N-body simulator - Parallelized with CUDA

Gabriele Bertinelli - 2103359 - Modern Computing for Physics - 24/25

The all-pairs approach to N-body simulation is a brute-force technique that evaluates alla pair-wise interactions among the N bodies. Given its $O(N^2)$ computational complexity, it's an interesting target for acceleration.
Many CPU codes exploit the force symmetry $F_{ij} = -F_{ji}$ to decrease the number of computation. However, I had difficulties to implement this strategy in my CUDA code due to the way I designed the algorithm.

1. A CUDA Implementation

One can think at the all-pairs algorithm as calculating each entry F_{ij} in a $N \times N$ grid of all pair-wise forces. The total force acting on body i, F_i, is obtained from the sum of all entries in row j. Each entry can be computed independently, so there is $O(N^2)$ available parallelism. However this approach is limited by memory bandwidth. To overcome this issue one can serialize some of the computation to achieve data reuse and to reduce the memory bandwidth required.
Let's introduce the concept of a computational **tile**, a square region of the grid of pair-wise forces consisting of *p* rows and *p* columns. Thus, only *2p* body descriptions are required to evaluate all p^2 interaction in the tile (p of which can be reused later). These body descriptions can be stored in **shared memory** or in registers. The total effect of the interactions in the tile is captured as an update to *p* acceleration vectors.
To achieve optimal reuse of the data, let's arrange the computation of a tile so that the interactions in *each row* are evaluated in *sequential* order, updating the acceleration vector, while the *separate row* are evaluated in parallel.

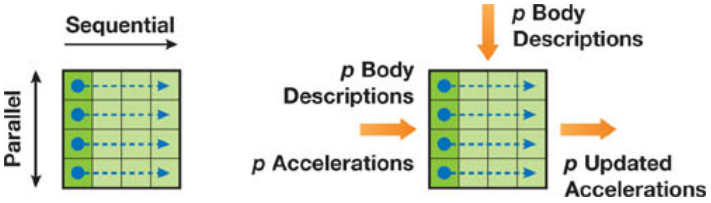

Fig.1.1 *The diagram on the left shows the evaluation strategy, and the diagram on the right shows the inputs and outputs for a tile computation. Source: NVIDIA*

1.1 Body-body Force Calculation

The interaction of a pair of bodies is implemented as an entirely serial computation. The function `computeAcceleration` computes the force of body i from its interaction with body j and updates acceleration a_i.

```
__device__ float3 computeAcceleration(float4 bi, float4 bj, float3 ai) {

    float3 r;

    // r_ij [3 FLOPS]
    r.x = bj.x - bi.x;
    r.y = bj.y - bi.y;
    r.z = bj.z - bi.z;

    // ||r_ij||^2 + eps^2 [6 FLOPS]
    float distSqr = r.x*r.x + r.y*r.y + r.z*r.z + EPS2;

    // 1/distSqr^(3/2) [4 FLOPS]
    float distSixth = distSqr * distSqr * distSqr;
    float invDistCube = rsqrtf(distSixth);

    // m_j * 1/distSqr^(3/2) [1 FLOP]
    float s = bj.w * invDistCube;

    // a_i = a_i + s * r_ij [6 FLOPS]
    ai.x += s * r.x;
    ai.y += s * r.y;
    ai.z += s * r.z;

    return ai; // tot [20 FLOPS]
}
```

The code uses CUDA's `float4` data type for body descriptions and accelerations stored in GPU device memory. Each body's mass is stored in the `w` field of the body's `float4` position. Using `float4` (instead of `float3`) data allows ***coalesced* memory** access to the arrays of data in device memory, resulting in efficient memory requests and transfers.

> *When threads in a warp access consecutive elements in global memory, the GPU can combine (or "coalesce") these individual memory requests into one large transaction. CUDA's memory system is optimized for transfers in chunks of 16 bytes (or multiples). Using a* `float4` *—which is exactly 16 bytes—ensures that each body's data is stored in a aligned, 16-byte unit.*
> *Instead* `float3` *elements don't align to the 16-byte boundaries required for optimal coalescing. This misalignment can force the hardware to perform extra memory transactions or add padding behind the scenes, reducing memory throughput.*

1.2 Tile Calculation

A tile is evaluated as *p* threads performing the same sequence of operations on different data. *Each thread updates the acceleration of one body as a result of its interaction with p other bodies*.
The algorithm loads *p* body descriptions from device memory into *shared memory* provided to each **thread block** by CUDA. Each thread block evaluates *p* successive interactions. The result of the tile calculation is *p* updated acceleration.
In the function `tileCalculation`, the input param `myPos` holds the position of the body for the executing thread and `shPosition` is an array of body descriptions in shared memory. *p* threads execute the function `computeAcceleration` in parallel, and each thread iterates over the same *p* bodies, computing the acceleration of its individual body.

```
__device__ float3 tileCalculation(float4 myPos, float3 accel) {

    // Shared memory for positions of particles in the tile
    // Memory shared across all threads in a block
    extern __shared__ float4 shPosition[];

    // Iterate through all p in this tile (tile size = blockDim.x)
    // Each thread processes interactions with all p in shared memory
    for (int i = 0; i < blockDim.x; i++) {
        accel = computeAcceleration(myPos, shPosition[i], accel);
    }

    // Return the accumulated acceleration in the tile
    return accel;
}
```

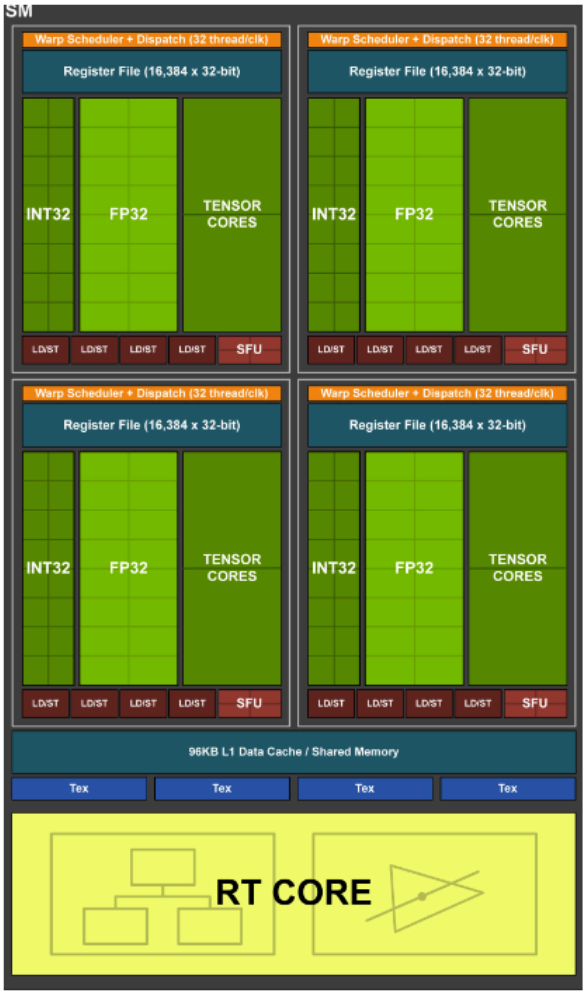

Fig. 1.2 *Turing TU102/TU104/TU106 Streaming Multiprocessor (SM). Source: NVIDIA*

1.3 Clustering Tiles into Thread Blocks

Let's define a **thread block** as having *p* threads that execute some number of tiles in sequence.

- The degree of parallelism, i.e. the number of rows, must be sufficiently large so that multiple warps can be interleaved to hide latencies in the evaluation of interactions.
- The amount of data reuse grows with the number of columns, and this parameter also govern the size of the transfer from device memory into shared memory.
- The size of the tile also determines the shared memory required. The code uses squared tiles $p \times p$. Before executing a tile, each thread fetches one body into **shared memory**, after which the threads synchronize. Then, each tile starts with *p* successive bodies in the shared memory.

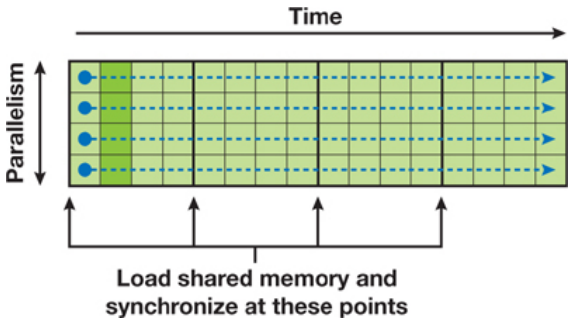

Fig. 1.3 *The CUDA kernel of pair-wise forces to calculate. Source: NVIDIA*

The figure shows a thread block that is executing multiple tiles. In a thread block there are ***N/p** tiles*, with *p* threads computing the forces on *p* bodies. A thread block reloads its shared memory every *p* steps to share *p* positions of data. **Each thread in a block computes all N interactions for one body**.
The code to calculate N-body forces for a thread block is in `forceCalculation`. The function calculates the acceleration of *p* bodies in a system, caused by their interaction with all *N* bodies in the system (*p* rows of a tile, *N* columns of the block).
The input parameters are pointers to device memory for the positions and the accelerations of the bodies.
The loop over the tiles requires two **synchronization points**:

1. The first synchronization ensures that all shared memory locations are populated before the computation proceeds;
2. The second ensures that all threads finish their computation before advancing to the next tile. *Without the second synchronization, threads that finish their part in the tile calculation might overwrite the shared memory still being read by other threads.*

```
__global__ void forceCalculation(void *d_pos, void *d_acc) {

    // Shared memory for storing particle data for the current tile
    extern __shared__ float4 shPosition[];

    // Cast void pointers to typed pointers for array indexing
    // so they can be indexed as arrays
    float4 *globPos = (float4*)d_pos;
    float4 *globAcc = (float4*)d_acc;
    float4 myPos;
    int i, tile;

    // Initialize acc accumulator for this thread's particle
    float3 acc = {0.0f, 0.0f, 0.0f};
    // Number of particles in a tile
    int p = blockDim.x;
    // Global index - unique identifier for each particle across all blocks
    int gtid = blockIdx.x * blockDim.x + threadIdx.x;
    // Load thread's assigned particle position
    myPos = globPos[gtid];

    // Process all particles in tiles
    for (i = 0, tile = 0; i < N; i += p, tile++) {

        // Calculate the global index for this thread's
        // assigned particle in the current tile
        int idx = tile * blockDim.x + threadIdx.x;
        // Each thread loads one particle from the current tile into shared memory
        shPosition[threadIdx.x] = globPos[idx];

        // Ensure all threads have loaded their data before computation
        __syncthreads();
```

```
        acc = tileCalculation<unrollLoop>(myPos, acc);
        __syncthreads();

    }

    // Save the result in global memory for the integration step
    // Convert float3 acceleration to float4
    float4 acc4 = {acc.x, acc.y, acc.z, 0.0f};
    globAcc[gtid] = acc4;

}
```

1.4 Defining a Grid of Thread Blocks

The script invokes the function in the section above on a **grid** of thread blocks to compute the acceleration of all N bodies. Let's define a 1D grid of size *N/p*. The result is a total of *N* threads that perform *N* force calculations each, for a total of N^2 interactions.

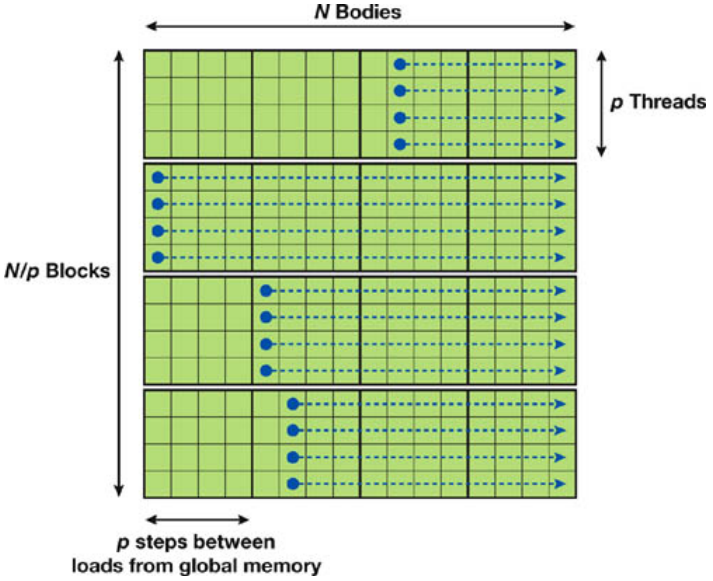

Fig 1.4 *The grid of thread blocks that calculates all forces. Source: NVIDIA*

---

2. Optimization

There are several possible layers of optimization. I tried to implement a couple of them in order to asses the performance of the algorithm.

2.1 Loop Unrolling

The first improvement comes from **loop unrolling**, where one replace a single body-body interaction call in the inner loop (in `tileCalculation`) with 2 to 32 calls to reduce loop overhead.

> This can be set using a `#pragma unroll` followed by the number of unrolls one wants the compiler to perform on a loop. On one hand, unrolling the loop can reduce the number of overhead instructions, and potentially reduce the number of clock cycles to process each p-space sample data. On the other hand, too much unrolling can potentially increase the usage of registers and reduce the number of blocks that can fit into a SM.

2.2 Performance Increases as Block Size Varies

Another tuning parameter is the value of *p*, the size of the tile. The total memory fetched by the program is N^2/p for each integration timestep, so increasing *p* decreases memory traffic. *p* must remain small such that $N/p \geq 40$ (i.e. number of SMs).
Another reason to keep *p* small is the concurrent assignment of thread blocks to multiprocessors. This technique provides more opportunity for the hardware to hide latencies of pipelined instruction execution and memory fetches. Otherwise, some multiprocessors will be idle.

2.3 Improving Performances for Small N (*not implemented*)

A final implementation is to use ***multiple threads*** *per body* to improve performance for $N < 4096$. As N decreases, there's not enough work with one thread per body to cover all the latencies, so performance drops rapidly.
One could therefore increase the number of active threads by using multiple threads on each row of a body's force calculation. If the additional threads are part of the same thread block, then the number of memory requests increases, as does the number of warps, so the latencies begin to

be covered again.
The Tesla T4 imposes a limit of 1024 threads per block, so one could split each row into q segments, keeping $p \times q \leq 1024$.

Normal loop	After loop unrolling

`int x;`
`for (x = 0; x < 100; x++)`
`{`
`  delete(x);`
`}`
	`int x;`
	`for (x = 0; x < 100; x += 5)`
	`{`
	`  delete(x);`
	`  delete(x + 1);`
	`  delete(x + 2);`
	`  delete(x + 3);`
	`  delete(x + 4);`
	`}`

Fig. 2.1 *Example of loop unrolling in C. Source: Wikipedia*

2.4 High-level Directives

Nowadays there exists many programming model that uses high-level compiler directives to expose parallelism in the code and parallelizing compilers to build the code. One of these is OpenACC. At its core OpenACC supports offloading of both computation and data from a *host* device to an *accelerator* device. In the case that the two devices have different memories the OpenACC compiler and runtime will analyze the code and handle any accelerator memory management and the transfer of data between host and device memory.
Certain optimizations are too low-level for a high-level approach like OpenACC or OpenMP. So, as usual, it depends.

1. NVIDIA Tesla T4 specs: https://www.techpowerup.com/gpu-specs/tesla-t4.c3316
2. Loop unrolling: https://en.wikipedia.org/wiki/Loop_unrolling
3. OpenACC: https://openacc-best-practices-guide.readthedocs.io/en/latest

---

3. CUDA Performance Analysis

By simply looking at the clocks and capacities of the Tesla T4 GPU, one observe that it is capable of 4070.5 gigaflops (2560 processors, 1.59 GHz each, one floating-point operation completed per cycle per processor). Multiply-add instructions (MADs) perform two floating-point operations every clock cycle, doubling the potential performance. The N-body simulator computes several MAD instructions, however the tests barely reach the peak aforementioned, probably either due to sub-optimal implementation and low number of bodies that did not “squeeze” the potential of the GPU.

When comparing gigaflop rates, I simply count the floating-point operations listed in the high-level code `computeAcceleration`: **20** floating-point operation per pair-wise force calculation. Inverse square root takes 16 clock cycles per warp of 32 threads, however I assigned a coast of 1 flop.

All the tests were made integrating 1000 steps.

3.1 Loop Unrolling

We can see, in Fig. 3.1, that without unrolling the code reaches similar performances as unroll factor equal to 16. This is could be due to the fact that modern compilers automatically unroll loops when beneficial, making manual unrolling redundant. With N=65536 we almost reach the theoretical peak of 4070.5 GFLOPS. It's likely that running the benchmark with more bodies it would have reached the peak, but the computation would have taken too long.
I choose to set unroll=16 from now on.

3.2 Number of Bodies

We can see, in Fig. 3.2, that the performance of the algorithm increases as the number of bodies grows. This is due to several reasons:

- More bodies allow to launch more threads, which increases the occupancy (i.e. more threads are active concurrently). This helps hide memory latency and other delays.
- Fixed overheads (like kernel launch and memory access initialization) are spread out over many more computations, making the overall execution more efficient.
- When the workload is small, a significant portion of the execution time is spent on overhead rather than actual computation. When the workload is increased, the proportion of time spent doing the actual computation increases.

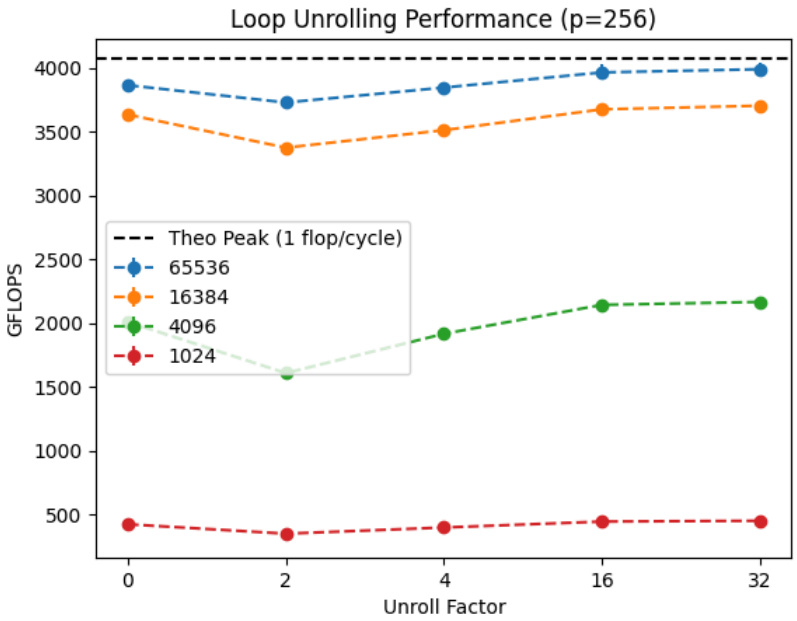

Fig. 3.1 *Comparison between different number of bodies and different unroll factors. The calculation was made setting a block size of 256 and integrating for 1000 steps. The difference between unroll=16 and unroll=32 is* $\sim 1\%$ *for all number of bodies. The benchmark was run for 10 time for each data point. The errors are comparable or smaller than the marker.*

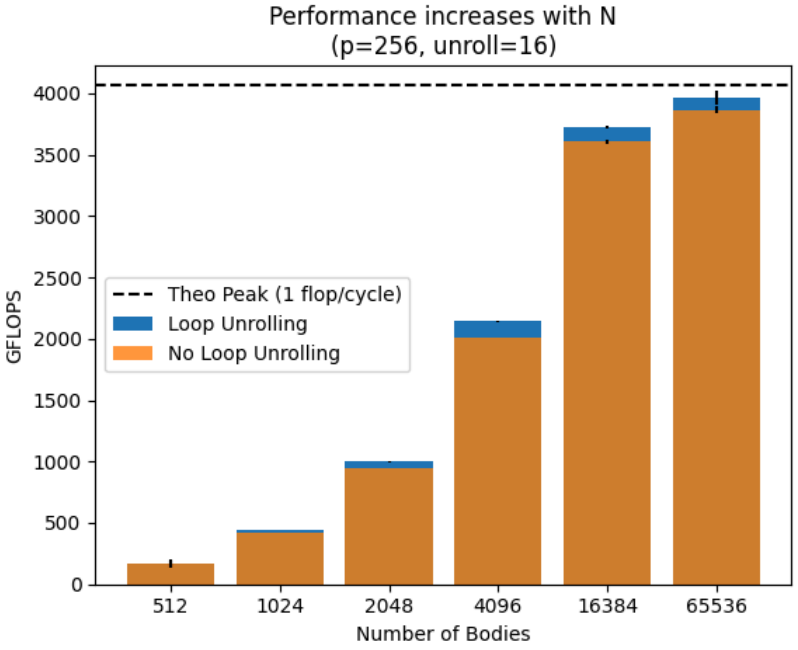

Fig. 3.2 *The performance increases as the number of bodies increases. The performance with loop unrolling is slightly better than than the benchmark without it. The benchmark was run for 10 time for each data point. The errors are the little black line at the top of each bar, but in many cases it is not visible.*

3.3 Block Size

For this benchmark I kept fixed the unroll factor at 16 and I let the block size vary. In order to not have any SM idle the theoretical lower bound it $N/p \geq 40$. N and p that satisfy this condition are highlighted in green in the table above. From Fig. 3.3, we can see that the best value of p is 128 (except for N=16384), even though for N=1024, 4096 the threshold is not met.
These discrepancies could be due to sub-optimal optimization of the code in other parts of it and/or due to some memory "traffic" between one benchmark and the other, even though memory cleaning, host and device wise, is performed and before any benchmark the GPU is primed to remove any one-time initialization overhead.
There's also to keep in consideration that the `forceCalculation` kernel was launched together with other two kernels - of lower complexity O(N) - for the integration part of the simulation. Even though the complexity is lower, the integration kernels could have led to a higher execution time and slightly worse performances.
For the last test and the simulation of a system of 6 bodies, I used p=128 and unroll=16.

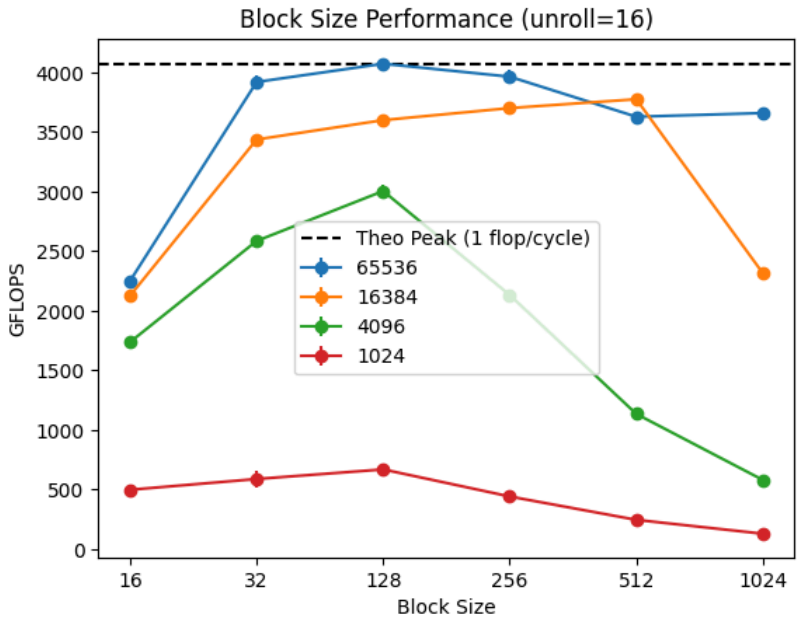

Fig. 3.3 *Comparison between different number of bodies and different block size, keeping the unroll factor fixed at 16. The errors are comparable or smaller than the marker.*

N/p ≥ 40		Block Size (p)					
		16	32	128	256	512	1024
Number of Bodies	1024	64	32	8	4	2	1
	4096	256	128	32	16	8	4
	16384	1024	512	128	64	32	16
	65536	4096	2048	512	256	128	64

Fig. 3.4 *Table that shows the value of N/p given the number of bodies (N) and the block size (p). The entries highlighted in green are the ones that satisfies the condition $N/p \geq 40$, where 40 is the number of streaming multiprocessors (SMs).*

3.4 Computational Time

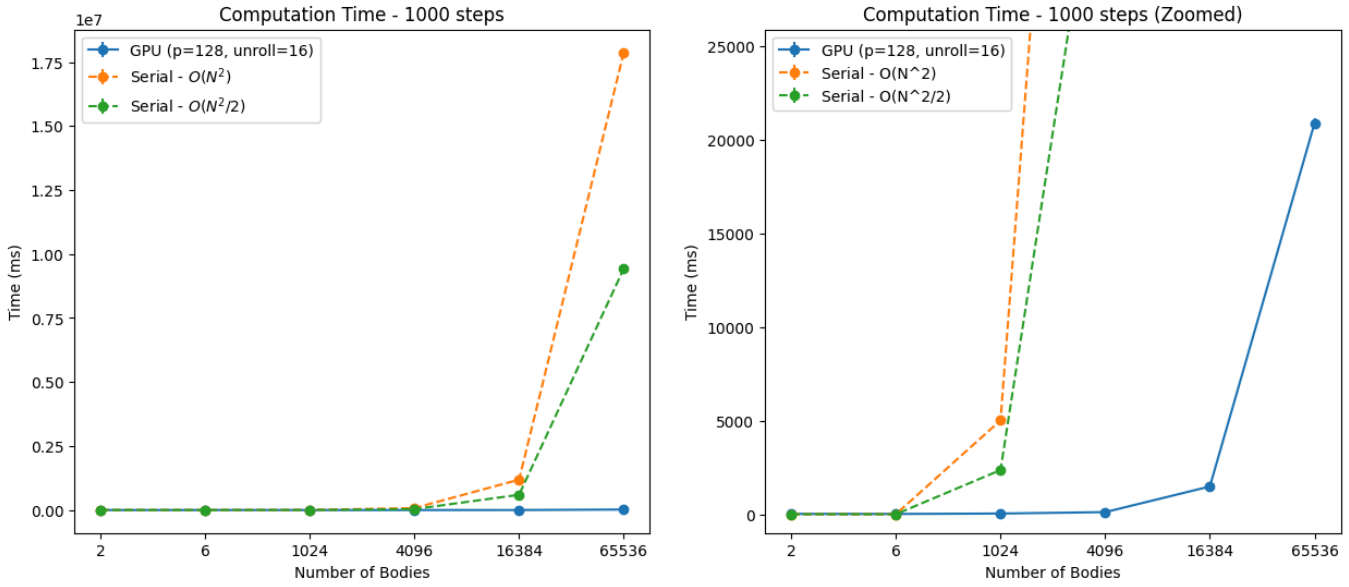

Fig. 3.5 *Comparison between the computational time for the parallel GPU code (blue solid line), the serial CPU version of it (orange dashed line) and the serial CPU version implementing force symmetry $F_{ij} = F_{ji}$. The plot on the right is a zoom-in of the plot on the left, for showing better the GPU time. The benchmark on serial code for N=16384, 65536 was run just 1 time, instead of 10, because it took to much time. The errors are comparable or smaller than the marker.*

From Fig. 3.5 we can appreciate the power of a simple GPU parallel implementation for a N-body simulator. If we consider the case of N=65536, the computational time taken by the GPU to integrate 1000 steps is $\sim 21\ s$, while the time for the CPU was almost 5 hours, and 2.5 hours for the serial code accounting for force symmetry. The CPU code was intentionally no parallelized and it has been run on a single Intel(R) Xeon(R) Gold 5218 CPU @ 2.30GHz (16 cores).

4. Six Bodies System Evolution

Simulation parameters:

- Block size=128
- Unroll factor=16
- Bodies=6
- dt=0.00001
- Steps=$5 \cdot 10^6$
- Total evolution time=50 arbitrary units of time
- Mass: uniform in $[1, 10]$
- Velocity: uniform in $[-1, 1]$
- Position: uniform in a cube $[-5, 5]$. I had to restrict the cube because the bodies were sampled to far away from each other, requiring a smaller integration steps and much much longer integration times. In fact, when particles are far from each other a faster method based on a far-field approximation is used. If one increases the number of bodies to 20 or even more, the cube of size L works just fine.
  The simulation required 2.94 minutes and below you can find the results. Further material, as gifs and plot at higher resolution, can be found in the GitHub repo for the project.

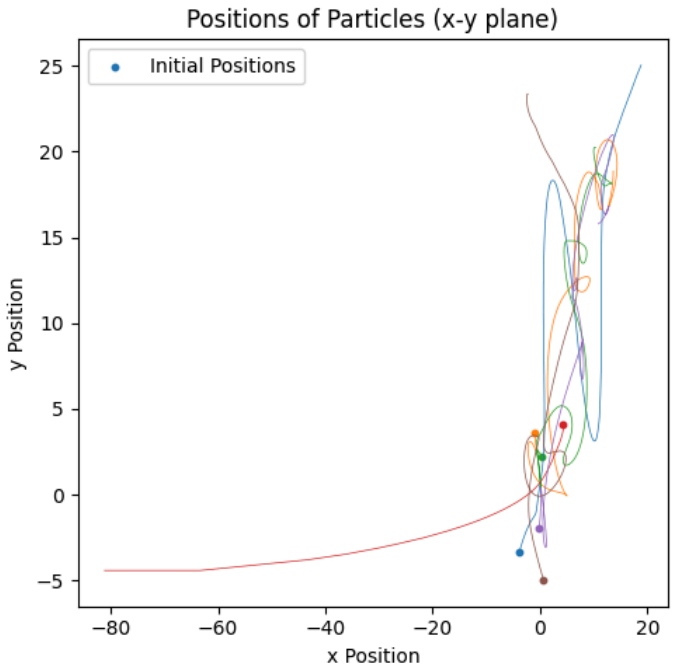

Fig. 4.1 *Evolution of the system seen on the x-y plane. The dots are the initial positions of the bodies.*

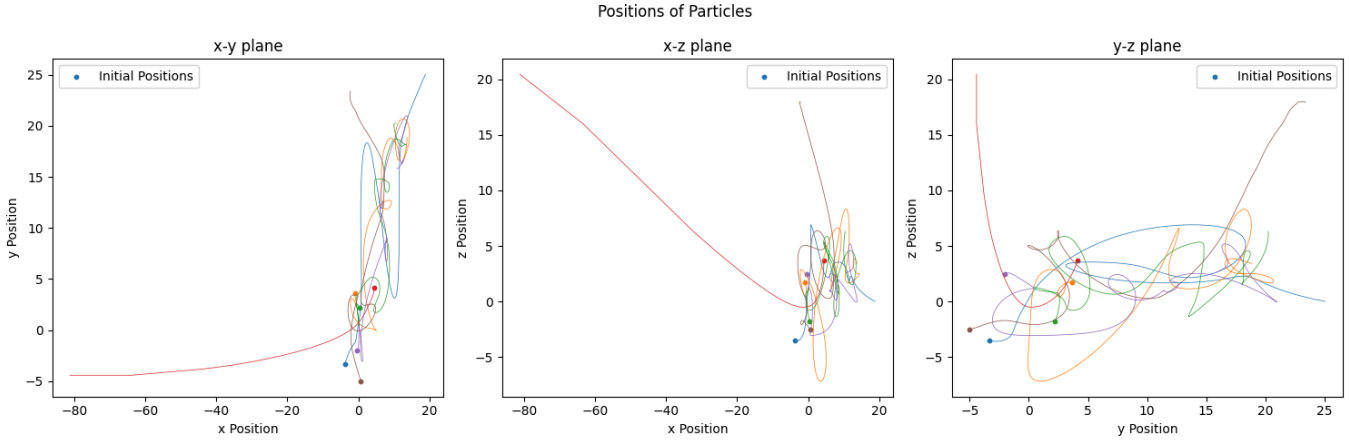

Fig. 4.2 *Evolution of the systema seen on all three planes. The dots are the initial positions of the bodies.*

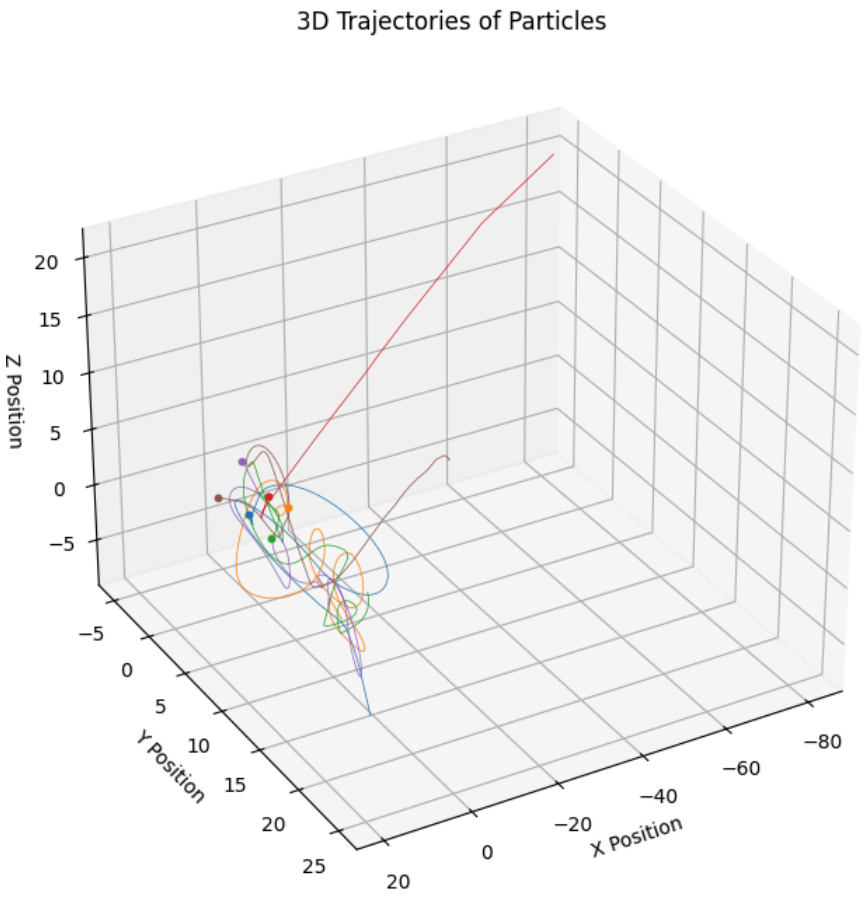

Fig. 4.3 *3D view of the evolution of the system. The dots are the initial positions of the bodies.*

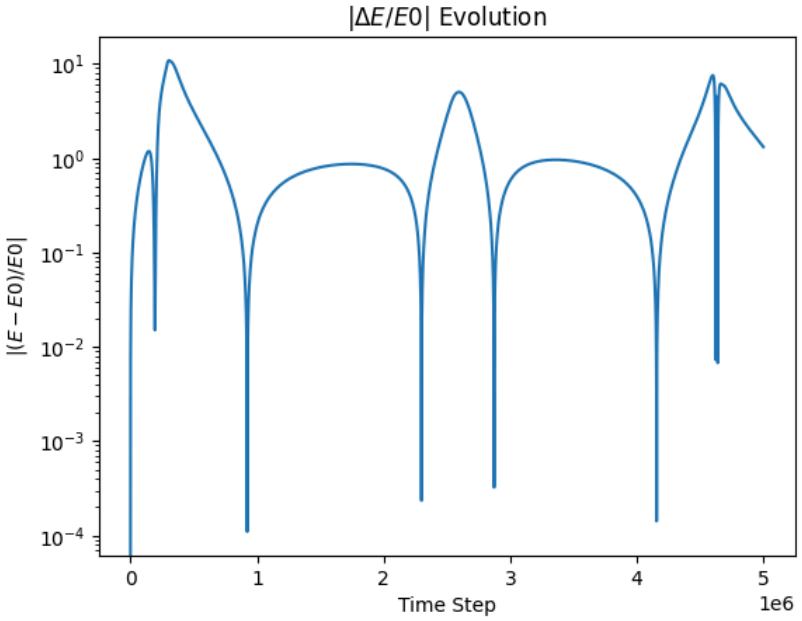

Fig. 4.4 *Evolution of the total energy E error with respect to the initial total energy E0.*

The energy seems to remain almost constant, but the pattern is not correct, for this kind of integrator. I tried to find the cause but the functions are the same to the one I've written for a Python N-body simulator called Fireworks. My guess is that this discrepancy is due to a different handling of the data in memory and rounding.

The integrator works just fine, because I've tested it with a 2-body system and the Lagrange problem of three masses in a equatorial triangle. Even though the integrator worked, I had to check many different integration time steps and softening factors because the values used in Firework, with this implementation (and also in the serial code, that is basically the same as the GPU code), didn't work out.

I also tried to translate the Python code in C language using the Verlet integrator and the values used with Fireworks the simulation and the energy error were fine.