Work Assignment - Phase 3

Parallel Computing

José Miguel Ferreira Barbosa
PG52689
Braga, Portugal

Tiago Adriano Gomes Moreira
PG52704
Braga, Portugal

Abstract—**This document is the written report of the third and final phase of work assignments of the curricular unit, Parallel Computing. The focus of this phase was how to design and implement an efficient parallel version of the case study.**

I. Introduction

The final phase of this work assignment aims to understand how to design and implement an efficient parallel version of the case study, eventually incorporating accelerators, with the main goal still being of reducing the execution time. In this report, we will explore the adopted strategies, the challenges encountered, and the proposed solutions to optimize the system's performance, emphasizing the crucial importance of parallel approaches in the pursuit of computational efficiency. It's worth noting that because of the issues we had in the two previous phases regarding results, we choose to pick up the original code and work from there again.

II. The Application Hot-Spots

As we figured out in the previous phases of this work assignment, we have two code blocks that call our hot-spots, those being:

- **Potential**
- **computeAccelarations**

It is imperative to acknowledge that both these functions were identified as hot-spots not only in the original code, that we will pick up again when working on this final phase, but also in the sequential version we had made and worked on during phase 2. So our main efforts will once again be directed towards optimizing and enhancing the performance of these components.

III. The Choosen Path

In this final work assignment, we had the freedom to choose one of three paths to improve the execution. After a discussing upon ourselves, we decided to look into the approach where we would be challenged with designing and implementing a new version for accelerators (GPU with CUDA). We choose this one because we find more suitable than the other two, as it also was the focus of the last classes of the semester of this curricular unit. We will then, work with CUDA, on the functions **Potential** and **computeAccelarations** specifically.

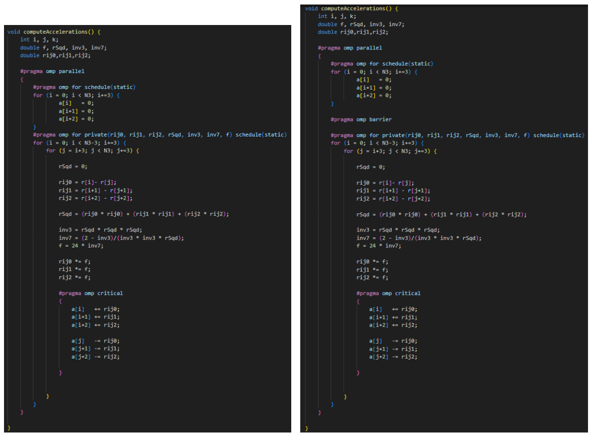
IV. Changes and Sequential code

As we mentioned in the introduction, we started from scratch again. Because of that, the code is different from the one we delivered in the versions prior to this one. First of all, to enhance our code's efficiency, we've shifted to Struct of Arrays (SoA). We made this change because it offers advantages like improved memory access patterns and better alignment with GPU parallelism.

We kept all the functions that are not the two hot spots, pretty much the same as they were in the original code (of course we adapted them to fit SoA), since they are not going to be worked on using CUDA. Also, we changed them in the first phase, but since we encountered problems with the simulation results we also felt like making more changes than required gave us a bigger chance of getting wrong results again, so we focused solely on **Potential** and **computeAccelarations** and worked carefully to make sure the same thing didn't happen again. We added a function called **power** (fig 11) with the sole purpose of replacing the **pow**, making the code a bit more efficient in the process. We only did this in natural powers, because non-natural powers are more complex to calculate, and it could lead to incorrect results. The function **Potential**, follows the same logic we applied in phase 1, when we worked on it, where we made slight changes in the operations made when calculating the potential, to improve the execution time, removing some time-consuming operations in the process, for example divisions and the call of the function **srqt**. When it comes to **computeAccelarations** we also followed the same logic we did in phase 1, where we removed of for cycles and also made slight changes to the operations made, for example we eliminated some redundant operations, and we introduced some intermediate variables to optimize the calculations and reduce repetition.

V. Phase II Corrections

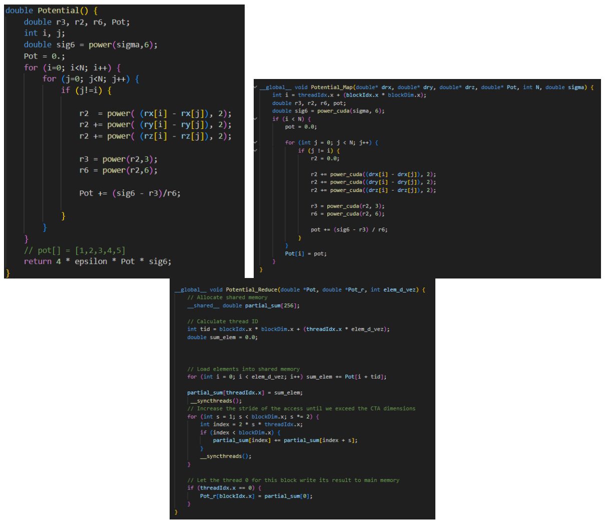
As we talked about, we started from scratch and made changes to the sequential code we had worked on previously. Since we changed the code we are working on and are working with CUDA for this final phase, we didn't pick up OpenMP again. However, we were informed that we had a data race in our last assignment, that went by unnoticed, and we identified it and decided to correct it and that correction can be seen in fig 1. To resolve this issue, we added a barrier like shown in the correction that will also guarantee that all the acceleration



```
void computeAccelerations() {
    int i, j, k;
    double f, rSqd, inv3, inv7;
    double rij0,rij1,rij2;

    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (i = 0; i < N3; i+=3) {
                a[i]   = 0;
                a[i+1] = 0;
                a[i+2] = 0;
        }
        #pragma omp for private(rij0, rij1, rij2, rSqd, inv3, inv7, f) schedule(static)
        for (i = 0; i < N3-3; i+=3) {
            for (j = i+3; j < N3; j+=3) {

                rSqd = 0;

                rij0 = r[i]- r[j];
                rij1 = r[i+1] - r[j+1];
                rij2 = r[i+2] - r[j+2];

                rSqd = (rij0 * rij0) + (rij1 * rij1) + (rij2 * rij2);

                inv3 = rSqd * rSqd * rSqd;
                inv7 = (2 - inv3)/(inv3 * inv3 * rSqd);
                f = 24 * inv7;

                rij0 *= f;
                rij1 *= f;
                rij2 *= f;

                #pragma omp critical
                {
                    a[i]   += rij0;
                    a[i+1] += rij1;
                    a[i+2] += rij2;

                    a[j]   -= rij0;
                    a[j+1] -= rij1;
                    a[j+2] -= rij2;

                }


            }
        }
    }

}
```

```
void computeAccelerations() {
    int i, j, k;
    double f, rSqd, inv3, inv7;
    double rij0,rij1,rij2;

    #pragma omp parallel
    {
        #pragma omp for schedule(static)
        for (i = 0; i < N3; i+=3) {
                a[i]   = 0;
                a[i+1] = 0;
                a[i+2] = 0;
        }

        #pragma omp barrier

        #pragma omp for private(rij0, rij1, rij2, rSqd, inv3, inv7, f) schedule(static)
        for (i = 0; i < N3-3; i+=3) {
            for (j = i+3; j < N3; j+=3) {

                rSqd = 0;

                rij0 = r[i]- r[j];
                rij1 = r[i+1] - r[j+1];
                rij2 = r[i+2] - r[j+2];

                rSqd = (rij0 * rij0) + (rij1 * rij1) + (rij2 * rij2);

                inv3 = rSqd * rSqd * rSqd;
                inv7 = (2 - inv3)/(inv3 * inv3 * rSqd);
                f = 24 * inv7;

                rij0 *= f;
                rij1 *= f;
                rij2 *= f;

                #pragma omp critical
                {
                    a[i]   += rij0;
                    a[i+1] += rij1;
                    a[i+2] += rij2;

                    a[j]   -= rij0;
                    a[j+1] -= rij1;
                    a[j+2] -= rij2;

                }


            }
        }
    }

}
```

Fig. 1. Correction of data race issue in phase 2



```
double Potential() {
    double r3, r2, r6, Pot;
    int i, j;
    double sig6 = power(sigma,6);
    Pot = 0.;
    for (i=0; i<N; i++) {
        for (j=0; j<N; j++) {
            if (j!=i) {

                r2  = power( (rx[i] - rx[j]), 2);
                r2 += power( (ry[i] - ry[j]), 2);
                r2 += power( (rz[i] - rz[j]), 2);

                r3 = power(r2,3);
                r6 = power(r2,6);

                Pot += (sig6 - r3)/r6;

            }
        }
    }
    // pot[] = [1,2,3,4,5]
    return 4 * epsilon * Pot * sig6;
}
```

```
__global__ void Potential_Map(double* drx, double* dry, double* drz, double* Pot, int N, double sigma) {
    int i = threadIdx.x + (blockIdx.x * blockDim.x);
    double r3, r2, r6, pot;
    double sig6 = power_cuda(sigma, 6);
    if (i < N) {
        pot = 0.0;

        for (int j = 0; j < N; j++) {
            if (j != i) {
                r2 = 0.0;

                r2 += power_cuda((drx[i] - drx[j]), 2);
                r2 += power_cuda((dry[i] - dry[j]), 2);
                r2 += power_cuda((drz[i] - drz[j]), 2);

                r3 = power_cuda(r2, 3);
                r6 = power_cuda(r2, 6);

                pot += (sig6 - r3) / r6;
            }
        }
        Pot[i] = pot;
    }
}
```

```
__global__ void Potential_Reduce(double *Pot, double *Pot_r, int elem_d_vez) {
    // Allocate shared memory
    __shared__ double partial_sum[256];

    // Calculate thread ID
    int tid = blockIdx.x * blockDim.x + (threadIdx.x * elem_d_vez);
    double sum_elem = 0.0;



    // Load elements into shared memory
    for (int i = 0; i < elem_d_vez; i++) sum_elem += Pot[i + tid];

    partial_sum[threadIdx.x] = sum_elem;
     __syncthreads();
    // Increase the stride of the access until we exceed the CTA dimensions
    for (int s = 1; s < blockDim.x; s *= 2) {
        int index = 2 * s * threadIdx.x;
        if (index < blockDim.x) {
            partial_sum[index] += partial_sum[index + s];
        }
        __syncthreads();
    }

    // Let the thread 0 for this block write its result to main memory
    if (threadIdx.x == 0) {
        Pot_r[blockIdx.x] = partial_sum[0];
    }
}
```

Fig. 2. Side by Side comparison of the CUDA and Sequential version of Potential

arrays are initialized correctly before the calculation starts, and we believe this small addiction solves our problem.

VI. Approach with CUDA

After finishing and making sure that the simulation results from the sequential version were correct, we went ahead and started our CUDA implementation.

In regards to the CUDA adaptation of the function **Potential** (fig 2) and to really explain what we did, let's just quickly get into what it does. The potential function calculates the potential energy of each particle in the system and sums them all, thus obtaining the total potential energy of the particle system in question. In the sequential version, the function performs these two steps almost simultaneously, where the accumulated value of the system's potential energy is stored in the variable Pot. So we came to the conclusion, that this function has two distinct parts: a mapping, corresponding to the stage of calculating the potential energy of all particles, and a reduction, corresponding to the sum of all these potential energies to obtain the potential energy of the system. It's important to note that the calculation of potential energies for different particles is embarrassingly parallel, as these calculations have no dependencies among them. The same can be said for the reduction phase, which aims to obtain the total potential energy of the system in a parallel fashion. With this understanding of the function in mind, we decided to divide this function into two different kernels, one for each component of this function: **Potential_Map**, which will fill an array with the potential energy values of each particle in the system, and **Potential_Reduce**, which will sum all the values calculated by **Potential_Map**, cumulatively placing the total sum value in the first position of the array. Additionally, we also launched a device called **power_cuda** (fig 12), which will solely perform the exponentiation of various values.

The **Potential_Map** Kernel will receive arrays with the positions of each particle (drx, dry, and drz), as well as their size (N) and sigma. It is worth mentioning that we are creating one thread for each particle in the system, that is, each thread will fill in the Pot array with the potential energy of its corresponding particle at the position corresponding to the number of that particle. For example, the potential energy of particle 0 should be located at position 0 in the Pot array. Each thread needs a specific identifier, 'i'. This way, it places the calculated potential energy in the correct location in the Pot array. This identifier should be calculated as follows: the identifier of each thread (ranging from 0 to the number of threads defined for each block), added to the identifier of each block (ranging from 0 to the number of blocks we decided on) times its dimension (which is equal to the total number of threads in the block). This formula allows us to correctly identify each thread independently of its block because we need to add the overhead given by the product between the block it is in and its dimension to its identifier within the block. In other words, we are positioning ourselves in the specific location.

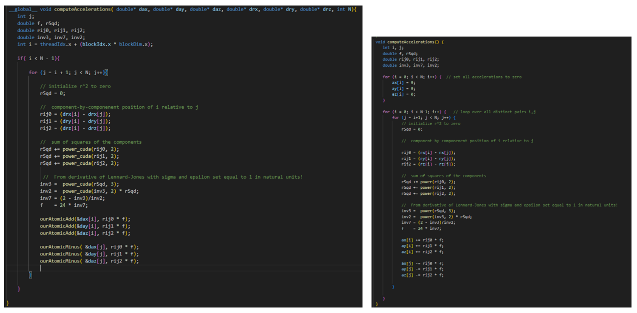
example:

with 256 threads per block, thread 7 in block 8, 'i' = 2055. It is worth noting that the calculation itself has not changed since our changes in phase 1, that we mentioned briefly in the previous section, so we will write the value of the potential energy of the respective particle into the Pot array at the correct position, as mentioned earlier. The other auxiliary variables (r3, r2, r6, pot, sig6), being local to the kernel, will be private to each thread.

Potential_Reduce will receive the array of previously calculated potential energies, as well as the number of elements that should undergo an initial reduction. The first reduction involves reducing the number of elements from 5000 to a number where we can have one thread per particle in a single block, in this case, 256 elements. In other words, this initial reduction involves going from 5000 elements to 256 elements. This necessity arises from how the reduction operation functions. The **Potential_Reduce** will perform accumulated sums of different elements. However, to do this, we must ensure synchronization at the thread level; otherwise, we would encounter data races. For this purpose, we are using the synchronization primitive **__syncthreads()**, which ensures that within the same block, threads cannot overtake each other. In other words, thread 1 cannot finish before thread 2. However, given the maximum block size of 1024 threads, to have one thread per element, we proceed with the initial reduction described above. Thus, with 256 threads, we can handle 5000 elements. It is worth noting that this initial **__syncthreads()** ensures that all threads write to the shared array before moving on to the next phase, it functions as a barrier in other words. Next, we will enter the loop of the second reduction, which is essentially a sequence of cascade sums, where in each iteration we will successively need half of the threads. For each iteration, a thread will sum two elements. For instance, thread 0 adds values 0 and 1, thread 1 adds values 2 and 3. In other words, with each iteration, we will halve the number of necessary threads until we only have to sum two values. At that point, thread 0 will add them and store the final value in the first position of the result array, Pot_r.

A small additional point to mention is that in this function, we leverage the concept of shared memory. The use of shared memory (__shared__ double partial_sum[256];) is implemented to minimize the number of accesses to global memory, which is slower compared to shared memory. Each thread block maintains a local copy of the partial_sum variable, which is utilized to accumulate the partial sum of the Pot vector elements for each thread.

These two kernels are launched by the **PotentialPrepare** function, which will be responsible for sending the values of the three position arrays (rx, ry, rz) to the GPU. It is worth noting that the array where the potential energies of each particle will be stored has a size of 5120, and not the expected 5000. This increase in size is done so that in the reduction phase, we have the correct calculations regarding the number of values to sum in the initial reduction phase which, in this case, is, 20 values at a time. These additional 120 positions will have no impact since this array is initialized with 0, meaning their presence will not affect the calculation of the system's potential energy in any way.



```
__global__ void computeAccelerations( double* dax, double* day, double* daz, double* drx, double* dry, double* drz, int N){
    int j;
    double f, rSqd;
    double rij0, rij1, rij2;
    double inv3, inv7, inv2;
    int i = threadIdx.x + (blockIdx.x * blockDim.x);

    if( i < N - 1){

        for (j = i + 1; j < N; j++){

            // initialize r^2 to zero
            rSqd = 0;

            //  component-by-componenent position of i relative to j
            rij0 = (drx[i] - drx[j]);
            rij1 = (dry[i] - dry[j]);
            rij2 = (drz[i] - drz[j]);

            //  sum of squares of the components
            rSqd += power_cuda(rij0, 2);
            rSqd += power_cuda(rij1, 2);
            rSqd += power_cuda(rij2, 2);

            //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
            inv3 =  power_cuda(rSqd, 3);
            inv2 =  power_cuda(inv3, 2) * rSqd;
            inv7 = (2 - inv3)/inv2;
            f    = 24 * inv7;

            ourAtomicAdd(&dax[i], rij0 * f);
            ourAtomicAdd(&day[i], rij1 * f);
            ourAtomicAdd(&daz[i], rij2 * f);

            ourAtomicMinus( &dax[j], rij0 * f);
            ourAtomicMinus( &day[j], rij1 * f);
            ourAtomicMinus( &daz[j], rij2 * f);

        }

    }

}
```

```
void computeAccelerations() {
    int i, j;
    double f, rSqd;
    double rij0, rij1, rij2;
    double inv3, inv7, inv2;

    for (i = 0; i < N; i++) {  // set all accelerations to zero
        ax[i] = 0;
        ay[i] = 0;
        az[i] = 0;
    }

    for (i = 0; i < N-1; i++) {   // loop over all distinct pairs i,j
        for (j = i+1; j < N; j++) {
            // initialize r^2 to zero
            rSqd = 0;

            //  component-by-componenent position of i relative to j

            rij0 = (rx[i] - rx[j]);
            rij1 = (ry[i] - ry[j]);
            rij2 = (rz[i] - rz[j]);

            //  sum of squares of the components
            rSqd += power(rij0, 2);
            rSqd += power(rij1, 2);
            rSqd += power(rij2, 2);

            //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
            inv3 =  power(rSqd, 3);
            inv2 =  power(inv3, 2) * rSqd;
            inv7 = (2 - inv3)/inv2;
            f    = 24 * inv7;

            ax[i] += rij0 * f;
            ay[i] += rij1 * f;
            az[i] += rij2 * f;

            ax[j] -= rij0 * f;
            ay[j] -= rij1 * f;
            az[j] -= rij2 * f;

        }

    }
}
```

Fig. 3. Side by Side comparison of the CUDA and Sequential version of computeAccelarations

Now let's get in to the other hot spot function **computeAccelerations** (fig 3). In our new version, the **computeAccelerationsPrepare** function is responsible for launching the kernel, initializing all accelerations to 0, and copying these values to the GPU. The **computeAccelerations** function in the sequential version calculates the accelerations of each particle from scratch every time it is called. For the CUDA version, the procedure is similar. To perform this task, we create one thread per particle and launch a single kernel that will perform this calculation for each particle. Just like in the potential function, all threads need a universal identifier calculated. The calculation itself is not altered from the sequential version to this CUDA version; what changes is the complexity of the function, decreasing from $O(N^2)$ to $O(N)$. In this function, we use the **ourAtomicAdd** and **ourAtomicMinus** (fig 7) functions from the device, which will ensure the absence of data races. This is the solution we came up with, and it works making sure that everything goes well and the one that is in the final code sent. However, we have another possible approach that we thought of, that we didn't implement, but we think is worth mentioning.

In this alternative approach (fig 8), instead of using the **ourAtomicAdd** and **ourAtomicMinus** devices, we employ the synchronization primitive **__syncthreads()**. As mentioned earlier, it serves as a barrier, preventing threads from the same block from overlapping. Each thread will have a different 'i', so we wouldn't have problems with data races. The value of 'j' depends on 'i', where 'j = i+1', and within the same block, thanks to **__syncthreads()**, no thread would surpass another. Therefore, for a single block, we wouldn't encounter any issues. However, as we are using more than one block, to achieve this, we would need to use an inter-block synchronization mechanism, ensuring that all threads from all blocks cannot surpass each other (it would be a kind of general barrier). We attempted to use **cooperative_groups**, but it didn't work.

Nevertheless, we believe that this approach would be very efficient, but since we couldn't guarantee the absence of "data races" we went with the strategy mentioned above and that gives us the assurances we desired.

VII. Results, Analysis and Discussion

In this section we will talk about the results we obtained in the phase of the work assignment and also talk about some tests we conducted and some metrics we think are valuable analyzing.

In this work assignment, we design and ran two different tests to see how our CUDA solution worked under some different circumstances. The tests were:

1) Changing the value of N in both the sequential and CUDA versions
2) Changing the number of threads per block in the CUDA version
3) Comparing the time and percentage of time for each hotspot function in both versions

A. Test 1

For the first test we conducted had a very simple principle. We wanted to see how the execution time changed in both the CUDA and the C++ versions when we changed the value of N, or in other words, we changed the number of particles. We tested 6 different numbers: 1000, 2000, 5000, 7500, 10000 and 15000 for both the sequential version(fig 5) and the CUDA version (fig 6).

We can see the results of the execution time of both versions in the graph (fig 4) and TABLE I

TABLE I
EXECUTION TIME FOR BOTH VERSIONS

N =	1000	2000	5000	7500	10000	15000
CUDA	2.0309s	4.0700s	10.2138s	15.3932s	21.2099s	35.7484s
Sequential	2.43169s	9.71843s	59.8467s	137.473s	187.933s	604.508s

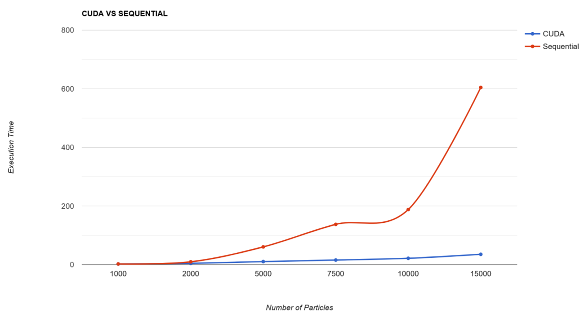


Fig. 4. Graph of execution time of CUDA and Sequential versions with different number of particles

By at looking and analyzing both of these we can see very easily, that as the value of the number of particles increases so does the difference in execution time between both versions. But we can get the full picture by calculating the speedup. Remembering the concept of speedup, it is a metric that indicates the performance gain achieved by utilizing parallelism compared to a sequential implementation. So will we use to really grasp, the difference between both versions.

In TABLE II, we can take a look at results of the calculations made to get this metric for each different value of N. Speedup is obtained by dividing the sequential time for parallel time, we can see the formula be applied in exemple below.

$$N = 5000 : 59.8467/10.2138 = 5.8594$$

TABLE II
SPEEDUP FOR EACH VALUE OF N

N =	1000	2000	5000	7500	10000	15000
SpeedUP =	1.1973	2.2052	5.8594	8.9308	8.8606	16.9101

We can notice once again, by analyzing this metric, that the difference rises with the number of particles (with the exception of 7500 and 10000). If talk about complexity, we can assess that the sequential version was a complexity of about $O(N^2)$for both the **Potential** and the **computeAccelarations** functions, while in the CUDA version the complexity of doing the same operations lowers to about $O(N)$, and so, with the input set at 'ar', this means each function is called about 200 times. So that this means that while in the CUDA version, the rise is pretty much linear, in the sequential version it's quadratic.

Let's talk about scalability analysis, where we examine how the performance of a system changes as the problem size or workload increases and what we can conclude with this test of changing the value of N. In the CUDA, execution time increases gradually as N increases, indicating a relatively good scalability. While in the sequential version, the execution time increases significantly as N increases, demonstrating poor scalability. The rate of increase is much steeper compared to the CUDA version. This shows us that the CUDA version is able to handle the workload more efficiently, whereas the sequential version struggles and becomes increasingly inefficient. Furthermore, upon observing the values presented for the number of particles, specifically 7500 and 10000, it is evident that they are quite close, with a slight decrease in the metric. Upon closer analysis, it becomes apparent that this decrease is part of a variation considered normal. This phenomenon typically precedes the next significant leap in Speedup, which is expected to occur between 10000 particles and 150000 particles.

The analysis consistently reveals that the CUDA approach demonstrates superior efficiency, particularly when dealing with considerably large values, aligning well with our intended goals. This finding underlines the scalability of our solution, highlighting that Speedup rises with the rise of the number of particles (N).

B. Test 2

The second test was changing the number of threads used in our CUDA version, with the number of particles set at 5000. We did this by changing the value of the variable 'TB_SIZE' in both **computeAccelerationsPrepare** and **PotentialPrepare**, which is the variable that defines the number of threads per block in our solution. The values we tested were: 64, 128, 256, 512 and 1024. 256, was the number that was delivered in the final code. It's important to mention that the number of threads is directly influences the number of blocks.

```
Example
const int TB_SIZE = 256;
int GRID_SIZE =
ceil((N * 1.0) / (TB_SIZE * 1.0));
// GRID_SIZE = 20
computeAccelerations
<<<GRID_SIZE, TB_SIZE>>>
```

Another quick thing worth mentioning is that in CUDA, we should always run the numbers of thereds as a multiple of 32 since GPU hardware is designed to perform operations in parallel in warps of 32 threads. If we have a total number of threads that is not a multiple of 32, the GPU will still execute the threads, but there will be a number of "ghost threads" that will be filled in to complete a warp. These additional threads may not be efficiently utilized and can result in some resource wastage. It's something that we have to be careful about when managing threads.

TABLE III
EXECUTION TIME IN RELATION TO NUMBER OF THREADS

Nº of Blocks	78	39	20	10	5
Threads per Block	64	128	256	512	1024
Execution Time	10.07s	10.15s	10.21s	10.45s	12.21s

Analyzing Table III we can see that the execution time is lower in the cases with more blocks and fewer threads, but they all remain around the 10 seconds mark, with the notable exception being the last example, where we run 1024 threads per block, with 5 blocks where it rises to 12 seconds. 1024 threads is the maximum amount we can run in a single block. We believe that the reason this case is such an outlier in this test is that the huge amount of threads in this block causes a slight overhead when trying to synchronize them all. In regards to the other values, as we just noted as well, they aren't that far apart from each other. We can attribute the fact the execution time is lower with more blocks to a couple of potential reasons. With a lower number of threads per block, it is possible that the GPU is using available resources more efficiently. This may occur because the GPU can allocate and manage blocks more effectively, avoiding underutilization of its resources. Another reason that may cause this, is that a lower number of threads per block can lead to better load balancing, ensuring that each thread has a significant amount of work to perform which can be important to avoid situations where some threads finish quickly while others are still processing, which could result in delays, and we believe these two reasons are why we obtained the results we did in regards to this specific test.

C. Test 3

Our goal with this third test is to do a more detailed comparison of the sequential version of the code with our CUDA solution in order to have a more in depth comprehension of how the implementations we made really affects performance. In this test, we compare the time taken by each hotspot function, taking into consideration as well the percentage of time taken regarding the total execution time. They are the only two relevant functions to analyze since, they are the only ones that were changed and also are the ones that have a relevant impact in the execution time of the program, in both versions. To do this test we used the command *'gprof'* for the sequential version (fig 17) and for the CUDA version we analyzed the profiling results, while in TABLE IV we have the values for time taken by each hotspot function, while in TABLE V we have the percentage of time each function in regards to the total time ((please note that in the CUDA version the value for 'Potential' includes both **Potential_Map** and **Potencial_Reduce**).

As we have discussed before, the total time between both versions is very different, with the parallel version being

TABLE IV
TIME TAKEN COMPARISON

	Potential	computeAccelarations	Total Time
Sequential Version	30.61s	29.24s	59.86s
CUDA Version	2.4387s	7.75105s	10.2138s

TABLE V
PERCENTAGE COMPARISON

	Potential	computeAccelarations	Total Time
Sequential Version	51.16%	48.87%	59.86s
CUDA Version	23.89%	75.89%	10.2138s

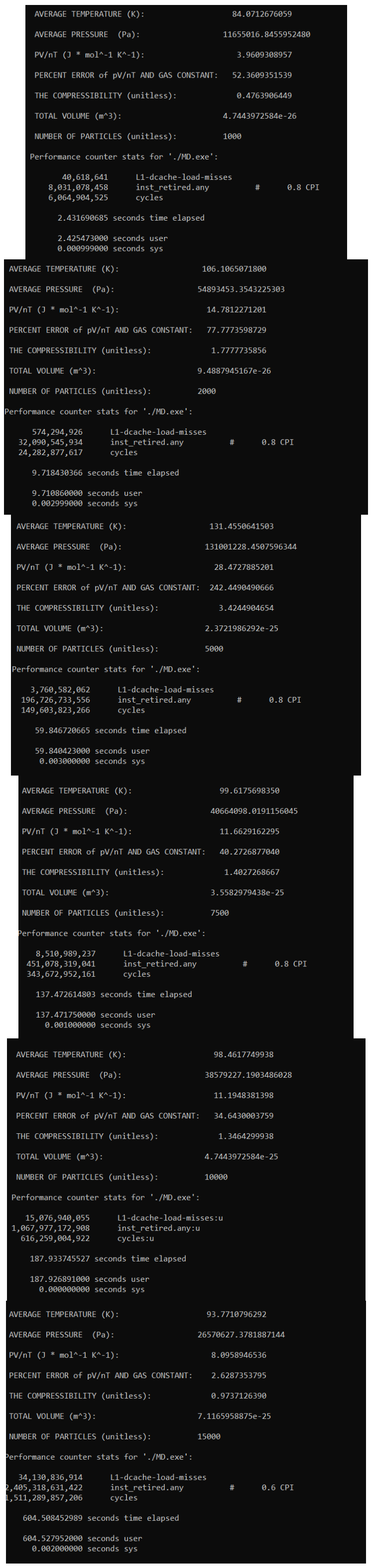
almost 6 times faster than the sequential one. However, this is not the point of this test and taking a look at both tables there is one thing that stands out immediately, that being the fact that while in the sequential version both functions take almost similar times (30.61s and 29.24s) and because of that the percentage of both is almost 50/50, in the CUDA version, there is a very big difference. The combination of both Potential functions that run on GPU (**Potential_Map** and **Potencial_Reduce**) only takes 23.89% of the total execution time, while **computeAccelarations** takes 75.89% of the time.

This can be explained by how both functions were implemented. Our implementation of the **computeAccelarations** function, works well and does exactly what it's required, and guarantees the absence of data races, but like we discussed in the previous section, isn't the most efficient solution possible, since it uses **ourAtomicAdd** and **ourAtomicMinus** devices that, while they are a good and safe way to guarantee functionality, it comes at a performance cost. By contrast, our **Potential** solution, implemented via **Potential_Map** and **Potencial_Reduce** (and of course **PotencialPrepare**), is very efficient. This is a lot due to the fact, that in both functions that run on the GPU, we don't really have any "critical zones", because it's extremely parallel, and adding the fact that we use shared memory while in the reduce function, it's going to come at a very good performance gain, explaining why it gives us such a low execution time.

VIII. CONCLUSION

To conclude this phase 3 and final report in regards to this work assignment, we feel like the work we did on this project as a whole, helped us understand a lot better how parallel programming works and how we can improve it and that we should be aware of when working on it. We faced a lot of difficulties during the whole work assignment, but we believe that these helped us see first hand what to avoid doing and what mistakes can happen. We feel that this final phase has helped us to better consolidate our knowledge in the realm of CUDA and to gain a deeper understanding of how this aspect of programming operates.

ATTACHMENTS



```
 AVERAGE TEMPERATURE (K):                   84.0712676059

 AVERAGE PRESSURE  (Pa):                  11655016.8455952480

 PV/nT (J * mol^-1 K^-1):                    3.9609308957

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   52.3609351539

 THE COMPRESSIBILITY (unitless):             0.4763906449

 TOTAL VOLUME (m^3):                       4.7443972584e-26

 NUMBER OF PARTICLES (unitless):          1000

Performance counter stats for './MD.exe':

      40,618,641      L1-dcache-load-misses
   8,031,078,458      inst_retired.any          #      0.8 CPI
   6,064,904,525      cycles

     2.431690685 seconds time elapsed

     2.425473000 seconds user
     0.000999000 seconds sys
```

```
 AVERAGE TEMPERATURE (K):                  106.1065071800

 AVERAGE PRESSURE  (Pa):                  54893453.3543225303

 PV/nT (J * mol^-1 K^-1):                   14.7812271201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   77.7773598729

 THE COMPRESSIBILITY (unitless):             1.7777735856

 TOTAL VOLUME (m^3):                      9.4887945167e-26

 NUMBER OF PARTICLES (unitless):          2000

Performance counter stats for './MD.exe':

     574,294,926      L1-dcache-load-misses
  32,090,545,934      inst_retired.any          #      0.8 CPI
  24,282,877,617      cycles

     9.718430366 seconds time elapsed

     9.710860000 seconds user
     0.002999000 seconds sys
```

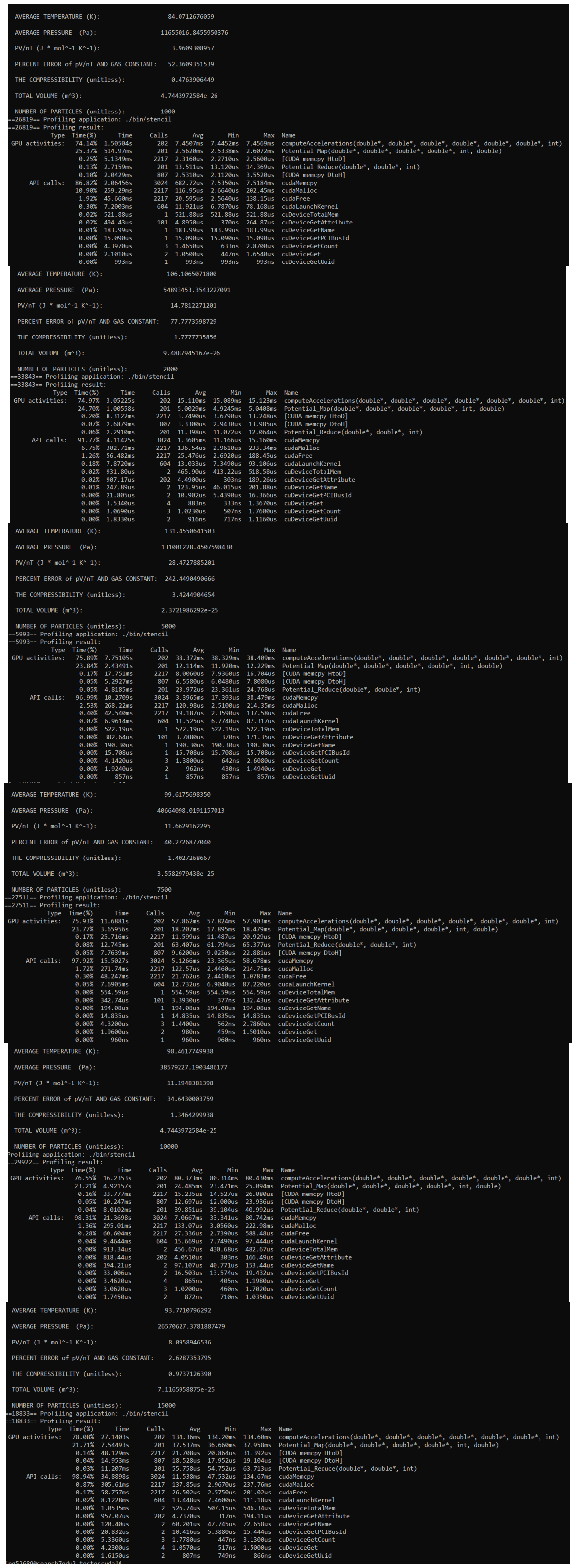
```
 AVERAGE TEMPERATURE (K):                  131.4550641503

 AVERAGE PRESSURE  (Pa):                  131001228.4507596344

 PV/nT (J * mol^-1 K^-1):                   28.4727885201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

 THE COMPRESSIBILITY (unitless):             3.4244904654

 TOTAL VOLUME (m^3):                      2.3721986292e-25

 NUMBER OF PARTICLES (unitless):          5000

Performance counter stats for './MD.exe':

   3,760,582,062      L1-dcache-load-misses
 196,726,733,556      inst_retired.any          #      0.8 CPI
 149,603,823,266      cycles

    59.846720665 seconds time elapsed

    59.840423000 seconds user
     0.003000000 seconds sys
```

```
 AVERAGE TEMPERATURE (K):                   99.6175698350

 AVERAGE PRESSURE  (Pa):                  40664098.0191156045

 PV/nT (J * mol^-1 K^-1):                   11.6629162295

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   40.2726877040

 THE COMPRESSIBILITY (unitless):             1.4027268667

 TOTAL VOLUME (m^3):                      3.5582979438e-25

 NUMBER OF PARTICLES (unitless):          7500

Performance counter stats for './MD.exe':

   8,510,989,237      L1-dcache-load-misses
 451,078,319,041      inst_retired.any          #      0.8 CPI
 343,672,952,161      cycles

   137.472614803 seconds time elapsed

   137.471750000 seconds user
     0.001000000 seconds sys
```

```
 AVERAGE TEMPERATURE (K):                   98.4617749938

 AVERAGE PRESSURE  (Pa):                  38579227.1903486028

 PV/nT (J * mol^-1 K^-1):                   11.1948381398

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   34.6430003759

 THE COMPRESSIBILITY (unitless):             1.3464299938

 TOTAL VOLUME (m^3):                      4.7443972584e-25

 NUMBER OF PARTICLES (unitless):          10000

Performance counter stats for './MD.exe':

    15,076,940,055      L1-dcache-load-misses:u
 1,067,977,172,908      inst_retired.any:u
   616,259,004,922      cycles:u

     187.933745527 seconds time elapsed

     187.926891000 seconds user
       0.000000000 seconds sys
```

```
 AVERAGE TEMPERATURE (K):                   93.7710796292

 AVERAGE PRESSURE  (Pa):                  26570627.3781887144

 PV/nT (J * mol^-1 K^-1):                    8.0958946536

 PERCENT ERROR of pV/nT AND GAS CONSTANT:    2.6287353795

 THE COMPRESSIBILITY (unitless):             0.9737126390

 TOTAL VOLUME (m^3):                      7.1165958875e-25

 NUMBER OF PARTICLES (unitless):          15000

Performance counter stats for './MD.exe':

    34,130,836,914      L1-dcache-load-misses
 2,405,318,631,422      inst_retired.any          #      0.6 CPI
 1,511,289,857,206      cycles

     604.508452989 seconds time elapsed

     604.527952000 seconds user
       0.002000000 seconds sys
```

Fig. 5. Results of C++ version with different Number of Particles



```
 AVERAGE TEMPERATURE (K):                   84.0712676059

 AVERAGE PRESSURE  (Pa):                  11655016.8455950376

 PV/nT (J * mol^-1 K^-1):                    3.9609308957

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   52.3609351539

 THE COMPRESSIBILITY (unitless):             0.4763906449

 TOTAL VOLUME (m^3):                      4.7443972584e-26

 NUMBER OF PARTICLES (unitless):          1000
==26819== Profiling application: ./bin/stencil
==26819== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   74.14%  1.50504s       202  7.4507ms  7.4452ms  7.4569ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   25.37%  514.97ms       201  2.5620ms  2.5338ms  2.6072ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.25%  5.1349ms      2217  2.3160us  2.2710us  2.5600us  [CUDA memcpy HtoD]
                    0.13%  2.7159ms       201  13.511us  13.120us  14.369us  Potential_Reduce(double*, double*, int)
                    0.10%  2.0429ms       807  2.5310us  2.1120us  3.5520us  [CUDA memcpy DtoH]
      API calls:   86.82%  2.06456s      3024  682.72us  7.5350us  7.5184ms  cudaMemcpy
                   10.90%  259.29ms      2217  116.95us  2.6640us  202.45ms  cudaMalloc
                    1.92%  45.660ms      2217  20.595us  2.5640us  138.15us  cudaFree
                    0.30%  7.2003ms       604  11.921us  6.7870us  78.168us  cudaLaunchKernel
                    0.02%  521.88us         1  521.88us  521.88us  521.88us  cuDeviceTotalMem
                    0.02%  494.43us       101  4.8950us     370ns  264.87us  cuDeviceGetAttribute
                    0.01%  183.99us         1  183.99us  183.99us  183.99us  cuDeviceGetName
                    0.00%  15.090us         1  15.090us  15.090us  15.090us  cuDeviceGetPCIBusId
                    0.00%  4.3970us         3  1.4650us     633ns  2.8700us  cuDeviceGetCount
                    0.00%  2.1010us         2  1.0500us     447ns  1.6540us  cuDeviceGet
                    0.00%     993ns         1     993ns     993ns     993ns  cuDeviceGetUuid
```

```
 AVERAGE TEMPERATURE (K):                  106.1065071800

 AVERAGE PRESSURE  (Pa):                  54893453.3543227091

 PV/nT (J * mol^-1 K^-1):                   14.7812271201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   77.7773598729

 THE COMPRESSIBILITY (unitless):             1.7777735856

 TOTAL VOLUME (m^3):                      9.4887945167e-26

 NUMBER OF PARTICLES (unitless):          2000
==33843== Profiling application: ./bin/stencil
==33843== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   74.97%  3.05225s       202  15.110ms  15.089ms  15.123ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   24.70%  1.00558s       201  5.0029ms  4.9245ms  5.0408ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.20%  8.3122ms      2217  3.7490us  3.6790us  13.248us  [CUDA memcpy HtoD]
                    0.07%  2.6879ms       807  3.3300us  2.9430us  13.985us  [CUDA memcpy DtoH]
                    0.06%  2.2910ms       201  11.398us  11.072us  12.064us  Potential_Reduce(double*, double*, int)
      API calls:   91.77%  4.11425s      3024  1.3605ms  11.166us  15.160ms  cudaMemcpy
                    6.75%  302.71ms      2217  136.54us  2.9610us  233.34ms  cudaMalloc
                    1.26%  56.482ms      2217  25.476us  2.6920us  188.45us  cudaFree
                    0.18%  7.8720ms       604  13.033us  7.3490us  93.106us  cudaLaunchKernel
                    0.02%  931.80us         2  465.90us  413.22us  518.58us  cuDeviceTotalMem
                    0.02%  907.17us       202  4.4900us     303ns  189.26us  cuDeviceGetAttribute
                    0.01%  247.89us         2  123.95us  46.015us  201.88us  cuDeviceGetName
                    0.00%  21.805us         2  10.902us  5.4390us  16.366us  cuDeviceGetPCIBusId
                    0.00%  3.5340us         4     883ns     333ns  1.3670us  cuDeviceGet
                    0.00%  3.0690us         3  1.0230us     507ns  1.7600us  cuDeviceGetCount
                    0.00%  1.8330us         2     916ns     717ns  1.1160us  cuDeviceGetUuid
```

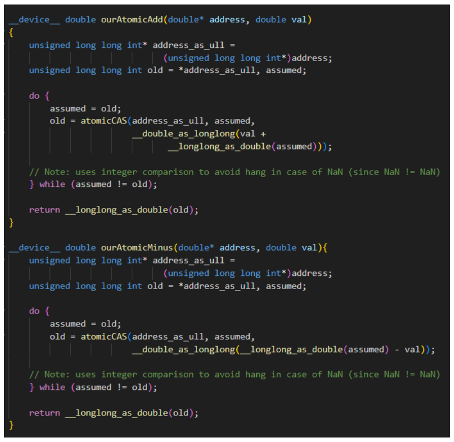
```
 AVERAGE TEMPERATURE (K):                  131.4550641503

 AVERAGE PRESSURE  (Pa):                  131001228.4507598430

 PV/nT (J * mol^-1 K^-1):                   28.4727885201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

 THE COMPRESSIBILITY (unitless):             3.4244904654

 TOTAL VOLUME (m^3):                      2.3721986292e-25

 NUMBER OF PARTICLES (unitless):          5000
==5993== Profiling application: ./bin/stencil
==5993== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   75.89%  7.75105s       202  38.372ms  38.329ms  38.409ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   23.84%  2.43491s       201  12.114ms  11.920ms  12.229ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.17%  17.751ms      2217  8.0060us  7.9360us  16.704us  [CUDA memcpy HtoD]
                    0.05%  5.2927ms       807  6.5580us  6.0480us  7.8080us  [CUDA memcpy DtoH]
                    0.05%  4.8185ms       201  23.972us  23.361us  24.768us  Potential_Reduce(double*, double*, int)
      API calls:   96.99%  10.2709s      3024  3.3965ms  17.393us  38.479ms  cudaMemcpy
                    2.53%  268.22ms      2217  120.98us  2.5100us  214.35ms  cudaMalloc
                    0.40%  42.540ms      2217  19.187us  2.3590us  137.58us  cudaFree
                    0.07%  6.9614ms       604  11.525us  6.7740us  87.317us  cudaLaunchKernel
                    0.00%  522.19us         1  522.19us  522.19us  522.19us  cuDeviceTotalMem
                    0.00%  382.64us       101  3.7880us     370ns  171.35us  cuDeviceGetAttribute
                    0.00%  190.30us         1  190.30us  190.30us  190.30us  cuDeviceGetName
                    0.00%  15.708us         1  15.708us  15.708us  15.708us  cuDeviceGetPCIBusId
                    0.00%  4.1420us         3  1.3800us     642ns  2.6080us  cuDeviceGetCount
                    0.00%  1.9240us         2     962ns     430ns  1.4940us  cuDeviceGet
                    0.00%     857ns         1     857ns     857ns     857ns  cuDeviceGetUuid
```

```
 AVERAGE TEMPERATURE (K):                   99.6175698350

 AVERAGE PRESSURE  (Pa):                  40664098.0191157013

 PV/nT (J * mol^-1 K^-1):                   11.6629162295

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   40.2726877040

 THE COMPRESSIBILITY (unitless):             1.4027268667

 TOTAL VOLUME (m^3):                      3.5582979438e-25

 NUMBER OF PARTICLES (unitless):          7500
==27511== Profiling application: ./bin/stencil
==27511== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   75.93%  11.6881s       202  57.862ms  57.824ms  57.903ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   23.77%  3.65956s       201  18.207ms  17.895ms  18.479ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.17%  25.716ms      2217  11.599us  11.487us  20.929us  [CUDA memcpy HtoD]
                    0.08%  12.745ms       201  63.407us  61.794us  65.377us  Potential_Reduce(double*, double*, int)
                    0.05%  7.7639ms       807  9.6200us  9.0250us  22.881us  [CUDA memcpy DtoH]
      API calls:   97.92%  15.5027s      3024  5.1266ms  23.365us  58.678ms  cudaMemcpy
                    1.72%  271.74ms      2217  122.57us  2.4460us  214.75ms  cudaMalloc
                    0.30%  48.247ms      2217  21.762us  2.4410us  1.0783ms  cudaFree
                    0.05%  7.6905ms       604  12.732us  6.9040us  87.220us  cudaLaunchKernel
                    0.00%  554.59us         1  554.59us  554.59us  554.59us  cuDeviceTotalMem
                    0.00%  342.74us       101  3.3930us     377ns  132.43us  cuDeviceGetAttribute
                    0.00%  194.08us         1  194.08us  194.08us  194.08us  cuDeviceGetName
                    0.00%  14.835us         1  14.835us  14.835us  14.835us  cuDeviceGetPCIBusId
                    0.00%  4.3200us         3  1.4400us     562ns  2.7860us  cuDeviceGetCount
                    0.00%  1.9600us         2     980ns     459ns  1.5010us  cuDeviceGet
                    0.00%     960ns         1     960ns     960ns     960ns  cuDeviceGetUuid
```

```
 AVERAGE TEMPERATURE (K):                   98.4617749938

 AVERAGE PRESSURE  (Pa):                  38579227.1903486177

 PV/nT (J * mol^-1 K^-1):                   11.1948381398

 PERCENT ERROR of pV/nT AND GAS CONSTANT:   34.6430003759

 THE COMPRESSIBILITY (unitless):             1.3464299938

 TOTAL VOLUME (m^3):                      4.7443972584e-25

 NUMBER OF PARTICLES (unitless):          10000
Profiling application: ./bin/stencil
==29922== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   76.55%  16.2353s       202  80.373ms  80.314ms  80.430ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   23.21%  4.92157s       201  24.485ms  23.471ms  25.094ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.16%  33.777ms      2217  15.235us  14.527us  26.080us  [CUDA memcpy HtoD]
                    0.05%  10.247ms       807  12.697us  12.000us  23.936us  [CUDA memcpy DtoH]
                    0.04%  8.0102ms       201  39.851us  39.104us  40.992us  Potential_Reduce(double*, double*, int)
      API calls:   98.31%  21.3698s      3024  7.0667ms  33.341us  80.742ms  cudaMemcpy
                    1.36%  295.01ms      2217  133.07us  3.0560us  222.98ms  cudaMalloc
                    0.28%  60.604ms      2217  27.336us  2.7390us  588.48us  cudaFree
                    0.04%  9.4644ms       604  15.669us  7.7490us  97.444us  cudaLaunchKernel
                    0.00%  913.34us         2  456.67us  430.68us  482.67us  cuDeviceTotalMem
                    0.00%  818.44us       202  4.0510us     303ns  166.49us  cuDeviceGetAttribute
                    0.00%  194.21us         2  97.107us  40.771us  153.44us  cuDeviceGetName
                    0.00%  33.006us         2  16.503us  13.574us  19.432us  cuDeviceGetPCIBusId
                    0.00%  3.4620us         4     865ns     405ns  1.1980us  cuDeviceGet
                    0.00%  3.0620us         3  1.0200us     460ns  1.7020us  cuDeviceGetCount
                    0.00%  1.7450us         2     872ns     710ns  1.0350us  cuDeviceGetUuid
```

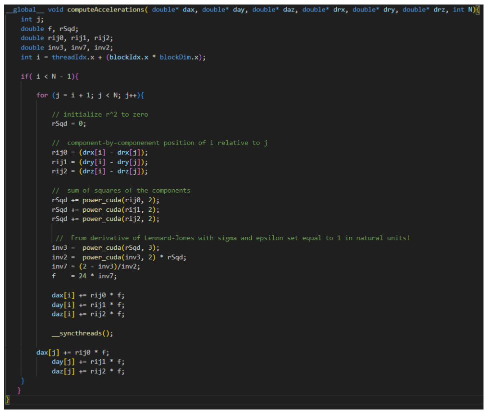
```
 AVERAGE TEMPERATURE (K):                   93.7710796292

 AVERAGE PRESSURE  (Pa):                  26570627.3781887479

 PV/nT (J * mol^-1 K^-1):                    8.0958946536

 PERCENT ERROR of pV/nT AND GAS CONSTANT:    2.6287353795

 THE COMPRESSIBILITY (unitless):             0.9737126390

 TOTAL VOLUME (m^3):                      7.1165958875e-25

 NUMBER OF PARTICLES (unitless):          15000
==18833== Profiling application: ./bin/stencil
==18833== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   78.08%  27.1403s       202  134.36ms  134.20ms  134.60ms  computeAccelerations(double*, double*, double*, double*, double*, double*, int)
                   21.71%  7.54493s       201  37.537ms  36.660ms  37.958ms  Potential_Map(double*, double*, double*, double*, int, double)
                    0.14%  48.129ms      2217  21.708us  20.864us  31.392us  [CUDA memcpy HtoD]
                    0.04%  14.953ms       807  18.528us  17.952us  19.104us  [CUDA memcpy DtoH]
                    0.03%  11.207ms       201  55.758us  54.752us  63.713us  Potential_Reduce(double*, double*, int)
      API calls:   98.94%  34.8898s      3024  11.538ms  47.532us  134.67ms  cudaMemcpy
                    0.87%  305.61ms      2217  137.85us  2.9670us  237.76ms  cudaMalloc
                    0.17%  58.757ms      2217  26.502us  2.5750us  201.02us  cudaFree
                    0.02%  8.1228ms       604  13.448us  7.4600us  111.18us  cudaLaunchKernel
                    0.00%  1.0535ms         2  526.74us  507.15us  546.34us  cuDeviceTotalMem
                    0.00%  957.07us       202  4.7370us     317ns  194.11us  cuDeviceGetAttribute
                    0.00%  120.40us         2  60.201us  47.745us  72.658us  cuDeviceGetName
                    0.00%  20.832us         2  10.416us  5.3880us  15.444us  cuDeviceGetPCIBusId
                    0.00%  5.3360us         3  1.7780us     447ns  3.1300us  cuDeviceGetCount
                    0.00%  4.2300us         4  1.0570us     517ns  1.5000us  cuDeviceGet
                    0.00%  1.6150us         2     807ns     749ns     866ns  cuDeviceGetUuid
```

Fig. 6. Results of CUDA version with different Number of Particles



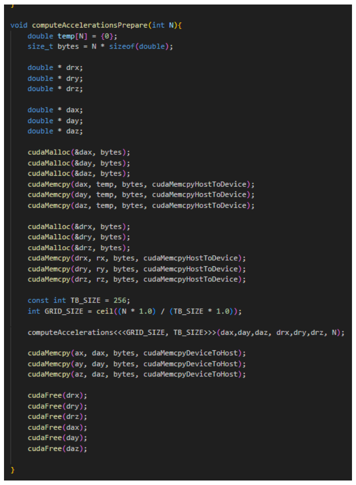
```
__device__ double ourAtomicAdd(double* address, double val)
{
    unsigned long long int* address_as_ull =
                            (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(val +
                            __longlong_as_double(assumed)));

    // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
    } while (assumed != old);

    return __longlong_as_double(old);
}

__device__ double ourAtomicMinus(double* address, double val){
    unsigned long long int* address_as_ull =
                            (unsigned long long int*)address;
    unsigned long long int old = *address_as_ull, assumed;

    do {
        assumed = old;
        old = atomicCAS(address_as_ull, assumed,
                        __double_as_longlong(__longlong_as_double(assumed) - val));

    // Note: uses integer comparison to avoid hang in case of NaN (since NaN != NaN)
    } while (assumed != old);

    return __longlong_as_double(old);
}
```

Fig. 7. ourAtomicAdd e ourAtomicMinus



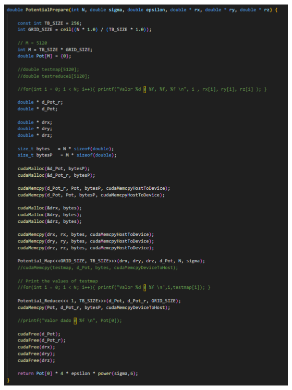
```
__global__ void computeAccelerations( double* dax, double* day, double* daz, double* drx, double* dry, double* drz, int N){
    int j;
    double f, rSqd;
    double rij0, rij1, rij2;
    double inv3, inv7, inv2;
    int i = threadIdx.x + (blockIdx.x * blockDim.x);

    if( i < N - 1){

        for (j = i + 1; j < N; j++){

            // initialize r^2 to zero
            rSqd = 0;

            //  component-by-componenent position of i relative to j
            rij0 = (drx[i] - drx[j]);
            rij1 = (dry[i] - dry[j]);
            rij2 = (drz[i] - drz[j]);

            //  sum of squares of the components
            rSqd += power_cuda(rij0, 2);
            rSqd += power_cuda(rij1, 2);
            rSqd += power_cuda(rij2, 2);

             //  From derivative of Lennard-Jones with sigma and epsilon set equal to 1 in natural units!
            inv3 =  power_cuda(rSqd, 3);
            inv2 =  power_cuda(inv3, 2) * rSqd;
            inv7 = (2 - inv3)/inv2;
            f    = 24 * inv7;

            dax[i] += rij0 * f;
            day[i] += rij1 * f;
            daz[i] += rij2 * f;

            __syncthreads();

        dax[j] += rij0 * f;
            day[j] += rij1 * f;
            daz[j] += rij2 * f;
    }
   }
}
```

Fig. 8. computeAccelarations CUDA Alternative



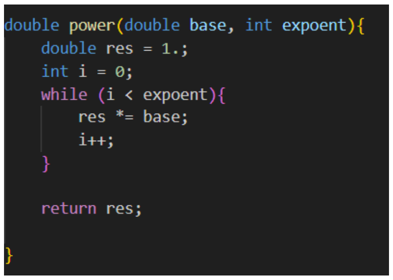
```
}

void computeAccelerationsPrepare(int N){
    double temp[N] = {0};
    size_t bytes = N * sizeof(double);

    double * drx;
    double * dry;
    double * drz;

    double * dax;
    double * day;
    double * daz;

    cudaMalloc(&dax, bytes);
    cudaMalloc(&day, bytes);
    cudaMalloc(&daz, bytes);
    cudaMemcpy(dax, temp, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(day, temp, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(daz, temp, bytes, cudaMemcpyHostToDevice);

    cudaMalloc(&drx, bytes);
    cudaMalloc(&dry, bytes);
    cudaMalloc(&drz, bytes);
    cudaMemcpy(drx, rx, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(dry, ry, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(drz, rz, bytes, cudaMemcpyHostToDevice);

    const int TB_SIZE = 256;
    int GRID_SIZE = ceil((N * 1.0) / (TB_SIZE * 1.0));

    computeAccelerations<<<GRID_SIZE, TB_SIZE>>>(dax,day,daz, drx,dry,drz, N);

    cudaMemcpy(ax, dax, bytes, cudaMemcpyDeviceToHost);
    cudaMemcpy(ay, day, bytes, cudaMemcpyDeviceToHost);
    cudaMemcpy(az, daz, bytes, cudaMemcpyDeviceToHost);

    cudaFree(drx);
    cudaFree(dry);
    cudaFree(drz);
    cudaFree(dax);
    cudaFree(day);
    cudaFree(daz);

}
```

Fig. 9. computeAccelarationsPrepare



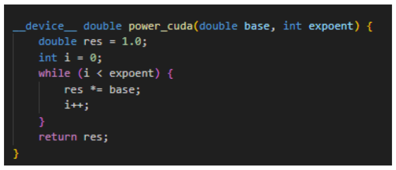
```
double PotentialPrepare(int N, double sigma, double epsilon, double * rx, double * ry, double * rz) {

    const int TB_SIZE = 256;
    int GRID_SIZE = ceil((N * 1.0) / (TB_SIZE * 1.0));

    // M = 5120
    int M = TB_SIZE * GRID_SIZE;
    double Pot[M] = {0};

    //double testmap[5120];
    //double testreduce1[5120];

    //for(int i = 0; i < N; i++){ printf("Valor %d [illegible] %f, %f, %f \n", i , rx[i], ry[i], rz[i] ); }

    double * d_Pot_r;
    double * d_Pot;

    double * drx;
    double * dry;
    double * drz;

    size_t bytes   = N * sizeof(double);
    size_t bytesP   = M * sizeof(double);

    cudaMalloc(&d_Pot, bytesP);
    cudaMalloc(&d_Pot_r, bytesP);

    cudaMemcpy(d_Pot_r, Pot, bytesP, cudaMemcpyHostToDevice);
    cudaMemcpy(d_Pot, Pot, bytesP, cudaMemcpyHostToDevice);

    cudaMalloc(&drx, bytes);
    cudaMalloc(&dry, bytes);
    cudaMalloc(&drz, bytes);

    cudaMemcpy(drx, rx, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(dry, ry, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(drz, rz, bytes, cudaMemcpyHostToDevice);

    Potential_Map<<<GRID_SIZE, TB_SIZE>>>(drx, dry, drz, d_Pot, N, sigma);
    //cudaMemcpy(testmap, d_Pot, bytes, cudaMemcpyDeviceToHost);

    // Print the values of testmap
    //for(int i = 0; i < N; i++){ printf("Valor %d [illegible] %f \n",i,testmap[i]); }

    Potential_Reduce<<< 1, TB_SIZE>>>(d_Pot, d_Pot_r, GRID_SIZE);
    cudaMemcpy(Pot, d_Pot_r, bytesP, cudaMemcpyDeviceToHost);

    //printf("Valor dado [illegible] %f \n", Pot[0]);

    cudaFree(d_Pot);
    cudaFree(d_Pot_r);
    cudaFree(drx);
    cudaFree(dry);
    cudaFree(drz);

    return Pot[0] * 4 * epsilon * power(sigma,6);
}
```

Fig. 10. Potencial Prepare



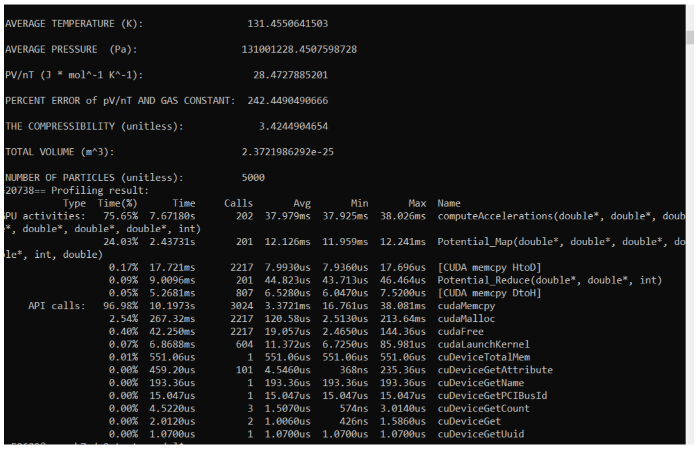
```
double power(double base, int expoent){
    double res = 1.;
    int i = 0;
    while (i < expoent){
        res *= base;
        i++;
    }

    return res;

}
```

Fig. 11. Power



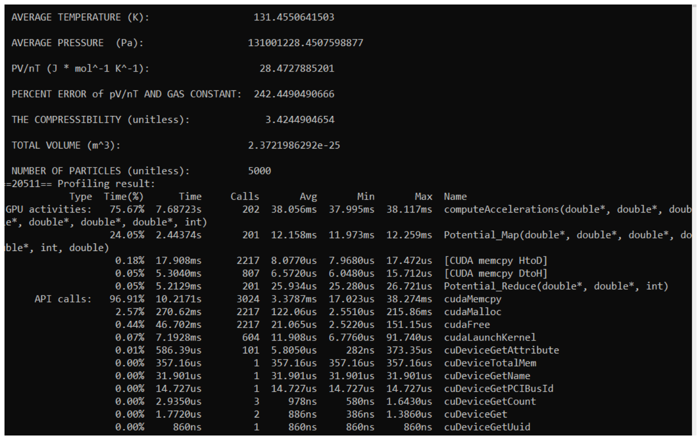
```
__device__ double power_cuda(double base, int expoent) {
    double res = 1.0;
    int i = 0;
    while (i < expoent) {
        res *= base;
        i++;
    }
    return res;
}
```

Fig. 12. Power Cuda



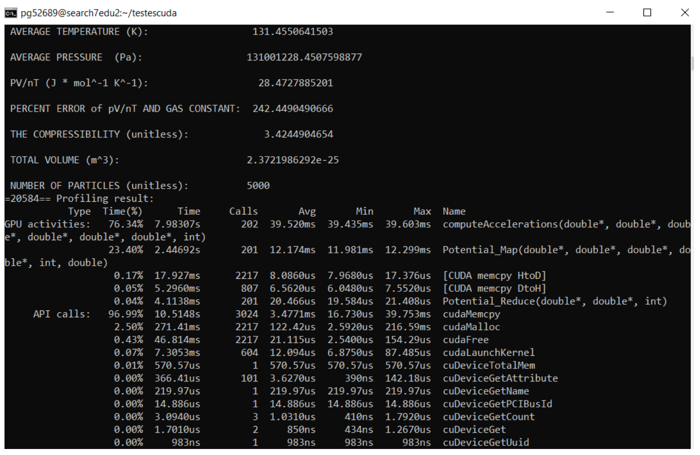
```
AVERAGE TEMPERATURE (K):                 131.4550641503

AVERAGE PRESSURE  (Pa):                 131001228.4507598728

PV/nT (J * mol^-1 K^-1):                  28.4727885201

PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

THE COMPRESSIBILITY (unitless):            3.4244904654

TOTAL VOLUME (m^3):                     2.3721986292e-25

NUMBER OF PARTICLES (unitless):         5000
20738== Profiling result:
          Type  Time(%)      Time     Calls       Avg       Min       Max  Name
PU activities:   75.65%  7.67180s       202  37.979ms  37.925ms  38.026ms  computeAccelerations(double*, double*, doub
*, double*, double*, double*, int)
                 24.03%  2.43731s       201  12.126ms  11.959ms  12.241ms  Potential_Map(double*, double*, double*, do
le*, int, double)
                  0.17%  17.721ms      2217  7.9930us  7.9360us  17.696us  [CUDA memcpy HtoD]
                  0.09%  9.0096ms       201  44.823us  43.713us  46.464us  Potential_Reduce(double*, double*, int)
                  0.05%  5.2681ms       807  6.5280us  6.0470us  7.5200us  [CUDA memcpy DtoH]
     API calls:  96.98%  10.1973s      3024  3.3721ms  16.761us  38.081ms  cudaMemcpy
                  2.54%  267.32ms      2217  120.58us  2.5130us  213.64ms  cudaMalloc
                  0.40%  42.250ms      2217  19.057us  2.4650us  144.36us  cudaFree
                  0.07%  6.8688ms       604  11.372us  6.7250us  85.981us  cudaLaunchKernel
                  0.01%  551.06us         1  551.06us  551.06us  551.06us  cuDeviceTotalMem
                  0.00%  459.20us       101  4.5460us     368ns  235.36us  cuDeviceGetAttribute
                  0.00%  193.36us         1  193.36us  193.36us  193.36us  cuDeviceGetName
                  0.00%  15.047us         1  15.047us  15.047us  15.047us  cuDeviceGetPCIBusId
                  0.00%  4.5220us         3  1.5070us     574ns  3.0140us  cuDeviceGetCount
                  0.00%  2.0120us         2  1.0060us     426ns  1.5860us  cuDeviceGet
                  0.00%  1.0700us         1  1.0700us  1.0700us  1.0700us  cuDeviceGetUuid
```

Fig. 13. Results of changing to 64 threads



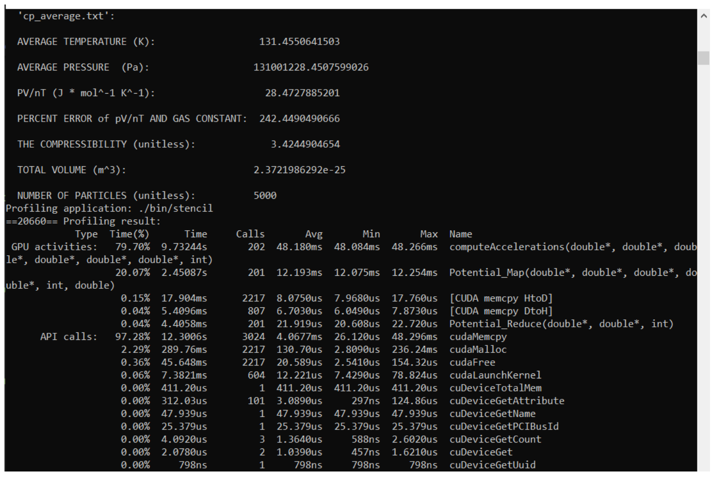
```
 AVERAGE TEMPERATURE (K):                 131.4550641503

 AVERAGE PRESSURE  (Pa):                 131001228.4507598877

 PV/nT (J * mol^-1 K^-1):                  28.4727885201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

 THE COMPRESSIBILITY (unitless):             3.4244904654

 TOTAL VOLUME (m^3):                       2.3721986292e-25

 NUMBER OF PARTICLES (unitless):          5000
=20511== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:   75.67%  7.68723s       202  38.056ms  37.995ms  38.117ms  computeAccelerations(double*, double*, doub
e*, double*, double*, double*, int)
                  24.05%  2.44374s       201  12.158ms  11.973ms  12.259ms  Potential_Map(double*, double*, double*, do
ble*, int, double)
                   0.18%  17.908ms      2217  8.0770us  7.9680us  17.472us  [CUDA memcpy HtoD]
                   0.05%  5.3040ms       807  6.5720us  6.0480us  15.712us  [CUDA memcpy DtoH]
                   0.05%  5.2129ms       201  25.934us  25.280us  26.721us  Potential_Reduce(double*, double*, int)
     API calls:   96.91%  10.2171s      3024  3.3787ms  17.023us  38.274ms  cudaMemcpy
                   2.57%  270.62ms      2217  122.06us  2.5510us  215.86ms  cudaMalloc
                   0.44%  46.702ms      2217  21.065us  2.5220us  151.15us  cudaFree
                   0.07%  7.1928ms       604  11.908us  6.7760us  91.740us  cudaLaunchKernel
                   0.01%  586.39us       101  5.8050us     282ns  373.35us  cuDeviceGetAttribute
                   0.00%  357.16us         1  357.16us  357.16us  357.16us  cuDeviceTotalMem
                   0.00%  31.901us         1  31.901us  31.901us  31.901us  cuDeviceGetName
                   0.00%  14.727us         1  14.727us  14.727us  14.727us  cuDeviceGetPCIBusId
                   0.00%  2.9350us         3     978ns     580ns  1.6430us  cuDeviceGetCount
                   0.00%  1.7720us         2     886ns     386ns  1.3860us  cuDeviceGet
                   0.00%     860ns         1     860ns     860ns     860ns  cuDeviceGetUuid
```

Fig. 14. Results of changing to 128 threads



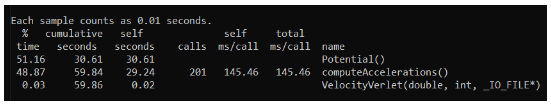
```
pg52689@search7edu2:~/testescuda
 AVERAGE TEMPERATURE (K):                 131.4550641503

 AVERAGE PRESSURE  (Pa):                 131001228.4507598877

 PV/nT (J * mol^-1 K^-1):                  28.4727885201

 PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

 THE COMPRESSIBILITY (unitless):             3.4244904654

 TOTAL VOLUME (m^3):                       2.3721986292e-25

 NUMBER OF PARTICLES (unitless):          5000
=20584== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
GPU activities:   76.34%  7.98307s       202  39.520ms  39.435ms  39.603ms  computeAccelerations(double*, double*, doub
e*, double*, double*, double*, int)
                  23.40%  2.44692s       201  12.174ms  11.981ms  12.299ms  Potential_Map(double*, double*, double*, do
ble*, int, double)
                   0.17%  17.927ms      2217  8.0860us  7.9680us  17.376us  [CUDA memcpy HtoD]
                   0.05%  5.2960ms       807  6.5620us  6.0480us  7.5520us  [CUDA memcpy DtoH]
                   0.04%  4.1138ms       201  20.466us  19.584us  21.408us  Potential_Reduce(double*, double*, int)
     API calls:   96.99%  10.5148s      3024  3.4771ms  16.730us  39.753ms  cudaMemcpy
                   2.50%  271.41ms      2217  122.42us  2.5920us  216.59ms  cudaMalloc
                   0.43%  46.814ms      2217  21.115us  2.5400us  154.29us  cudaFree
                   0.07%  7.3053ms       604  12.094us  6.8750us  87.485us  cudaLaunchKernel
                   0.01%  570.57us         1  570.57us  570.57us  570.57us  cuDeviceTotalMem
                   0.00%  366.41us       101  3.6270us     390ns  142.18us  cuDeviceGetAttribute
                   0.00%  219.97us         1  219.97us  219.97us  219.97us  cuDeviceGetName
                   0.00%  14.886us         1  14.886us  14.886us  14.886us  cuDeviceGetPCIBusId
                   0.00%  3.0940us         3  1.0310us     410ns  1.7920us  cuDeviceGetCount
                   0.00%  1.7010us         2     850ns     434ns  1.2670us  cuDeviceGet
                   0.00%     983ns         1     983ns     983ns     983ns  cuDeviceGetUuid
```

Fig. 15. Results of changing to 512threads



```
  'cp_average.txt':

  AVERAGE TEMPERATURE (K):                 131.4550641503

  AVERAGE PRESSURE  (Pa):                 131001228.4507599026

  PV/nT (J * mol^-1 K^-1):                  28.4727885201

  PERCENT ERROR of pV/nT AND GAS CONSTANT:  242.4490490666

  THE COMPRESSIBILITY (unitless):             3.4244904654

  TOTAL VOLUME (m^3):                       2.3721986292e-25

  NUMBER OF PARTICLES (unitless):          5000
Profiling application: ./bin/stencil
==20660== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   79.70%  9.73244s       202  48.180ms  48.084ms  48.266ms  computeAccelerations(double*, double*, doub
le*, double*, double*, double*, int)
                   20.07%  2.45087s       201  12.193ms  12.075ms  12.254ms  Potential_Map(double*, double*, double*, do
uble*, int, double)
                    0.15%  17.904ms      2217  8.0750us  7.9680us  17.760us  [CUDA memcpy HtoD]
                    0.04%  5.4096ms       807  6.7030us  6.0490us  7.8730us  [CUDA memcpy DtoH]
                    0.04%  4.4058ms       201  21.919us  20.608us  22.720us  Potential_Reduce(double*, double*, int)
      API calls:   97.28%  12.3006s      3024  4.0677ms  26.120us  48.296ms  cudaMemcpy
                    2.29%  289.76ms      2217  130.70us  2.8090us  236.24ms  cudaMalloc
                    0.36%  45.648ms      2217  20.589us  2.5410us  154.32us  cudaFree
                    0.06%  7.3821ms       604  12.221us  7.4290us  78.824us  cudaLaunchKernel
                    0.00%  411.20us         1  411.20us  411.20us  411.20us  cuDeviceTotalMem
                    0.00%  312.03us       101  3.0890us     297ns  124.86us  cuDeviceGetAttribute
                    0.00%  47.939us         1  47.939us  47.939us  47.939us  cuDeviceGetName
                    0.00%  25.379us         1  25.379us  25.379us  25.379us  cuDeviceGetPCIBusId
                    0.00%  4.0920us         3  1.3640us     588ns  2.6020us  cuDeviceGetCount
                    0.00%  2.0780us         2  1.0390us     457ns  1.6210us  cuDeviceGet
                    0.00%     798ns         1     798ns     798ns     798ns  cuDeviceGetUuid
```

Fig. 16. Results of changing to 1024 threads



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 51.16     30.61    30.61                             Potential()
 48.87     59.84    29.24      201   145.46   145.46  computeAccelerations()
  0.03     59.86     0.02                             VelocityVerlet(double, int, _IO_FILE*)
```

Fig. 17. gprof for sequential version