Performance Programming Coursework

Exam Number: B129230

1. Introduction

Performance optimization is a significant process to accelerate an existing program in a particular environment. Optimization is an experimental process that there are many theories and tools to help this process. There are different types of optimization such as compiler optimization, auto-tuning and hand optimization.

The purpose of this report is to describe optimization processes and performance analysis of a single thread simple molecular dynamics program on the back end of Cirrus. In addition, the author will also explain the original code problem and results of change made to the code.

2. Experimental Preparation

2.1 Tools and Environment

Visual Studio Profiling Tools is an integral part of Visual Studio 2017 which can be used to measure different applications performance characteristics. This performance analysis suite can detect performance issues of source code and output performance report including CPU usage, GPU usage and memory usage.

Intel VTune Amplifier is a performance analysis program which can be used to find the algorithm hotspots and code bottleneck by collecting and analyzing important profiling data.

Intel Advisor is an assistant tool which can help developers build well-threaded and vectorized code that is based on hardware.

Bitbucket is a version control repository hosting service which is developed by Atlassian. The author uses Bitbucket to track code change and only available between due time and post time.
https://Epcc_Master@bitbucket.org/Epcc_Master/performanceprogramming.git

The author's work environment is a x64 architecture workstation with 16GB memory, 9MB cache, 6 cores and 12 logic processors. The CPU is Intel (R) Core (TM) i7-8750H @ 2.20GHZ and can be overclocked to 4.1GHZ. The author uses this workstation with Visual Studio IDE and its Profiling Tools and embedded Intel VTune Amplifier to analyse program.

'Cirrus' is a SGI ICE XA system with 280 computes nodes 10, 080 cores, utilizing InfiniBand interconnect. Each node provides 36 physical cores which use Intel(R) Xeon(R) E5-2695 v4, clocked at 2.1GHZ, and contains 256GB RAM per node. The optimized program and all results should be run and based on the back end of Cirrus. In addition, the author also analyses the program on Cirrus by using Intel Advisor with -qopt-report command.

All results and analysis are all based on the Intel compiler 17.0.2.

2.2 Methodology and Plan

Performance Optimization usually follows a cycle: measuring performance, utilizing theory analysis, proposing change or exit, changing the code, verifying results and then measuring performance again.

The author always bears this cycle in mind and then creates the following plan:

- the first step is to measure the program performance on Cirrus to have a general performance concept without any code change and compiler optimization;
- the second process is to use Visual Studio Profiling Tools and Intel VTune Amplifier to find hotspots and performance bottleneck of the program;
- the third step is to spend the most time on the most time-consuming part. The important thing that should be kept in mind is that optimization is a continuous work. Only when satisfied with code optimization, turn optimization on and test again.
- the fourth step is to spend some time on not very time-consuming sections, and turning on compilers optimization and auto-tuning;
- finally, the author will summarize performance improvement in different parts compared with the original part.

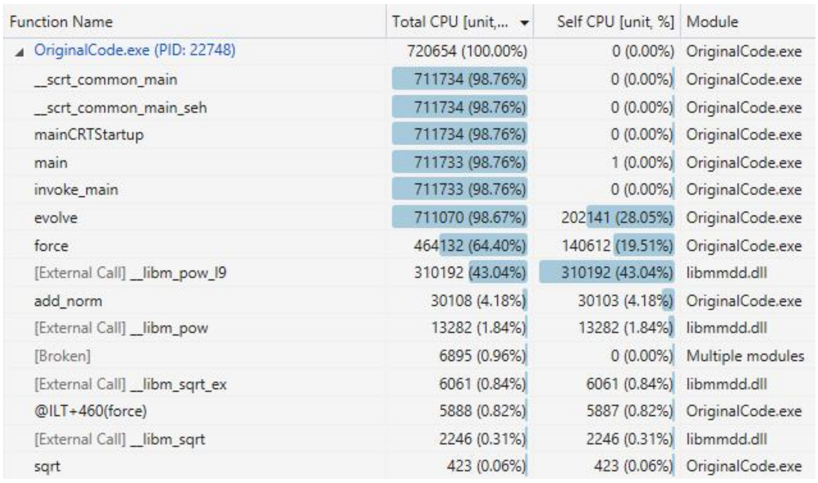
On the other hand, the author will always verify results after each optimization step compared with original results on the workstation and Cirrus respectively.

3. Code and Performance Issues

3.1 Analysis of Performance Issues

The author uses Visual Studio Profiling Tools and Intel VTune Amplifier to get the following performance hotspots analysis.

The author uses the VS Profiling Tools and Intel VTune Amplifier to understand hotspots in the program.



Function Name	Total CPU [unit,...	Self CPU [unit, %]	Module
OriginalCode.exe (PID: 22748)	720654 (100.00%)	0 (0.00%)	OriginalCode.exe
__scrt_common_main	711734 (98.76%)	0 (0.00%)	OriginalCode.exe
__scrt_common_main_seh	711734 (98.76%)	0 (0.00%)	OriginalCode.exe
mainCRTStartup	711734 (98.76%)	0 (0.00%)	OriginalCode.exe
main	711733 (98.76%)	1 (0.00%)	OriginalCode.exe
invoke_main	711733 (98.76%)	0 (0.00%)	OriginalCode.exe
evolve	711070 (98.67%)	202141 (28.05%)	OriginalCode.exe
force	464132 (64.40%)	140612 (19.51%)	OriginalCode.exe
[External Call] __libm_pow_l9	310192 (43.04%)	310192 (43.04%)	libmmdd.dll
add_norm	30108 (4.18%)	30103 (4.18%)	OriginalCode.exe
[External Call] __libm_pow	13282 (1.84%)	13282 (1.84%)	libmmdd.dll
[Broken]	6895 (0.96%)	0 (0.00%)	Multiple modules
[External Call] __libm_sqrt_ex	6061 (0.84%)	6061 (0.84%)	libmmdd.dll
@ILT+460(force)	5888 (0.82%)	5887 (0.82%)	OriginalCode.exe
[External Call] __libm_sqrt	2246 (0.31%)	2246 (0.31%)	libmmdd.dll
sqrt	423 (0.06%)	423 (0.06%)	OriginalCode.exe

Figure1: Original Code Hotspots

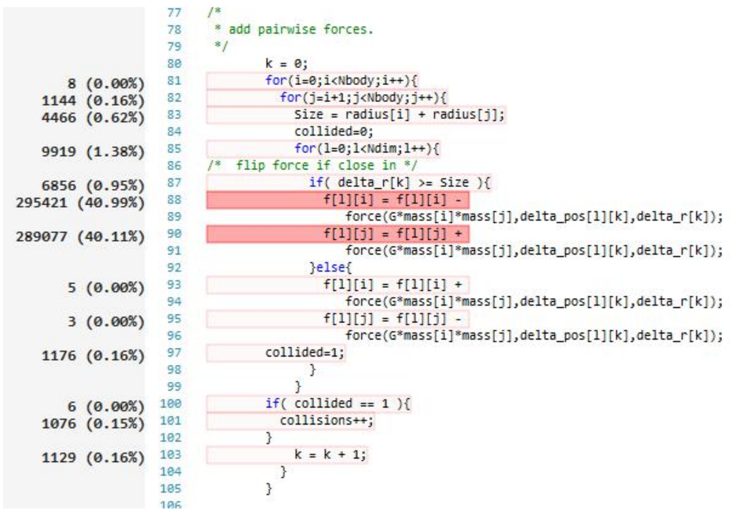
As shown in Figure 1, it is easy to find that the most workload exists in evolve() function and force() function. In addition, the function of pow() and add_norm() are also time consuming. The function of evolve is the most workload and it accounts for 98.67% in the whole program.

According to Intel VTune Amplifier analysis, it has a similar top hotspots as VS profiling tool. The functions doing most individual work as shown in figure 2.

Top Hotspots		
Function	Module	Exclusive Samples
_libm_pow_l9	libmmdd.dll	42.88%
evolve	MD.exe	27.48%
force	MD.exe	20.66%
add_norm	MD.exe	4.08%
pow	libmmdd.dll	3.31%

Figure 2: Intel VTune Performance Analysis

The author looks into evolve() function to find specific performance issues. As shown before, the part of adding pairwise forces is the most workload and especially the calculating f[][] array has around 81% of 98%.

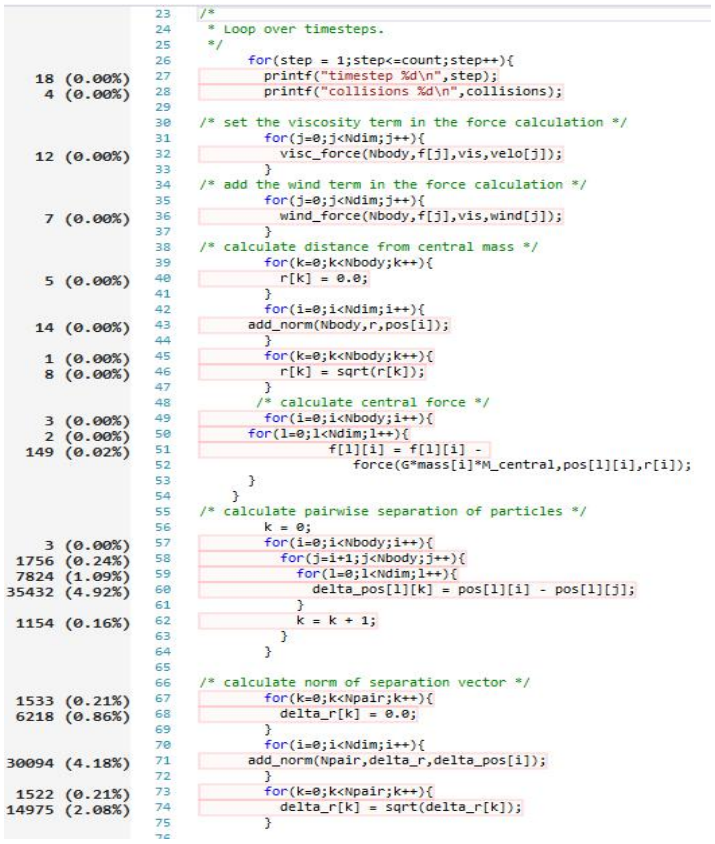


```
                     /*
                      * add pairwise forces.
                      */
                             k = 0;
       8 (0.00%)             for(i=0;i<Nbody;i++){
    1144 (0.16%)               for(j=i+1;j<Nbody;j++){
    4466 (0.62%)                 Size = radius[i] + radius[j];
                                 collided=0;
    9919 (1.38%)                 for(l=0;l<Ndim;l++){
                     /*  flip force if close in */
    6856 (0.95%)                   if( delta_r[k] >= Size ){
  295421 (40.99%)                    f[l][i] = f[l][i] -
                                        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
  289077 (40.11%)                    f[l][j] = f[l][j] +
                                        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
                                   }else{
       5 (0.00%)                     f[l][i] = f[l][i] +
                                        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
       3 (0.00%)                     f[l][j] = f[l][j] -
                                        force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
    1176 (0.16%)             collided=1;
                                   }
                                 }
       6 (0.00%)             if( collided == 1 ){
    1076 (0.15%)               collisions++;
                             }
    1129 (0.16%)                 k = k + 1;
                               }
                             }
```

Figure 3: Performance Analysis of evolve()

The 295421 means evolve() and its children used 295421 ms of CPU time (295421 sample(s) at 1000 samples/sec) and then the percentage means 40.99% of CPU time. Thus, the author will start optimization and analysis from this the most workload part.

Other workload is mainly distributed in the parts of calculating pairwise separation of particles and calculating norm of separation vector with 6.5% and 7.5% respectively as shown in figure 4.



```
                  /*
                   * Loop over timesteps.
                   */
                       for(step = 1;step<=count;step++){
   18 (0.00%)            printf("timestep %d\n",step);
    4 (0.00%)            printf("collisions %d\n",collisions);

                  /* set the viscosity term in the force calculation */
                         for(j=0;j<Ndim;j++){
   12 (0.00%)              visc_force(Nbody,f[j],vis,velo[j]);
                         }
                  /* add the wind term in the force calculation */
                         for(j=0;j<Ndim;j++){
    7 (0.00%)              wind_force(Nbody,f[j],vis,wind[j]);
                         }
                  /* calculate distance from central mass */
                         for(k=0;k<Nbody;k++){
    5 (0.00%)              r[k] = 0.0;
                         }
                         for(i=0;i<Ndim;i++){
   14 (0.00%)          add_norm(Nbody,r,pos[i]);
                         }
    1 (0.00%)            for(k=0;k<Nbody;k++){
    8 (0.00%)              r[k] = sqrt(r[k]);
                         }
                        /* calculate central force */
    3 (0.00%)            for(i=0;i<Nbody;i++){
    2 (0.00%)          for(l=0;l<Ndim;l++){
  149 (0.02%)                    f[l][i] = f[l][i] -
                                    force(G*mass[i]*M_central,pos[l][i],r[i]);
                       }
                     }
                  /* calculate pairwise separation of particles */
                         k = 0;
    3 (0.00%)            for(i=0;i<Nbody;i++){
 1756 (0.24%)              for(j=i+1;j<Nbody;j++){
 7824 (1.09%)                for(l=0;l<Ndim;l++){
35432 (4.92%)                  delta_pos[l][k] = pos[l][i] - pos[l][j];
                             }
 1154 (0.16%)                k = k + 1;
                           }
                         }

                  /* calculate norm of separation vector */
 1533 (0.21%)            for(k=0;k<Npair;k++){
 6218 (0.86%)              delta_r[k] = 0.0;
                         }
                         for(i=0;i<Ndim;i++){
30094 (4.18%)          add_norm(Npair,delta_r,delta_pos[i]);
                         }
 1522 (0.21%)            for(k=0;k<Npair;k++){
14975 (2.08%)              delta_r[k] = sqrt(delta_r[k]);
                         }
```

Figure 4: Performance Analysis of evolve()

Apart from these three the most workload parts, other parts almost do not occupy the CPU time. These three parts occupy nearly 95% time. Thus, the author will spend time on these critical parts.

The original code will use the following compiler flag:

-g -O0 -check=uninit -check-pointers:rw -no-vec

-g

It is used to tell the compiler to generate a level of debugging information in the object file. If the program wants to get the better performance, it should be compiled in release version rather than the debug version. Thus, the compiler flag of -g should be removed for maximal performance

-O0

the project can be executed correctly but slowly with O0. O0 means no compiler optimization is used in the program

-check=uninit

This compiler flag determines whether checking occurs for uninitialized variables. Option -check-uninit means enabling this checking to ensure there is not read before writing for a variable. Otherwise, there will be a run-time error.

According to icc manual, run-time checking of undefined variables is only implemented on local, scalar variables rather than dynamically allocated variables. However, the main variables are global and dynamical in the program. Thus, this compiler flag is not very necessary, and it will slow performance.

-check-pointers:rw
The compiler flag determines whether the compiler check bounds for memory access through pointers. “rw” means checks bounds for reads and writes through pointers.

-no-vec
This compiler flag will disable all auto-vectorization such as vectorization of array notation statements.

Compiler flags	*500 iteratins*
Originl Makfile	*5195*
-O0 Only	*1014.90s*
-O1 Only	*671.30s*
-O2 Only	*138.30s*
-O3 Only	*140.60s*

Tablet 1: The Results of Original program

The results of original code with different compiler flags are shown in Tablet 1.

3.2 Analysis of Code

As analysis before, there are three the most workload loop blocks which should be spent the most time on. They are part of calculating pairwise separation of particles, part of calculating norm of separation vector and part of adding pairwise forces. They are explained in detail in the following sections from 3.2.3 to 3.2.5.

Modern compilers are very good at optimization except for memory optimizations, which means compiler optimizations are not all done by compilers so that programmers should help compiler not only including adding compiler flags. For example, the compiler is hard to change memory layouts and moreover always chooses correctness rather than performance. Thus, programmers need to optimize code to help compiler so that the author focuses on these three parts in terms of vectorization, memory optimizations and data alignment, and also investigates some basic performance issues of the entire program.

3.2.1 Basic Optimization

There are lots of basic optimization methods which can be done for any programs such as constant folding, redundancy elimination, copy propagation and constant propagation.

Constant folding means replacing variables with constants at compile time to reduce memory access. The value can be stored in instruction. Thus, the author changes the enum variable Ndim to constant 3 during the compiling time rather than accessing enum variable every time at the running time.

```
#define Ndim 3
```

In addition, there are two examples referring to temporal locality and spatial locality optimizations as shown below:
The one is part of setting the viscosity term and the wind term. When looking into the visc_force() and wind_force(), these two functions to calculate the f[j] in serial. The author combines two independent parts in one part to reduce the number of references required and increase options for code scheduler and moreover for optimal cache use.

```
/* set the viscosity term in the force calculation */
        for(j=0;j<Ndim;j++){
          visc_force(Nbody,f[j],vis,velo[j]);
        }
 /* add the wind term in the force calculation */
        for(j=0;j<Ndim;j++){
          wind_force(Nbody,f[j],vis,wind[j]);
        }
```

Change to:

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

The advantage of this optimization is that all computation of f[j] can be finished in one time without caching f[j] many times, which means this change is beneficial to cache temporal locality.
Another example is calculating central force.

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

Change to:

```
         for (l = 0; l < Ndim; l++) {
                    for (i = 0; i < Nbody; i++) {
                        f[l][i] = f[l][i] - force(G*mass[i] * M_central, pos[l][i], r[i]);
                    }
                }
```

Loop order is important for exploiting spatial locality in caches and C are laid out by rows. Thus, swapping l loop and j loop can make the memory access more optimal, which means has a good spatial locality and in order.

3.2.2 Data Structure

```
double *pos[Ndim], *velo[Ndim];
double *f[Ndim], *vis, *mass, *radius;
double *delta_pos[3];
double *r;
double *delta_r;
double wind[Ndim];

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

The calloc() allocates the memory dynamically and then initializes their value to zero. However, the original program doesn' t deallocate memory by using free(), which is a potential problem of memory leak, because they are all global variables which are allocated on the heap so that the programmer must allocate and free them manual.
The advantage of calloc() dynamical array is flexible and can be determined during the running period because C requires array dimensions to be known during the compiling period and cannot change size afterwards. In addition, these variables are all used double-precise float points to avoid alignment problem in some degree during computation. If using calloc() to allocate memory, there is a compiler flag -qopt-calloc to increase the performance of long-running programs which use calloc() frequently. According to Intel description, they recommend to combine -inline-calloc and -qopt-malloc-options=3 to reduce memory fragmentation.

However, using these pointers allocates the two-dimensional elements but can not guarantee each the first dimensional pointers to be continuous. Take delta_pos [3] as an example, the program just applies for three pointers and then uses these pointers to allocate a continuous memory respectively, which means the entire two-dimensional pointer array is not continuous from the head to the end and need multiple memory accesses per element access. This method will add the overhead of addressing and accessing. In addition, dynamic memory structures will also have some other problems. For example, it will consume more memory and cause a higher ratio of cache misses. Moreover, pointers have the possibility to inhibit compiler optimizations so that it is hard to vectorise. The reason is that the compiler can not specify where the data should be before running. Thus, this is an optimization point for performance.

In the author' s opinion, a possible method is that they will be changed to static multi-dimensional arrays in order to get better performance. Thus, the author uses static multi-dimensional arrays instead of dynamical arrays, because related variables of static arrays are stored close together may help cache use and moreover, dynamical arrays are not continuous from the aspect of the first dimension. The most important reason is that when the author allocates the array statically, the compiler can specify where the data should be within the memory.

```
double pos[Ndim][Nbody], velo[Ndim][Nbody];
double f[Ndim][Nbody], vis[Nbody], mass[Nbody], radius[Nbody];
double r[Nbody];
double delta_r[Nbody][Nbody];
double delta_pos[Ndim][Ndim*Nbody*Nbody];
double wind[Ndim];
```

	100 iterations	500 iterations
global dynamical variables(-o3 only)	28.20s	140.60s
global dynamical variables(-o3 -ipo -no-prec-div)	16.12s	80.69s
global static variables(-o3 -ipo -no-prec-div)	15.25s	76.76s

Tablet 2: comparison of dynamical variables and static variables

So far, 76.76s is the best result that the author can achieve.

However, these variable are also global variables and allocated on the heap. The compiler is more effective with local variables. Thus, the possible further optimization point of these variables is to modify these global variables as local variables by the called procedure. This coursework mainly focuses on MD.c file. Changing to local variables will be future work.

3.2.3 Code block 1: Calculate Pairwise Separation of Particles

This part accounts for 6.5% of 98% evolve().

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

In the code block 1, the first pos [l][i] is going through the full pos array and the second pos [l][j] is going through most of the pos array, which means the pos array is being loaded from memory many times during the simulation.

In addition, there is an iterated counter k of delta_pos [l][k] in this loop, which means the compiler cannot vectorize the delta_pos [l][k] easily, because each k and k+1 are dependent. It is similar to the Code Block 3, because k is also an iterated counter for delta_r [k]. The recommendation is to use calculated values (based on loop index) rather than iterated counters. Thus, the k should be replaced by index counters i, j.

The first trial is to use i, j to replace k to get better optimal memory access:

```
k = i*(2Nbody - i - 1)/2 +j -i
```

However, the author finds this optimization adds more calculation of k. Thus, the author decides to change the memory structure to eliminate k totally.

According to an analysis of the entire program, k is only for delta_pos and delta_r as an iterated counter. Thus, these variables memory structure are important. In this part, the author finds that delta_pos[l][k] can be replaced by pos[l][i] - pos[l][j] directly without iterated counter k in the following computation.

3.2.4 Code Block 2: Calculate Norm of Separation Vector

The code block 2 accounts for 7.5% and inner add_norm() accounts for 4.2%.

```
/* calculate norm of separation vector */
        for(k=0;k<Npair;k++){
          delta_r[k] = 0.0;
        }
        for(i=0;i<Ndim;i++){
         add_norm(Npair,delta_r,delta_pos[i]);
        }
        for(k=0;k<Npair;k++){
          delta_r[k] = sqrt(delta_r[k]);
        }
```

In the code block 2, sqrt() cannot pipeline usually but other instructions can be issued during the sqrt() progresses. Thus, the author has to find instructions that don’ t require the sqrt() unit and put instructions between sqrt() issue and use of result to help compiler optimization. Fusing three loops together could be an optimization method. The author analyses all three loops of code block 2 and finds that they are calculating the distance between two points repetitively so that applying Pythagorean theorem to fuse three loops, because the distance of sqrt(delta_r[k]) between two points is also appropriate to multipoint.

All three loops can be expanded inner function and fused by adjusting loop sequences.

```
for (k = 0; k < Npair; k++){
    delta_r[k] = 0.0;
    for (i = 0; i < Ndim; i++){
        delta_r[k] = delta_r[k] + (delta_pos[i][k] * delta_pos[i][k]);
    }
    delta_r[k] = sqrt(delta_r[k]);
}
```

	500 iterations
Before ‘FUSE’ optimization	76.76s
After ‘FUSE’ optimization	55.18s

Tablet 3: Combination of The Code Block 2

According to analysis, the author finds that the total number of i loop and j loop equals Npair. Thus, the author considers changing delta_r[k] memory structure to the two-dimensional array to remove iterated counter k to be suitable for the Code Block 1 and the Code Block 3.

Then change memory structure:
`double *delta_r <=> double delta_r[Nbody][Nbody].`
`delta_pos[l][k] <=> pos[l][i] - pos[l][j],` which has been described in section 3.2.3.

Thus, the current code combining the Code Block 1 and Code Block 2 is optimized to:

```
for(i=0;i<Nbody;i++){
for(j=i+1;j<Nbody;j++){
    delta_r[i][j] = 0.0;
    for ( l = 0; l < Ndim; l++){
        delta_r[i][j] = delta_r[i][j] + (pos[l][i] - pos[l][j])*(pos[l][i] - pos[l][j]);
            }
        delta_r[i][j] = sqrt(delta_r[i][j]);
    }
}
```

Finally, as shown before in section 3.22 Data Structure, the compiler is more effective with local variables, and it is recommended to use local variables. Thus, adding one more local temporary variable to help the compiler to reduce the overhead of accessing a global variable.

```
double tmp = 0
   for ( i = 0; i < Nbody; i++){
       for ( j = i+1; j < Nbody; j++){
           tmp = 0;
           delta_r[i][j] = 0;
           for ( l = 0; l < Ndim; l++){
               tmp = tmp + (pos[l][i] - pos[l][j])*(pos[l][i] - pos[l][j]);
           }
           delta_r[i][j] = sqrt(tmp);
       }
   }
```

	500 iterations
After 'FUSE' optimization	55.18s
Change memory structure and add temporary variables	40.28s

Tablet 4: Further Modification of The Code Block 2

According to tablet3, as expected, changing delta_r[][] structure and adding the local temporary variable have a huge performance improvement. In addition, this part has been shown in the -qopt-report=5 report, pos[l][j] and delta_r[i][j] has aligned access and the loop has been vectorized.

3.2.5 Code Block3: Add Pairwise Forces

```
/* add pairwise forces */
       k = 0;
       for(i=0;i<Nbody;i++){
         for(j=i+1;j<Nbody;j++){
           ..........
           for(l=0;l<Ndim;l++){
             if( delta_r[k] >= Size ){
               f[l][i] = f[l][i] -
                  force(G*mass[i]*mass[j],delta_pos[l][k],delta_r[k]);
            ..........
        collided=1;
              }
           }
       if( collided == 1 ){
         collisions++;
       }
           k = k + 1;
         }
       }
```

In the code block 3, this is the most workload accounting for around 85%. According to analysis, the most obvious performance bottleneck is that force () is called too many times to do the same computation, which means it has repetitive computation. In addition, there are also if () statements in the loops to disrupt vectorization.

The author also looks into force() function as shown below and finds there are not lots of things which can be done. This is only a very small function. Thus, inlining this function could be better for speed by increasing the code size. The author still does it manually without using compiler flag during the optimization period, although inlining can be done by the compiler automatically and adding -ipo compiler flag. The more details of -ipo will be explained in section 3.2.6.

```
        double force(double W, double delta, double r){
          return W*delta/(pow(r,3.0));
        }
```

Thus, the first job is to expand force() and only calculate it one time. In addition, changing the memory structures as shown in the section 3.2.3 and 3.2.4:

```
        delta_pos[l][k] <=> (pos[l][i] - pos[l][j])*(pos[l][i] - pos[l][j])
        delta_r[k] <=> delta_r[i][j]
```

Thus, the optimization result is that:

```
forceResult =
G*mass[i] * mass[j] * (pos[l][i] - pos[l][j])*(pos[l][i] - pos[l][j])/ delta_r[i][j]* ...... *delta_r[i][j]
```

The author tries to remove if() statement outside by splitting two if() branches in two parts with the same loops index. The purpose is to remove the first if() branch, but it will cause wrong results because of the program algorithm. Thus, the author gave up removing if() statement outside during the coursework. This part will be future work.

The author then analyses hotspots again and finds that the amount of computation only exist in the first branch when delta_r[k] > Size as shown in Figure 3. Thus, the author decides to let the program calculate the first most workload branch first and only check the second branch when it is necessary. The method is to change two branches as one branch.

```
            for (l = 0; l < Ndim; l++){
                forceResult = ......
                f[l][i] = f[l][i] - forceresult;
                f[l][j] = f[l][j] + forceresult;
                if (delta_r[i][j] < Size){
                    f[l][i] = f[l][i] + 2 * forceresult;
                    f[l][j] = f[l][j] - 2 * forceresult;
                    }
                }
```

There is one small optimization point: pow(delta_r[i][j], 3) has been expanded and moved to the Code Block 2 to reduce the number of references required and increase for optimal cache use. That’s the reason why tmp variable is multiplied three times.

In addition, the author modifies the logic of assigning collisions variable and eliminate judgement of the collided variable so that the logic of this part becomes clear, which increase the readability of program in some degree.

```
                for (i = 0; i < Nbody; i++){
                    for (j = i + 1; j < Nbody; j++){
                        Size = radius[i] + radius[j];
                        for (l = 0; l < Ndim; l++){......}
                        if (delta_r[i][j] < Size){......}
                    }
                    if (delta_r[i][j] < Size) { collisions++; }
                }
            }
```

	500 iterations
The best result before optimization	40.28s
The best result after optimization of the Code Block 3	30.20s

Tablet 5: Further Modification of The Code Block 3

3.2.6 Compilers Flags

As shown before paragraphs, programmers have various ways to optimize code to help compiler optimize the program, but the easier way is to use suitable compiler flags correctly. The author uses the Intel compiler 17.0.2 on Cirrus to optimize the program. The author checks the documentation carefully and uses these flags sparingly. Moreover, the author measures their time to determine whether keeping them for the program. The following compiler flag will be used in the program and the result will be shown in section 4.

-O3

The compiler flags of -O2 and -O3 have a noticeable influence on the program. The O2 and O3 option are both for code speed. According to Intel’s website, O2 option is the recommended optimization level, but the Intel claims that O3 option is better for the program which has loops that heavily use floating-point calculations and process large data sets. Thus, the author will test -O2 and -O3.

-ipo

This compiler flag is interprocedural optimization (IPO) and used to enable interprocedural optimization between files. This flag tells the compiler to perform inline function expansion for calls to functions defined in separate files. According to Intel introduction, this compiler flag can do lots of basic optimizations such as dead code elimination, constant propagation and procedure reordering. Moreover, it can enhance optimization when combining with other compiler features.

The author finds that this compiler flag is powerful. For example, the separate util.c file provides some functions for MD.c. These util.c functions are not long but called many times so that inlining should be used here to increase body size to reduce the overhead of functions call. Thus, the author didn’t use this compiler flag at the beginning and move the calls of util.c functions outside to calculate directly in the most workload places.

However, according to experiments, the program achieves a similar performance compared with using -ipo and even -ipo can get better performance, because there are some places that the author has not realized. Thus, although inline function expansion can be done by hands, such as wind_force() and visc_force(), it is still worthy to use this compiler flag to make codes cleaner and help some potential basic optimization points.

Compiler Flags	*-O0*	*-O1*	*-O2*	*-O3*
without -ipo	*327.11s*	*64.98s*	*36.3s*	*33.40s*
with -ipo	*328.1s*	*60.70s*	*35.68s*	*33.37s*

Tablet 6: Comparison of -ipo

-xCORE-AVX2

This compiler flag can enable processor-specific optimization. Hardware configuration is a key point to improve performance. Especially, Intel(R) Xeon(R) E5-2695 v4 supports Instruction Set Extensions: Intel AVX2. However, according to Intel documentation, the generated executable will neither run on non-Intel processors nor Intel processors that do not support AVX2, which means this compiler flag reduce portability. In addition, according to tests, this compiler flag does not improve performance. Conversely, it increases running time slightly. Thus, the author does not recommend using this flag but treat it as a potential optimization flag for the future.

Compiler Flags	500 iterations
-O3 -ipo -no-prec-div	30.2s
-O3 -ipo -no-prec-div -xCORE-AVX2	31.35s

Tablet 7: Comparison of -xCORE-AVX2

-no-prec-div
-no-prec-sqrt

This compiler flag is used to improve the precision of floating-point divided and square root implementations which have a slight impact on speed. Thus, the author will use -no-prec-sqrt and -no-prec-div to reduce the loss of performance. However, correctness is an important consideration for this program because the result of this program could be affected by optimization activity slightly. And the provided test tool can not detect the presence of NaN values in the input. According to tests, they both can improve performance slightly, but the author recommends only use -no-prec-div flag and ***does not use -no-prec-sqrt for the accuracy of the program.***

Compiler Flags	500 iterations
-03	33.4s
-O3 -ipo	33.37s
-O3 -ipo -no-prec-div	30.20s
-O3 -ipo -no-prec-div -no-prec-sqrt	30.07s

Tablet 8: Comparison of -no-prec-div and -no-prec-sqrt

-qopt-report=5

This compiler flag will output an optimization report in terms of loop nest, vector and auto-parallelization optimizations. After optimization, the feedback of the Code Block 2 report describes that pos[l][i] and delta_r[i][j] has aligned access and it has been vectorized.

```
remark #15388: vectorization support: reference pos[l][j] has aligned access   [ MD.c(116,55) ]
remark #15388: vectorization support: reference delta_r[i][j] has aligned access   [ MD.c(118,5) ]
  ......
remark #15300: LOOP WAS VECTORIZED
```

Thus, the author decides that the final compiler flag should be: -O3 -ipo -no-prec-div for this program.

4. Results

The author has compared performance improvements in different parts as shown before. This section focus on the overall performance compared with the original code.

4.1 Result

The author analyses all performance improvement of all optimization activities, which is shown below.

Optimization Activities	500 iterations
Originl Makfile	5195s
-O0 Only	1014.90s
-O1 Only	671.30s
-O2 Only	138.30s
-O3 Only	140.60s
Original code (-O3 -ipo -no-prec-div)	80.69s
Change to global static variables (-O3 -ipo -no-prec-div)	76.76s
Fused the part of sqrt() (-O3 -ipo -no-prec-div)	55.18s
Change memory structure and add temporary variables (-O3 -ipo -no-prec-div)	40.28s
modify the Code Block 3 (-O3 -ipo -no-prec-div)	30.20s

Tablet 9: Improvement of Optimization Activities

Finally, the program gets the following result:

Original Program	The Best Optimized Version	Speedup
5195s	30.2s	172 times

Tablet 10: Final Result

4.2 Correctness

The whole program always uses the final collisions to check the correctness of the program. The author thinks that if the result of collisions equals 68540, the program is correct. In addition, all optimization activities can meet the requirements of that verification tool.

4.3 Maintainability and Readability

The program mainly focuses on optimizing memory structure and increasing vectorization about three most workload part and do not change the algorithm.

The optimization of the Code Block 3 simplifies the condition statement and the logic of assigning of collisions variable so that it increases readability and maintainability of the program to some degree.

5. Conclusion and Future Work

5.1 Future Work

As shown in the before, there are some future work which can be done. The first one is trying to change all global variables to local variables and transferred them by procedure or pointers.

In addition, removing if() branch outside is another future work because this part is related to the algorithm and not very easy.

The most important future work is to vectorize f[][] array of the Code Block 3 to be able to calculate each dimension at the same loop, which means to eliminate for(l=0; l<3; l++) this loop. The author thinks that this vectorization can make a huge contribution to performance.

The future code could be similar to the following code:

```
for (l = 0; l < Ndim; l++){
    f[l][i] = f[l][i] - forceresult;
    f[l][j] = f[l][j] + forceresult;
}
```

Change to

```
    f[0][i] = f[0][i] - forceresult0;
    f[0][i] = f[0][i] + forceresult0;
    f[1][i] = f[1][i] - forceresult1;
    f[1][i] = f[1][i] + forceresult1;
    f[2][i] = f[2][i] - forceresult2;
    f[2][i] = f[2][i] + forceresult2;
```

5.2 Conclusion

During this optimization process, the author mainly focuses on loops optimization in terms of the memory structure, vectorization and compilers, because they are key points to the improvement of performance for loops. The author spends time on the most workload parts and always follows the optimization plan as shown in the section 2.2.

Finally, the entire optimization process of this program gets the ideal result and the author learns a lot.

References

1. Corden, Martyn. ntel® Compiler Options for Intel® SSE and Intel® AVX generation (SSE2, SSE3, SSSE3, ATOM_SSSE3, SSE4.1, SSE4.2, ATOM_SSE4.2, AVX, AVX2, AVX-512) and processor-specific optimizations. [Online] Intel, January 24, 2010. [Cited: April 2, 2019.] https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations.

2. Le, Thai. How Intel® AVX2 Improves Performance on Server Applications. [Online] Intel, September 5, 2014. [Cited: April 1, 2019.] https://software.intel.com/en-us/articles/how-intel-avx2-improves-performance-on-server-applications.

3. Intel® C++ Compiler 19.0 Developer Guide and Reference. [Online] [Cited: April 4, 2019.] https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-qopt-calloc.