Optimizing Molecular Dynamics (MD) Simulation

Afonso Franco
Universidade do Minho
Braga, Portugal
pg53595@alunos.uminho.pt

Alice Teixeira
Universidade do Minho
Braga, Portugal
pg52670@alunos.uminho.pt

***Abstract*— The virtual simulation of physics can often times be a time-consuming, resource-heavy procedure. In this paper, we apply several optimization techniques in order to enhance the performance of a molecular dynamics simulation, showcasing the advantages of their use and the results obtained.**

***Index terms*—optimization, programming, strenght-reduction, loop-invariant code motion, molecular dynamics**

I. Introduction

In some scientific fields, the simulation of entities' behaviours is useful since it provides researchers a visual resource which they may control, facilitating their task of correlating input data and effects produced. Moreover, the more complex such simulations progressively become, the more resource-costly and time-intensive they are potentially rendered. This way, it is crucial that their development is over viewed under a resource-aware lens. The application of optimization techniques proposes the preservation of essential code while making use of more efficient instructions, minimizing the strain on available resources.

A. Paper overview

In this paper we propose an alternate version of the c program provided within the course of Computação Paralela, taught at University of Minho, which represents a molecular dynamic's simulation of Argon atoms.

The original code is part of FoleyLab/MolecularDynamics: Simple Molecular Dynamics, which introduces a simulation of particle movements over time by calculating the force, accelaration and position of the atoms. The calculations are based on the classical understanding of mechanics and Newton's laws of movemement, in order to compute thermodynamic properties of materials.

The optimized version of the code we present aims to minimize the time and resources consumed in the rendering of this aforementioned simulation. Furthermore, the techniques applied are presented in the following sections of this paper, alongside an explanation on how they better its performance.

II. Performance before optimizing code

Before starting to optimize our code, we need to first have a baseline of how the code runs without any optimizations.

A. Baseline compiler flags

For our baseline run we used no compiler optimization flags (as provided by the professors).

```
 Performance counter stats for 'MD-original.exe':

 1,243,860,205,351      inst_retired.any
                        # 0.65 CPI
   812,320,853,369      cycles

     279.583571300 seconds time elapsed
```

As you can see, the unoptimized code runs in just under 5 minutes with a CPI of 0.65.

III. Compiler flags

This obviously isn't optimal, so we applied the following optimization flags:

```
-march=native -mtune=native -mavx -O2 -ftree-
vectorize
```

These flags tell the compiler to try to vectorize the code and optimize according to the available CPU's architecture and capabilities.

Using the above compiler flags, we reduced the original code to 3.5 minutes.

```
 Performance counter stats for 'MD-original.exe':

   986,829,986,961      inst_retired.any
                        # 0.66 CPI
   654,476,484,890      cycles

     215.458661315 seconds time elapsed
```

IV. Optimization techniques

Now that we had chosen compiler flags that make sense to our program, we started optimizing the code itself. We used gprof, perf and valgrind to see where the hotspots of our code were, and found out that more than 99% of the execution time of our code was in the “Potential” and “computeAccelerations” functions.

A. Minimizing Repetitive Calculations

In the original code, separate loops for potential energy and particle acceleration used common variables, causing computational overhead. To improve efficiency, we consolidated these calculations into a single function, “PotentialAndAcceleration,” reusing intermediate results. Additionally, we optimized loop iterations to avoid redundant potential calculations for both directions between particles, boosting performance.

```
for i: 0 -> N
  for j: 0 -> N
    //Perform calculations
```

To the more efficient form:

```
for i: 0 -> N-1
  for j: i+1 -> N
    //Perform calculations
```

we have made it so that **j** is always in front of **i**, reducing the number of iterations from

$$N * N \tag{1}$$

to the much smaller:

$$\frac{N * (N - 1)}{2} \tag{2}$$

Then the function only does calculations between points once for each pair of points, and adds the value doubled to out total potential.

Additionally, this modification allowed us to eliminate the need for a check to verify if “i” and “j” are the same since they were now guaranteed to be distinct. This streamlining completely removes N*N branch predictions and further contributes to a reduction in the number of instructions and branch misses.

To enhance efficiency even further and reduce redundant memory accesses and dependencies, we have reorganized the code. Specifically, we moved the sections of code that depend solely on the “i” variable outside the “j” loop. This modification prevents the need to fetch these values during each iteration of “j” since they remain constant until the next iteration of the “i” loop. This optimization minimizes unnecessary data retrieval.

B. Removing pow, sqrt, and Reducing Arithmetic Operations

Since the **pow** function is more expensive than manually multiplying, we decided to turn all calls to that function into a series of multiplications. Furthermore, we decided to simplify the expressions to minimize the ammount of multiplications and divisions. Square roots can also be removed since they are canceled by each other (The value inside is absolute).

Starting with this:

$$\left(\frac{\sigma}{\sqrt{r^2}}\right)^{12} - \left(\frac{\sigma}{\sqrt{r^2}}\right)^{6} \tag{3}$$

Distributing the outer powers we can get rid of the square roots:

$$\frac{\sigma^{12}}{r^6} - \frac{\sigma^6}{r^6} \tag{4}$$

Expanding powers:

$$\frac{\sigma * \sigma * \ldots}{r * r * \ldots} - \frac{\sigma * \sigma * \ldots}{r * r * \ldots} \tag{5}$$

Then, we simplify the multiplications down by using intermidiate variables:

$$\text{quot} = \frac{\sigma * \sigma}{\text{r2}} \tag{6}$$

$$\text{term2} = \text{quot} * \text{quot} * \text{quot} \tag{7}$$

$$\text{finalValue} = \text{term2} * (\text{term2} - 1.) \tag{8}$$

Now we only have 4 multiplications and one division.

We then apply this same principle to the other arithmetic heavy areas of our code.

V. Results

After all these changes, and using the previoulsy mentioned gcc flags, this was our best run.

```
 Performance counter stats for '/home/pg53595/MDS/
MD.exe':

    19,357,701,649      inst_retired.any
                        #  0.7 CPI
    14,293,931,277      cycles

       4.471249537 seconds time elapsed
```

We reduced the number of instructions by more than **6000%** and our number of cycles by a little over **5500%**.

The time to run our program went down by 6500% from around **5 minutes** to just under **4.5 seconds**.

VI. Conclusion

Our study focused on optimizing resource-intensive molecular dynamics (MD) simulations. By employing advanced optimization techniques, we significantly enhanced our MD simulation code’s efficiency. Initially, compiler optimization flags improved performance. Subsequent refinements, including the elimination of redundant calculations and simplification of arithmetic operations, drastically reduced instructions and cycles. This led to an impressive 6500% decrease in execution time, with instructions and cycles reduced by over 6000% and 5500%, respectively. Our research highlights the vital role of code optimization in physics simulations for more resource-aware scientific research.