the CPU will fetch many elements of the image array which are stored next to the element we want to fetch. So, when fetching image[13312], the CPU will also fetch image[13313], image[13314] etc. So, I decided to take advantage of that and "synchronize" storing and

loading of the elements and since storing was made in row-major order, I decided to also do loading in row-major order (from column-major order). I did this by inverting the for loops. Now, on the 15th iteration of the for loop (for j=14 and i=0), the CPU fetches the elements image[1038], image[13] and image[15]. On the 16th iteration of the for loop (for j=15 and i=0), the CPU fetches the elements image[1039], image[14] and image[16]. It is obvious, that loading now becomes a lot more efficient as every time the CPU fetches an element, the elements which will be fetched with it will be used in the next iterations. The SpeedUp is representative of that efficiency.

Grid Size	Previous Run-Time	Run-Time	SpeedUp
1024x1024	2.174	0.181	**12.01x**
4096x4096	93.690	4.590	**20.41x**
8000x8000	106.479	17.019	**6.26x**

5th OPTIMIZATION: DATA TYPES

Examining the Data Types as instructed by the Assignment Description, I realized that the *image* and *tmp_image* pointers were pointing to a double precision variable. So, I tried changing the double (precision) variables to float (single precision) variables. The reason I did that is that doubles require 8 bytes of memory, whereas floats only require 4. As a result, storing and loading those variables now becomes twice as fast(about **2x** SpeedUp), as the bus can now transfer double the elements from the cache to the RAM (while storing) and from the RAM to the cache (while loading).

Grid Size	Previous Run-Time	Run-Time	SpeedUp
1024x1024	0.181	0.088	**2.07x**
4096x4096	4.590	2.443	**1.89x**
8000x8000	17.019	8.471	**1.99x**

CONCLUSIONS & SUMMARY TABLE

There are many and very useful conclusions to be made from the data presented in the tables above. First of all, intelligent compilers such as Intel's ICC make very important optimizations while compiling the code. Although if statements are considered "expensive" operations, removing the if statements from the stencil code resulted in my run-time increasing (SpeedUp less than 1x). This is because the ICC compiler manages to remove the if statements and does so more efficiently than me. Another very important optimization made by the ICC compiler is **vectorization** or "the process of converting an algorithm from a scalar implementation, which does an operation one pair of operands at a time, to a vector process where a single instruction can refer to a vector" *[5]*. The compiler peels the double for loop and realizes that these operations can be done simultaneously using the unused space in the SIMD (same instruction multiple data) registers. Although the compiler does the vectorization on its own, we still need to help the compiler "see" that those instructions can be operated simultaneously. The 4th Optimization, where we load the elements in row-major since they were stored in this way, does exactly that and loads the elements that will be used on those multiple data in the same operation. So, hypotheses are important, but testing is even more important.

Other than this, it is also important to try different **orders of optimizations**. In a previous order of optimizations, I first changed the Data Types (5th Optimization) and I then performed the Storing/Loading Optimizations (4th Optimization). In that order, the run-time results from changing the Data Types (without Storing/Loading Optimizations) showed very small speedup or even slower run time. This makes sense. Changing the Data Types, allows the bus to transfer 16 instead of 8 elements (considering 64 bytes each transfer). However, without synchronizing Storing and Loading so that the bus brings elements used in sequent iterations, changing the Data Types is useless. If the bus transfers only one "useful" element per transfer, the run-time is the same regardless of the number of elements it transfers.

The following table compares the run-time of the code without any optimizations and compiled by **GCC 4.8.4** to the run-time of the code with the optimizations described earlier (Compiler Flags, Divisions Removal, Storing/Loading Optimization, Data Types) and compiled by **ICC 16.0.2**.

Grid Size	Initial Run-Time	Final Run-Time	Total SpeedUp
1024x1024	8.248	0.088	**93.73x**
4096x4096	357.600	2.443	**146.38x**
8000x8000	639.696	8.471	**75.52x**

REFERENCES

[1] 3.10 Options That Control Optimization, https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

[2] Compilers, https://github.com/UoB-HPC/hpc-course-getting-started/blob/master/4_Compilers.md

[3] GCC 7 Release Series - Changes, New Features, and Fixes https://gcc.gnu.org/gcc-7/changes.html

[4] Intel C++ Compiler 16.0 Update 4 for Linux* Release Notes for Intel Parallel Studio XE 2016 https://software.intel.com/en-us/articles/intel-cpp-compiler-release-notes#2016u2

[5] A Guide to Vectorization https://software.intel.com/sites/default/files/8c/a9/CompilerAutovectorizationGuide.pdf