Parallel Computing
Work Assignment Phase 3

Gonçalo Costa
PG55944
Universidade do Minho
Braga, Portugal

Marta Rodrigues
PG55982
Universidade do Minho
Braga, Portugal

Pedro Lopes
PG55992
Universidade do Minho
Braga, Portugal

***Abstract*—This report covers the implementation of optimizations in a 3D fluid dynamics simulation across three phases. The first phase focused on improving execution time through various code-level enhancements, the second phase introduced shared memory parallelism using OpenMP for strong scalability and the final phase explores parallel programming on GPUs using CUDA, further improving performance and demonstrating the benefits of hetero-geneous computing.**

***Index terms*—3D Fluid Solver, Parallelism, OpenMP, CUDA**

I. Introduction

The main goal of this project is to analyse a range of strategies and tools designed to optimizing a given program,with the ultimate goal of enhancing its performance.

This paper provides a detailed examination of the strategies applied across the project's various stages, from the optimization of a sequential code to the implementation of a efficient parallel version using accelerators, and an evaluation of the resulting performance improvements. As the provided code underwent modifications during different phases of the project, this document is structured into distinct sections, each addressing a specific stage of development.

II. WA1 - Optimizations on Sequential Code

The first phase of this project consisted of implementing optimization techniques to a single threaded version of a provided code to reduce execution time.

Through the analysis of the source code using the *perf* command, we identified performance metrics, such as CPU cycles, cache misses and more and found the program's execution time to be approximately **26.92** seconds.

The functions that contributed the most to the total execution time were: *vel_step* (**83,30%**, complexity: O(M×N×O)), *diffuse* (**33.20%**, complexity: O(M×N×O)), *project* (**57,88%**, complexity: O(M×N×O)) and most importantly *lin_solve* (**86,81%**, complexity: O(M×N×O)). This information was obtained with *gprof* tool, as shown in the following figure:

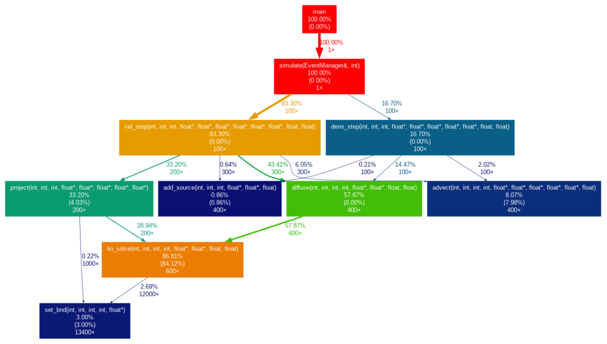
main
100.00%
(0.00%)
100.00%
1×
simulate(EventManager&, int)
100.00%
(0.00%)
1×
83.30%
100×
16.70%
100×
vel_step(int, int, int, float*, float*, float*, float*, float*, float*, float, float)
83.30%
(0.00%)
100×
dens_step(int, int, int, float*, float*, float*, float*, float*, float, float)
16.70%
(0.00%)
100×
33.20%
200×
0.64%
300×
43.41%
300×
6.05%
300×
0.21%
100×
14.47%
100×
2.02%
100×
project(int, int, int, float*, float*, float*, float*, float*)
33.20%
(4.03%)
200×
add_source(int, int, int, float*, float*, float)
0.86%
(0.86%)
400×
diffuse(int, int, int, int, float*, float*, float, float)
57.87%
(0.00%)
400×
advect(int, int, int, int, float*, float*, float*, float*, float*, float)
8.07%
(7.98%)
400×
28.94%
200×
57.87%
400×
0.22%
1000×
lin_solve(int, int, int, int, float*, float*, float, float)
86.81%
(84.12%)
600×
2.69%
12000×
set_bnd(int, int, int, int, float*)
3.00%
(3.00%)
13400×

Fig. 1: Initial Performance Graph for WA1 created by *gprof2dot*.

A. *Code Optimization*

All optimization efforts aimed to improve execution time, reduce the number of instructions executed and minimize program execution cycles. The changes made and the resulting run time after each step are listed below:

- Checking and Improving Macros : After the initial code analysis and improving code readability, we started doing the performance upgrades to the code by analysing existing macros and optimizing wherever we could improve anything significant. **RUN TIME: 8.6 s**

- Loop Permutation : As seen during the classes, to enhance memory and cache access, we improved spatial data locality and reduced memory dependencies. By reorganizing data access in triple or more nested loops, we brought the different data values closer in memory and, in doing so, they are more easily accessed between loop iterations, reducing cache misses massively and cutting runtime by a second. **RUN TIME: 7.5 s**

- Data Processing through Blocks : It became clear after the previous modification that there was still room for improvement to the data and cache accesses in the loops. Thinking about the way the code works, it essentially processes a 3D cube of data throughout its 3 axis and, during this step of the optimization, we noticed that the code was processing the whole block of data all at once, which was overflowing the cache. So, in order to put the cache to better use, we divided the data into smaller cubes, big enough to fit in the cache, greatly improving space/time locality and reducing runtime. However, the size of the blocks needs to be carefully calculated, which will be discussed further below. **RUN TIME: 5.38 s**

- Division Handling and Other Improvements : From this point, optimization options to reduce runtime, cache misses, and clock cycles became limited. We focused on division operations, as they are CPU-intensive. We discovered a division within the inner loop of the lin_solve function, significantly impacting performance. To optimize, we replaced divisions with precalculated values:

```
float div = 1/c;
x[idx] = (x0[idx] + a * (x_im1 + x_ip1 +
x_jm1 + x_jp1 + x_km1 + x_kp1)) * div;
```

To reduce the number of calls to the IX macro , we calculated the difference between IX(i, j, k) and its neighbors along each axis. This minimized multiplication operations by replacing them with addition and allowed us to call IX once per J iteration instead of every I iteration.

Unfortunately, we couldn't optimize IX to be called only every K iteration due to discrepancies in expected results, and the need to avoid repeating *idx* and *idx-1* values between iterations.

Finally, we noticed that the function *set_bnd* was running an *if* statement in every loop iteration. To optimize this, we pre-calculated the three boolean values for these *if* expressions before entering the loops. This allowed us to execute the if statements only once during the function's execution. The calculation of one of the boolean values can be seen below :

```
  auto neg_mask = (b == 3) ? -1.0F : 1.0F;
```

RUN TIME: 5.05 s

- Change in Block Size : After several optimizations, we struggled to find further improvements in the code. However, by adjusting the block size from 8 to 4, we saw one of the most significant runtime reductions. The smaller blocks brought the data closer together, improving access speed and reducing cache overhead. **RUN TIME: 4.16 s**

B. Results

Finally, with all the improvements mentioned before, we obtained the following statistics:

Code	**Statistics**		
	Instructions (u)	***Cycles (u)***	***Time (s)***
Original	166,608,012,110	51,532,633,271	26.915
Optimized	21,055,041,190	13,430,363,565	4.165

The optimizations implemented resulted in a speedup of approximately 6.5.

III. WA2 - Parallelism with OpenMP

The goal of this phase was to explore parallelism techniques for the previously optimized sequential version of the code.

We began by replacing the *lin_solve* function with a more efficient red-black solver, modifying it to closely align with the approach used in the first phase, as the previous one was very difficult to parallelize. Additionally, we increased the data size from SIZE = 42 to SIZE = 84.

Using the *perf* command, we profiled the code to detect the following hotspots, with the associated overhead:

Function	**Overhead**
lin_solve	68,87%
project	16,79%
advect	13,09%

Table II: Results of the command perf

A. Implementations

We analysed and optimized the functions listed above using *#pragma* directives to minimize overhead and enhance performance. Below are the specific strategies employed for each function:

- ***lin_solve***: The red and black loops in this solver were parallelized. Since both loops are nested, we applied the collapse(2) directive. The value 2 was chosen instead of 3 because the innermost loop depends on the outer loops, violating an important premise of this directive: the independence between the loops, to linearize them.

  To ensure there were no data races with *max_c*, we utilized the *reduction(max:max_c)* directive, which creates private copies of this value for each thread, to prevent any risk of threads reading wrong values, ensuring thread safety.

  Similarly, variables like *old_x*, *IDX*, and *change* were declared inside the loops to ensure thread-specific access, which could also be achieved by using the *private* directive.

  We also used the directive *schedule(static)* to balance the workload of the linearized iterations across the threads available.
- ***advect***: We only used the for *collapse(3) schedule(static)* directive to parallelize the function. This time, we figured that value 3 was the best by testing different values and checking that all loops were independent.
- ***project***: We only parallelized the loops through the same strategy mentioned in the previous *advect* function, leaving the function calls outside the loops sequential because *lin_solve* was already parallelized and *set_bnd* was problematic, so we ended up leaving this last function as is.
- ***add_source***: Although this function is called less frequently, we used *parallel for schedule(static)* for maximum parallelization, even if it didn't improve the performance as much as the other functions.

Other functions, such as set_bnd, were left unchanged due to either parallelization difficulties or negligible performance improvements.

B. Results

We evaluated the performance of the code across multiple threads, calculating the achieved speed-up and comparing it to the theoretical ideal speed-up predicted by Amdahl's Law. resulting in the graph provided.

#T	TExec	SpeedUp	#I$\times 10^9$	#CC$\times 10^9$	#M$\times 10^6$	CPI
1	14.4301	1	81.99	45.49	1.89	0.6
2	13.5818	1.06	82.33	83.65	1.92	1.0
4	7.1313	2.02	82.73	85.16	1.92	1.0
8	3.9483	3.65	83.21	88.23	1.92	1.1
16	2.1808	6,62	84.65	95.59	1.95	1.1
20	1.818	7,94	85.46	99.26	1.99	1.2
22	1.6776	9,82	85.72	100.64	1.99	1.2
32	3.369	4,28	133.83	285.18	2.10	2.1
48	2.8302	5,1	115.69	370.76	2.11	3.2

Table III: Results obtained using different number of threads

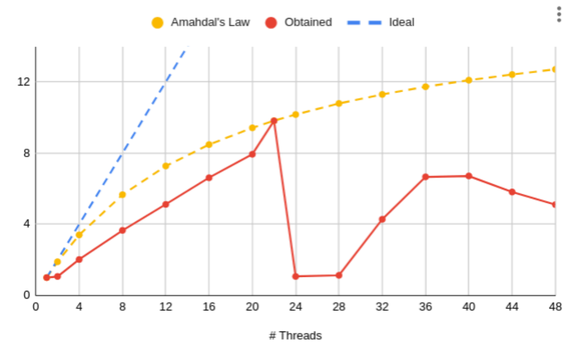

Fig. 2: Speed-Up Graph using *srun* with *cpus-per-task* = *threads*

Our implementation outperformed the sequential code, achieving a mostly linear speed-up. However, the parallelization fell short of the ideal speed-up for a fully parallelizable algorithm. Notably, there was an unusual sharp drop in performance between 24 and 28 threads before returning to normal behaviour. Interestingly, running the script *run.sh* through every thread with 48 CPUs per task produced a better result.

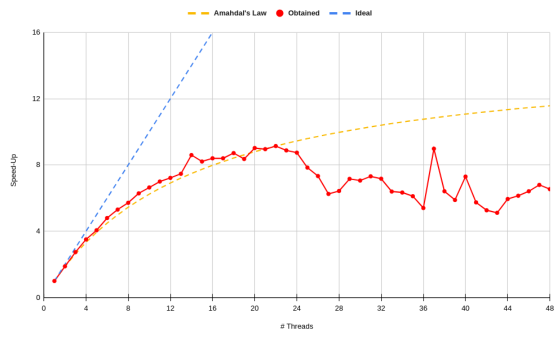

Fig. 3: Speed-Up Graph using the script through 48 threads

IV. WA3 - GPU's Parallelism

In this final phase, after focusing on the sequential and OpenMP-parallelized versions of the base code in previous phases, we focused on optimizing the code using GPU acceleration through CUDA by executing the majority of the workload regarding floating-point operations and others in the GPU. CUDA, an NVIDIA API, allows massive parallelism through the use of the thousands of cores available in modern GPUs, making it ideal for computationally intensive tasks like fluid simulations. Code can be executed by the GPUs through the use and development of global kernels, which utilize thread indexing to handle operations that require multiple iterations, such as a triple-nested loop that determines the sum of 3 arrays.

We began by identifying the computational hotspots and loops with a big workload, especially those involving floating-point operations. With this done, we started the development of the kernels for these specific situations that we found throughout the code. This process required careful consideration of how to effectively translate these code sections from CPU to GPU, evaluating whether the transformation effort was justified, always ensuring the overhead was kept to a minimum.

A. Memory Management between CPU (Host) and GPU (Device)

As previously mentioned, one of the most important aspects of CUDA programming is the memory exchanges between the CPU and GPU. Since the GPU and CPU have distinct memory storage, in order for the code to execute in the GPU, we must copy the input memory from the host (CPU) to the device memory (GPU). With these memory exchanges possible, there's also the possibility of creating significant overhead, caused by excessive and unnecessary exchanges between both devices.

To create efficient memory management between the CPU and GPU throughout the program's execution, we first had to ensure that the use of the *cudaMalloc()* and *cudaFree()* functions was set to a minimum while ensuring the correct memory usage and exchanges in the GPU.

To achieve this, we decided, as soon as the input arrays are allocated and set to zero in the *allocate_data()* and *clear_data()* functions, since these are ready to be processed by the *simulate()* function, to immediately copy these arrays to the device memory and, this way, by ensuring that the *vel_step()* and *dens_step()* functions work with the pointers allocated in the GPU, we only need to make N memory exchanges, being N the number of data pointers needed in the GPU for the kernels to execute.

For the *apply_events()* function, despite handling small event arrays (length 1 or 0), we developed a kernel to process event data with length 1 on the GPU, further leveraging the GPU's parallel processing capabilities.

After completing the *simulate()* function, only the density data is copied back to the CPU to calculate the total density and display the result, following the approach used in earlier phases.

B. *Solver Implementation*

With the changes regarding the memory exchanges between CPU and GPU and some other changes made to the *main.cu*, we started to work on implementing the solver and adapt it, in order for it to only use data that is located in the GPU.

Firstly, all loops that involved heavy operations were translated into specific kernels that were called by the CPU. Examples include the *add_source_kernel()*, the loop1 and loop2 project kernels, the advect_kernel() and, mainly, the *lin_solve_kernel()*, which was the most important to develop due to it being the largest computational hotspot. Each kernel was designed to operate on data blocks, with each thread within a block handling a specific data element.

Extensive tests were conducted to determine the optimal number of threads per block and the number of blocks, based on the data size and the GPU architecture. These tests aimed to maximize throughput by finding the configuration that provided the best performance.

The kernels were responsible for updating the velocity fields (*du, dv, dw*) and density field (*ddens*) during each simulation timestep. To enhance efficiency, memory access patterns were optimized. Specifically, we took advantage of the high throughput of the GPU's global memory for the simulation data and ensured that threads accessed memory in a coalesced manner, minimizing memory latency.

C. *Optimizations to the Kernels*

In the CUDA version of the simulation, several optimizations were introduced to improve performance and thread execution efficiency.

The *lin_solve()* function was the most important optimization needed for this phase, so we immediately started working on it. As mentioned previously, we first converted the loops for each color into a specific kernel, with a boolean argument distinguishing the colors.

Despite these changes, after almost all the loops in the code were turned into kernels, in order for them to run in the GPU, and the *lin_solve* ones too, the *lin_solve* remained a big hotspot due to the incorrect implementation of the *max_c* check in the kernel. So, to fix this, we started researching about reduction in CUDA Programming and found a way to implement a **Multiple Elements per Thread Reduction** method, based on NVIDIA's guidelines and, then, adapting the reduction from handling the sum of elements to the check of the max value between elements of an array, that were added by each thread in each kernel, red or black colored. The way this reduction works is, firstly, we must call the *max_reduce_kernel()* twice: The first one is going to reduce all the elements from different blocks into 1 with all the contenders to the maximum and, in the second call, the reduction of the final block is done and the max value is found.

This way, after both *lin_solve_kernel()* calls fill the array *changes_d* with the values from `fabsf(x[idx] - old_x)`, the kernel *launch_max_reduce*() is called and writes the max value within the changes_d array into max_c, which is then verified in the while condition of the do-while loop. Even though this reduction improves the performance of the lin_solve kernel, we need to copy the value from max_c from the device memory to the host, which leads to a memory exchange for each iteration of the do-while loop for each lin_solve call, making the *CudaMemcpy* one of the hotspots in API calls.

The 2 loops in the *project()* function into 2 separate kernels and, to decrease the number of CudaMallocs, CudaFree and CudaMemcpys calls, we expanded the number of arguments in the *vel_step* and *dens_step* functions, in order for them to use the arguments needed for some kernels in the GPU and bring them to said kernels, just like we can see in the *diffuse()* and *project()* functions, for example, which include calls for the *lin_solve_kernel()*, or in other words, the last kernel that we want making memory exchanges between the Host and Device memories.

Finally, when trying to improve the *lin_solve_kernel*'s performance, we discovered that there was a better way of finding out when an iteration of the do-while loop had found out a value bigger than the tolerance allowed. Basically, instead of storing all the change values into an array and, then, after the execution of both red and black kernels, launching a reduction into this array to figure out the biggest of all of them and, then, comparing this value with the tolerance, we copied a boolean value, used as a flag, into the GPU to check for convergence and, in every iteration of the do-while, in case the change value exceeded the tolerance, this flag is altered and the flag is checked after the kernels' executions, instead of launching a reduction, which ended up improving the *lin_solve* kernel performance a lot.

To provide a comprehensive view of the optimization techniques employed, we decided to include both versions in our code. Below, we present test results comparing the performance of these two approaches.

D. *Thread Configuration: Threads per Block*

After completing the kernel development, we focused on determining the optimal number of threads per block, a critical decision in CUDA programming for achieving peak performance. In theory, threads within a block share resources like shared memory and are executed by the same Streaming Multiprocessor (SM). Choosing a proper configuration ensures efficient GPU resource utilization, minimizes idle time, and avoids bottlenecks.

We conducted extensive tests with thread counts ranging from 64 to 1024, selecting the number of Threads per Block that yielded the best execution time among them. Based on these tests, we concluded that the best and optimal number of Threads per Block is **128**, as we can see in the tests below.

As for the number of Blocks in the Grid, we generally followed a consistent formula, except for the *lin_solve_kernel()*, which used half the Blocks for X dimension, as the kernel was called separately for each half of the data (Red and Black).

V. Tests Analysis

With all the implementation done, we proceeded to test the program to evaluate its performance and, mainly, how the execution time works under various circumstances.

A. Strong Scalability

To test the strong scalability of this implementation, we ran the code with different number of Threads per Block, all with SIZE = 168. The results, summarized in Tables IV and V, show the performance for the version without reduction (using a flag check instead) and the version with reduction, respectively.

#Threads per Block	**Execution Time (*s*))**
64	13.955
128	13.953
256	14.420
512	14.277
1024	15.342

Table IV: Execution time evolution with the number of threads per block without reduction and flag check

#Threads per Block	**Execution Time (*s*))**
64	20.428
128	17.365
256	18.543
512	19.148
1024	19.854

Table V: Execution time evolution with the number of threads per block with reduction of max_c

With all the values in place, we realized that the best version, in terms of overall execution time was the one without reduction, that used flag check instead. Additionally, we confirmed that the optimal number of Threads per Block is **128**.

B. Weak Scalability

For this type of scalability, we tested the code with 128 Threads per Block across different problem sizes, as shown in Figures 4 and 5 along with their corresponding Tables VI and VII.Similar to the previous scalability analysis, each figure and table represents a different solution.

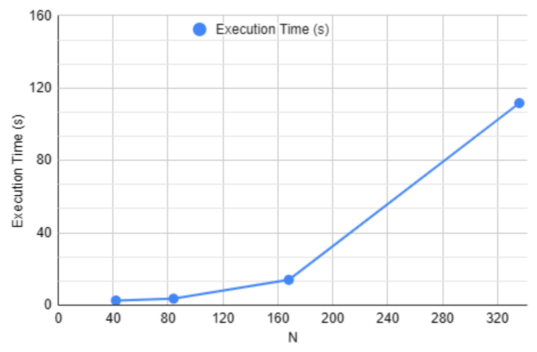

Fig. 4: Execution Time without reduction for different problem sizes with 128 Threads.

SIZE	**Execution Time (*s*))**
42	2.538
84	3.572
168	13.960
336	1m51.677

Table VI: Execution times of Fig. 4

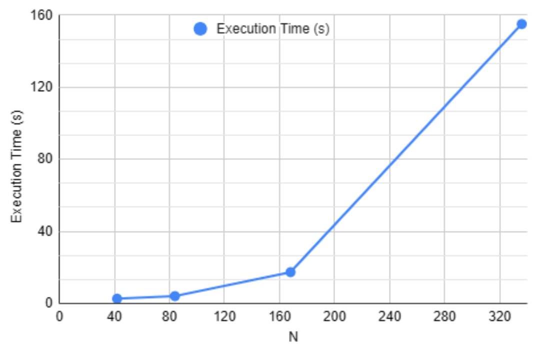

Fig. 5: Execution Time with reduction for different problem sizes with 128 Threads.

SIZE	**Execution Time (*s*))**
42	2.634
84	4.093
168	17.345
336	2m35.050

Table VII: Execution times of Fig. 5

C. Testing all other phases with SIZE = 168

While testing the performance during this final phase, since we were gonna write about all phases and its details, we decided to also test the WA1 and WA2 implementations with SIZE = 168. The results of these tests are presented in Tables IX and X.

#Threads	Execution Time (s))
1	149.59
2	135.36
4	70.76
8	38.73
12	27.45
16	21.91
19	19.80
20	18.69
21	18.33
22	17.56
24	16.64
28	16.72
32	14.84
36	14.91
40	17.12
44	16.80
48	54.20

Table VIII: Execution time evolution of WA2 implementation with SIZE = 168

#Threads	Execution Time (s))
1	297.91

Table IX: Execution time of WA1 implementation with SIZE = 168

After completing the tests, it became clear the WA1 isn't very efficient with larger problems, while the WA2 turned out to be very efficient and competent for handling bigger problem sizes.

VI. Conclusion

After analysing the statistics of each phase's performance with SIZE = 168 and other SIZEs, we can conclude that the WA1 version is only efficient for small problems, since it doesn't involve as much Parallel Programming concepts as the other WA's. As for the last 2, both these WA's proved to be quite competent in processing bigger problems, as we can see in the figures and table showed previously. Overall, the **best performance**, up to SIZE = 168, was achieved by the **WA3 without reduction, using the converged flag**, with **13.953s**.

In the first phase we gained valuable insights into ILP improvements, different compilation flags and their respective advantages and disadvantages, and learned about good memory organization and vectorization.

In the second phase, we learned to use OpenMP and Multi-Threading to our advantage, testing different quantities of threads to optimize code execution and, also, the good practices that we should follow while programming with OpenMP and Multi-Threading in general.

Finally, in this phase, we learned about and used the CUDA programming model, understanding its advantages and what we should look out for when programming with it. We also learned how the GPUs work and contribute in this model, creating an intriguing relationship with the CPU that significantly enhances code performance.

The different stages of this project provided us a great learning opportunity and a hands-on experience with a large variety of different tools and techniques that can be used to improve the performance od computationally intensive and heavy code through parallelism and good practices.

VII. Appendix

A. Parallelization performance results

In table III:

- #T - Number of threads
- TExec - Execution Time
- #I - Number of instructions $\times 10^9$
- #CC - Number of clock cycles $\times 10^9$
- #M - Number of Misses $\times 10^7$
- CPI - Cycles per Instruction

B. Amahdal's Law

The speed-up for p processes (*S(p)*) is calculated using Amdahl's Law, defined as:

$$S(p) = \frac{1}{f + \frac{1-f}{p}}$$

Where p is the number of threads and *f* is the serial fraction of workload. To determine *f*, we used:

$$f = \frac{\frac{1}{S} - \frac{1}{P}}{1 - \left(\frac{1}{P}\right)}$$

with $S = \dfrac{Tseq}{Tpar}$, where $Tseq$ is sequential execution time and $Tpar$ is the parallel execution time with P=22 threads, chosen for optimal performance.

This calculation yielded *f*=0.0591

Based on Amdahl's Law, the ideal speed-up for various thread counts was determined as follows:

#T	**1**	**2**	**4**	**8**	**16**	**20**	**22**	**32**	**48**
SpeedUp	1	1.89	3.41	5.71	8.6	9.58	9.99	11.53	13

Table X: Ideal Speed-Up for different number of threads