Performance Programming Coursework

Exam Number: B136013

April 4, 2019

Contents

1 Introduction

This is the coursework for the Performance Programming class of High-Performance MSc program at the University of Edinburgh. The goal of this course is to provide students with the awareness of how an application can be modified in order to harvest the capabilities of the underlying machine and achieve competitive performance in terms of time and memory.

In the following report, the author describes the scientific project that is under examination. In terms of the methodology, the optimisation mechanisms will be introduced with the target platform that the algorithm is designed to run. Moreover, there is a brief discussion about building and testing tools that are a necessary part of a standalone software project. The next chapter analyses the applied optimisations, providing detailed information for each of them. It states the initial issue, describes the optimisation role, advantages, disadvantages and where they are applied in the code. After optimisation discussion for each category, a general performance analysis of the optimisations is mentioned justifing the choice of optimisations. In the end, there is further work worth to mention exploring more options and brief conclusions for this project.

2 Project Description

This project is a computer simulation that investigates the movement of molecules in a 3D space. The molecules collide with each other, and their movement takes into consideration external factors like the wind. Newton's equations of motion are used to calculate the trajectories of the molecules.

The given algorithm of a simplified MD (molecular dynamics) simulation is computationally intensive. That happens based on the fact that the code contains a significant number of iterations over arrays with double precision numbers. As a result, the CPU is continuously busy doing floating point operations, and the memory receives a high number of requests to access the corresponding data.

The challenge of this project is that these characteristics make the simulation algorithm to require adjustments in order to produce the results in a reasonable amount of time. The improvements in the performance of the code lurk dangers. The developer should be careful not to damage the readability of the code or not to make it unsustainable.

3 Methodology

3.1 Optimisation Methods

In the project, a variety of code optimisations has been investigated, and those that proved to improve the code have been applied. These optimisations are classified in some basic types based on the nature of the performance issue that they resolve. These types are:

- Compiler Flags
- Data Structures
- Code Structure
- Vectorisation

These categories are extensively analysed in Section 4 that contains the optimisation methods.

3.2 Platform

The optimised software is designed and built to be a single threaded application running primarily on the back end of Cirrus supercomputer. Generally, the changes in the code, data structures and building tools take into consideration the specifications of the system. The compiler is configured to export AVX2 instructions. This instructions set targets the Intel(R) Xeon(R) CPU E5-2695(Broadwell) v4 @ 2.10GHz processors, with 64 byte cache line. This is the dominant model of processors on Cirrus. Thus, trying to run the executable on another device with different CPU architecture will possibly fail.

3.3 Measurements

In order to monitor the performance of the optimisations, a variety of measurements has been recorded. The most important of these measurements is the running time for 500 timesteps. In terms of profiling tools, Gprof has been used for timing this process. Cirrus provides Valgrind which is used to produce report analysis of the memory in the application. Detailed measurements like memory access and cache miss ratio for both first and last level (LL in other words L3) caches are generated and used in the report. At this point, we have to point out that memory investigation captures only the first iteration of **evolve** function because of the time-consuming nature of memory analysis. This is not an issue as long as we compare all memory measurements across different optimisation for just one timestep. In total, there is a detailed record of the measurements for each optimisation in Appendix A.

3.4 Build Tools

The original code is written in C language. The compiler used to translate the source code is ICC [1]. ICC is developed by Intel, and it is available on Cirrus. ICC offers the option to optimise the executable for the underlying Intel processor technology. GNU Make [2] is the selected mechanism to describe the compiling and linking procedure of the application. In terms of the user interaction, two shell scripts are responsible for coordinating these actions for a successful building and running of the project. The first script `run.sh` runs the simulation in the frontend of Cirrus, and it is located in the top folder. The second script `bench.pbs` performs the same procedure submitting the job in the backend of Cirrus, this file is located as well in the top folder of the project.

3.5 Testing

Testing is essential for software projects. Especially on this occasion, where the aggressive changes of the compiler to the machine code can lead to wrong results. For this reason, a correctness test program of the output files has been developed. The course organiser has developed this program which calculates the difference between two output files of the simulation measuring the deviation of one to another. Another testing procedure includes the check for NaN values in the output files of the simulation which occur by undefined mathematical operations like 0/0.

These correctness tests have been gathered in the python script: `correctness.py`. This script iterates all of the output files checking for the existence of NaN values. Then compares them with the output files of the initially supplied code using the given program: `diff-output`. In case that the results are as expected it prints "Pass" otherwise "Fail".

4 Implementation

The optimisations are divided into four categories as stated in the methodology chapter. The report presents these categories from the easiest to apply to the hardest. For each optimisation, there is a short explanation of the issues in the original code and the places that the same issue appeared. Succeeding that there is a description of the optimisation techniques that used to solve the problem accompanied by the original and transformed code. Moreover, there is a short comment from the perspective of maintainability and readability. At the end of each optimisation category there a performance analysis of the applied methods focusing on the running time and memory measurements.

4.1 Compiler Flags

Compilers provide a great variety of configurations that can effortlessly optimise the code behind the scenes and improve the performance of the code drastically [3]. Most of the following flags are specifically for the Intel Compiler [4]. In case of compiling the application with another compiler, e.g. GCC, some flags have to be replaced.

4.1.1 Optimisations

Removed Flags Knowing that the delivered code is correct, checking the correctness of the program as described above with the python script, is sufficient. As a result, there is not a reason to keep unnecessary flags that prevent optimisations or reduce execution time for the sake of correctness. The removed flags are:

- **-check=uninit** which checks occurs for uninitialised variables
- **-check-pointers:rw** checks bounds memory access for r/w through pointers
- **-no-vec** prevents vectorisation

Debugging and Profiling Flags The following flags are responsible for warning generation, producing profiling information in the compiled files, or enabling features in the code. These flags have neglectable impact on the overall performance of the code.

- **-Wall** enables warning and error diagnostics
- **-Winline** warns when a function that is declared as inline is not inlined
- **-g** instructs the compiler to generate debugging information
- **-pg** compiles a source file for profiling
- **-restrict** pointers can be qualified with the restrict keyword

- **-qopenmp-simd** enables OpenMP SIMD compilation for the vectorisation technique that will be presented later

ICC provides a reporting tool to log the compiler actions and avoid the user from analysing assembly code; this tool is accessible by inserting: **-qopt-report-file=FILENAME** and **-qopt-report=5** flags. Listing 1 shows a sample of the produced report.

```
LOOP BEGIN at MD.c(28,9)
  remark #15388: vectorization support: reference vis[i] has aligned access
  remark #15388: vectorization support: reference mass[i] has aligned access
  remark #15305: vectorization support: vector length 4
  remark #15301: SIMD LOOP WAS VECTORIZED
  remark #15448: unmasked aligned unit stride loads: 4
  remark #15452: unmasked strided loads: 15
  remark #15453: unmasked strided stores: 6
  remark #15475: --- begin vector cost summary ---
  remark #15476: scalar cost: 243
  remark #15477: vector cost: 103.250
  remark #15478: estimated potential speedup: 2.350
  remark #15488: --- end vector cost summary ---
```

Listing 1: ICC report

Optimisation Flags This category of flags is specific for performance improvements. From this point onwards performance analysis metrics are recorded after each experiment. The first one is **-Ofast** which sets drastic optimisations, e.g. loop unrolling, variable renaming, and floating-point optimisations that improve the performance significantly. The other one is **-ipo** ICC specific flag which enables inter-procedural optimisation between files.

ISA Specific Flags As mentioned before, the code is specifically designed to run on Cirrus processors. For this reason, **-xBROADWELL** compiler flag has been used. This flag indicates the compiler that the simulation runs on a Broadwell processor. In this way, the compiler is instructed to generate appropriate instructions that target its architecture.

4.1.2 Performance Analysis

After deleting the initial checking flags, the program has run in the back end of Cirrus. Based on Appendix A, the simulation completed in **1242.52** seconds having **33.4** billion data references, **128.4** million L1 and **78.7** million LL cache misses. Then the optimisation and ISA specific architecture flags added. The running time reduced significantly to **81.28** seconds noting roughly **15x** speedup, which are great results having in mind the invested effort. Moreover, also the memory measurements improved. The data references reduced to **1.7** billion and the L1, and LL cache misses to **116.8** and **73.4** respectively.

4.2 Data Structures

Refactoring the data structures can improve the performance of the application exceedingly. Changing the memory allocation type of the variables, performing array interchange, aligning and padding the data are some of the actions that need considering.

4.2.1 Data Structure Optimisations (Part I)

Memory Allocation Type First of all, changing the type of memory allocation for the most used data is essential. Using dynamic allocation the compiler is not able to take advantage of the data and apply critical optimisations. Changing the data to be static enables the compiler to choose where to store them in memory in compile time and ensure contagious memory allocation for the arrays. This optimisation is not always plausible because the programmer has to know the size of the array. In our case, the size of the data arrays, e.g. pos, velo, f in coord.h is known, and the dynamic allocations can become static. This modification changes a statement from this form: `double *mass = calloc(Nbody,sizeof(double));` to this: `double mass[Nbody];`. This change improves the readability of the code by the elimination of malloc function calls.

Array Interchange At this point, 2D arrays like pos are declared like:
`double pos[Ndim][Nbody];` which Ndim is the X, Y, Z dimension and Nbody the number of particles. If the dimensions are swapped then spatial locality can be increased because the Ndim is the fastest changing dimension in most of the nested loops [1]. Hence, the compiler will be able to perform advanced optimisations like loop unrolling for Ndim and vectorisation for Nbody, which initially was not profitable due to the size of the dimensions [2]. At this point, some changes to the code structure have to occur as well.

Due to array interchange some of the **util.c** functions have to move temporarily to the **MD.c** file knowing that this will improve performance and harm readability due to the avoidance of function calls.

This optimisation is performed in all of the 2D arrays in the simulation. It worth to mention that this process was challenging because it easily creates algorithmic mistakes and it needs a lot of modifications in the code.

[1]The loops which set the viscosity and wind term in the force calculation does not have Ndim as fastest changing dimension, but we address this issue later

[2]The three elements in Ndim dimension does not perform well on vectorisation.

Memory Alignment The data arrays of the simulation are fetched into multiple cache lines. Each array can be aligned to start from the beginning of a cache line. In this way, the number of used cache lines and therefore the number of cache misses is reduced. This is achieved by setting the aligned attribute to 64 (cache line size in Bytes) in the declaration of the array e.g.
`double mass[Nbody] __attribute__((aligned(64)));`. This technique also aids the compiler to produce fewer code lines for vectorised loops as it does not need to take care of misalignment data. Memory alignment techniques affect the readability slightly because of the extra information in the declaration.

Array Padding Compiler put static arrays back to back in memory. It is plain to see this phenomenon if we print the memory addresses. Based on this fact, array's data may share the same cache line. In that case, if both of two back to back arrays are used in the same loop, then data are evicted from the cache due to conflicts. Adding additional unused space of size at least the size of the cache line reduces the cache misses. In the code the padding size has been defined as 64 and the declaration of an array is made as `double mass[Nbody + PADDING];`. This optimisation has been performed for all of the data arrays.

4.2.2 Performance Analysis

Running Time Figure 1 shows the running time improvements for each of the data structure optimisations. It is plain to see that changing the memory allocation type and array interchange reduced the overall time significantly from **81.28** to **73.97** and then to **66.03**. On the contrary, memory alignment and array padding did not affect the running time of the simulation.

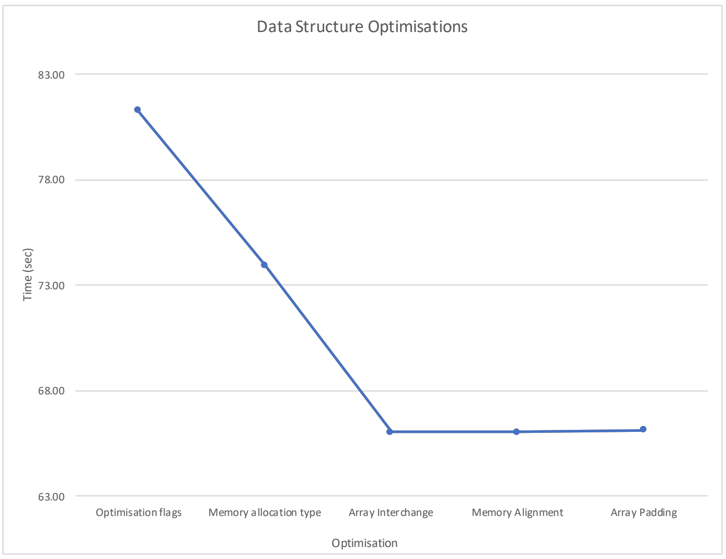
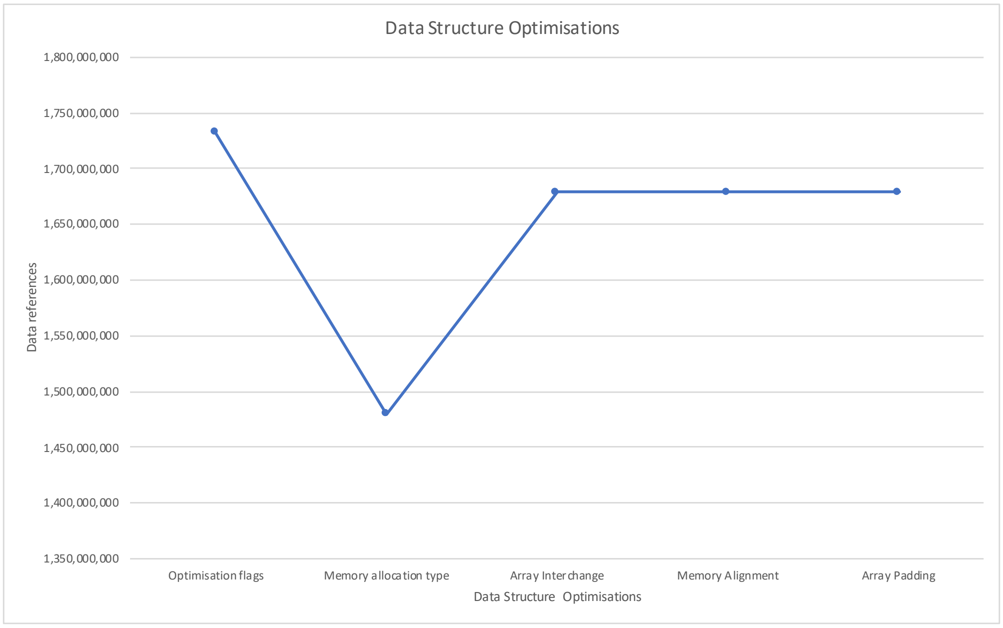
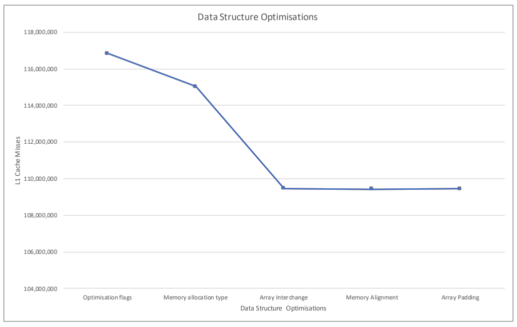


Figure 1: Data Structure Optimisations Timing Measurements

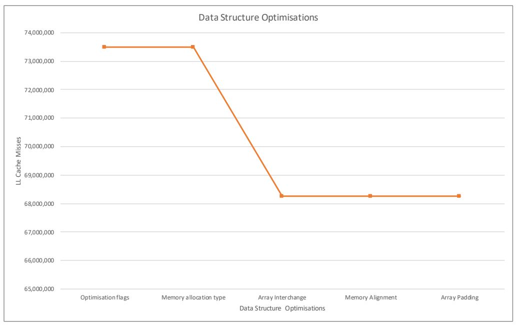
Memory When it comes to memory, according to Figure 2 (a) the allocation type optimisation was the only one that reduced the data references from **1.7** to **1.4** billion accesses. On contrary, the array interchange had a negative impact on increasing the accesses from **1.4** to **1.6** billion without been a problem as long as the running time has been declined. Despite that fact, further investigation is needed to justify this change. The other changes had not any noticeable impact. In terms of cache misses, based on Figures 2 (b) and (c) memory allocation type and array interchange optimisation reduced the misses for L1 and LL from **116.8** and **73.4** to **109.4** and **68.2** accordingly. The rest of the optimisation as happened with data references did not contribute to improvements.



(a) Data Structure Optimisations Data References



(b) Data Structure Optimisations L1 cache misses



(c) Data Structure Optimisations LL cache misses

Figure 2: Data Structure Optimisations Memory Measurements

4.3 Code Structure

In this section, the code structure modifications are discussed. These changes include any code refactoring actions that made affecting the code maintainability, readability and performance.

This type of optimisations modifies the code to take into advantage the memory hierarchy of the system and minimise the latency of data access or even change the algorithm avoiding unnecessary calculations. In this section, there is a discussion of these modifications as proposed in the lectures to achieve performance improvements [5].

Indentation Before the actual code modifications GNU C coding style has been applied to the code [6] using the GNU indent program [7]. These action has been done in order to improve the readability of the code without any influence on the performance of the code.

4.3.1 Code Structure Optimisations

Initialisation In the previous section, replacing **calloc** with static allocation does not initialise the data. As a result, we have to ensure that the elements in the arrays are zero. This is not necessary, all of the arrays are explicitly initialised by the input files or by the algorithm in the main loop of **evolve** function. Hence, the initialisation part has removed.

Loops Interchange This optimisation takes into consideration data locality in nested loops on 2D arrays, by changing faster the last index to access continuous data. Currently, the fastest changed array dimension, access data that belong to distant cache lines. In the experiment, every nested loop that contains Nbody and Ndim range nested loops have been modified, so the outer loop does Nbody iterations and the inner Ndim iterations. Listing 2 shows this modification, which a Ndim loop contains a function that has a loop of Nbody range. After the modification, the loops are exchanged, exploiting data locality.

```
for(j=0;j<Ndim;j++){ /*before*/
    wind_force(Nbody,f[j],vis,wind[j]);
}

for(i=0; i<Nbody; i++) { /*after*/
    wind_force(Ndim, f[i], vis[i], wind);
}
```

Listing 2: Loops Interchange

Generally, the indexes have also changed in this part for consistency reasons. Indexes: i and j are used for Nbody size arrays, k for Npair size arrays and l for Ndim size arrays. At this point, util functions can be used again in MD.c file with slight modifications in their body to match the array dimensions.

Routine in-lining Functions are essential in software development to separate the logic and reduce the size of the code blocks. In this way, readability and portability are improved significantly. The disadvantage of this practice is that it reduces the performance, as a function call needs hundred's of cycles to complete. Making inline functions solves this issue. During compilation, the inline function call is replaced by the actual code. For this reason the functions in **util.c** file moved to **util.h** (new header file) and are declared using **inline** keyword, e.g.
`inline double force (double W, double delta, double r);`

Loops Fusion Many loops in the simulation share the same iteration space. The multiple instances of the same loop body can be avoided with loops fusion. This technique merges two loops of the same iteration range. In this way, temporal locality is improved and the running time is lessened significantly. This action expands the body of the new loop damaging the readability of the code.

At this stage, two loop fusions have been done. The first one merges four loops. The calculation of the distance from the central mass, the addition of viscosity and wind term in the force calculation and finally the calculation of central force. The second one fuses the updates of positions and velocities. Listing 3 contains an example of the latter improvement in the given code.

```
for(i=0; i<Nbody; i++) /*before loop 1*/
    for(l=0; l<Ndim; l++)
        pos[i][l] += dt * velo[i][l];

for(i=0; i<Nbody; i++) /*before loop 2*/
    for(l=0; l<Ndim; l++)
        velo[i][l] += dt * (f[i][l]/mass[i]);

for(i=0; i<Nbody; i++) { /*after loop 1 & 2*/
    for(l=0; l<Ndim; l++) {
        pos[i][l] += dt * velo[i][l];
        velo[i][l] += dt * (f[i][l]/mass[i]);
    }
}
```

Listing 3: Loop Fusion

Loop fusion of different subroutines is also possible and profitable. In our case the inline functions **visc_force** and **wind_force** have been combined in a single function merging the encapsulated loops. Listing 4 shows this optimisation.

```
/*before*/
inline void visc_force(int N,double *f,double *vis,double *velo) {
          for(int i=0;i<N;i++)
            f[i] = -vis[i] * velo[i];
}
inline void wind_force(int N,double *f, double *vis, double velo) {
          for(int i=0;i<N;i++)
            f[i] = f[i] -vis[i] * velo;
}

inline void visc_wind_force(int N,double *f, \
double *vis,double *velo,double wind) {/*after*/
    for(int i=0;i<N;i++)
         f[i] = -vis[i] * (velo[i] + wind);
}
```

Listing 4: Loop Fusion Inline Functions

Loops Reversal In the simulation the "add pairwise forces" and "update positions and velocities" loops use both **f** and **pos** arrays. Based on this fact, we invert the iteration space of the second outer loop from **Nbody-1** to **0** as shown in Listing 5. In this way, the last loop accesses at first the already cached data from the first loop. This method will reduce the cache misses of the arrays.

```
for(i=Nbody-1; i>=0; i--) {
  for(l=0; l<Ndim; l++) {
    pos[i][l] += + dt * velo[i][l];
    velo[i][l] += + dt * (f[i][l]/mass[i]);
  }
}
```

Listing 5: Loop Reversal

Conditionals The existence of an if-then-else statement in a loop obligates the program to check the condition in each iteration. We can avoid that and perform the checking just one time. Moving the conditional statement outside of the loop and duplicating the loop inside the condition blocks, achieve this goal.

Listing 6 show the one example that this optimisation has performed in the code in the "add pairwise forces" loop. This improvement harms both readability and maintainability because it doubles the code size and every change in the loop has to be performed in two parts of the code.

Data Flow and Loop Fusion 2 This optimisation made by taking an in-depth look at the data flow of three blocks of loops that:

```
/* before */
for(l=0; l<Ndim; l++) {
    if( delta_r[k] >= Size ) /* if statement */
    else /* else statement */
}
/* after */
if( delta_r[k] >= Size )
    for(l=0; l<Ndim; l++) /* if statement */
else
    for(l=0; l<Ndim; l++) /* else statement */
```

Listing 6: Conditionals

- calculate pairwise separation of particles
- calculate the norm of separation vector
- add pairwise forces

First of all, the second block has one loop of **Npair** iterations. The loop can be transformed into a nested loop, with both of the new loops iterating **Nbody** times. This transformation gives the opportunity fusing these 3 loop blocks to exploit data locality of **delta_r** and **delta_pos** arrays as mentioned above in Section 4.3.1 (Loop Fusion).

Other optimisations

- **Dead Code Elimination** Usually compiler are good in eliminating code that has not any usage in the results. Despite that fact, dead code has to be eliminated because it increases the complexity of the algorithm. For example, **collided** variable has been removed and the variable **collisions** have been used instead.
- **Avoid Recalculations** Avoiding recalculations of the same expression save a meaningful amount of execution time. For this reason, the function **force(mass_square,delta_pos[k][l],delta_r[k]);** have been calculated once and its results is saved in a variable for later usage in multiple statements. Moreover, results of arithmetic operation that contain pre defined values have pre defined as well. For example, **G*M_central** operation has been defined in the preprocessor as: `#define G_mul_M_central G*M_central`. This kind of optimisation has been done by the compiler automatically, but in this way once again, we ensure that it is applied.

4.3.2 Data Structure Optimisations (Part II)

The following paragraphs in this section contain 2 additional data structure optimisation that occurred after the code transformations above.

Memory Accesses Taking a good look at the algorithm and the data usage in the loops, we can denote that the arrays: **r** in "calculate distance from central mass" loop and **delta_pos**, **delta_r** in "add pairwise forces" loop are used as variables in each iteration without any reference or reuse of the stored array data again. Changing these three arrays to variables the memory size, therefore, the memory accesses have decreased significantly making it the most crucial optimisation in terms of memory usage. The detailed implementation of this improvement is presented in Appendix B.

Data Scope The data arrays of the simulation are declared in global scope. In this way, any function that includes **coord.h** file can use the arrays. Unfortunately, the compiler cannot optimise the data structures the best possible way [3]. Moving the declaration of the arrays from the global space of **coord.h** file to the main function of **control.c** file solves this issue. This means that the variables have to pass as arguments in **evolve** function. This action harms readability, overloading the function's declarations with the parameters.

Moreover, because the arrays in C are passed as pointers, **restrict** type qualifier has to be added. This indication instructs the compiler that a pointer to an object is only accessible through this pointer and modifications by another pointer will lead to undefined behaviour. The qualifier enables features like vectorisation because the compiler knows that different pointers address to different data.

4.3.3 Performance Analysis

Running Time Optimisations in code structures managed to reduce running time to half. Time performance improvement for each of the applied optimisations is shown in Figure 3.

On the one hand, improvements such as loop interchange, routine inlining, loop reversal, data flow and data scope had not any impact in the running time. On the other hand, the rest of the methods had worth noting contribution. First of all, Loop fusion offered significant speedup, improving the running time of the simulation from **66.03** to **50.53** seconds. Secondly, moving the condition statements outside of the loop gave a boost of around **3** seconds. Last but not least, memory access and other optimisations played a vital role to lessen time from **47.04** to **34.95**.

Memory Once more the the optimisations reduced in a vast amount the number of data references. Based on Figure 4(a) Code optimisations led to almost 9 times less references from **1.8** to **0.2** million data accesses which is noticeable achievement. According to Figures 4(b) and 4(c), similar pattern exists in the change rate of L1 and LL cache misses. The cache misses reduced from **109,438,025** to **8,259,432** and from **68,248,004** to **27,766** for L1 and LL respectively.

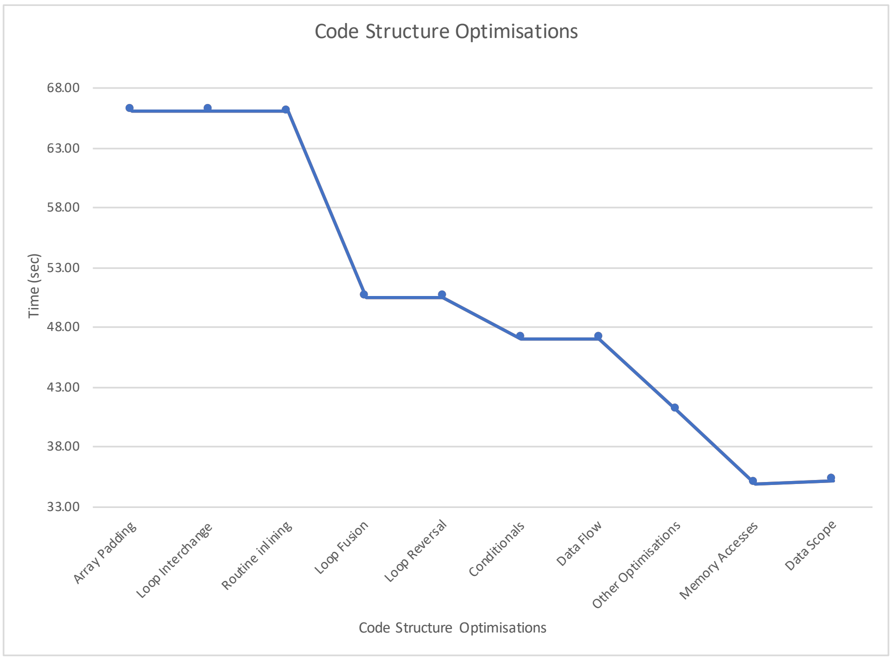
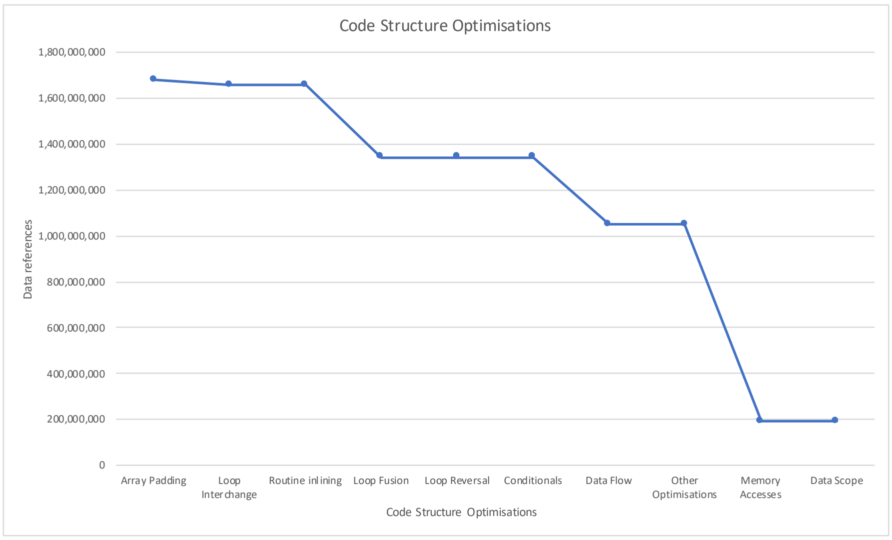


Figure 3: Code Structure Optimisations Timing Measurements

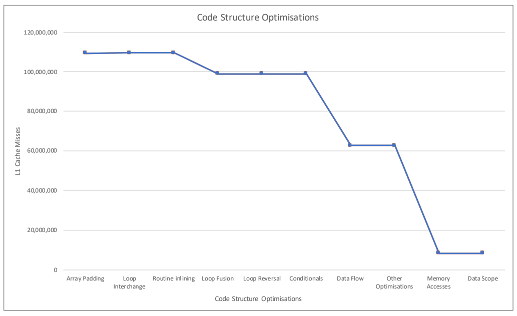
Loop interchange, routine inlining, loop reversal, conditionals and data scope optimisations did not have any drastic deviations. In contrast, loop fusion offered some improvements taking into consideration that already cached loops are reused as shown in the presented Listing 3. In terms of data flow and loop fusion, they made the data references to drop by 293 thousand. More importantly, L1 but especially LL cache misses decrease remarkably due to data locality as mentioned before. With respect to memory accesses optimisation that removed **r**, **delta_pos**, and **delta_r** arrays reduced the data accesses from **1** to **0.19** million. This logical because the last two of the three arrays were located into the most computationally heavy loop of the simulation. When it comes to cache misses for both L1 and LL they are reduced as well, because fewer data in the cache means fewer conflicts between different data.

4.4 Vectorisation

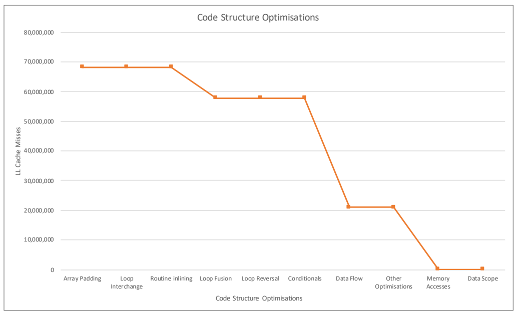
The simulation is a single threaded application. For this reason, in order to improve the performance it has to exploit parallelism that can be achieved using the hardware inside the same core. Vectorisation of loops is a way to achieve this goal. Using SIMD (single instruction multiple data) instructions in the program enables the hardware to run the same command on multiple data concurrently which is a common pattern in the simulation.



(a) Code Structure Optimisations Data References



(b) Code Structure Optimisations L1 cache misses



(c) Code Structure Optimisations LL cache misses

Figure 4: Code Structure Optimisations Memory Measurements

4.4.1 Optimisations

Loops Using -O2 and above optimisation flags, vectorisation of loops is enabled. Adding the appropriate pragma directives e.g. `#pragma simd` above the loop instructs the compiler to vectorise the loop. Vectorising the code with `#pragma simd` directive brings new issues under consideration. These are data dependencies between different iterations or further optimisation that can be done. Bellow there is list of the used clauses explaining their characteristics.

- **aligned** Tells the compiler that all data inside the loop are aligned
- **private** Specifies that the data scope of the variables inside the loop are private
- **reduction** Performs the declared operation over the data across iterations
- **linear** Specifies data with linear relationship with the loop variable

Listing 7 shows two examples of vectorisation in the simulation that use these clauses.

```
#pragma simd aligned private(delta_r, delta_pos,\
force_val, mass_square, radius_sum) \
reduction(+:tmp_f, collisions)
for(j=i+1; j<Nbody; j++) {
    /*code*/
}

#pragma simd aligned linear(mass)
for(i=Nbody-1; i>=0; i--) {
    /*code*/
    velo[i][l] += + dt * (f[i][l]/mass[i]);
}
```

Listing 7: Vectorisation

Inline Functions In the simulation, inline functions reside inside vectorised loops or even inside branches of vectorised loops. The developer has to inform the compiler about that; otherwise, we do not know if the loop will be vectorised. As a result, `#pragma omp declare simd` or `#pragma omp declare simd inbranch` directives have to be added on top of the inline function declaration accordingly.

Unfortunately, this modification was not recorded in the measurements because there was some unexpected error and due to limited amount of time it was not able to be resolved and run on the back end.

4.4.2 Performance Analysis

Vectorisation offered a slightly decreased running time from **35.12** to **30.02** seconds and reduced the data references from **190** to **142** millions thanks to SIMD instructions. This improvement has been achieved exclusively by the concurrent execution of the calculations in the vectorised loops. More details about the measurements are presented in Appendix A.

5 Further Work

5.1 Other optimisations

During experimenting with optimisation methods, some of them did not have any significant impact or even worsened the performance. Some of those worth noting are loop distribution and loop tiling that did not make it to the final version.

5.2 Compilers

GCC Experimenting with GCC (GNU Compiler Collection) has been investigated as well. The corresponding flags of the GCC compiler have replaced the ICC flags for reporting and architecture specific code generation. The new flags are: **-march=core-avx2**, **-fdump-tree-vect-all=report_gcc.txt**, and **-fopt-info -fopt-info-loop-optimized**. Moreover, the ICC directives for vectorisation have been replaced by **#pragma GCC ivdep** directive to achieve similar vectorisation results. Running the experiment the running time increased comparing to the final submission of ICC from **30.02** to **33.02** seconds. Interestingly enough, while the memory references increased from **142** to **182** million the cache misses for L1 and LL decreased from **8.3** to **8.2** million and from **29** to **27** thousand respectively.

LLVM Another compiler has been examined for this experiment. LLVM compiler accompanied by Clang frontend is a relatively new C compiler available on Cirrus [8] [9]. Once again the Intel ICC specific flags have been replaced. In this implementation, **-march=broadwell** is used to specify the Broadwell architecture of the compiler and **-Rpass=loop-vectorize** to produce the vectorisation report. Moreover, the compiler has been instructed to vectorise the loops using **#pragma clang loop vectorize(enable)** directive. In terms of performance analysis the running time increased from **30.02** to **35.10** seconds, in addition the data references inclined drastically from **142** to **588** million. The cache misses followed the same fashion, noting increase. LLVM could achieve better performance, but due to lack of time further investigation was not made.

5.3 Skylake architecture

Cirrus has recently acquired GPU nodes; each of these nodes contains two new more advanced CPUs than the most of Cirrus nodes. These CPUs are 2.4 GHz, 20-core Intel Xeon Gold 6148 (Skylake) series processors. Changing the flag that produces Broadwell ISA specific instructions to create Skylake instructions will harvest the capabilities of this processor. The flag of the Intel compiler ICC to produce architecture specific code is **-xSKYLAKE**. As a result, the running time declined dramatically from **30.02** to **22.59** seconds. The memory references and cache misses remained almost the same.

6 Conclusions

To sum up, there are a variety of optimisations from flags to code restructure that improve the performance of the application. A good practice would be to have a clear image of the performance target to be achieved. Once the developer has this target, then optimisations starting from the easiest to the most difficult can be applied until the target performance goal has been achieved.

A Measurements

Optimisation Name	Time(sec)	Data References	Cache misses L1	Cache misses LL
Initial (Removed Flags)	1242.25	33,476,427,235	128,462,338	78,731,996
Optimisation flags added	81.28	1,732,437,793	116,839,251	73,489,919
Memory allocation kind	73.97	1,479,760,028	115,025,410	73,489,814
Array Interchange	66.03	1,679,160,292	109,468,569	68,248,087
Memory Alignment	66.03	1,679,165,963	109,430,721	68,248,238
Array Padding	66.12	1,679,160,229	109,438,025	68,248,004
Loop Interchange	66.12	1,658,347,606	109,448,340	68,248,006
Routine Inlining	66.03	1,658,347,344	109,448,276	68,248,004
Loop Fusion	50.53	1,343,470,127	98,922,904	57,764,160
Loop Reversal	50.53	1,343,397,314	98,922,649	57,764,160
Conditionals	47.04	1,343,472,251	98,922,984	57,764,241
Data Flow	47.04	1,049,927,848	62,786,942	21,056,970
Other Optimisations	41.08	1,049,925,847	62,786,869	21,056,883
Memory Accesses	34.95	190,480,283	8,287,860	27,830
Data Scope	35.21	190,452,602	8,259,432	27,766
Vectorisation	30.02	142,751,132	8,348,651	29,799
Skylake Architecture	22.59	142,740,201	8,349,023	29,902
GCC	33.33	182,211,709	8,269,075	27,370

B Evolve function in MD.c

```c
void
evolve(int count, double dt)
{
  int  step;
  int  i, j, k, l;
  double  radious_sum, force_val, mass_square;
  /*
   * Loop over timesteps.
   */
  count = 1;
  for (step = 1; step <= count; step++) {
    printf("timestep %d\n", step);
    printf("collisions %d\n", collisions);

    for (i = 0; i < Nbody; i++) {

      /* calculate distance from central mass */
      r[i] = 0.0;
      add_norm(Ndim, &r[i], pos[i]);
      r[i] = sqrt(r[i]);

      /* set the viscosity term in the force calculation */
      /* add the wind term in the force calculation */
      visc_wind_force(Ndim, f[i], vis[i], velo[i], wind);

      /* calculate central force */
      for (l = 0; l < Ndim; l++) {
        f[i][l] -= force(G_mul_M_central * mass[i], pos[i][l], r[i]);
      }
    }

    /* add pairwise forces. */
    k = 0;
    for (i = 0; i < Nbody; i++) {
      for (j = i + 1; j < Nbody; j++) {
        /* calculate pairwise separation of particles */
        for (l = 0; l < Ndim; l++) {
          delta_pos[k][l] = pos[i][l] - pos[j][l];
        }

        /* calculate norm of separation vector */
        delta_r[k] = 0.0;
        add_norm(Ndim, &delta_r[k], delta_pos[k]);
        delta_r[k] = sqrt(delta_r[k]);
```

```
            radious_sum = radius[i] + radius[j];
            mass_square = G * mass[i] * mass[j];

            /* flip force if close in */
            if (delta_r[k] >= radious_sum) {
              for (l = 0; l < Ndim; l++) {
                force_val = force(mass_square,delta_pos[k][l],delta_r[k]);
                f[i][l] -= force_val;
                f[j][l] += force_val;
              }
            } else {
              for (l = 0; l < Ndim; l++) {
                force_val = force(mass_square,delta_pos[k][l],delta_r[k]);
                f[i][l] += force_val;
                f[j][l] -= force_val;
              }
              collisions++;
            }
            k = k + 1;
          }
        }

        for (i = Nbody - 1; i >= 0; i--) {
          for (l = 0; l < Ndim; l++) {
            pos[i][l] += +dt * velo[i][l];
            velo[i][l] += +dt * (f[i][l] / mass[i]);
          }
        }
      }
    }
```

Listing 8: Array declaration in control.c

C Array declaration in control.c

```
double pos[Nbody][Ndim] __attribute__((aligned(64)));
double velo[Nbody][Ndim] __attribute__((aligned(64)));
double f[Nbody][Ndim] __attribute__((aligned(64)));
double vis[Nbody] __attribute__((aligned(64)));
double radius[Nbody] __attribute__((aligned(64)));
double mass[Nbody] __attribute__((aligned(64)));
double wind[Ndim + PADDING] __attribute__((aligned(64))) = {0.9, 0.4, 0.0};
```

Listing 9: Array declaration in control.c

References

[1] Intel Software. Intel C++ Compiler. https://software.intel.com/en-us/c-compilers. (Accessed on 03/02/2019).

[2] GNU. GNU make. https://www.gnu.org/software/make/manual/make.html. (Accessed on 03/02/2019).

[3] Stephen Booth. Optimising with the Compiler, Lecture Notes in Performance Programming, February 2019.

[4] Manpage of icc. http://www.dartmouth.edu/ rc/HPC/man/icc.html. (Accessed on 03/27/2019).

[5] Stephen Booth. Compiler optimisation III, Lecture Notes in Performance Programming, February 2019.

[6] GNU. GNU Coding Standards: Writing C. https://www.gnu.org/prep/standards/html_node/writing-C.html. (Accessed on 03/02/2019).

[7] GNU. indent: Indent and Format C Program Source. https://www.gnu.org/software/indent/manual/indent.html. (Accessed on 03/02/2019).

[8] Chris Lattner. LLVM and Clang: Next Generation Compiler Technology, 2008.

[9] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL, Dec 2002. *See* `http://llvm.cs.uiuc.edu`.