**Abstract**

The stencil is a single instruction multiple data function, a few ideas on optimizing stencil is vectorization include using compiler flags and tiling. In order to analyse the stencil.c code, I have decided to use Intel Advisor (shortened to IA below) which is a specialist tool in SIMD vectorization optimization for C and a bit of trial and improvement. IA is ran on my laptop which has an Intel CORE i5 CPU. Running stencil.c with flags "gcc -g -std=c99 -Wall stencil.c -o stencil" took 5.907822 seconds to run on the Blue Crystal Phase 4 (shortened to BCP4). Using IA on my laptop and running the original stencil.c, there are some very interesting results and suggested improvements detailed below.

## 1 Optimization

### 1.1 Flags With GCC

First of all, there was no vectorization therefore adding optimization flags "-O2/-O3" and "-ftree-vectorize" to enable auto-vectorization will improve the performance. After testing different optimization flags, I originally decided to use "-Ofast -mtune=native" which resulted in the fastest running times (See table below for results on BCP4). The "-mtune=native" flag informs the compiler to generate optimised code that runs best on the native machine i.e. BCP4. "-Ofast" flag enables all "-O3" optimizations as well as other floating point arithmetic optimising flags such as "-ffast-math".

| Command | Run Times |
|---|---|
| gcc -O1 | 2.007511 |
| gcc -Ofast | 1.18876 |
| gcc -Ofast | 1.19336 |
| gcc -Ofast -mtune=native -ftree-vectorize | 1.185200 |
| gcc -O3 -mavx | 2.001167 |

Under 'Per Program Recommendations', IA mentioned that a higher instruction set architecture is available, currently stencil is compiled using the SSE2 instruction set which is a lower ISA compared highest available ISA on both my laptop and BCP4 which is AVX2. Using the flag "-mavx2", the application now runs at 1.179462 seconds (all timings quoted below will be an average from 5 runs) which is 80.0% faster. This is a result of targeting the higher architecture of the BCP4 CPU.

### 1.2 Vectoriazation

After re-analysing the code using "-Ofast -mtune=native -mavx2" flags, I have discovered that the "-Ofast" flags makes the "-mavx2" flag redundant. Therefore I have opted for the "-O3" flag and optimize the floating-point arithmetic myself through rewriting stencil.c. In addition, I also added restrict before image and tmp_image in the stencil function to reduce memory handling disambiguation inside the for-loops to tell the compiler that pointers image and tmp_img are not aliased with anything else, telling the compiler to do auto-vectorization in these loops. The code is mainly memory bound but may also be compute bound. I can first see that the division in the stencil function is an unoptimised floating point operation and I will therefore eliminate so there is only multiplication which requires less CPU cycles (https://gmplib.org/~tege/x86-timing.pdf shows that despite the fluctuations, MUL is faster than DIV across different CPUs). The runtime on BCP4 is now 0.977067s (83.4% speed up ). From the roofline analysis, the stencil function can improve in terms of efficiency through improving caching efficiency.

### 1.3 ICC

After looking at optimization during compilation using gcc, I also looked at how well the Intel C++ compiler can optimise my code. I discovered that using icc with the same optimization flags i.e. "-O3 -mavx2" the runtime is now down to 0.213633s (96%). IA also shows me a more detailed report on the vectorised looks in line 62 where the nested for-loops locate in the stencil function. The vectorization efficiency shown is >=100%. However, from the roofline analysis, I can still see that the code is both memory bound and compute bound. The roofline conclusion shows a few key points: 1. Improve caching efficiency 2. Potential underutilization of FMA instructions which can be improved by the following changes - Target the higher ISA - Explore further compiler options (which may vary depending on the CPU micro-architecture) I have decided to target the highest instructions set as possible which is AVX2 which resulted in a slightly faster runtime compared to AVX. AVX introduces fused multiply-accumulate operations (FMA mentioned in the suggested improvement) which is beneficial in this case as the weight is multiplied to each cell and sum the results. The precision is increased as well. AVX2 then introduces three operand general-purpose bit manipulation and multiply, which might improve the operations described by the stencil function. To do this I used the "-xCORE-AVX2" flag on icc. 3. Ineffective peeled/remainder loop(s) present - Disable dynamic alignment

### 1.4 Cache Optimization

The first cache improvement I have explored is to use float instead of double as datatypes for image and tmp_image. Theoretically, the cache should now be able to store more information and hence improve the hit rate. Having done this, the runtime of the stencil function is now down to 0.094846s (98.3% speed up). Another method of optimization is unrolling the loops, instead of having one calculation done per loop, it now does two instead of one. This proved to have no effect on the runtime and after research, I realised that the compiler automatically unrolls the loop during compilation, from IA the compiler unrolls the loops by 2. As mentioned in the instruction, I have also discovered that tiling does not speed up the runtime. I tried both using tiles of 2x2 and tiles of 64x64, but in both cases, tiling slows the function down. In theory, tiling maximises cache hits hence improving cache efficiency. An explanation for the slow down is that the compiler already has a more efficient tiling strategy and by adding an extra two loops, it affects the compiler's optimising abilities.

The times recorded below shows the runtime differences after compilation using gcc and icc. stencil compiled using icc performs well at smaller sizes however, at 8000x8000 is slower compared to compilation by gcc.

| Command | 1024 | 4096 | 8000 |
|---|---|---|---|
| gcc -O3 -mavx2 | 0.117368 | 2.982543 | 11.005303 |
| icc -O3 -xCORE-AVX2 | 0.101539 | 2.936012 | 11.152176 |

## 2 Conclusion

| Code | 1024 | 4096 | 8000 |
|---|---|---|---|
| Optimized | 0.094846 | 2.936012 | 11.152176 |
| Unoptimized | 5.908108 | 130.144126 | 561.314572 |
| % Speed Up | 98.3 | 97.7 | 98.0 |

Stencil compiled using gcc performs better on my laptop with a performance of 8 GFLOPS on average whereas compiled with icc performs slower at an average of 3 GFLOPS. A note must be made that the version of icc on BCP4 is 18.0.3 whereas on my laptop the version is 19.0.5. Many new SIMD features are added to 19.0, for example, the "-qopenmp-simd" flaf is set by default. icc compiled source code performs better on BCP4 and can be explained by the fact that the instructions are specified for intel CPUs (BCP4 uses intel E5-2680 v4 Broadwell CPUs). In conclusion, most of the serial code optimization is done automatically by the compiler.