Matthew Nagy by19729

Introduction

This is a report on optimizing a d2q9-bgk lattice boltzmann scheme. There are two major stages to these optimizations. Serial optimizations, and parallel optimizations. The serial optimizations can be categorised into non SIMD and SIMD changes. These reflect if they use Single Instruction, Multiple Data (SIMD) CPU instructions, also called vectorization. The SIMD serial and parallel optimizations were performed with the help of the OpenMP framework[1].

It is assumed the reader has some knowledge of the starting code, the C programming language, SIMD and parallelization already.

Serial Non-SIMD Optimizations

The first series of changes focused on serial optimization. These changes used neither SIMD instructions, nor multiple threads.

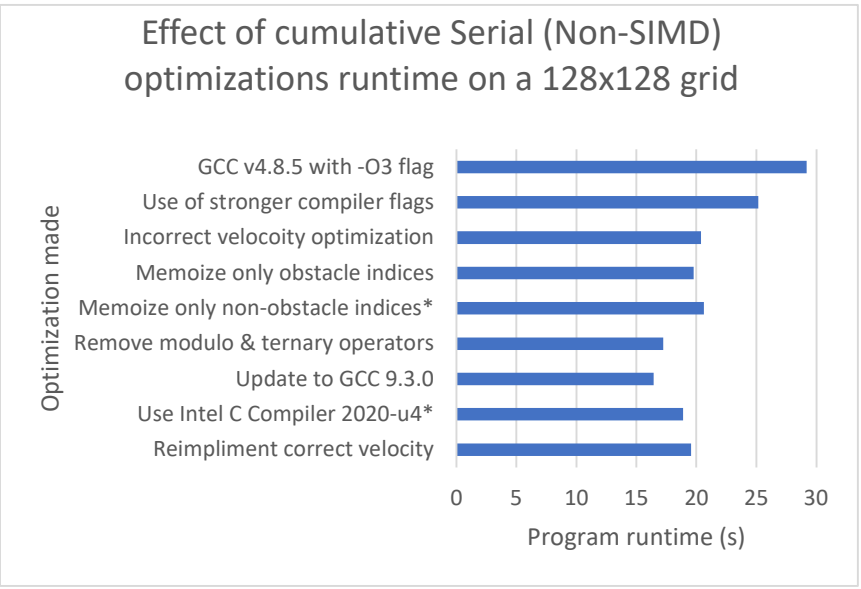


Figure.1: A graph to show the cumulative effect non SIMD optimizations made to the program's runtime. The time was collected on a 128x128 grid. A '' means this change was reverted.*

The first change was to give optimization flags to the compiler. This resulted in a 15.39% speedup. The specific flags used were `-ofast` and `-march=native`. The first flag tells the compiler to apply the most aggressive optimizations. This may include breaking the C standard to optimize floating point maths. The second flag tells the compiler to assume the code will only run on the CPU it is being compiled on. This lets it take information such as cache size and special CPU-specific instructions into account when optimizing the program.

After this I tried to only calculate velocity once. This resulted in me recording the average velocity of the system before the relaxation state. The error this caused was below the threshold necessary to be reported by the check program. However it was now incorrect. It was rectified later, but is needed here to put some future timings into perspective. The runtime of the program was 20.40 seconds after this change.

In the `collide` and `rebound` functions an if-statement is used in a for-loop. This checked if the cell was an obstacle or not. As this is value is constant, I would store the indices that passed this conditional in a separate list. This prevented looping through all cells and running a comparison. Furthermore, rather than summing velocities and number of obstacles in `collide`, only velocities were summed. As the number of obstacles are constant, this is pre-calculated. As obstacles grow in order of n, while non obstacles grow in order on n^2, this sped up `rebound` by 3.19%, but decreased performance for `collide`. Therefore, this optimization was only applied to `rebound`.

Updating the GCC compiler suite from version 4.8.5 to 9.3.0 gave a 4.81% speedup. This comes from improvements in the compilers ability to optimize C code[2][3].

Figure 5 in the conclusion shows the final runtimes on different grid sizes once the velocity calculations were corrected.

Adjustments Needed to Vectorize

The code could not be vectorized as is. The number of obstacles is not known at compile time. Therefore, when vectorizing, runtime checks would be used in `rebound` to ensure the vector would not index past the end of the list. This lost more performance than gained by memoizing, so it was removed.

The `propagate`, `rebound` and `collide` functions were merged into one function containing a nested loop. Temporary calculations were done using a new 9-length array. At the end of a loop the result is written into `tmp_cells`. At the end of the function, a pointer swap is performed between `tmp_cells` and `cells` so that `cells` now holds the new values. The OpenMP pragma `omp simd reduction(+:tot_u)` was used to tell the compiler to vectorize this loop as a reduction on the velocity accumulator `tot_u`.

Merging the loops while serial caused the program to run almost twice as slow. This heavily implies it is computationally bound, thus the additional reads to memory did not hamper runtime.

The format of cells was changed from an array of structs to array of arrays (`float**`). They were passed into functions with both pointers `restrict` and `const`. This helped the compiler vectorize the loop. The arrays were initialized with c11's `aligned_alloc` functions to ensure they are on 64-byte boundaries. This reduces the number of runtime checks needed to vectorize the code.

Finally, the Intel C Compiler[4] v.2020-u4 was use. The compiler flags were `-std=c11 -Wall -fast -xHOST -qopenmp qopenmp-link=static`. These flags are analogous to the GCC flags used and statically link the OpenMP API.

After these changes, the compiled program used SIMD instructions to increase performance.

Serial SIMD Optimizations

All runtimes here are on a 256x256 grid of cells. The runtime prior to addition vector based optimizations was 59.67 seconds. The compiler report said vectorizing would give a 1.61 fold predicted performance increase (ppi).

Having the loops unrolled to calculate edge cells separately was a good optimization for serial code. However this prevented certain cells from being calculated in a SIMD manner. Furthermore, all grids are a multiple of 8 in size. As the compiler report states a vector length of 8 is chosen, merging the loops would mean every cell could be vectorized with no remainder.

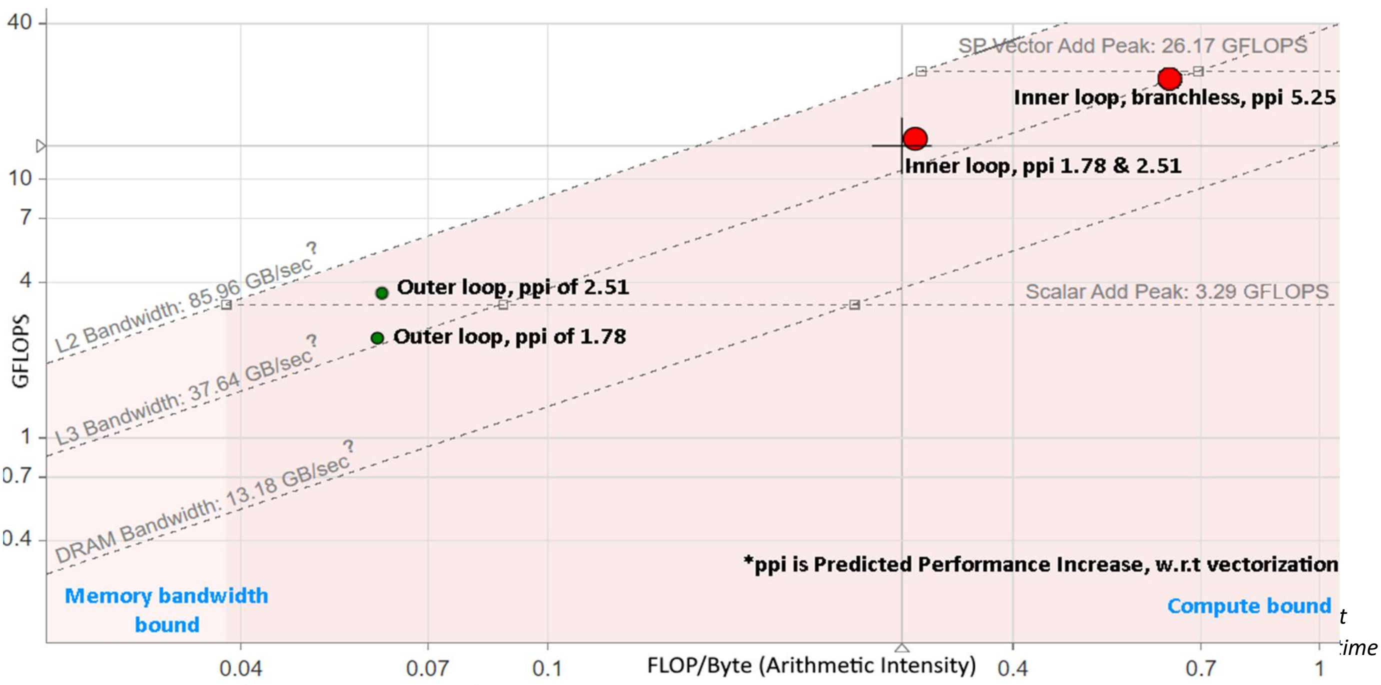
The loops were re-merged, using a bit-mask and bitwise AND operation for bounds checking. The Godbolt Compiler Explorer[5] shows that this saves 5 instructions per cell.

All grid sizes were a power of 2, therefore a bitmask of grid-size - 1 allowed efficient bound checking. A value too large would be wrapped around from zero. A negative value of -1 in two's compliment would be wrapped around to grid-size - 1. This performs the same operation as the modulo and ternary operator in the original code but with less operations. This gave a 6% performance increase and 5.85% ppi increase.

The compiler report stated that the all array indexing in the merged loop had unaligned access. OpenMP provides pragmas to tell the compiler about pointer alignment. Using these gave a small performance and ppi increase. However the compiler still had many array accesses as unaligned. This is because OpenMP can assume `cell`'s alignment but not `cell[n]`, with n between 0 and 8.

Intel provides a pragma, `vector aligned`, that assumes all pointers in a loop are aligned on optimal byte boundaries. `__attribute__((aligned(64)))` was used on local arrays in the loop to satisfy this. Despite a 46.8% increase in ppi, the observed performance increase could have been noise. Figure 2 shows a roofline analysis on these two scenarios gathered by the Intel Advisor[6] tool.

Figure 2 shows pointer alignment provided a 54.9% speedup in the outer loop, and a negligible change to the inner loop. The outer loop takes up only 0.20% of the program runtime. The inner loop takes 89% of the runtime, and was unaffected. The compiler report can take only computation time into account. This suggests that the inner loop is memory bandwidth



bound.

Godbolt did not display any change in the code of the loop when this optimization was applied. Compilers sometimes generate multiple versions of the same piece of code and branch to them based on some prior condition. The lack of change suggests that this optimization only changes the metric which determines which branch to choose. However, there is no conclusive proof to these theories.

The greatest performance increase came from removing branching. This increased ppi from vectorizing by 109%. This is because every cell now undergoes the same calculations, therefore the entire loop can utilize SIMD commands. Rather than a conditional branch, both `collide` and `rebound` values are calculated. A cell is an obstacle if the array `obstacle` has the value 1 in it. Therefore, you can assign `tmp_cells` as the `collide` and `rebound` values multiplied by `1-obstacle` and `obstacle` respectively. This lets the compiler use FMA vector commands rather than branching, and gave a 29% improvement in performance. This is also shown in Figure 2, showing the incredible increase in operations that can be carried out per byte.

With this, it almost reached the limit for single precision vector addition, and is compute bound on the L1 cache. As there are some FMA instructions too, there is still some room for improvement. A compiler report stated that a further 2.62% speedup may be possible. However the effort for such a small change may not be worth it

Parallel Optimizations

The SIMD section was a reduction over the total velocity. This would cause race condition in parallel code. Therefore the SIMD section became a reduction under a temporary variable. This temporary variable would be added to the total velocity. Therefore the outer loop now became a reduction over the total velocity. The pragma used was `omp parallel for reduction(+:tot_u)`.

As shown by Figure.3, the code scales sub-linearly. As the parallelizable areas of the code execute faster, the serial areas become a more significant proportion of the runtime.

Larger grids spend more time updating cells. This is a parallelizable action. Therefore on larger grids a greater proportion of the runtime may be parallelized. This explains why Figure 3 showed an increase in scalability as the grid sizes grow larger.

A roofline analysis on a 128x128 grid with 28 threads showed the serial `accelerate_flow` took as long to run as the parallelized merged loop.

Parallelizing the `accelerate_flow` function did not improve performance. As each loaded byte has only 1 operation performed on it, this function would be extremely memory bandwidth bound. Thus CPU optimizations were ineffective.

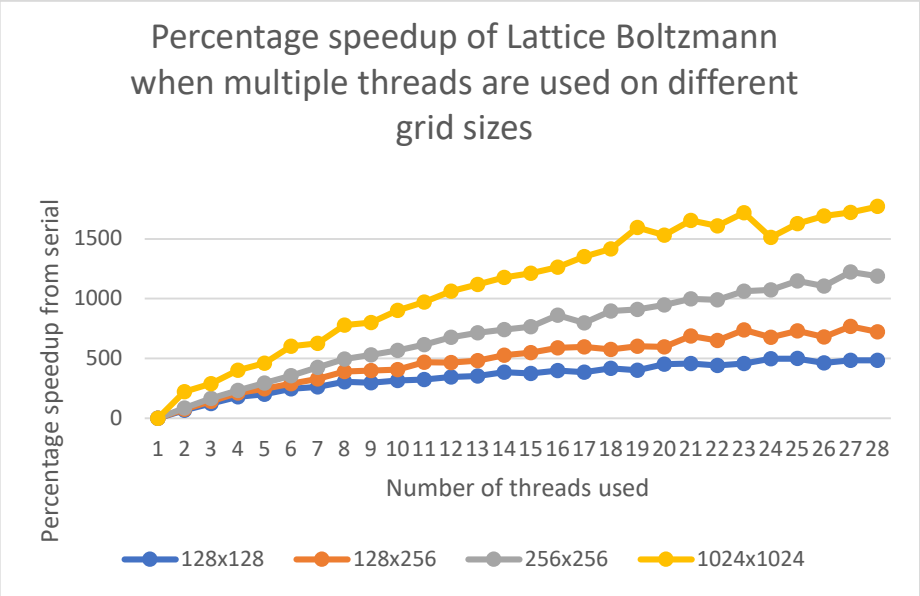


Figure.3: A graph to show the percentage speedup shown from using different numbers of threads.

The runtime on a 1024x1024 grid has a high degree of uncertainty past 14 threads. On 28 threads the runtime can be between 10.9 to 18.7 seconds. This may be because of how the threads and memory was being used. As Blue Crystal uses a NUMA architecture, that could cause the uncertainty. Using environment variables and the use of pragmas I attempted to make the code NUMA aware. `OMP_PROC_BIND=true` made sure threads wouldn't change logical core. `OMP_PLACES=cores` ensured each core would get one thread. The initializer loop had the `omp parallel for` pragma used on it. This made sure access patterns matched the propagation loop. Memory was assigned on the same thread that would access it using the first-touch principle. However the uncertainty remained. Intel's VTune tool[7] reports 6.8% of memory accesses require using memory not directly available to the core requesting it. Therefore my changes were insufficient.

Fixing this may lead to a great increase in performance, as VTune reports 28% of CPU time was spent spinning due to load imbalance. All work schedules were static. Therefore the only cause for these unbalances must be the erratic memory access speeds.

Figure 4 shows that a majority of my final implementation was compute bound as opposed to memory bandwidth bound. It also shows that the cache was well used. 75% of all memory bottlenecks are in the faster cache rather than DRAM.

Despite this, VTune reports 49.8% of cpu time was spent poorly. This is shown in Figure 6 This means there was a lot of waiting for memory requests. Future optimizations should use the cache more.

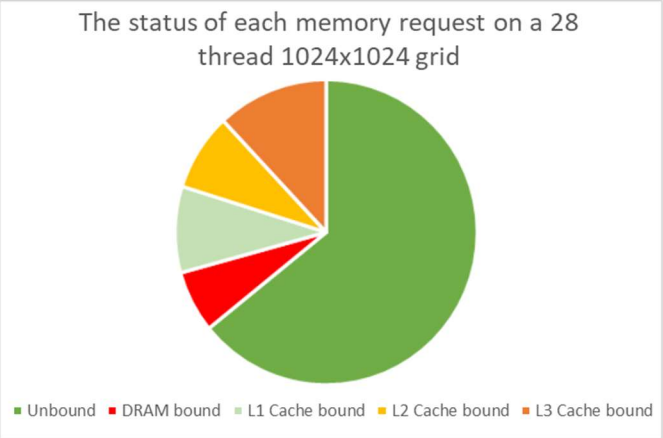


Figure 4: The status of memory requests in the final parallel program

An example of such an optimization could be tiling the grid. Each thread currently updates several rows of the grid. If they updated some N by N tile of the grid, this would allow for better cache use. This is because each cell requires the cells above and below them to be loaded to update. If a small tile was used, these values would still be in the cache when they or their neighbours are updated, reducing memory bottleneck.

Conclusions

Figure 5 clearly shows that parallelization improves performance compared to serial execution.

Figure 5: A table of average execution times between serial and parallel implimentations, on different grid sizes

Size of grid	Initial execution (no optimizations) (-O3 flag only)(s)	Serial execution (with SIMD) (s)	Parallel execution (28 threads) (s)
128x128	28.8	5.40	0.892
256x256	224	43.3	3.08
1024x1024	959	236	13.42

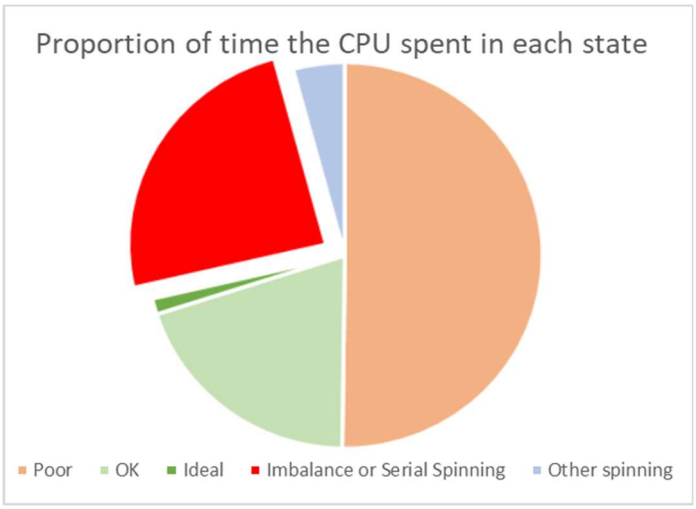


Figure 6: A pie chart showing the class of performance a CPU reached proportional to its runtime

A more perfect implementation would ideally solve the CPU and memory bottlenecks I experienced. Comparing Figures 4 and 6, I believe memory bandwidth to be the greatest bottleneck in this program. As figure 5 shows, half of CPU time was spent poorly waiting for memory, and memory speed imbalance wasted a further quarter of CPU time. One way to resolve this would be to add more CPU cache. CPU developers seem to have decided the same, with AMD researching new technologies to increase cache capacity[8].

The techniques discussed in this report show how far performance can be pushed with the correct techniques. That, with the correct optimizations, the same problem can be solved up to 7000% faster (Figure 5) highlights the importance optimization and high performance holds.

Bibliography

1: *OpenMP* [online] Available at: https://www.openmp.org/

2: *GCC 6 update log* [online] Available at: https://gcc.gnu.org/gcc-7/changes.html

3: *GCC 7 update log* [online] Available at: https://gcc.gnu.org/gcc-7/changes.html

4: *Intel oneAPI DPC++/C++ Compiler* [online] Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html

5: *Godbolt Compiler Explorer* [online] Available at: https://godbolt.org/

6: *Intel® Advisor* [online] Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/advisor.html

7: *Intel® VTune™ Profiler* [online] Available at: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

8: *AMD 3D V-Cache™ Technology*. [online] Available at: https://www.amd.com/en/campaigns/3d-v-cache