|epcc|

Performance Programming Report

B159973

April 10, 2020

Contents

1 Introduction

1.1 Machine

The machine that all the experiments ran was the EPCC's Cirrus back-end[1]. All the jobs were deployed using *qsub* command on 1 node. Each node on Cirrus has a massive amount of 256GB RAM and an x86 architecture with 36 CPU cores Intel(R) Xeon(R) CPU E5-2695. The cache memory consist of: L1 cache of 32KB, L2 cache of 256KB and L3 cache of 46080KB. No multi-threading or multi-processing methods were used for decreasing runtime. In essence, the program runs on serial without exploiting any parallelism using programming techniques or running on multiple threads or CPUs.

1.2 Software Compiler

The current compiler that is used in the whole experiment is Intel C/C++ Compiler 18.0 for Linux. The module is loaded on Cirrus using *intel-compilers-18*. For compiling the command used is *icc*. The Intel compiler is a state of the art software compiler and provides the best code optimizations in the market. Cirrus nodes are composed of Intel cores with Broadwell micro-architecture. Intel's compilers can effectively handle many optimization techniques that rely on their micro-architectures. The other reason for using this compiler rather than any other one is due to that there is a compatibility using the Intel Vtune profiler.

1.3 Measuring Standards & Output Validation

For each optimization that is stated inside the report, the program ran on the back end Cirrus for five times and each time the total CPU time was recorded. After that, the time statistics are gathered using UNIX shell scripting. The script uses awk for extracting the timings and these are saved in a .csv format. Afterwards a python script briefly calculates the average run-time and standard deviation using in-build statistics modules. This approach is used for having clear view of the executable runtime for each modification and minimizing time observational errors. Validating the correctness of the output of the program was done using the *diff-check* that was provided. Every modification mentioned in the report was tested for correctness and then it was inserted and discussed in this report.

```
#!/bin/bash
# Accumulate runtime of the five runs
echo "runtimes" > res.csv
for i in {1..5}
do          # Grab the total time
    ./MD | awk '/500 timesteps / {print $4}' >> res.csv;
done
```

Listing 1: Running executable multiple times and gathering statistics

```
import pandas as pd

df = pd.read_csv('res.csv')
x = df['runtimes'].std()
y = df['runtimes'].median()
print ('Average: '+ str(y))
print ('STD: '+ str(x))
```

Listing 2: Calculating average run time and standard deviation statistics

1.4 Initial Profiling

An initial profiling investigation was performed using the Intel's VTune Amplifier[2]. The loaded module on Cirrus was *intel-vtune-19/2019.0.2.570779*. The tool figured out were the hot-spots were located as well as severe other important CPU parameters. The profile was done using *amplxe-cl -collect hotspots* which analyzes the hotspots of on the serial code. The data collected by VTune was stored and organized inside a directory and then a report is generated using *amplxe-cl -report hotspots -r -dir*.

2 Program Flow

First of all the program checks if there is an input from the user for setting the *Nsteps* variable. A time-step is an iteration of the main loop evolve() function. By default the number of Nsteps is 100, if the user doesn't change it. The program gathers data from the input file with a simple fscan() that reads 9 columns is used. Each column is a 16 digit double and data are stored in these variables and multidimensional arrays:

Variable	Description	Default Value
Nbody	Number of particles	4096
Ndim	Number of dimensions	3
Npair	Number of pair particles	8386560
Nstep	Number of time-steps	100
Nsave	Number of times to run Nstep time-steps	5
collisions	Number of collisions between particles	0
dt	Timestep value	0.2

Data Structure	Description	Memory Size
pos	Position of the particles	12288 doubles (98.30KB)
velo	Velocity of the particles	12288 doubles (98.30KB)
r	Distance of particles from central mass	4096 doubles (32.76KB)
f	Forces that act on particles	12288 doubles (98.30KB)
mass	Particle mass	4096 doubles (32.76KB)
radius	Particle radius	4096 doubles (32.76KB)
vis	Particle velocities	4096 doubles (32.76KB)
delta_r	Particles pairs separation distance	16777216 doubles (134.21MB)
delta_pos	Particle pairs separation positions	50331648 doubles (402. 65MB)
wind	Wind force of the three dimensions	3 doubles (24B)

After reading the data from an input file, it performs many calculations using the *evolve()* function on some of these arrays and outputs the manipulated data to separate files. Inside the report each array is analyzed for its contribution to the program. For *nsave = 5*, the program outputs five output.dat files (output.dat100 ... output.dat500), which each one has the same file size and type. These arrays are being used for every nsave iteration and their result is passed for the next nsave iteration. Therefore output.dat500 has a different result than output.dat100.

File name	Type	Memory Size
input.dat	ASCII text	580KB
output.dat	ASCII text	580KB

The outputs of every optimization in the report were tested for correctness with diff-check and passing as argument the output.dat500 of the initial attempt (original program as given).

The *evolve()* function is called in *main()* contains several other functions within it:

Function Name	Parent Function	Description
void evolve((int count, double dt)	void main(int argc, char *argv[])	Updates the input velocity and position by calculating and adding the pairwise forces for each particle.
void vis_force(int N, double *f, double *vis, double *vel)	void evovle (int count, double dt)	Calculates the viscosity force and store the result in the *f* array
void wind_force(int N, double *f, double *vis, double *vel)	void evovle (int count, double dt)	Calculates the wind force and store the result in the *f* array
void add_norm(int N, double *r, double *delta)	void evovle (int count, double dt)	Normalizes the **delta* array and store the result inside **r* array
void force(double W, double delta, double r)	void evovle (int count, double dt)	Calculates the central force by the given *delta* position *r* distance and *W* mass

3 Compiler Optimizations

3.1 Compiler Flags

The first observation is that inside the Makefile there are three flags enforcing some form of checking and restrictions :

```
-check=uninit -check-pointers:rw -no-vec
```

Check-pointers: Enable the Intel's pointer checker which checks bounds for reads and writes that are done using pointers.[4]
No-vec: Disables the any vectorization in the code with SSE2 instructions that is inserted by calling 02 flag and above. [5]
Check=uninit: Figures out on whether to check for uninitialized variables. A run-time error will be executed if a variable is read before it is modified with a write operation. [7]

Using the Vtune Profiler, these are the average runtimes of the top 5 most heavy CPU resource consuming functions:

Module Name	Function Name	Average Runtime (seconds)
MD	_libm_pow_l9	1637.660s
MD	evolve	1484.757s
libchkp.so	__chkp_check_bounds	1207.172s
MD	force	956.713s
libchkp.so	__chkp_map	639.356s

The Vtune tool gave an approximation that most of the time is spent inside the function *_libm_pow_* which is an internal implementation of the *pow()* function. This is probably called by the second most consuming function, the evolve() which is the heart of the program. Moreover, the *force()* function is again called by *evovle()*. The *__chkp_check_bounds* is introduced by the flag *-check-pointers* and checks the bounds the pointers. These flags introduce functions that make the program inefficient because checking for reading/writing bounds and uninitialized variables has a lot of overhead. Moreover they restrict vectorization which is undoubtedly helpful for achieving higher performance. Therefore they must be discarded. After discarding these flag the output was inspected and it was correct. The runtime tremendous decreased. The program ran again using Intel's optimization compiler flags. The following table shows the runtimes:

Compiler Flag \ Statistics	Average Runtime	Standard Deviation	Speed Up
-O (Disable Optimizations)	2954.99	18.2	1
-O1	661.34	15.3	4.5
-O2	136.56	10.34	21
-O3	142.43	4.53	21
-Ofast	143.749	6.12	21
-ipo -mtune=broadwell	85.579	0.52	35

4 Code Optimization

4.1 Data Structures

There are multiple data structures inside the program. Inside the *coord.h* all the arrays that are describe on table 3 are initialized using *extern* keyword. This keyword extends the whole visibility of the variables and functions thought all the files that compile and produce the executable. In other words these arrays are initialized inside the BSS segment which contains all the global and static variables. Any function in any file has access these globals. An optimization would be to replace the *extern* keyword by calling the evolve() function and passing all the necessary parameters that are needed (all the arrays that are used). All the arrays were initialized inside the main() function instead of the coord.h file.

```
void evolve(int count,double  dt, double *pos[Ndim], double *velo[Ndim],double *f[Ndim
    ], double *vis,double  *mass, double *radius,double *delta\_pos[3],double *r,
    double *delta\_r,double wind[Ndim],int collisions);

void evolve(int count,double dt)
```

Data Definition	Average Runtime	Standard Deviation
extern keyword	85.35	0.52
local scope (Declared in main())	85.64	0.45

There is no special decrease in run-time after changing the declaration of the evolve() function. This proposed optimization has no significant meaning as only the scope of the variables is changed.

4.2 Aligned Arrays

```
  r =             _mm_malloc(Nbody*
    sizeof(double),32);
  delta_r =       _mm_malloc(Nbody *
    Nbody* sizeof(double),32);
  mass =          _mm_malloc(Nbody*
    sizeof(double),32);
  radius =        _mm_malloc(Nbody*
    sizeof(double),32);
  vis =           _mm_malloc(Nbody*
    sizeof(double),32);
  f[0] =          _mm_malloc(Ndim *
    Nbody* sizeof(double),32);
  pos[0] =        _mm_malloc(Ndim *
    Nbody* sizeof(double),32);
  velo[0] =       _mm_malloc(Ndim *
    Nbody* sizeof(double),32);
  delta_pos[0] =    _mm_malloc(Ndim *
    Nbody * Nbody* sizeof(double),32);
```

Listing 3: Dynamic Allocation

```
double *__attribute__((aligned(32))) pos[
    Ndim];
double *__attribute__((aligned(32))) velo[
    Ndim];
double *__attribute__((aligned(32)))f[Ndim
    ], *vis, *mass, *radius;
double *__attribute__((aligned(32)))
    delta_pos[3];
double __attribute__ ((aligned(32))) wind[
    Ndim];
```

Listing 4: Static Allocation

Aligned arrays are essential such as for vectorization. The data was aligned on definition and on dynamic allocation. The Intel Intrinsics Guide guide [6] suggest to use *_mm_malloc()* and the compiler flag -qopt-dynamic-align for dynamic allocation. The AXV instructions are more compatible using 32 bytes aligned data structures because the arrays are loaded faster inside the vector registers.

4.3 Loop Optimizations

Inside the evovle() function, there are multiple loops that perform a vast amount of calculations. Initially the evovle() function is separated into two distinct parts for showing the workflow: a)Before calculating the pairwise separation - which it is consisted of 7 different loops and the following are calculated: viscosity and wind values in the force computations and figuring out the distances from the central mass and the central forces, b)after calculating the pairwise separation of the particles which is consisted of 7 loops again, where the normalization of separation vector, addition of the pairwise forces and update of the positions and the velocities take place. The code as handed, utilizes to many resources and produces a massive amount of unnecessary computation to execute the algorithm. After careful examination, the algorithm can be composed by two different loops. Four pseudocodes are shown below. The first two are dedicated to show the difference of the first part and the other two for the second part of evovle() function. It is inevitable to code the same algorithm without two loops.
The first redefined loop combines all the seven loops from the first part and the second redefined loop all the seven loops from the second part. In essence the first part, initializes the f array which is the forces that act on each particle. For the second part, the pairwise force (which are $((Nbody * (Nbody - 1))/2)$) are added and used to update positions and velocities. The update positions and velocities arrays are the outputs of the evolve() function.

Algorithm 1 Envolve() First Part - Unchanged

$N \leftarrow NumberOfParticles$
$D \leftarrow NumberOfDimensions$
for $i \leftarrow 0, D$ **do** ▷ Loop 1 - Viscosity Term
  for $j \leftarrow 0, N$ **do**
    $f[i][j] \leftarrow -vis[j] * velo[i][j]$
  end for
end for
for $i \leftarrow 0, D$ **do** ▷ Loop 2 - Wind Term
  for $i \leftarrow 0, N$ **do**
    $f[i][j] \leftarrow f[i][j] - vis[j] * wind[i]$
  end for
end for
for $k \leftarrow 0, N$ **do** ▷ Loops 3,4,5 - Distance from central mass
  $r[k] \leftarrow 0.0$
end for
for $i \leftarrow 0, D$ **do**
  for $k \leftarrow 0, N$ **do**
    $r[k] \leftarrow r[k] + (pos[i][k] * pos[i][k])$
  end for
end for
for $i \leftarrow 0, N$ **do**
  $r[k] \leftarrow sqrt(r[k]))$
end for
for $k \leftarrow 0, N$ **do** ▷ Loop 6 - Central force
  for $i \leftarrow 0, D$ **do**
    $f[i][k] = f[i][k] - force(G * mass[i] * M\_central, pos[i][k], r[k]);$
  end for
end for

The Loop 1 and 2 can be easily merged because the iterations are the same (loop iterations: number of particles). There is however a data dependency between them, so in order to combine them in one equation, the same order has to be kept inside the equation. The loop 3,4,5 also curry the same amount of iterations (1 to number of particles) and can be merged into one loop. The array r[k] which is the distance of particles from the central mass is first initialized to zero, then distance from the central masses are added for the three dimensions and finally the result is squared. Finally the central forces in loop 6 are calculated by subtracting the force that are generated by the mass of the particle, the gravitational force and a constant with the delta position and is divided with distance of the particle from the central mass. It is important to distinguish that this happens for each dimension (x,y,z).

Algorithm 2 Envolve() First Part - Redefined

$N \leftarrow NumberOfParticles$
$D \leftarrow NumberOfDimensions$
procedure LOOP(Calculate Loops 1 - 6)
  for $k \leftarrow 0, N$ **do**
    $r \leftarrow sqrt((pos[0][k]^2) + (pos[1][k]^2) + (pos[2][k]^2))$

$$f[0][k] = f[0][k] - (vis[k] * velo[0][k]) - (vis[k] * wind[0]) - ((G * m * M\_central * pos0)/(r * r * r))$$
$$f[1][k] = f[1][k] - (vis[k] * velo[1][k]) - (vis[k] * wind[1]) - ((G * m * M\_central * pos1)/(r * r * r))$$
$$f[2][k] = f[2][k] - (vis[k] * velo[2][k]) - (vis[k] * wind[2]) - ((G * m * M\_central * pos2)/(r * r * r))$$

  end for
end procedure

In the redefined loop, the loop 1 and 2 are written as two subtractions on each element of the f array $f[0,1,2][k] - = (vis[k] * velo[0,1,2][k]) - (vis[k] * wind[0,1,2])$. The outer loop that consists of three iterations is diminished by performing loop unrolling. The loops 3,4 and 5 are substituted with one simple equation $r = sqrt((pos[0][k]^2) + (pos[1][k]^2) + (pos[2][k]^2)$. Again here the loop is unrolled and each dimension distance is added. The whole addition is then squared. The result is not stored into the array r as previously, because it is only needed for the following three calculations locally inside the loop (scalar/array replacement). This saves about 32KB of memory as the r array uses 4096 doubles. The force() function is also replaced by in-lining the equation for the three dimensions.

4.3.1 Efficient Code for Redefined Loop

```
for (k = 0; k < Nbody; k++){
      double pos0 = *(*(pos + 0) + k);
      double pos1 = *(*(pos + 1) + k);
      double pos2 = *(*(pos + 2) + k);
      double m = G *  M_central * mass[k];
      double r = pos0 * pos0 + pos1 * pos1 + pos2 * pos2;
      double v = vis[k];
      r = sqrt(r);
      r = r*r*r;
      f[0][k] = -(v * velo[0][k]) - (v * wind[0]) - (m*pos0/r);
      f[1][k] = -(v * velo[1][k]) - (v * wind[1]) - (m*pos1/r);
      f[2][k] = -(v * velo[2][k]) - (v * wind[2]) - (m*pos2/r);
      }
```

Here just to show another way to access the array, the C code *(*(pos + 0) + k) and pos[0][k] is converted to the same assembly code! For reducing array accesses, we assign them into doubles. The forces() function is replaced using inline calculations with the m variable being calculated only one time. The loop is unrolled into three distinct expressions.

Algorithm 3 Evolve() - Second Part - Unchanged

$N \leftarrow NumberOfParticlePairs$
$D \leftarrow NumberOfDimensions$
procedure LOOP 1(Pairwise particle separation)
  for $i \leftarrow 0, N$ **do**
    for $j \leftarrow i, N$ **do**
      for $l \leftarrow 0, D$ **do**
        $delta\_pos[l][k] \leftarrow pos[l][i] - pos[l][j]$
      end for
      $k \leftarrow k + 1$
    end for
  end for
end procedure
procedure LOOPS 2,3,4(Normalization of separation vector)
  for $k \leftarrow 0, N$ **do**
    $delta\_r[k] \leftarrow 0.0$
  end for
  for $i \leftarrow 0, D$ **do**
    for $k \leftarrow 0, N$ **do**
      $delta\_r[k] \leftarrow r[k] + delta\_pos[i][k] * delta\_pos[i][k])$
    end for
  end for
  for $k \leftarrow 0, N$ **do**
    $delta\_r[k] \leftarrow sqrt(r[k]))$
  end for
end procedure
procedure LOOP 5(Add pairwise forces)
  $k \leftarrow 0$
  for $i \leftarrow 0, N$ **do**
    for $j \leftarrow i, N$ **do**
      $size = radius[i] + radius[j]$
      $collided = 0$
      for $l \leftarrow 0, D$ **do** ▷ Flip force if close in
        if $delta\_r > size$ **then**
          $f[l][j] \leftarrow f[l][j] - force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
          $f[l][j] \leftarrow f[l][j] + force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
        else
          $f[l][j] \leftarrow f[l][j] + force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
          $f[l][j] \leftarrow f[l][j] - force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
          $collided \leftarrow 1$
        end if
        if $collided == 1$ **then**
          $collisions++$
        end if
      end for
      $k \leftarrow k + 1$
    end for
  end for
  procedure LOOP 6(Update Positions)
    for $i \leftarrow 0, N$ **do**
      for $j \leftarrow i, N$ **do**
        $pos[j][i] \leftarrow pos[j][i] + dt * velo[j][i]$
      end for
    end for
  procedure LOOP 7(Update Velocities)
    for $i \leftarrow 0, N$ **do**
      for $j \leftarrow i, N$ **do**
        $velo[j][i] = velo[j][i] + dt * (f[j][i]/mass[i])$
      end for
    end for
end procedure

The first loop of the second part is iterating thought the number of the particle pairs and performs a subtraction between the positions of the particles and saves that distance in an array delta_pos. After that, loops 2,3 and 4 normalize that vector, following the same method of normalization as in the first part. Afterwards, the pairwise forces are added inside an if-statement. Finally, loop 6 and loop 7 updates the positions and the velocities into two arrays velo and pos, which these are the outputs of the function.

Algorithm 4 Evolve() - Second Part - Redefined Loop

$N \leftarrow NumberOfParticles$
$D \leftarrow NumberOfDimensions$
procedure LOOP 1- 7(Calculate pair wise forces and update velocities and positions)
  for $i \leftarrow 0, N$ **do**
    for $j \leftarrow i, N$ **do**
      $size \leftarrow radius[i] + radius[j]$
      $m \leftarrow mass[i] * mass[j] * G$
      $collided \leftarrow 0$
      for $k \leftarrow 0, D$ **do** ▷ Three iterations only
        $deltaPos[k] \leftarrow pos[k][i] - pos[k][j];$
      end for
      $delta\_r \leftarrow 0$
      for $k \leftarrow 0, D$ **do** ▷ Three iterations only
        $delta\_r \leftarrow delta\_r + deltaPos[k] * deltaPos[k]$
      end for
      $delta\_r \leftarrow sqrt(delta\_r)$
      if $delta\_r > size$ **then**
        $f[l][j] \leftarrow f[l][j] - force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
        $f[l][j] \leftarrow f[l][j] + force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
      else
        $f[l][j] \leftarrow f[l][j] + force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
        $f[l][j] \leftarrow f[l][j] - force(G * mass[i] * mass[j], delta\_pos[l][k], delta\_r[k])$
        $collided \leftarrow 1$
      end if
      if $collided == 1$ **then**
        $collisions++$
        $k \leftarrow k + 1$
      end if
    end for
    for $k \leftarrow 0, D$ **do** ▷ Three iterations only
      $pos[k][i] \leftarrow pos[k][i] + dt * velo[k][i]$
      $velo[k][i] \leftarrow velo[k][i] + (dt * (f[i]/mass[i]))$
    end for
  end for
end procedure

As observed, all the loops follow the same amount of iterations $((Nbody * (Nbody - 1))/2)$, and can be merged together in a single one. The r array can be discarded, because it is only needed as a divisor for calculating the forces that act for each particle pair inside one iteration. The delta_r can be computed on spot by calculating the delta_positions and adding them together and perform a normalization on the addition. Proceeding, a branch decision is made and the f array is updated accordingly. Finally, inside the same loop again, the update of the two arrays take place on the three dimensions. From $9((Nbody * (Nbody - 1))/2) = 75479040$(loop 3 is ((Nbody*(Nbody-1))*3) iteration, now only $\frac{1}{9}$ which are 8386560 take place with different calculation in them. All of these contributes to a lot of dead code elimination.

Any code simplification that results in less computations inside the inner loop of the second part of the evolve() function is extremely crucial. Obviously, that block of code is executed 8386560 for five

times. After modifying the second part of the evolve() function the output was tested and the diff-output executable showed difference max=0.000000) The following performance was recorded:

Statistics	Average Runtime	Standard Deviation
-O0	151.46	4.65
-O1	75.69	1.52
-O2	53.94	0.12
-O3	52.32	0.45
-Ofast	51.749	0.34
-ipo -mtune=broadwell	50.45	0.31

4.3.2 Efficient Code for the Redefined Loop

For re-coding the second part, the main idea was to modified just like the above pseudo-code. But there are many modification that can result in a more efficient program.

```
for (i = 0; i < Nbody; i++)
   {
     double f0_i = f[0][i];
     double f1_i = f[1][i];
     double f2_i = f[2][i];
     double pos0_i = pos[0][i];
     double pos1_i = pos[1][i];
     double pos2_i = pos[2][i];

     for (j = i + 1; j < Nbody; j++)
     {
       /* double size = radius[i] + radius[j];*/
       double m = mass[i] * mass[j] * G;
       double delta_pos0 = pos0_i - pos[0][j];
       double delta_pos1 = pos1_i - pos[1][j];
       double delta_pos2 = pos2_i - pos[2][j];
       double delta_r = delta_pos0 * delta_pos0;
       delta_r += delta_pos1 * delta_pos1;
       delta_r += delta_pos2 * delta_pos2;
       delta_r = delta_r * delta_r * delta_r;
       delta_r = sqrt(delta_r);
       collided = 0;

       if (delta_r >= size)
       {
          f0_i -= (m * delta_pos0) / delta_r;
          f1_i -= (m * delta_pos1) / delta_r;
          f2_i -= (m * delta_pos2) / delta_r;
          f[0][j] += (m * delta_pos0) / delta_r;
          f[1][j] += (m * delta_pos1) / delta_r;
          f[2][j] += (m * delta_pos2) / delta_r;
       }
       else
       {
          f0_i += (m * delta_pos0) / delta_r;
          f1_i += (m * delta_pos1) / delta_r;
          f2_i += (m * delta_pos2) / delta_r;
          f[0][j] -= (m * delta_pos0) / delta_r;
          f[1][j] -= (m * delta_pos1) / delta_r;
          f[2][j] -= (m * delta_pos2) / delta_r;
         collided = 1;
       }
       if (collided == 1)
       {
         collisions++;
       }
       k = k + 1;
     }
     pos[0][i] += dt * velo[0][i];
```

```
50        velo[0][i] += dt * (f0_i / mass[i]);
51        pos[1][i] += dt * velo[1][i];
52        velo[1][i] += dt * (f1_i / mass[i]);
53        pos[2][i] += dt * velo[2][i];
54        velo[2][i] += dt * (f2_i / mass[i]);
55      }
56    }
57  }
```

For starting, the pairwise forces are added on the f array which iterates from number i to the number of j. The code induces decreases in array accesses. As we can see, the f[0][i], f[1][i], f[2][i] values for each i are used only one time in the inner loop (loop-invariant code motion). Therefore it's better if these are assigned as doubles and to access them as variables rather than an array with multiple (3) indexes. The same applies also to pos[0][i], pos[1][i], pos[2][i] values. In the force function $G * mass[i]mass[j]$ is evaluated 6 times. This product is calculated only one time instead of six at the beginning of the loop (*double m*). Therefore 12 array accesses and 12 multiplication are reduced to one two multiplications and array accesses. Loop unrolling is performed on code lines 14 - 16 for the loop that calculates the position distance. Unfortunately we cannot bypass the array access for pos[0][j], pos[1][j], pos[2][j]. Delta_pos array is discarded and only three variables are used. Loop unrolling is also applied for the second loop inside the inner loop which calculates the delta_r. The delta_r array is discarded because only one variable is needed locally. The size of delta_r array is about 134.21 MB and the size of delta_pos is about 402.65 MB. These arrays are quite huge and since they are global, they consume a lot of memory that usually created on BSS / data segments. So here we are essentially doing an array replacement. As long as these results are needed locally only on one iteration there is no point in allocating and using these huge arrays that lead to loss of performance. The delta_r variable is calculated by performing 4 multiplications and three additions with one square root function. The *sqrt(delta_r)*, as examined, is by far the single most heavy resource calculation. Due out curiosity, this was tested and it costs about 1.5 seconds more for each 100 iterations, which is about 7.5 seconds for 500 timesteps! The if statement is proceeding normally, but the force() function is replaced with by inlining the calculation *mass * delta_pos / delta_r*. Moreover, loop unrolling is also performed by adding the three dimension forces on i column, and subtracting on the j column and vice versa for the flip forces. At the end, the update loops for position and velocities are discarded and six operations for all dimensions for a complete update iteration. The update procedure is outside of the inner loop, it is part of the outer loop at the end and therefore any optimization here brings minuscule performance elevation. The block of code that utilizes the most CPU resources is in the inner loop. Before discussing further, an experiment was conducted. From the evovle() function, inner loop was removed and the program finished in just 0.74 seconds! This agrees with the previous observation in the end of page 8. Any further optimization will be dedicated inside the scope of the inner loop.

4.4 Modifications based on input

These are a series of modifications that are primarily based on the input of the program. If the input is changed, the program will not behave the same and will not have the same output as a different input file. However, these adjustment can be consider as a performance enhancement only for the given input file.

After a careful observation of the input file, the summation of radius for any particle pair is always one (1.00). The radius of each particle is the second column inside the input.csv file. To obtain the size variable in the second part of the evovle() function, two array access are needed and an addition. These can be discarded and constant propagate the number 1 to be compared with delta_r inside the if-statement. Therefore the size variable can be replaced by number 1.

```
1     if (delta_r >= 1){  /* Calculate forces */}
2     else{ /* Calculate flip forces */}
```

Listing 5: Constant propagation inside if-statement

Another observation on the input is that all the values in velo[2]* and pos[2]* which correspond to the velocity and the position of the z axes are set to zero (0.00E+0). Therefore it is possible not to include

these variables and their calculations inside the inner loop: delta_pos_2, f[2][j],f2_i. This also applies for the update of the pos and velo arrays in the end of the outer loop as all the values in pos[2]* and velo[2]* are zero. The following table runtimes is based on discarding the z axis calculations.

Compiler Flags	Average Runtime	Standard Deviation
-O3	40.57	0.15
-Ofast	40.12	0.46
-ipo -mtune=broadwell	39.45	0.11

5 Vectorization

As we can see inside the evovle() function the CPU spends the most time for executing the inner for loop of the second part. These for-loops can however be vectorized when they fulfilled some conditions. The Intel Programming Guidelines for vectorization [5] suggests that any function calls, data dependencies, non vectorizable operations and mixing vectorizable types in the same loop must be avoided. Furthermore the loops must contain simple of block of code with straight-line code. In any case any branches such as if ,return statements must be avoided. They also strongly suggest the use of array notations rather than pointers, because misidentify pointers can lead to unforeseen data dependencies. For an efficient memory access, inner loops with sparse arrays are preferred (data structures with stride content). Moreover data alignment to 16-byte boundaries is essential and can be done using Intel SSE instructions and 32-byte for AVX instructions. The success of a vectorized compiler lies on an appropriate data layout. Intel suggest to use these specific modifiers provides aligned data structure: __declspec(align()) or __attribute__((aligned())) when defining a structure or a union. These already are introduced to the current code. In the end, the guidelines suggest to use structure of arrays (SoA) rather than array of structures(AoS). AoS are preferred for encapsulation but they can delay vectorization. Cirrus nodes have Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz processors. According to Intel manuals[8] these support support a 256 bit AVX2 instruction set extension. The AVX2 instructions set support Intel AVX2, Intel AVX and Intel SSE instructions with versions 4.2, 4.1, 3, 2 and 1.

5.1 User Mandated Vectorization

SIMD directives can be used in order to force a user mandated vectorization. The directives provide essential details to the compilers to enable vectorization. When *pragmas* are inserted, the compiler heuristics are completely overwritten if there is no logical fault. Therefore they give the authority to the programmer to control vector processing. This is not always a good case. If the programmer doesn't know how to use these powerful directives it may harm the performance, because compiler could apply more appropriate vectorization techniques for the specific loop. User mandate vectorization has a less complex usage than SIMD intristic classes, but generally it give less programming control.

A simple SIMD pragma instruction was used for calculating the delta positions, the f array

```
#pragma omp simd
        for (h = 0; h < Ndim; h++)
        {
          delta_pos[h] = pos[h][i] - pos[h][j];
          delta_r += delta_pos[h] * delta_pos[h];
        }
        delta_r = sqrt(delta_r);
        delta_r = delta_r * delta_r * delta_r;

        if (delta_r >=size)
        {
#pragma omp simd
          for (h = 0; h < Ndim; h++)
          {
            f[h][i] -= ((m * delta_pos[h]) / delta_r);
            f[h][j] += ((m * delta_pos[h]) / delta_r);
```

```
          }
        }
        else
        {
#pragma omp simd
          for (h = 0; h < Ndim; h++)
          {
            f[h][i] += ((m * delta_pos[h]) / delta_r);
            f[h][j] -= ((m * delta_pos[h]) / delta_r);
          }
          collided = 1;
        }
```

Listing 6: Using SIMD pragma

Compiler Flags	Average Runtime	Standard Deviation
No pragmas	50.45	0.31
pragma omp simd	59.14	0.24

Unfortunately, forcing vectorization using *pragmas* here, rewrites more efficient vectorization techniques that the compiler introduced before. The problem with using these SIMD instructions is that they must be applied with care as discussed before and on this case the compiler had done a lot better job than this block of code. Nevertheless this is simplest and easiest way to vectorize. We must proceed by trying different vectorization techniques.

5.2 Replacing Branch

```
if (delta_r >= size){  /* Calculate forces */}
else{ /* Calculate flip forces */}
```

Listing 7: Original Branch

```
    union{double _f; unsigned long long _x;} _u;
    double  r = delta_r - size;
    _u._f = r;
    int re = (int)(_u._x >>63);
    re = 1 - 2*re;

/*Calculate force*/
    /*1*/ f0_i    += (-re)*((m * delta_pos0)/delta_r);
    /*2*/ f1_i    += (-re)*((m * delta_pos1)/delta_r);
    /*3*/ f2_i    += (-re)*((m * delta_pos2)/delta_r);
    /*1*/ f[0][j] += (+re)*((m * delta_pos0)/delta_r);
    /*2*/ f[1][j] += (+re)*((m * delta_pos1)/delta_r);
    /*3*/ f[2][j] += (+re)*((m * delta_pos2)/delta_r);
```

Listing 8: Branch replacement

A union structure is created that contains a double and an unsigned long long which both are 64 bits. The *r* variable is the subtraction of the normalize particle pairwise distance and the size of the two particles. We need to extract the sign of this subtraction for proceeding with branch-less code. The result of the subtraction is quite hard to handle because sometimes it is a huge positive double with up to 5 digits, where as other times, a tiny fraction. For handling this, that subtraction result is stored inside the union as a double. The are three outcomes: either a positive double, a negative double or zero. If it is zero, the case must behave just like the positive double, it must enter the if statement (delta_r>=size). Then a bit-wise operation takes place, a right shifting for 63 bit positions inside the 64-bit union. The outcome is modified with this equation $re = 1 - 2 * re$. If there is a positive or zero outcome after the subtraction it becomes 1, else if there is a negative outcome it becomes -1. Therefore, the variable *re* becomes a co-efficient which can by multiplied with the force function. If the size is bigger the *re* becomes -1 else it becomes +1. This optimization is essential to be able to vectorize the inner loop. The casting in union is very efficient and fast. However by just applying this optimization without proceeding to vectorization does not decrease runtime.

```
    for (i = 0; i < Nbody; i++)
    {
      double f0_i = f[0][i];
      double f1_i = f[1][i];
      double f2_i = f[2][i];
      double pos0_i = pos[0][i];
      double pos1_i = pos[1][i];
      double pos2_i = pos[2][i];
      double m, size, delta_r =0 ;
      int re;
#pragma omp simd reduction(+ \
                           : f0_i, f1_i, f2_i,k,delta_r) private(m, size, re) aligned(
    pos : 32,f:32)

#pragma prefetch pos,f
      for (j = i + 1; j < Nbody; j++)
      {

        size = radius[i] + radius[j];
        m = mass[i] * mass[j] * G;

        double delta_pos0 = pos0_i - pos[0][j];
        double delta_pos1 = pos1_i - pos[1][j];
        double delta_pos2 = pos2_i - pos[2][j];
        delta_r = delta_pos0 * delta_pos0;
        delta_r += delta_pos1 * delta_pos1;
        delta_r += delta_pos2 * delta_pos2;
        delta_r = pow(delta_r,1.5);

        union {
          double _f;
          unsigned long long _x;
        } _u;
        double r = delta_r - 1;
        _u._f = r;
        re = (int)(_u._x >> 63);
        re = 1 - 2 * re;

        double s = m/delta_r;
        double a =  delta_pos0 * s;
        double b = delta_pos1  * s;
        double c = delta_pos2  * s;
        /*1*/ f0_i += (-re) * a;
        /*2*/ f1_i += (-re) * b;
        /*3*/ f2_i += (-re) * c;
        /*1*/ f[0][j] += (+re) * a;
        /*2*/ f[1][j] += (+re) * b;
        /*3*/ f[2][j] += (+re) * c;


        collisions -=  (re>>15);

        k += 1;
      }
      pos[0][i] += dt * velo[0][i];
      velo[0][i] += dt * (f0_i / mass[i]);
      pos[1][i] += dt * velo[1][i];
      velo[1][i] += dt * (f1_i / mass[i]);
      pos[2][i] += dt * velo[2][i];
      velo[2][i] += dt * (f2_i / mass[i]);
    }
  }
}
```

Listing 9: Best Solution using SIMD directives

Here the most optimal solution is presented. The SIMD reduction does a really good job on performance. It aggregates the variables *f0_i,f1_i,f2_i,delta_r and k* in that scoping clause by summing them

up. Generally, reductions operations reduce data structures by aggregating the values over one or multiple dimensions by summation, multiplication, etc. Only those variables are able to enter into that clause. However, attempts by inserting *delta_pos0, delta_pos1, delta_pos2* in the clause were made but they tend to slower the performance because these are not reduced. The if statement was replaced with the union data type and the *re* variable as mentioned before. However there was a huge obstacle on counting the particle collisions. In order to replace the if statement of counting the collisions we have to introduce a function $f(re)$ which it must produce 0 if the re is positive and 1 if the re is negative. The *collided* flag variable becomes one when it enters the else branch, so *re* must be negative as explained before. However no algebraic equation compose that function, due to the fact that $f(re)$ is not a continues function because the value range of *re* is only two distinct integers (1,-1).

$$re \in \{-1, 1\}, f(re) = \begin{cases} 1, & \text{if } re = -1 \\ 0, & \text{if } re = 1 \end{cases}$$

For bypassing this, bitwise operations come in handy. Signed integers in C are 16 bits. The binaries of -1 and 1 are:

$$\text{Binary: } 1 = 0000000000000001, \text{Binary: } -1 = 1111111111111111$$

If we left swift by 15 bits, 1 becomes 0 and -1 becomes 1. Therefore the if statement for counting the collisions can be discarded:

```
 if (collided == 1) {collisions++;}
 /*==*/
collisions -=  (re>>15);
```

Delta_r is calculated by using one time the *pow()* function instead of 2 multiplications and a *sqrt()* function which differ by a minimal amount of time. As tested the gains in performance where about 0.5 seconds. The target is to always reduce calls from math library functions.

```
 delta_r = pow(delta_r,1.5);
   /*==*/
 delta_r = delta_r * delta_r * delta_r;
```

For calculating the forces array, some modifications again where made: The $m * delta\_pos0/delta\_r$ is calculated two times. This applies for the three delta positions. A variable can be used to store that value and calculate it only time. Moreover using the following equation:

$$(m * delta\_pos0)/delta\_r = \frac{m}{delta\_r} * delta\_pos0$$

we can calculate on one time $\frac{m}{delta\_r}$ and save it in a variable and then reuse it again. This technique is lowering the total runtime by about 6-7 seconds.

```

       double s = m/delta_r;
       double a =  delta_pos0 * s;
       double b = delta_pos1  * s;
       double c = delta_pos2  * s;
       /*1*/ f0_i += (-re) * a;
       /*2*/ f1_i += (-re) * a;
       /*3*/ f2_i += (-re) * c;
       /*1*/ f[0][j] += (+re) * a;
       /*2*/ f[1][j] += (+re) * b;
       /*3*/ f[2][j] += (+re) * c;

       /*==*/

       /*1*/ f0_i += (-re) * ((m * delta_pos0) / delta_r);
       /*2*/ f1_i += (-re) * ((m * delta_pos1) / delta_r);
       /*3*/ f2_i += (-re) * ((m * delta_pos2) / delta_r);
       /*1*/ f[0][j] += (+re) * ((m * delta_pos0) / delta_r);
       /*2*/ f[1][j] += (+re) * ((m * delta_pos1) / delta_r);
       /*3*/ f[2][j] += (+re) * ((m * delta_pos2) / delta_r);
```

For all these modifications, the flags used CFLAGS= -g -mtune=broadwell -xCore-AVX2 -qopenmp -qopt-dynamic-align which enable the AVX2 instructions and dynamic data alignment. Every array inside the evolve() function was 32 bytes aligned. An attempt to prefetch arrays was made but the Intel guidelines say that pragma prefetch applies only to Intel's Advanced Vector Extensions 512 (AVX-512) [10].

Compiler Flags	Average Runtime	Standard Deviation
Without User mandated	50.45	0.31
Pragma omp reduction	23.21	0.04

5.3 Intel Intristics

5.3.1 Intrinsics for Stream Operation - mm256_stream_pd()

A special type of load(stream) that was used to fetch from the memory was *_mm256_stream_pd_*. However this special type of load that bypasses the cache and loads from the memory in a direct manner. Furthermore it inhibits the processors to write to the cache, because we are indicating that the data is non-temporal, they won't be used soon again so don't feed in cache. By using this intristic command, the performance was terrible and actually it decreased. Inside the Makefile, for a correct compilation, the flag -xCORE-AVX2 was used for generate specialized code on AVX architecture. The following headers file must be imported:

```
#include "immintrin.h" /* AVX Extensions Intrinsics include file */
#include "xmmintrin.h" /* Streaming SIMD Extensions Intrinsics include file */
```

```
/* Applying vector intristics on three arrays mass,delta_pos and delta_r*/
    __m256d deltapos =  _mm256_stream_pd(delta_pos, deltapos);
    __m256d msd =  _mm256_stream_pd(m, msd);
    __m256d deltard =   _mm256_stream_pd(delta_r, deltard);
```

5.3.2 Intrinsics for Load Operation - mm256_load_pd()

This command moves integer values from a memory location to a vector. For applying this intristic, Intel's guideline mention that the address must be 32-byte aligned, as they slide in directly into the destination vector. For compilation again the flag -xCORE-AVX2 was used but the Intel guidelines also command to use.

```
/* Applying vector intristics on three arrays mass,delta_pos and delta_r*/
     delta_pos[0] = pos0_i - pos[0][j];
     delta_pos[1] = pos1_i - pos[1][j];
     delta_pos[2] = pos2_i - pos[2][j];
     /*calcluate delta_r*/
double  deltar[4] __attribute__((aligned(32))) = {delta_r, delta_r, delta_r, 0};
double  m[4] __attribute__((aligned(32))) = {m, m, m, 0};
    __m256d deltapos =  _mm256_load_pd(delta_pos, deltapos);
    __m256d msd =  _mm256_load_pd(m, msd);
    __m256d deltard =   _mm256_load_pd(delta_r, deltard);
```

Structure of arrays(SoA) were also used for vectorization as Intel strongly encourages to do.

```
/* Applying vector intristics on three arrays mass,delta_pos and delta_r*/
      struct __attribute__((aligned(32))) SoA
    {
      double delta_pos[4];
      double ms[4];
      double deltar[4];
    } SoA;

/*********************************************
* After entering nested for-loop
```

```
**********************************************/
        /*calcluate m*/
        SoA.ms[0] = m;
        SoA.ms[1] = m;
        SoA.ms[2] = m;
        SoA.deltar[0] = delta_r;
        SoA.deltar[1] = delta_r;
        SoA.deltar[2] = delta_r;
        /*calcluate delta_r*/
        SoA.delta_pos[0] = pos0_i - pos[0][j];
        SoA.delta_pos[1] = pos1_i - pos[1][j];
        SoA.delta_pos[2] = pos2_i - pos[2][j];
        deltapos = _mm256_load_pd(SoA.delta_pos);
        msd =      _mm256_load_pd(SoA.ms);
        deltard =  _mm256_load_pd(SoA.deltar);
```

But again there was no any deliberating increase in performance. Maybe the way that these are load into the cache is inefficient. The intristics classes are hard to handle without having the proper knowledge and experience on them. The best out of these for loading the variables was:

```
    delta_pos[0] = pos0_i - pos[0][j];
    delta_pos[1] = pos1_i - pos[1][j];
    delta_pos[2] = pos2_i - pos[2][j];
    /*calcluate delta_r*/
    deltapos = _mm256_load_pd(delta_pos);
    msd =      _mm256_broadcast_sd(&m);
    deltard =  _mm256_broadcast_sd(&delta_r);
```

The *_mm256_broadcast_sd* is a broadcast command to load directly a double inside the vector register. It was significantly faster for loading a single double value inside the vector register. After loading the values the following calculation were applied:

```
    union __attribute__((aligned(32))) {
          __m256d k5;
          double a[4];
        } _k;
/*************************************************
* After entering nested for-loop
*************************************************/
    delta_pos[0] = pos0_i - pos[0][j];
    delta_pos[1] = pos1_i - pos[1][j];
    delta_pos[2] = pos2_i - pos[2][j];
    /*calcluate delta_r*/
    deltapos = _mm256_load_pd(delta_pos);
    msd =      _mm256_broadcast_sd(&m);
    deltard =  _mm256_broadcast_sd(&delta_r);
    __m256d temp = _mm256_mul_pd(deltapos, msd);
    _k.k5 = _mm256_div_pd(temp, deltard);

        f0_i += (-re) * _k.a[0];
        f1_i += (-re) * _k.a[1];
        f2_i += (-re) * _k.a[2];
        f[0][j] += (+re) * _k.a[0];
        f[1][j] += (+re) * _k.a[1];
        f[2][j] += (+re) * _k.a[2];
```

The union is allocated before entering the nested for-loop. The union has size 32 bytes and contains a 32 byte register and 4 8-bytes doubles in array. It is a nice way to access each the vector register without a storing operation which costs. A multiplication with *deltapos* (delta_pos values) and *msd*(mass) is performed using *_mm256_mul_pd*. The result is divided with *deltard* which is the vector register that contains *delta_r*. The division is happening with the *_mm256_div_pd()*. The result is stored in *_k5*. Vector intristics classes should give a better performance than this, but again they require an excellent way to handle them efficiently.

Technique	Average Runtime	Standard Deviation
Intel Intristics	111.7	2.83

6 Conclusions

Initially the program needed a huge amount of time to execute. First a good observation by Vtune Profiler helped to find the hot-spots. After that by closely examining the program work flow and by using and evaluating many performance enhancement techniques such as many different compiler flags, algebraic simplifications, scalar replacements, constant propagation, loop unrolling, loop invariant code motion, dead code elimination, SIMD vectorization, vector intristic classes, automatic vectorization, alignment in data structures, replacing functions by in-lining them, decreases in array accesses and by slightly modifying the algorithm, there was a speed up of 128 times.

Technique	Average Runtime	Standard Deviation
Initial Program	2954.99	18.2
SIMD reduction	23.21	0.044

References

[1] Cirrus Facilites - EPCC
https://www.epcc.ed.ac.uk/facilities/demand-computing/cirrus

[2] Intel® VTune™ Profiler (Formerly Intel® VTune™ Amplifier)
https://software.intel.com/en-us/vtune

[3] Compiler Switches for Performance Analysis on Linux* Targets
https://software.intel.com/en-us/vtune-help-compiler-switches-for-performance-analysis-on-linux-targets

[4] Pointer Checker Feature Summary -Intel Guides
https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-pointer-checker-feature-summary

[5] Intel® C++ Compiler 19.1 Developer Guide and Reference
https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-programming-guidelines-for-vectorization

[6] The Intel Intrinsics Guide
https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[7] 1996-2011, Intel Corporation ,Compiler XE 12.1
https://scc.ustc.edu.cn/zlsc/sugon/intel/compiler_c/main_cls/copts/ccpp_options/option_check-uninit.htm

[8] Intel® Xeon® Processor E5-2695 v4
https://ark.intel.com/content/www/us/en/ark/products/91316/intel-xeon-processor-e5-2695-v4-45m-cache-2-10-ghz.html

[9] Data Alignment to Assist Vectorization -Intel
https://software.intel.com/en-us/articles/data-alignment-to-assist-vectorization

[10] Using Intel's Prefetch Pragma
https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-prefetch-noprefetch