Serial Code Optimisation: 5 Point Stencil by Samia Mohinta (User_id: dc18393, Candidate No. : 97293)

The 5-point stencil code applies a weighted 5-point stencil on a rectangular checkerboard grid. The value in each cell of the grid is updated based on the average values of its neighbouring North, South, East and West cells. This program is verified against three different input matrix sizes, 1024X1024, 4096X4096 and 8000X8000 respectively. The code also iterates 100 times for each input to calculate the final weighted average of the image matrix.

Choose Compiler

A step by step approach to optimize the code is undertaken, with an initial focus on the choice of compiler. The original source code without any optimisation flags is compiled with GCC (gcc-4.8.4) and ICC (intel-compiler-16-u2) compilers.

The results after running the code on GCC and Intel Compiler without optimisation flags (-O0 is the default flag, even when not specified) are as follows:

GCC (gcc-4.8.4)				ICC (intel-compiler-16-u2)			
Version	Runtime 1024*1024 (in seconds)	Runtime 4096*4096 (in seconds)	Runtime 8000*8000 (in seconds)	Version	Runtime 1024*1024 (in seconds)	Runtime 4096*4096 (in seconds)	Runtime 8000*8000 (in seconds)
Source Code	9.757482	354.529419	607.643823	Source Code	9.449316	49.437813	175.381485

Table 1: Comparison between GCC and ICC compiled codes

It is observed from *Table 1* that code compiled with the Intel compiler runs faster than the one with GCC. However, for a matrix size of 1024 X 1024, there is not much noticeable difference in the run-times. The Intel compiled code is about 7x faster than the GCC on the 4096 X 4096 matrix size and nearly 3x faster on the 8000 X 8000 matrix. As a justification for the above results, it can be assumed that the Intel compiler works better because it takes advantage of the underlying Intel architecture of Blue Crystal (Intel Sandy Bridge). Moreover, it can also be said that ICC works better on codes with loops and floating point calculations than GCC, when the input size increases. As a result, Intel compiler (intel-compiler-16-u2) is used hereafter for further optimisations.

Using Compiler's Optimisation Flags

In the next step, Intel Compiler's Optimisation flags were introduced (from degree 1 to degree 3) during compilation. The results are as follows:

Optimisation Level	Runtime 1024*1024 (in seconds)	Runtime 4096*4096 (in seconds)	Runtime 8000*8000 (in seconds)	Optimisation Flags (Intel Compiler)
1	6.795352	119.903797	358.201789	-O1
2	3.398959	49.285918	175.828319	-O2
3	6.814238	110.000777	356.293968	-O3

Table 2: Runtimes with Optimisation flags in Intel Compiler

It is evident from *Table 2* that O2 is the recommended level of optimisation. This is because compiler vectorisation was enabled at O2. With this option, the compiler performed some basic loop optimisations, intrinsic in-lining, and other common compiler optimisations, which led to improved performance (Reference – [0]). The O1 flag enabled speed optimisation but disabled compiler's auto-vectorisation functionality and hence, it did not perform well in this scenario. On the other hand, O3, the highest optimisation level flag, enabled O2 optimisations along with aggressive loop transformations. It is believed that the forced loop and memory access transformations with O3 were inefficient on this code and hence slowed the code down.

Finding Bottleneck and Code Modifications

In order to identify the bottleneck in the code, GPROF profiler was used (code compiled with GCC). The profiler suggested that about 99% of the total time was being spent inside the void stencil() function. Furthermore, PERF was used to obtain the percentage of cache misses, which was nearly 33% for 1024X1024 input. Therefore, to further improve the performance, the following code modifications were undertaken.

- **Divisions Replaced with Multiplications**: Floating point divisions were replaced with floating point multiplications (e.g. A*3.0/5.0 to A*0.6), because multiplications are cheaper than divisions. Also, the compiler was hinted to perform single precision multiplications instead of double precision (e.g. A*0.6f). With this, the runtimes decreased to **~0.36s (1024 X1024), ~8.07s (4096 X 4096) and ~30.36s (8000 X 8000)** from ~3.3s, ~49.28s and ~175.82s respectively. [~ stands for approximately]

- **Pointer Data-type Change:** Pointer data-types were changed next, from Double to Float to improve memory utilization, since double takes twice the storage space of float (double = 8 bytes and float = 4 bytes). After assessing the code using the Roofline model, it was understood that the given Stencil Code is Memory Bound (more governed by load and store operations). Hence, it was crucial to choose the correct data-type to make the optimum use of the cache memory. This change also significantly dropped the runtimes, which were now **~0.16 s (1024X1024), ~3.72 s (4096X4096) and ~14.43 s (8000X8000).**
- **Restrict Keyword Used**: Keyword 'restrict' was included while declaring pointer variables (image & tmp_image) in function definitions. This declaration hint prevented pointer aliasing, i.e, it informed the compiler that the memory location should only be accessed using the pointer defined and the compiler need not do additional checks. However, this just marginally improved the runtimes, by about 0.01 s, possibly because the compiler was already applying anti-aliasing optimisations as part of the –O2 flag.

The above changes cumulatively reduced the runtimes **from ~3.3 s to ~0.16 s (*1024X1024 100 iterations*), ~49 s to ~3.70 s (*4096X4096 100 iterations*) and ~175s to ~14.42s (*8000X8000 100 iterations*)** respectively, when compiled with -O2 flag. The percentage of cache misses also dropped from ~33% to ~6% for 1024X1024 input. Subsequently, this modified code was compiled with **–xHOST** and –O2 flags. This flag, xHOST, enabled processor-specific optimisations (targeting highest level instruction sets), resulting in runtimes **of ~0.14 s (*1024X1024 100 iterations*), ~3.60 s (*4096X4096 100 iterations*) and ~14.41 s (*8000X8000 100 iterations*)** respectively**.**

Vectorization and Data Access and Alignment

As per the vectorisation reports generated using –qopt-report and –qopt-report-phase, this code still needed improvement, since it was not exploiting optimum vectorisation levels due to presence of costly 'IF' statements inside the nested for loops in function void stencil(). Also, the memory was being accessed in a column-major order, while C is a row-major order language. In order to realize these observations, the code was modified to replace 'IF' statements with 'FOR' loops to calculate the weighted average of each cell (algorithmic complexity remained O (N^2)). The memory access pattern was also altered to access cells row-wise in the loops. Furthermore, instead of multiplying each surrounding cell value with the weight, all surrounding image cell values were added up first and then that value was multiplied with the weight and then assigned to the current cell. This was done because additions are computationally less expensive than multiplications. This reduced the number of load-store operations as well. These changes reflected in the runtimes of the code, **dropping it to ~0.08 s (*1024X1024 100 iterations*), ~2.47 s (*4096X4096 100 iterations*) and ~8.51 s (*8000X8000 100 iterations*) for individual inputs**. Despite the changes, not all loops could be vectorised due to the compiler's assumption of a vector dependence inside the loops. Dynamic memory alignment options like _mm_malloc (a, 64) and __assume_aligned(a, 64) were explored as well. However, these did not improve performance, rather deteriorated it. It is reasoned that data alignment with stencils does not work efficiently because there is data dependency on the surrounding cells for a single cell operation, so contiguous memory locations are generally not accessed. For example, in a 3X3 matrix if A[4] is the current cell, its weighted average value is dependent on the values of A[1], A[7], A[3] and A[5] cells.

On the whole, even though there is still room for performance enhancement by solving the above mentioned vectorisation and aliasing bottlenecks, it has improved significantly (evident decrease in runtime). Moreover, the final cache miss percentage was only 0.96 for 1024X1024 input. A comparison of runtimes achieved sequentially starting with the source code (unoptimised) to vectorisation (optimised) is presented in *Table 3* for the three different inputs.

Optimisations Applied	Runtime for 1024 X1024	Runtime for 4096 X4096	Runtime for 8000X 8000
Source Code with Intel compiler	**9.449316 seconds (Unoptimised)**	**49.437813 seconds (Unoptimised)**	**175.381485 seconds (Unoptimised)**
Intel's –O2 Flag	3.398959 seconds	49.285918 seconds	175.828319 seconds
Code Change – Divisions to Multiplications, Double to Float, Restrict	0.16 seconds	3.713436 seconds	14.424658 seconds
-xHOST optimisation	0.14 seconds	3.60 seconds	14.41 seconds
Vectorisation : Replacing IF with For Loops, From Column Major Memory Access to Row Major, Reducing Number of Multiplications and Load Store Operations	**0.08 seconds (Optimised Run-time)**	**2.47 seconds (Optimised Run-time)**	**8.517159 seconds (Optimised Run-time)**

Table 3: Comparison of Run-times from Source Code till Vectorisation

References:

[0]: https://software.intel.com/en-us/articles/step-by-step-optimizing-with-intel-c-compiler

[1]: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html