Uppsala University

High Performance Programming

Assignment 3

Serial Code Optimization

Author:
Olle Virding

February 7, 2023

Contents

1 Introduction

The purpose of this report is to evaluate the efficiency of different methods for *serial code optimization* and analyze time of execution.

The program written in this assignment simulates a constellation of stars or particles in a configuration similar to that of a galaxy and the using the numerical method *symplectic Euler method*. The accuracy of this method is of first order and the use of a higher order method, such as *classic Runge-Kutta* or *RK4* would have most likely yielded a better result which was not a part of this assignment.

This report will cover the following types of optimizations:

- Keywords
- Vectorization
- Cache Optimization
- Arithmetic Optimization
- Optimization flags for the GNU compiler

In the following sections discussing each of the different types of optimization all subsequent time measurements for performance gain will be based on the optimal time with all optimizations applied. The file for reference will be `ellipse_N_03000.gal` and `nsteps = 100`.

2 Keywords

The following keywords have been used:

- `Restrict` *Tells the compiler that the adress of this pointer will not change for the duration of the program*
- `Const` *Tells the compiler that for the duration that this variable is in use the value of the variable will not change*
- `Static` *Tells the compiler that this function will only be called from within the program*
- `Packed` *Used to effectivly read binary data from files in to structs. It forces the computer to store the attributes of the struct next to eachother in memory*
- `Inline` *Tells the compiler that this function can be inserted from where it was called*

The performance gain with the keywords stated above did not cause a noteworthy improvement with only an approximated improvement of 6%. With a runtime of 2.5s this only correspond to an improvement of 0.12s. It should be noted that when writing the program `inline` and `const` showed the most significant improvement but where diluted in comparison to the other optimizations. It should be made a habit to always use these keywords as they can, if used correctly, have a bigger impact than what was depicted here. The keyword `Packed` should be used with care since this can in some cases cause memory misalignment to occur.

3 Vectorization

Auto-vectorization is a technique where the compiler converts loops into vectors and therefor performing multiple calculations in parallel. This was most prominent in the function `calculate_forces` and `calculate_position`. This was tested by using the following compiler flags `-O2 -ftree-vectorize -ffast-math -g -march=native` and the for testing without vectorization `-O2 -ffast-math -g -march=native`

The performance gain when using vectorization when then calculated to 7.5% which corresponded to a decrease in runtime from 2.77 to 2.51.

4 Cache Optimization

Cache optimization is the concept of optimizing the order of instruction to reduce what is called *cache miss*. Cache is a type of memory that is divided into several levels denoted `L1`, `L2` and in some cases `L3`. Access to data in L1 memory is faster than the deeper levels and one should strive to reduce the numbers of L1 misses. This can be analyzed with the tool Valgrind.

A technique called *Cache blocking* have been included in the code to improve cache memory access by dividing long lists of data into smaller blocks. This improves the ability to access memory already loaded into the L1-cache.

For smaller sizes of N *(number of particles)*, cache blocking can be a redundant implementation and was therefor only activated when $N > 100$. *Note that no implementations for when N is not divisible by 100 was made since the input files when* $N > 100$ *are all divisible by 100.*

Using *Valgrind* one can see the effect of cache blocking. This was performed with input file `ellipse_N_01000.gal` and `nsteps = 10`, since valgride increase the runtime by a large margin. this was the following output with cache blocking:

```
==175849== D   refs:       16,730,279  (16,534,769 rd   + 195,510 wr)
==175849== D1  misses:         82,870  (    80,840 rd   +   2,030 wr)
==175849== LLd misses:          4,928  (     3,077 rd   +   1,851 wr)
==175849== D1  miss rate:         0.5% (       0.5%     +     1.0%  )
==175849== LLd miss rate:         0.0% (       0.0%     +     0.9%  )
```

And this was the output when running without cache blocking:

```
==175805== D   refs:       16,384,949  (16,279,529 rd   + 105,420 wr)
==175805== D1  misses:      7,518,722  ( 7,516,692 rd   +   2,030 wr)
==175805== LLd misses:          4,928  (     3,077 rd   +   1,851 wr)
==175805== D1  miss rate:        45.9% (      46.2%     +     1.9%  )
==175805== LLd miss rate:         0.0% (       0.0%     +     1.8%  )
```

5 Arithmetic Optimization

Using the profiler `operf` and `gprof` it was concluded that the most costly function was `calculate_forces`. The assignment was to use the following formula to calculate the force

$$\mathbf{F}_i = -Gm_i \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{(r_{ij} + \epsilon_0)^3} \mathbf{r}_{ij} \tag{1}$$

Equation (1) can be implemented in C as following:

```
for (i = i_block; i < min(i_block + blockSz, N); i++)
{
target = particles + i;
target_pos_x = target->pos_x;
target_pos_y = target->pos_y;
inc_x = 0;
inc_y = 0;
for (j = j_block; j < min(j_block + blockSz, N); j++)
    {
    radius_x = ((particles + j)->pos_x) - target_pos_x;
    radius_y = ((particles + j)->pos_y) - target_pos_y;
    r_ij_square = (radius_x * radius_x + radius_y * radius_y);
    inverse_square = 1 / (sqrt(r_ij_square) + epsilon);
    tmp = factor * (particles + j)->mass * inverse_square ...
* inverse_square * inverse_square;
    inc_x += tmp * radius_x;
    inc_y += tmp * radius_y;
    }
target->vel_x += inc_x;
target->vel_y += inc_y;
}
```

By assigning `target = particles + i` we make the inner loop invariant of index i and enabling vectorization by the compiler. We declare `inc_x` and `inc_y` before the outer for loop and assigning it to 0 before the inner loop. This will add one *"large"* value to `target->vel_x` instead of adding several *"small"* values.

In the innermost for loop one have tried to reduce the amount of computational force required to solve the problem while still retaining the minimum required accuracy. This was done by reducing function calls for `pow(inverse_square, 3)` and instead manually multiplying `inverse_square * inverse_square * inverse_square`. One also tried to reduce the amount of time division was called by assigning
`inverse_square = 1 / (sqrt(r_ij_square) + epsilon)` and then multiplying with this factor.

the multiplication `delta_t * G` was included in every iteration of the inner loop, and since these values never change during runtime, one could assign this to be the value of `factor`.

There was an attempt to implement the infamous `Q_rsqrt()` that was used in the game

Quake III arena but it failed to meet the accuracy criteria. `Q_rsqrt()` is a function that approximates $\frac{1}{\sqrt{x}}$ without using the built in function `sqrt()` from `math.h` or using any floating point division, and instead exploits a clever use of datatypes and bit manipulation.

The last optimization that was implemented was the removal of the *if-statement* `if (i!=j)`. This was removed since for that case `radius_x = 0` and `radius_y = 0`, meaning that the incrementation from this case will also be 0. It was decided that doing this *useless* computation was less costly than for each iteration over the inner loop evalute the if-statement `if (i!=j)`.

6 Optimization flags for the GNU compiler

In the final version of the assignment, these were the flags used by the GNU compiler

```
-O3 -ffast-math -march=native
```

`O3` for the most aggresive optimization and the inclusion of auto-vectorization. `-ffast-math` to speed up calculations against the cost of precision. The precision was still in the acceptable range and could therefor be included. `-march=native`, tells the compiler information about the CPU it is being compiled on and can therefor utilize this to make quicker computations at the cost of making the program non-portable accross different CPUs.