Optimizing Single-Threaded Molecular Dynamics Simulation for Argon Gas

Eduardo Figueiredo
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52679@uminho.pt

Gonçalo Senra
MEI: Parallel Computing
Universidade do Minho
Barcelos, Portugal
Email: pg52683@uminho.pt

***Abstract*—This article discusses optimization techniques applied to a single-stream molecular dynamics simulation program for argon gas. The goal of this work assignment is to improve program performance and reduce execution time. We are exploring code analysis and profiling tools to achieve this goal. This report summarizes our findings, methods, and results.**

I. Introduction

The program we analyzed and optimized is a simple molecular dynamics simulation code for argon gas. The original version of the code is available in the Foley Lab/Molecular Dynamics, specifically in the "Simple Molecular Dynamics" section. In this report, we will detail the optimization techniques used to improve its performance and reduce execution time.

***Index Terms*—Optimizing, Instructions, Execution time, Performance, Loops, Code, Potential, Accelerations, Power operations, Flags, Vectorize**

II. Optimizing the Molecular Dynamics Simulation Code

This section discusses various strategies for optimizing argon gas's molecular dynamics simulation code.

A. Abbreviations and Acronyms

a) SSE4: Streaming SIMD Extensions 4
b) AVX: Advanced Vector Extension

B. Optimizing the Potential Calculation

The original code of the Potential() function calculates the potential energy in a system of interacting particles via the Lennard-Jones potential. The optimizations we made, which can be seen in Figures 1 and 2, are as follows: We eliminated the loop named "k", which traverses three columns of table "r". This way we have reduced the number of instructions. In loop 'j' we notice that there are repeated calculations. In other words, the square of the difference between "r[i]" and "r[j]" is the same as the square of the difference between "r[j]" and "r[i]. We simplified the code by considering all pairs of 'i' and 'j' and then multiplying the result by 2. To reduce cache misses, we stored the values we accessed in the 'i' loop in variables so that we could use them in the second loop. To minimize the number of instructions, we eliminated a calculation that was performed N*N times and was equal to 4 times epsilon. Instead, we moved this calculation outside of the loops. These optimizations were made to improve the efficiency and performance of the code.

C. Optimizing the Compute Accelerations

After the pow() functions removal, the second target of optimization was the deletion of the loops named 'k' (as we can see in Figures 3 and 4), which were causing a significant increase of instructions. Beyond the optimization mentioned before, in every single iteration of the loop named 'i' we stored the matrix line 'i' in temporary variables ('ri0', 'ri1', 'ri2'), so that the program could avoid cache misses and consequently have to fetch the values from memory.

D. Combining Potential Calculation and Accelerations

In the original code, the calculation of the potential energy and the calculation of the particle acceleration are performed in separate steps. To optimize the code, we created a function that combines functions optimized for potential energy and acceleration(Figure 5). Taking advantage of the fact that these functions are identical regarding loops and memory array accesses, we combined them into a single function, thus reducing the number of instructions and memory accesses.

E. Optimizing the Power Operations

The original program involved several power operations that consumed a significant part of the execution time (Figure 6), due to the computational cost of lookup operations. Because of that, we decided to replace all the pow() calls, (except the numbers that were powered to 1/n), with basic multiplications. This replacement reduced the number of instructions, and the execution time decreased significantly.

F. Eliminating Unnecessary Loops

While optimizing the code, we discovered unnecessary loops that were negatively affecting performance. These loops, named "k", are abundant in native code. Since 'k' loops only have a maximum range of 3, we chose to remove them and replace them with repeated instructions. This adjustment helped reduce the total number of instructions and improve efficiency. Finally, in the 'initializeVelocities()' function, we

merged some loops, further reducing the number of instructions.

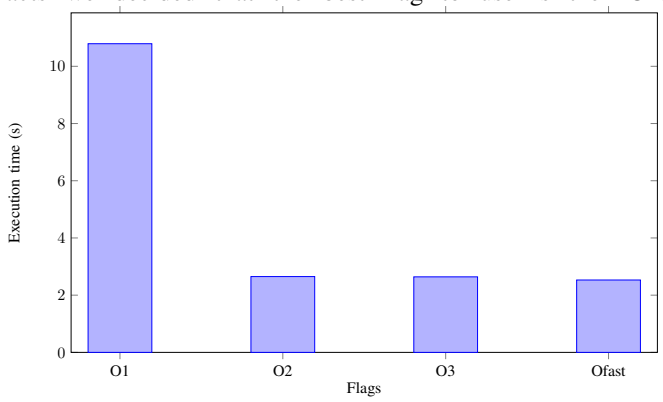
III. Analysis of Optimizations

A. Flags

a) -pg, -g: These flags enable debugging information in the executable file, the -pg flag adds the execution time of each function

b) -fno-omit-frame-pointer: When enabled, the compiler keeps the frame pointer in the call stack, making debugging and tracking function calls easier. This is useful for accuracy and visibility in the call stack. Therefore, when using this option, the compiler does not optimize away the frame pointer, making it visible and accessible for debugging and analysis purposes.

c) -O2: This flag represents a moderate level of optimization, we chose this level of optimization due to the great balance between execution time and compilation time. Additionally, we tried the -O3 and -Ofast flags, but we didn't get a much better execution time and the compilation time increases, if we consider that these flags use much more aggressive optimizations than the -O2 flag, (which normally leads to bigger binary files), with all these facts we decided that the best flag to use is the -O2.



d) -ftree-vectorize, -msse4, -mavx: The flag -ftree-vectorize enables tree vectorization, this means that the compiler will attempt to generate instructions to process multiple data elements in parallel, the -msse4 flag enables support for SSE4 instructions, which are used to accelerate parallel data operations. On the other hand, the -mavx flag enables support for AVX instructions, providing additional resources for the parallel processing of data vectors, a feature commonly used throughout the code. When used together, these flags deliver optimized performance on the processor, improving overall code efficiency.

IV. Conclusion

2*Code Type	Performance Metrics	
	Execution Time	Number of Instructions
Original Code	85.05 seconds	730652174066 instructions
Optimized Code	2.64 seconds	17933299277 instructions

TABLE I
Comparison of Original and Optimized Code

By running the code on our machines, we were able to optimize the execution time by approximately 32.22 times and reduce the number of instructions to about 40.74 times compared to the original code, using a more efficient programming language.

In summary, the exhaustive search for patterns in the code to discard unnecessary loops, and parallelization were the best ways to optimize this single-threaded program. In the future, we could turn this program multi-threaded so we can obtain an even better execution time.

References

[1] Computer Organization and Design: The Hardware/Software Interface, David Patterson and John Hennessy, 5th Ed., Morgan Kaufmann, 2013
[2] Course slides

V. ANNEXES

```c
double Potential() {
    double quot, r2, rnorm, term1, term2, Pot;
    int i, j, k;
    Pot=0.;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (j!=i) {
                r2=0.;
                for (k=0; k<3; k++) {
                    r2 += (r[i][k]-r[j][k])*(r[i][k]-r[j][k]);
                }
                rnorm=sqrt(r2);
                quot=sigma/rnorm;
                term1 = pow(quot,12.);
                term2 = pow(quot,6.);

                Pot += 4*epsilon*(term1 - term2);
            }
        }
    }
    return Pot;
}
```

Fig. 1. Original function Potential()

```c
double Potential(){
    double quot, r2, rnorm, Pot, term3, term6, term12, aux, r0i, r1i, r2i,term1,term2;
    int i, j, k;
    double var = 8 * epsilon; //(2 * (4 * epsilon));
    Pot = 0.;
    for (i = 0; i < N; i++){
        r0i = r[i][0];
        r1i = r[i][1];
        r2i = r[i][2];
        for (j = i + 1; j < N; j++){
            r2 = 0.;
            aux = r0i - r[j][0];
            r2 += aux * aux;
            aux = r1i - r[j][1];
            r2 += aux * aux;
            aux = r2i - r[j][2];
            r2 += aux * aux;
            rnorm = sqrt(r2);
            quot = sigma / rnorm;
            term3 = quot * quot * quot;
            term6 = term3 * term3;
            term12 = term6 * term6;
            Pot += term12 - term6;
        }
    }
    Pot = Pot * var;
    return Pot;
}
```

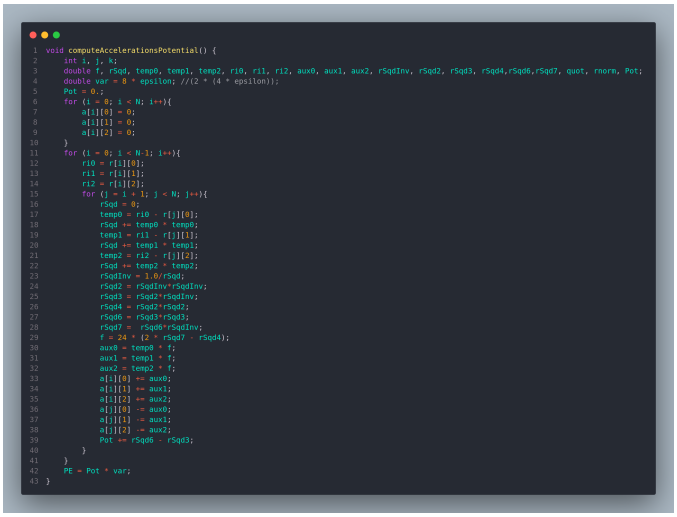
Fig. 2. Optimized function Potential()

```c
void computeAccelerations() {
    int i, j, k;
    double f, rSqd;
    double rij[3]; // position of i relative to j
    for (i = 0; i < N; i++) {  // set all accelerations to zero
        for (k = 0; k < 3; k++) {
            a[i][k] = 0;
        }
    }
    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {
            rSqd = 0;
            for (k = 0; k < 3; k++) {
                //  component-by-componenent position of i relative to j
                rij[k] = r[i][k] - r[j][k];
                //  sum of squares of the components
                rSqd += rij[k] * rij[k];
            }
            f = 24 * (2 * pow(rSqd, -7) - pow(rSqd, -4));
            for (k = 0; k < 3; k++) {
                //  from F = ma, where m = 1 in natural units!
                a[i][k] += rij[k] * f;
                a[j][k] -= rij[k] * f;
            }
        }
    }
}
```

Fig. 3. Original function computeAccelerations()

```c
void computeAccelerations(){
    int i, j, k;
    double f, rSqd, temp0, temp1, temp2, ri0, ri1, ri2, aux0,aux1,aux2,rSqdInv,rSqd2,rSqd4,rSqd7;
    for (i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    for (i = 0; i < N - 1; i++){
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = i + 1; j < N; j++){
            rSqd = 0;
            temp0 = ri0 - r[j][0];
            rSqd += temp0 * temp0;
            temp1 = ri1 - r[j][1];
            rSqd += temp1 * temp1;
            temp2 = ri2 - r[j][2];
            rSqd += temp2 * temp2;
            rSqdInv = 1.0/rSqd;
            rSqd2 = rSqdInv*rSqdInv;
            rSqd4 = rSqd2*rSqd2;
            rSqd7 =  rSqd4*rSqd2*rSqdInv;
            f = 24 * (2 * rSqd7 - rSqd4);
            aux0 = temp0 * f;
            aux1 = temp1 * f;
            aux2 = temp2 * f;
            a[i][0] += aux0;
            a[i][1] += aux1;
            a[i][2] += aux2;
            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
        }
    }
}
```

Fig. 4. Optimized function computeAccelerations()



```
void computeAccelerationsPotential() {
    int i, j, k;
    double f, rSqd, temp0, temp1, temp2, ri0, ri1, ri2, aux0, aux1, aux2, rSqdInv, rSqd2, rSqd3, rSqd4,rSqd6,rSqd7, quot, rnorm, Pot;
    double var = 8 * epsilon; //(2 * (4 * epsilon));
    Pot = 0.;
    for (i = 0; i < N; i++){
        a[i][0] = 0;
        a[i][1] = 0;
        a[i][2] = 0;
    }
    for (i = 0; i < N-1; i++){
        ri0 = r[i][0];
        ri1 = r[i][1];
        ri2 = r[i][2];
        for (j = i + 1; j < N; j++){
            rSqd = 0;
            temp0 = ri0 - r[j][0];
            rSqd += temp0 * temp0;
            temp1 = ri1 - r[j][1];
            rSqd += temp1 * temp1;
            temp2 = ri2 - r[j][2];
            rSqd += temp2 * temp2;
            rSqdInv = 1.0/rSqd;
            rSqd2 = rSqdInv*rSqdInv;
            rSqd3 = rSqd2*rSqdInv;
            rSqd4 = rSqd2*rSqd2;
            rSqd6 = rSqd3*rSqd3;
            rSqd7 =  rSqd6*rSqdInv;
            f = 24 * (2 * rSqd7 - rSqd4);
            aux0 = temp0 * f;
            aux1 = temp1 * f;
            aux2 = temp2 * f;
            a[i][0] += aux0;
            a[i][1] += aux1;
            a[i][2] += aux2;
            a[j][0] -= aux0;
            a[j][1] -= aux1;
            a[j][2] -= aux2;
            Pot += rSqd6 - rSqd3;
        }
    }
    PE = Pot * var;
}
```

Fig. 5. Function computeAccelerationsPotential()

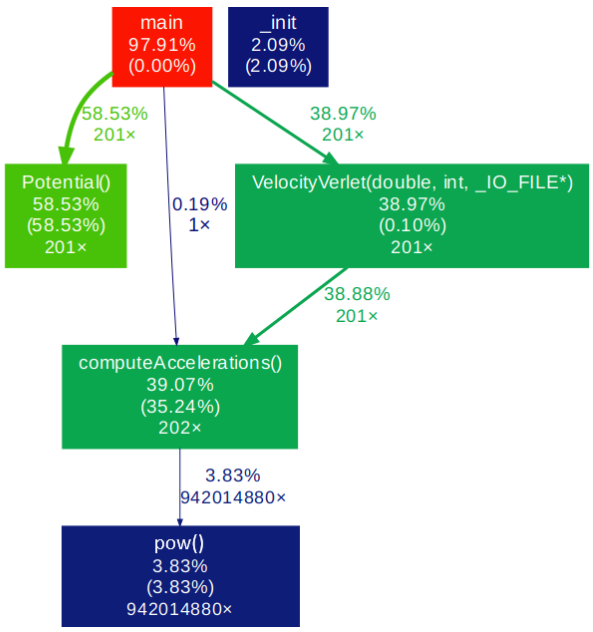


Fig. 6. Original call-graph

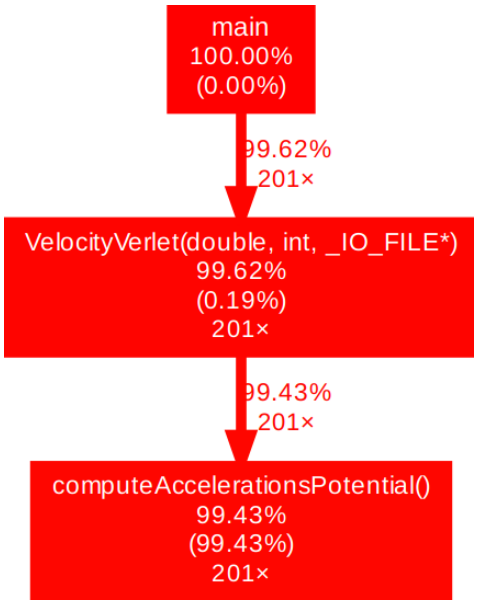


Fig. 7. Optimized call-graph