Computação Paralela

1st Inês Ferreira
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG53879

2nd Marta Sá
Informatics Department
Master in Informatics Engineering
University of Minho
Braga, Portugal
PG54084

***Abstract*—This document is about the project proposed in the Curricular Unit of Parallel Computing in the 2023/2024 school year.**

***Index Terms*—OpenMP, C, CUDA, Parallelism**

I. Introduction

The aim of this phase is to explore parallelism in order to improve overall execution time of the code enhanced in the previous phase, we began by identifying the application's hot-spots, i.e., the code blocks with the highest computation time and tried to analyse and explore parallelism within these blocks. We concluded that the function that we should focus was *computeAccelerations()*. With this in mind, we used the API CUDA to run the function *computeAccelerations()* in the GPU.

Furthermore, we carry out an analysis of scalability and, at the end, discuss the results obtained. Note that every test was executed 3 times in order to avoid incorrect results due to the impact of other processes.

II. First Phase

In this phase we explored various methods to increase the program's performance that we explain in the following subsections.

A. Removal of constants

When analyzing the functions, we noticed that some contained operations within cycles where, for example, two constants multiplied. And in cases where such a scenario occurred, we chose to create variables outside the cycle to contain the results of these operations so that it could later be used within the cycles without always repeating calculations leading to an increase of the execution time.

Moreover, the variables sigma and epsilon always had the value 1 and were not relevant for the calculations in the *computeAccelerations()* function, so we choose to remove them from the calculations in order to make the code cleaner and the program a little more efficient.

B. Transposing matrices and changing the order of cycles

As we know, data in computers is stored following a memory hierarchy. When the CPU fetches data from memory, it stores in the cache along with the data that was allocated in the nearby positions. So if it is then necessary to access nearby positions it will take less time, as it is faster to access the cache than the main memory. In the case of matrices, data is stored in rows, so it is faster to access positions in the same row and different columns than the other way around, due to spatial locality. It was taking this information into account that we decided to transpose the matrices *v*, *a* and *r* leading to a decrease of the number of cache misses.

C. Merging of functions

Afterwards, we noticed that both functions *Potential()* and *computeAccelerations()* perform the same calculation to determine the value of the variable *rSqd*. Therefore, in order to reduce the number of instructions, we choose to merge both functions in the *computeAccelarations()* function and to make the variable *PE* global. As a consequence, we had to adapt the content of the cycles as a way to obtain the same final result.

Furthermore, we choose to iterate over the matrices using the original *computeAccelerations()* cycles, avoiding the use of *if* statements and making the code more vectorizable. However, in order to obtain the same final result, we had to calculate the *Pot* variable differently. Besides this we also noticed that due to the distributive property it is possible not to multiply by *epsilon* inside the cycle to obtain the *Pot* result but to multiply later outside the cycle.

D. Loop Unrolling

Finally, we decided to use loop unrolling in the cycle where we initialize the matrix *a* to allow two instructions to be executed at the same time, thus reducing execution time and enhancing vectorization. In addition to this, inside the function *computeAccelerations()*, we removed completely the cycles iterated by the variable *k* as it only iterated the same operation 3 times with different values of *j* and *i* and replaced it with the 3 operations and variables that were previously required in the cycle.

TABLE I
Results Obtained

Metrics (Average of 10 runs)	Execution Time (s)	Number of Instructions	Clock Cycles	CPI	Cache Misses
-O0 (Default)	93,87+-38,20	622 946 629 283	302 136 452 682	0.48	221:u
-O3	46,70 +- 17,74	299 678 928 707	149 679 834 080	0.50	68:u
-Ofast	10,034 +- 0,390	52 233 737 783	32 970 353 734	0.63	78,497
Removal of Pow and Sqrt	8,206 +- 0,205	50 822 365 131	26 943 332 320	0.53	69 276
Removal of constants	8,022 +- 0,167	50 821 441 903	26 291 488 016	0.51	66 707
Transpose and Exchange of Cycles	8,580 +- 0,161	50 366 475 392	28 106 748 244	0.56	71 663
Merge	5,397 +- 0,140	30 214 432 538	17 454 459 118	0.57	46 608
Loop unrolling	5,340 +- 0,132	30 214 154 137	17 444 238 735	0.57	45 541

Observing the results obtained, we conclude that the best approach is to use the flag -Ofast along with the implementation of loop unrolling, vectorization and techniques that allow the reduction of the number of instructions.

III. Second Phase

In the second phase, in order to improve the execution time of the code, we used OpenMP directives such as **pragma omp parallel for**, **reduction** and **private**.

A. Scalability Analyses

For the purpose of choosing an approach to explore the application's parallelism we executed an analysis of scalability and represented the results in the tables bellow. We tried to remake this analysis with two values more comparable, i.e., one N equal to 5000 and the other equal to 2500 but due to functionality problems in the *cluster* this wasn't accomplished.

TABLE II
SCALABITY ANALYSIS - SEQUENTIAL

N	Texec (s)	#I	Cycles	CPI	LLC Misses
3500	11.3604	13,468,418,801	28,380,474,148	2,11	2,668
5000	23.2026	27,412,901,323	57,983,525,282	2.11	4,047

From the results of the previous table we can see that as the value of N decreases, the execution time doesn't decrease in the same proportion.

TABLE III
SCALABITY ANALYSIS - PARALLEL

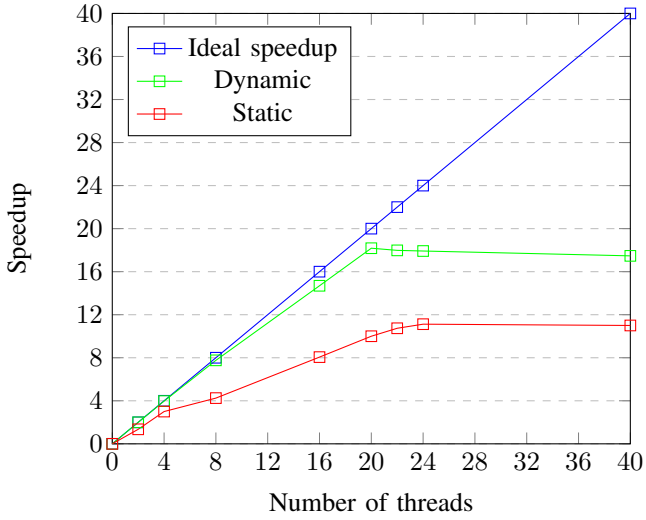
N	Threads	Texec (s)	#I	Cycles	CPI	LLC Misses	#I / Threads	#I / LLC Misses
3500	**2**	8.5563	13,918,555,218	29,491,754,964	2.12	660,143	6,959,277,609	21,084
	8	2.7024	16,499,154,658	36,058,109,886	2.18	2,285,624	2,062,394,332	7,218
	16	1.4272	19,662,306,443	42,897,253,770	2.18	4,535,542	1,228,894,152	4,335
5000	**2**	17.2240	27,872,855,861	58,609,273,115	2.10	736,735	13,936,427,930	37,832
	8	5.4543	30,643,488,667	65,649,689,748	2.14	2,076,333	3,830,436,083	14,758
	16	2.8768	34,216,087,120	74,290,786,341	2.17	4,805,880	2,138,505,445	7,119

Furthermore, the same thing happens when comparing the number of threads and the number of instruction per thread: as the first one increases, the second doesn't decrease in the same proportion. This suggests that the program has a load balancing problem between threads which can lead to an increase of the execution time due also to the fact that some threads remain idle while others are still performing computations.

In order to try to solve this problem, we used the OpenMP directive *scheduling* to experiment different approaches modifying the value of *chunk size* for both the static and the dynamic types of scheduling. Note that for the experiments carried out, we defined that the number of threads would be 24 based on the best speedup obtained.

Furthermore, we concluded that the best execution time was acquired with a dynamic type of scheduling, like we supposed, associated with the value 50 for the *chunk size*. The remaining values can be explained taking into account the overhead associated with the dynamic type of scheduling. If the size of each chunk is too small then the overhead correlated with the dynamic distribution of the iterations during the runtime will compromise the execution time. On the other hand, if the size of each chunk is too big then the dynamic type of scheduling would be similar to the static type of scheduling.

At this point, we can analyse the behaviour of the final code for a varying number of threads and compare it to the ideal behaviour by calculating the speedups.



The ideal speedup would be proportional to the number of cores, which is not the case. The difference between the ideal and the obtained speedup can probably be explained because of a memory wall problem since the CPI increases with the number of threads, as we can verify in the *Table II*. This means that, as the number of threads grows, the waiting time for each thread to access memory also grows and, consequently, the number of LLC misses and cycles increases.

IV. Third Phase

A. Implementation

With the goal of avoiding unnecessary memory allocations and given that, on one hand, the **total** number of threads is defined by the number of blocks multiplied by the number of threads per block and, on the other hand, this number should be at least equal to N, we defined the following constants:

```
#define NUM_THREADS_PER_BLOCK 16

#define NUM_BLOCKS static_cast<int>(ceil(N /
NUM_THREADS_PER_BLOCK))
```

In this associations we defined the number of thread per block (NUM_THREADS_PER_BLOCK) and the number of blocks (NUM_BLOCKS) in which we used the function *ceil()* to round up the results in order to avoid the creation of less than N total threads.

With regard to the implementation of the function *computeAccelerations()*, we used the keyword *__global__* in its declaration to indicate that it is a kernel function, i.e., a function that should only be called on the host and executed on the device. Moreover, we initialized the variable *i* with the following expression:

```
int i = bid* blockDim.x + tid;
```

In this expression we multiplied the current block by its dimension. After that, we added the thread's position in the block to be able to assign a number to each thread. With this approach, while the thread's number plus 1 is less than N, the inner cycle (iterated by *j*) is executed and the outer cycle (iterated by *i*) can be removed given that its iterations are now made through threads.

1) ***Function computeAccelerations****:* Furthermore, since all variables defined inside the function *computeAccelerations()* (see Annex II) are private to each thread, we only had to consider the race conditions regarding the variables *a* and *Pot*. As a first approach, we made the accesses to the variables *a* and *Pot* sequential through the function *atomicAddDouble()*, defined in the nvidia documentation [1]. This function is responsible for adding two doubles in a sequential way, which we thought it wasn't the best approach to deal with the problem considering that sequential sections take a lot of time executing.

Therefore, we decided to do a reduction of the variable *Pot* using the *__shared__* directive to define the array *aux*. This directive is responsible for declaring a variable in shared memory, making it shared between the threads in a block but not visible to threads in other blocks.

Before implementing the reduction we had to change the way the addictions are made to the variable *Pot*. Initially, this addictions were made directly in the variable but now they are made in the array *aux* in the position corresponding to the id of the block's thread that is currently doing the computations, which was useful to avoid race conditions.

Moreover, as we can see in Annex I, the logic behind the Sequential Addressing reduction starts by iterating a for loop where we initialize the variable *s* with the position corresponding to the middle of the block. Inside the loop, for every thread corresponding to the first half of the array *aux*, we will add to that same position the corresponding on the second half of the array. At the end of every iteration we sync the threads since we will be accessing some of the same positions of the array when changing the variable *s* for the next iteration. At the end of this loop we should have in the position 0 of the array *aux* the summation of every value stored in *aux* at the beginning of the loop.

Note that, before the implementation of the reduction of the variable *Pot*, we had to make sure that the array *aux* was completely filled. To do this we used the statement *__syncthreads()* which is responsible for making all the threads within a block wait at the barrier for each other before remaining with the program's execution.

After the reduction, we stored the value of *aux[0]*, in the array *Pot* at the position identified by the current block's id. Once again, this approach was choose in order to avoid race conditions since that adding *aux[0]* directly to a variable *Pot* required a sequential execution of the code.

It's important to refer that we also tried to reduce the array *a* but this caused a decrease of the execution time. We think that this happened because of the size of the array which is a significantly so the time that it takes to allocate it in the device's memory ends up being longer than accessing it sequentially with the function *atomicAddDouble()*.

2) ***Function computeAccelerationsAndPotential****:* In order to be able to run the program in the GPU, we had to create the function *computeAccelerationsAndPotential()*, responsible for the following tasks:

- allocate in the device's global memory the necessary data;
- copy the input data from the host's memory to the device's memory;
- execute the necessary functions on the device;
- copy the output results from the device's memory to the host's memory;
- deallocate the device's memory.

As we can see in the Annex III, the data structures that had to be allocated in the device's memory were the arrays *d_r*, *d_a* and the array *d_Pot*, which has the same size as the number of blocks. Furthermore, the arrays *d_a* and *d_Pot* were initialized with zeros and the array *d_r* with values copied from the array *r*, allocated in the host's memory. Note that, in order to simplify this process we choose to transform the variables *r* and *a*, initially matrices, into arrays.

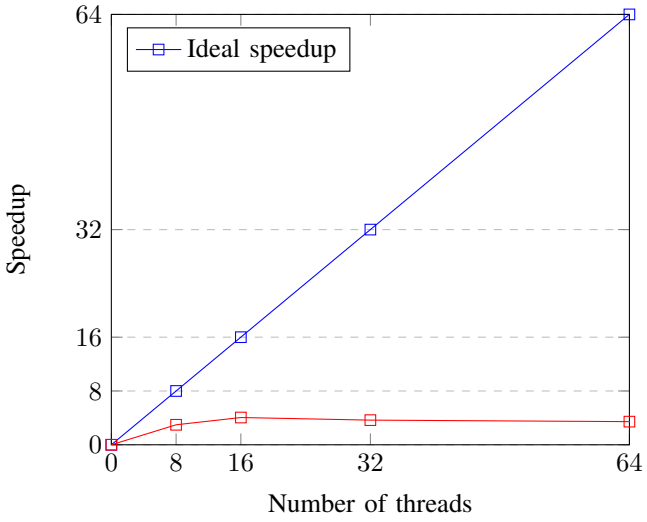
As mentioned before, we executed the necessary functions, in this case, the function *computeAccelerations()* with the arguments *d_a*, *d_r* and *d_Pot* and the parameters NUM_BLOCKS and NUM_THREADS_PER_BLOCK.

Finally, we store the new information in the data structures in the host's variables and deallocated the device's memory. One of this variables, the *POT_RESULTS*, was iterated in order to store in the variable *PE* the sum of every position, i.e., the value of *Pot* that each block calculated.

B. Scalability Analyses

To perform the scalability analysis, we analyzed strong scalability and weak scalability. In strong scalability, the goal is to analyze the way the speedup increases with the number of PUs for a fixed problem data size. In weak scalability we analyse the increase problem data size as the number of PU increases.

1) ***Strong Scalability****:* Since we are using GPU programming with CUDA, in order to analyse strong scalability we defined the number of threads per block and the number of blocks per grid in our code and observed how that affected the execution time. The results can be seen on the following graphic with an N equal to 5000:



From this results, we can see that the speedup's peak was reached with 16 threads so we will be using this number when appropriated during the following tests.

Furthermore, the ideal speedup would be proportional to the number of cores, which is not the case. The difference between the ideal and the obtained speedup can probably be explained due to the overhead associated to each thread.

Besides, we assumed that the best execution time would be accomplished by a multiple of 32 threads since a warp is a group of 32 threads. The fact that this didn't happen suggests that the GPU used was optimized to execute half-warps (16 threads).

*2) **Weak Scalability**:* Regarding the analysis of the weak scalability we measured the execution time while increasing the number of threads in the same proportion as we increased the variable N and calculated the speedup. The results can be seen on the following graphic:

TABLE IV
WEAK SCALABILITY ANALYSIS

N	Number of Threads	Execution Time (s)
2500	8	5.394
5000	16	7.768
10000	32	12.920
20000	64	28.465

From this approach, we should expect that, ideally, the execution time remains constant as we increase the value of N and number of threads. This wasn't accomplished because of various reasons, such as:

- The overhead of launching a kernel can become more noticeable as the data size and number of threads increase.
- Idle time as some threads within a block may remain idle while others are still performing computations due to, for example, synchronisation barriers.

*3) **Execution time on a different machine**:* To complete our analysis, we also decided to evaluate the variation of the excution time in another machine which has the following specifications of the GPU [2]:

- GPU - NVIDIA RTX 3070
- NVIDIA CUDA Cores - 5888
- Standard Memory Configuration - 8 GB GDDR6

Note that for this evaluation used the fixed value of 16 for the number of threads.

TABLE V
SCALABILITY ANALYSIS

N	2500	5000	10000	20000
Execution Time (s)	2.27	3.24	9.108	28,166
Execution Time Cluster (s)	5.762	7.768	14.752	51.957

C. Conclusion

With the accomplishment of this practical work, the group concludes that it was possible to consolidate knowledge taught during the classes. It was also a great way for us to explore CUDA and practise more about optimizing code. The group believes that there is room for improvement in future work, especially the challenge of exploring further and solving scalability issues, in order to further improve the performance of the program. Furthermore, we think it would be interesting to analyse the execution time with the Amdahl's law but unfortunately this wasn't possible.

REFERENCES

[1] Nvidia, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[2] Nvidia, https://www.nvidia.com/en-eu/geforce/graphics-cards/30-series/rtx-3070-3070ti/

V. ANNEXES

Annex I

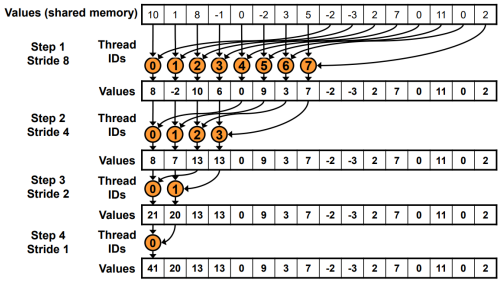
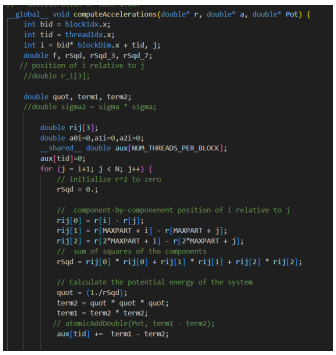


Fig. 1. Reduction algorithm.

Annex II



```
__global__ void computeAccelerations(double* r, double* a, double* Pot) {
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    int i = bid* blockDim.x + tid, j;
    double f, rSqd, rSqd_3, rSqd_7;
    // position of i relative to j
     //double r_ij[3];

    double quot, term1, term2;
    //double sigma2 = sigma * sigma;

        double rij[3];
        double a0i=0,a1i=0,a2i=0;
        __shared__ double aux[NUM_THREADS_PER_BLOCK];
        aux[tid]=0;
        for (j = i+1; j < N; j++) {
            // initialize r^2 to zero
            rSqd = 0.;

            //  component-by-componenent position of i relative to j
            rij[0] = r[i] - r[j];
            rij[1] = r[MAXPART + i] - r[MAXPART + j];
            rij[2] = r[2*MAXPART + i] - r[2*MAXPART + j];
            //  sum of squares of the components
            rSqd = rij[0] * rij[0] + rij[1] * rij[1] + rij[2] * rij[2];

            // Calculate the potential energy of the system
            quot = (1./rSqd);
            term2 = quot * quot * quot;
            term1 = term2 * term2;
           // atomicAddDouble(Pot, term1 - term2);
            aux[tid] +=  term1 - term2;
```

Fig. 2. Function *computeAccelerations* - 1.

```
        // atomicAddDouble(Pot, term1 - term2);
        aux[tid] +=  term1 - term2;


        //  from derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
        rSqd_3 = rSqd*rSqd*rSqd;
        rSqd_7 = rSqd_3*rSqd_3*rSqd;
        f = 24. * (2./rSqd_7 - 1./(rSqd_3*rSqd));

        //  from F = ma, where m = 1 in natural units!
        a0i += rij[0] * f;
        atomicAddDouble(&a[i], -rij[0] * f);
        a1i += rij[1] * f;
        atomicAddDouble(&a[MAXPART + j], -rij[1] * f);
        a2i += rij[2] * f;
        atomicAddDouble(&a[2*MAXPART + j], -rij[2] * f);
    }
    atomicAddDouble(&a[i], a0i);
    atomicAddDouble(&a[MAXPART + i], a1i);
    atomicAddDouble(&a[2*MAXPART + i], a2i);

__syncthreads();
// Perform parallel reduction using an inverted tree

for (unsigned int s = blockDim.x / 2; s > 0; s >>= 1) {
    if (tid < s && tid + s < blockDim.x) {
        aux[tid] += aux[tid + s];
    }
    __syncthreads();
}
// Store the result in the output array
if (tid == 0) {
    Pot[bid] = aux[0];
}
}
```

Fig. 3. Function *computeAccelerations* - 2.

Annex III

```
void computeAccelerationsAndPotential() {
    double *d_r, *d_a, *d_Pot;
    cudaMalloc((void**)&d_r, sizeof(double) * MAXPART * 3);
    cudaMalloc((void**)&d_a, sizeof(double) * MAXPART * 3);
    cudaMalloc((void**)&d_Pot, sizeof(double)*NUM_BLOCKS);
    checkCUDAError("mem allocations");

    cudaMemcpy(d_r, r, sizeof(double) * MAXPART * 3, cudaMemcpyHostToDevice);
    cudaMemset(d_a, 0, sizeof(double) * MAXPART * 3);
    cudaMemset(d_Pot, 0, sizeof(double)*NUM_BLOCKS);
    checkCUDAError("set memory");

    computeAccelerations<<<NUM_BLOCKS,NUM_THREADS_PER_BLOCK>>>(d_r, d_a, d_Pot);
    checkCUDAError("memcopy1");

    cudaMemcpy(a, d_a, sizeof(double) * MAXPART * 3, cudaMemcpyDeviceToHost);
    checkCUDAError("memcopy2");

    cudaMemcpy(POT_RESULTS, d_Pot, sizeof(double)*NUM_BLOCKS, cudaMemcpyDeviceToHost);
    checkCUDAError("memcopy3");


    // // Free device memory
    cudaFree(d_r);
    cudaFree(d_a);
    cudaFree(d_Pot);
    //cudaFree(aux);
    checkCUDAError("FREEs");

    //calcular o PE
    PE = 0.;
    for(int i=0;i<NUM_BLOCKS;i++){
        PE+=POT_RESULTS[i];
    }

    PE*=8;
```

Fig. 4. Function *computeAccelerationsAndPotential*.