Work Assignment — Phase 3

João Luís Ferreira Magalhães
MS Computer Engineering
University of Minho
Braga, Portugal

Jorge Nuno Gomes Rodrigues
MS Computer Engineering
University of Minho
Braga, Portugal

Rodrigo Ferreira Gomes
MS Computer Engineering
University of Minho
Braga, Portugal

***Abstract*—This paper presents three phases of the optimization of a fluid dynamics simulator, starting by optimizing the code in a instruction level, then moving to the use of OpenMP for multi thread processing in the CPU, and finally through GPU acceleration using CUDA. Our last implementation features specialized kernels with a Red-Black solver approach and block-level reductions.**

***Index Terms*—Parallelism, CUDA, Threads**

I. Introduction

This paper presents a systematic approach to optimizing a fluid dynamics simulator through three distinct phases, each targeting different levels of parallelization and hardware acceleration. Our work begins with instruction-level optimizations, focusing on improving cache utilization, reducing redundant computations, and enabling better compiler optimizations. We then progress to exploiting thread-level parallelism using OpenMP for multi-core CPU execution, and finally implement GPU acceleration using CUDA to harness the parallel processing capabilities of graphics processors.

The remainder of this paper is organized as follows: Section 1 details our instruction-level optimizations and their impact on CPU performance. Section 2 explores our OpenMP implementation for multi-threaded execution on CPU architectures. Section 3 presents our CUDA-based GPU implementation, including kernels and optimization strategies for parallelism. Each section includes performance analyses and discussions of the challenges encountered and solutions developed.

II. Phase 1

A. Analysis Of The Original Program

The program without any optimizations, acquired the results exposed in table I under the entry "No change". These results serve as a starting point for our optimizations. From the profiling, we found out the *hot spot* was the function *lin_solve* with 84.55% of total execution time, that used four *for* nested loops for calculations over an *Array* . The other functions with significant percentage execution time were *advect* with 8.24%, *project* with 4.15% and finally *set_bnd* with 2.47%. With this profiling, our goal is clearly to minimize the execution time of the function *lin_solve*. We also decided to add the flags mentioned prior and measure those results, for the sake of comparison between the optimizations we would add to the code.

B. Optimizations

1) Separating Independent Computations: To allow the compiler to make space for parallelism, we should separate independent computations as much as we possibly can, manually. Considering the *hotspot* mentioned in the analysis, we found out that the operations are very much dependent on each other. Therefore, we separated the computation of the summation of the points and then performed the multiplication and division by the two constants. This reordering should allow the CPU to compute parts of the summation while data loads are performed in parallel. The results of the addiction of Optimization A can be found in table I. We expected a lower CPI value, which in reality kept the same, but the overall number of instructions and cycles decreased.

2) Redundancy avoidance: To calculate the index of the *x Array*, the code mostly uses a macro that returns the index value, given the *i,j and k* values. In some instances, we noticed that the macro is repeated for the exact same parameters. With this in mind, we can instantiate a temporary value that will reduce the number of times the macro is called. For the level two of optimization, as we can see on table I, the changes made the code perform worst. We were only able to obtain the expected results in the level 3 of optimization.

3) Loop Interchange: As referenced in this paper, the *hotspot* of the code is a function that contains four nested *for* loops. Each loop has a specific order that directly affects the calculations of the macro *IX*. Also, is relevant to mention that the order does not affect the result of the function, just the order in which the values are being calculated. With this in mind, it is important to have a look at the macro.

$$\mathrm{IX}(i, j, k) = i + (M + 2) \cdot j + (M + 2) \cdot (N + 2) \cdot k \quad (1)$$

Now its noticeable which variable (*i,j and k*) as the biggest impact in value produced, being clearly the index *k*. Therefore, an optimized order for the loops should be *k then j then i*, to minimize the jumps in the array, which will attenuate the number of misses in the cache. Like other optimizations, it can be applied in other sections of the code. The results of this optimization can be found in the table I. The results matched some of the expectations. We noticed a big decrease in the Cache Misses, but the number of Clock Cycles had increased

with the changes. It is important to mention that, for the third level of optimizations, there was a decrease in execution time, contrary to what we registered for the second level.

4) Loop Unrolling: Loop unrolling is a common optimization technique that can significantly improve performance by reducing branch overhead and increasing instruction-level parallelism. By unrolling loops, the compiler can generate more efficient machine code that executes multiple loop iterations in a single pass, minimizing the cost of loop control instructions such as increments, comparisons, and branches. In the context of our fluid simulator, the primary performance bottleneck is the *lin_solve* function, which contains a set of nested loops. Unrolling these loops can potentially yield substantial performance gains by allowing the CPU to execute more iterations concurrently and reducing the overall number of instructions. Also, we noticed that the memory accesses would be reduced, due to pre-fetching of the values for the second iteration.To implement loop unrolling, we modified the innermost loop of *lin_solve* to execute four iterations at a time, rather than a single iteration. This was achieved by duplicating the loop body and adjusting the loop indices accordingly. By unrolling the innermost loops, we can execute four iterations of the key calculations in a single pass, reducing the overall number of loop control instructions and allowing the CPU to better utilize its resources for parallel execution. This optimization, combined with the previous improvements, resulted in a significant performance boost, as shown in the optimization results table I, specially in the reduction of Clock Cycles.

5) Flags: The most efficient optimizations that reduced the biggest amount of execution time were the flags that we thought were appropriate for the compiling of the program. Initially, optimization level "-O2" was used, but switching to "-O3" improved instruction-level parallelism, reduced branching, and optimized loop transformations for better performance. The "-march=native" flag tailors the code to the CPU architecture, while "-funroll-loops" enhances parallelism by explicitly unrolling loops. The "-ffast-math" flag speeds up floating-point operations, trading some accuracy for performance. Finally, "-flto" boosts overall efficiency by optimizing function calls and memory access across multiple files. Results using these flags are shown in table I.

6) Replacing Macros for Standard Library Functions: A very simple improvement to the program is to swap some functions that already have a standard C implementation. In our program, we found a macro for calculating the maximum value between to integers, which was way less efficient than the standard version. The result of this optimization can be found in table I.

7) Regarding vectorization: Unfortunately, we were unable to alter the *hotspot* of the code so that vectorization could take place effectively. Nonetheless, some other not so time bounding functions were vectorized by the usage of the flag "-O3" that enables vectorization by default.

C. Theoretical proof for results obtained

1) Separating Independent Computations (A): The inner loop involves $L \times N^3$ iterations, where "L" represents LineSolver and $N^3 = O \times N \times M$. The code executes 16 operations: six memory loads, five summations, one multiplication, one load of x0, one addition with x0, one division, and one store. Of these, six memory loads, three summations (pairwise), and the load of x0 are independent, reducing potential cycles from 16 to 7 in parallel execution. This optimization theoretically increases speed by 2.29×. However, results in Table I show minor clock reduction, indicating the compiler already optimized this section.

2) Loop Interchange (C): Changing the loop order in *lin_solve* from i-j-k to k-j-i improves memory access patterns and cache efficiency. In C and C++, arrays are stored in row-major order, meaning elements with the same first index are stored consecutively. The original order caused poor cache locality as non-adjacent elements were accessed, leading to frequent cache misses. Reordering to k-j-i allows the innermost loop to access consecutive memory locations, enhancing spatial locality and reducing cache misses. This optimization leverages modern CPU design for sequential access, significantly improving performance, especially in large 3D grids.

3) Loop Blocking (D2): Loop blocking enhances cache utilization by dividing the grid (array) into smaller blocks that fit into the CPU cache, keeping data for neighboring cells accessible and reducing cache misses. This technique, while increasing code complexity, improves memory locality and parallelism. The original computation:

$$\sum_{i=1}^{M}\sum_{j=1}^{N}\sum_{k=1}^{O}\text{computation}(i,j,k) \tag{2}$$

is transformed into a blocked version:

$$\sum_{i=i_b}^{\min(i_b+B_i-1,M)}\sum_{j=j_b}^{\min(j_b+B_j-1,N)}\sum_{k=k_b}^{\min(k_b+B_k-1,O)}\text{computation}(i,j,k) \tag{3}$$

Results show a higher instruction count but a larger reduction in cycles, with only a slight rise in cache misses, confirming better locality and parallelism.

4) Loop Unrolling (D): The original version of the *lin_solve* function uses three nested loops:

$$\sum_{l=0}^{\mathrm{L}-1}\sum_{k=1}^{O}\sum_{j=1}^{N}\sum_{i=1}^{M}\text{computation}(i,j,k) \tag{4}$$

with a total iteration count of:

$$\mathrm{L} \times O \times N \times M = \mathrm{L} \times N^3. \tag{5}$$

In the optimized unrolled version:

$$\sum_{l=0}^{\mathrm{L}-1}\sum_{k=1}^{O,\ \text{step=2}}\sum_{j=1}^{N,\ \text{step=2}}\sum_{i=1}^{M}\text{comp} \tag{6}$$

where:

$$\text{comp} = \text{c}(i,j,k) + \text{c}(i,j+1,k) + \text{c}(i,j,k+1) + \text{c}(i,j+1,k+1), \quad (7)$$

each iteration performs four computations, halving the loop counts for j and k. The unrolled loop's iteration count is:

$$\text{L} \times \left\lceil \frac{O}{2} \right\rceil \times \left\lceil \frac{N}{2} \right\rceil \times M = \text{L} \times \left\lceil \frac{N^3}{4} \right\rceil. \quad (8)$$

This reduces iterations by a factor of 2 in the j and k loops while keeping the i loop constant, theoretically increasing speed if the compiler efficiently parallelizes unrolled instructions. However, practical gains may be constrained by cache effects, branch prediction, and CPU parallelism limits.

III. Phase 2

A. Solver Analysis

In this phase, the grid size was adjusted to 84, and a new solver more suitable for parallel execution was implemented. Some optimizations from the previous phase were retained and applied to the new solver. An initial profiling with the previous optimization level yielded an execution time of 13.37 seconds, with the *lin_solve* function identified as the primary *hot_spot*.

B. Data Race and Parallelization Study

The *lin_solve* function employs a Red-Black strategy that naturally promotes code parallelization. This method manages shared *x* and *x0* arrays, ensuring that simultaneous writes to the same memory location are avoided if each phase executes sequentially. The *set_bnd* function acts as a synchronization point since it operates outside the phases. Additionally, special care is required for the *max_c* convergence check, as changes must be safely visible across threads. Overall, most of the code was deemed parallelizable due to the absence of significant data race conditions. Maintaining instruction order and identifying single-threaded sections were the primary considerations.

With the identified data races in mind, parallelization was implemented using *OpenMP* directives to distribute tasks among threads. The outermost loop was parallelized using *pragma omp parallel for*, allowing each thread to handle a portion of the grid. For clarity, the *shared(x, x0)* clause was explicitly added, even though these arrays are inherently shared. Consistent treatment was applied across all phases, preserving the original code sequence. To optimize convergence, the *reduction(max:max_c)* clause was used, enabling each thread to maintain a local *max_c* copy, later combined into a single maximum value. Additionally, *private(old_x, change)* was declared for local copies, a requirement of the algorithm, though it could alternatively be scoped within the loop.

For the *add_source* and *advect* functions, *pragma omp parallel for* was utilized, with appropriate handling of shared variables. Similar logic was applied to *set_bnd* and *project*, requiring private copies of *i* and *j* in the former, and maintaining sequential dependencies between loops in the latter.

C. Scalability Analysis And Discussion

After implementing the parallelization, we ran the program for $1 \leq N \leq 40$, so we could analyze the optimal number of threads. The speedups obtained can be found in Fig 1

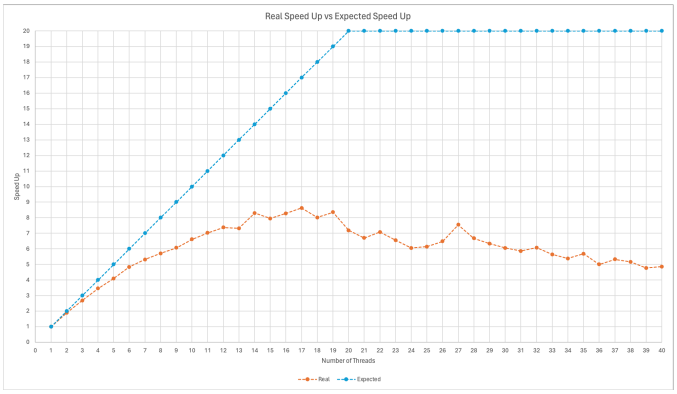

Fig. 1. Speedup of the program for the different number of threads, with the expected ideal speedup for the "cpar" partion.

As shown in the graph, the experimental speedup increases up to 17 threads but begins to decline as the thread count rises further. The number of cache misses was anticipated to grow with the addition of threads. Surprisingly, however, cache misses remained stable. In contrast, memory load increased as the number of threads grew. The primary factor affecting execution time was the memory bandwidth, which proved insufficient to provide data to all cores simultaneously, causing delays in thread processing. This issue can be easily identified in the reports, where *gomp_team_barrier_wait_end* dominates the time when running with 25 or more threads. It's also important to note that there is some overhead with certain routines, such as *reduction*, which, although being the most efficient option, is evident from the increase in *CPI*. Finally, it is essential to consider that the workload distribution across threads can vary, potentially decreasing performance. The data for the analysis can be found in Table III on the *Appendix*.

IV. Phase 3

A. Implementation Overview

The parallel version developed aims to leverage the use of powerful accelerators, more precisely Nvidia GPUs using the CUDA programming language. This approach resulted in many changes to the solver, keeping in mind the knowledge obtained by using OpenMP for the Phase 2, because most of the problems relating to data races would affect the new version the same way as it did to the previous. Fortunately, the solver, as observed in Phase 2, does not have many obstacles for parallelization, as will be covered in this section.

Programming in CUDA does not mean that every line of code executes in the GPU device. The CPU has a very important job of launching the CUDA kernels that actually do the computation. This is an important note, because some of the functions logic that we will present need some form

of synchronization or convergence check that requires CPU coordinatination.

The function lin_solve is the *hot spot* of the code, which means is the most important target for parallelization. This function, for CUDA adaptations, cumulated in 4 kernels, named *red_phase_kernel*, *black_phase_kernel*, *compareMaxChangesKernel* and *reduceMaxKernel*. Starting off by the *red and black* kernels, that are responsible for updating the *x* array. The particularity of our design is that, after updating the x value, each block will perform a reduction, to acquire the maximum value of the change variable within the block. We found this idea interesting because many threads we needed to launch were not actually performing any work on finding *x* due to the *red and black* approach, where even and odd calculations are separated to avoid data races. This divergence in the kernels are not optimal and, in this case, unavoidable, so we opted to try and make as efficient as possible. From the *host* perspective, each phase works on 3 dimension and requires its own *change buffer* that later will be combined in one single buffer. These phases are launched sequentially, starting by the *red phase* followed by the *black phase*. The *compareMaxChangesKernel* handles the computation of merging the 2 buffers used in the phases to one, so it can ultimately be reduced to only one value. The last part is the *reduceMaxKernel* that will run as many times it needs until only one value remains as the maximum. This loop might be needed for larger array sizes that need many folds.

The function set_bnd was also split in 4 kernels, where 3 of them leverage well the use of parallelism and the last one only really needs one thread to make the update. Starting by the 3 main kernels, each thread updates the *x* array in a different position. Contrary to all the other launches, these kernels function with a grid layout, due to the fact that the computation only requires 2 variables. For the last kernel, only one thread is really needed for the operation, the only requirement is to be the last step after the other kernels. Interestingly, it's common practice to still launch 32 threads, even if only one is really needed. This behavior is favored over launching only one thread, because in GPUs keeping the warp alignment is a priority.

The function add_source was the easiest to adapt for the new version due to the fact of corresponding to a simple update to independent positions in the *x* array, without any data races. The main difference from the other mentioned launchers is that only one dimension is truly necessary.

The advect function employs the *advect_kernel*. Each thread calculates the new position of a point in the x array, clamps it within the grid boundaries, and interpolates values using surrounding grid points. This kernel uses a three-dimensional grid layout, with threads corresponding to the spatial dimensions of the problem. The function concludes with a call to set_bnd to apply boundary conditions.

The project function employs 2 kernels: *project_kernel_a*, which calculates the divergence and initializes the pressure array, and *project_kernel_b*, which adjusts the velocity field based on the pressure gradient. These 2 phases need to happen sequentially, but can be parallelized. Boundary conditions are applied using set_bnd after each step, and the lin_solve function is also invoked.

The diffuse, dens_step and vel_step functions encapsulate the simulation's main steps. These functions rely heavily on the optimized kernels for efficient execution, which means no modification was necessary. Its relevant to mention that the pointers used and passed for the kernels are actually allocated on the GPU and not in the CPU.

Finally, we opted to change the main file of the program. The goal was to minimize how much data was sent from the GPU to the CPU. The first steps were easy, which were changing the data allocation, data initialization and clear to the GPU. Then, we only needed to change the function that is responsible to apply the events to also work on GPU.

B. Other Options

One of our goals in the implementation we chose was to show technical knowledge within CUDA programming, specially in the use of reduction techniques. However, there is another approach that requires no reductions that can be very beneficial for the program. This change happens to the lin_solve function and its kernels. As mentioned, we reduce the values of "change" to find the maximum so we can make a convergence check. This manner follows exactly the logic from the sequential version of the code. Nonetheless, it´s possible to calculate the exact same results without reducing values by changing how the convergence check is calculated. The change value is irrelevant, in the sense that it´s only purpose is to see if any value had a change from its previous value greater than the tolerance. Taking this in consideration, if every thread knows the tolerance, individually, they can check if the convergence happen within the designated workload. After that, a global boolean variable can be updated to let know the cpu that no more iteration need to be done. This approach is faster, because it mitigates the need for a reduction to find the biggest "change" value, but differs from the original concept of the solver of finding the biggest value per phase.

C. Scalability Analysis and Results

After implementing the parallelization, we ran the program for 3 different block sizes, for 2 different machines, 2 different tests and for 2 different sizes, so that we could analyze the impact of those variables on the execution time of the program.For the "Home" machine, it's important to mention that the GPU is a GeForce RTX 3060.

In GPU programming, the choice of thread block size is crucial for optimizing performance. For this implementation, we selected thread block sizes that are multiples of 128, as this provides several important advantages. GPUs execute threads in groups called warps, each containing 32 threads. Using a thread block size of 128, which is a multiple of 32, we ensure that the threads in each block align perfectly with the GPU's warp structure, maximizing execution efficiency and minimizing idle threads.

We also chose to consider how the program scales with the grid size. Scalability refers to the ability of the program to maintain or improve performance as the size of the problem increases. This is particularly important in GPU programming, where the computational workload often increases significantly with larger problems. The number of iterations was also tested in Test 2, which doubled compared to Test1

Finally, we tested the program on two different machines, referred to as "Search" and "Home," because testing on different hardware configurations is crucial for a comprehensive performance analysis. Each machine has distinct hardware characteristics, such as the number of cores, memory bandwidth, and GPU architecture, which significantly influence the program's performance. The graphics and data related to phase 3 can be found in the VI-C subsection.

D. Discussion

From the results presented in the above section, several observations can be made about the performance and scalability of the CUDA implementation across different machines, thread block sizes, and test/input sizes.

When looking at the impact of thread block sizes on performance, one of the most significant findings is that the choice of thread block size has a considerable impact on program performance. On the Search machine, the best performance was achieved with a thread block size of 256, particularly for larger input sizes. However, this was not universally true, as shown by the results on the Home machine. On the Home machine, performance trends varied, with optimal configurations depending on the specific test and hardware conditions. This variability underscores the importance of balancing the number of threads per block with the number of blocks. An optimal thread block size ensures efficient utilization of GPU resources by aligning with the warp structure and avoiding idle threads. However, achieving the best performance requires tuning based on hardware characteristics, such as the number of multiprocessors and the memory bandwidth, as shared memory often can be a bottleneck (Figures 5 and 6).

The results also highlight the scalability of the implementation. For both machines, the speedup achieved was consistent even when the number of iterations was doubled (as seen in Test 2). This consistency indicates that the program scales effectively with the grid size, maintaining its performance advantage for larger grids. However, it is evident that larger input sizes generally benefit more from GPU acceleration, as the overhead of parallelization becomes less significant relative to the computation time (Figures 7 and 8).

The execution time graph does not follow our initial idea for the scaling of the problem. Considering the size of the grid to double its value *N*, the following equation explains how much the grid size increases for the initial value.

$$\text{Grid Size} = SIZE^3 = (2N)^3 = 8N^3 \tag{9}$$

Based on the equation, we thought every execution would take 8 times more time to execute based on the problem size only. With the graph from Figure 9, we can see that it's not true for the considered sizes. For the smaller sizes, below 168, the execution time only doubles the time from the previous size. Considering the last 2 sizes, approximately, we can see a growth of almost eight times, which falls within our first theory.

The performance trends observed in Test 1 and Test 2 were remarkably similar, despite the double iterations in Test 2. This suggests that the implementation trends observed are not specific to the number of iterations. Instead, they reflect the underlying performance characteristics of the implementation on the given hardware.

Lastly, we refer to the Phase 2 parallel version for a final comparison. The comparison between the CUDA implementation and the OpenMP version highlights the strong dependence of performance on the underlying hardware. In the Search machine, the OpenMP version achieved its best performance with the optimal number of threads identified in Phase 2. However, on the Home machine, this trend was not replicated, as the OpenMP implementation did not perform as effectively, underscoring the variability caused by hardware differences.

This reinforces the conclusion that performance optimization is not only tied to the algorithm or parallelization strategy but also highly influenced by hardware's specific characteristics, such as the number of cores, memory bandwidth, and GPU/CPU architecture. Consequently, achieving the best performance requires tuning and configuration for each hardware environment.

E. Future Improvements

For our CUDA implementation, a few impediments can be considered when speaking about performance. Some are very well known problems in GPU programming. Several potential improvements could enhance the performance and functionality of the current version. Due to time constrainsts, this sections details potencial improvements based on our tests and observations of the current implementation.

The current implementation suffers from thread divergence in the red-black pattern solver, where threads within the same warp take different execution paths. This could possibly be optimized by reorganizing the grid layout to ensure threads within the same warp follow similar paths.

As seen in the Figures 5 and 6, for different hardware the optimal block configuration may differ. Given the observed variability in performance across different block sizes and hardware configurations, implementing an auto-tuning mechanism could be valuable. This system could dynamically select optimal thread block sizes based on the specific GPU architecture and problem size, ensuring better performance across different hardware configurations.

As discussed in the "Other Options" section, implementing the boolean-based convergence check could improve performance by eliminating the need for reduction operations. This could be extended further by implementing a hybrid approach that switches between methods based on problem size and hardware characteristics.

V. Conclusion

In this work assignment, we have presented a study of optimizing a fluid dynamics simulator through three distinct phases of implementation, each building upon the insights and achievements of the previous phase. Our work demonstrates the significant performance improvements that can be achieved through a systematic approach to optimization, from instruction-level enhancements to full GPU acceleration.

The first phase focused on instruction-level optimizations, where we identified and addressed several key performance bottlenecks. By implementing techniques such as loop interchange, loop unrolling, and redundancy elimination, along with careful compiler optimization flags, we achieved substantial improvements in cache utilization and overall execution time. The analysis of the lin_solve function as the primary hotspot guided our optimization efforts effectively.

In the second phase, we leveraged OpenMP to implement CPU-based parallel processing. Our analysis revealed that thread scalability was limited by memory bandwidth constraints, with optimal performance achieved at around 17 threads. Beyond this point, the overhead of thread management and memory access synchronization began to outweigh the benefits of additional parallelism. This phase highlighted the importance of careful thread management and the impact of hardware limitations on parallel performance.

The final phase, implementing GPU acceleration using CUDA, demonstrated the most significant performance improvements. Our implementation featured specialized kernels with a Red-Black solver approach and block-level reductions, effectively utilizing the massive parallel processing capabilities of modern GPUs. However, our testing across different machines and configurations revealed that performance optimization is highly hardware-dependent. The optimal thread block size varied between different GPU architectures, and the scaling behavior with problem size showed interesting nonlinear characteristics.

VI. Appendix

A. Optimization Statistcs

TABLE I
Best runs for each degree of Optimization

Optimization	Time (s)	Number of Instructions	Clock Cycles	L1 Cache Misses
No change	27.71	1.666×10^{11}	8.884×10^{10}	2.310×10^{9}
No change + O2	11.09	1.891×10^{10}	3.477×10^{10}	2.321×10^{9}
Run A	10.79	1.815×10^{10}	3.467×10^{10}	2.312×10^{9}
Run B	10.85	1.856×10^{10}	3.510×10^{10}	2.319×10^{9}
Run C	12.15	1.677×10^{10}	3.941×10^{10}	2.007×10^{8}
Run D	5.42	1.511×10^{10}	1.746×10^{10}	2.22×10^{8}
Run D2	7.52	2.511×10^{10}	2.400×10^{10}	2.199×10^{8}
Run E	2.85	1.642×10^{10}	9.118×10^{9}	2.229×10^{8}
No change + Flags	7.00	1.852×10^{10}	2.266×10^{10}	2.312×10^{9}
Run F	2.77	1.656×10^{10}	8.900×10^{9}	2.235×10^{8}

TABLE II
Speedup and percent difference for each optimization
(Comparing with previous entry)

Optimization	Speedup	Percent Difference		
		Number of Instructions	Clock Cycles	L1 Cache Misses
No change	0	0	0	0
No change + O2	2.499	88.65	60.86	-0.47
Run A	1.028	4.02	0.29	0.39
Run B	0.994	-2.26	-1.24	-0.30
Run C	0.893	9.65	-12.27	91.33
Run D	2.242	9.89	55.68	-10.63
Run E	2.032	8.67	-47.77	0.41
No change + Flags	0.529	25.50	-48.21	-3.93
Run F	1.966	-10.63	49.60	90.39

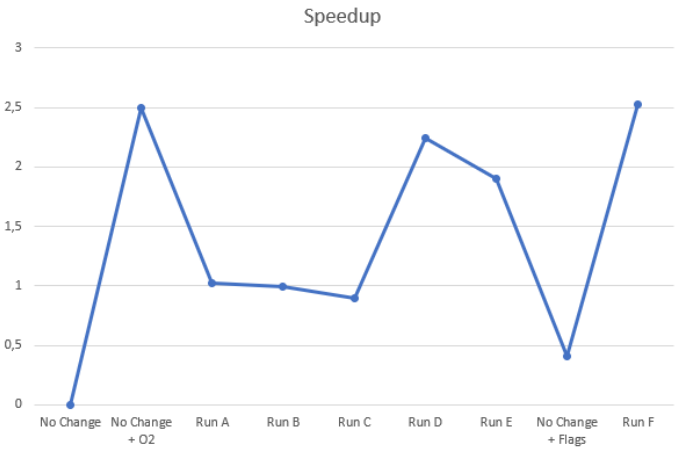

Fig. 2. Speedup, relative to previous state of code

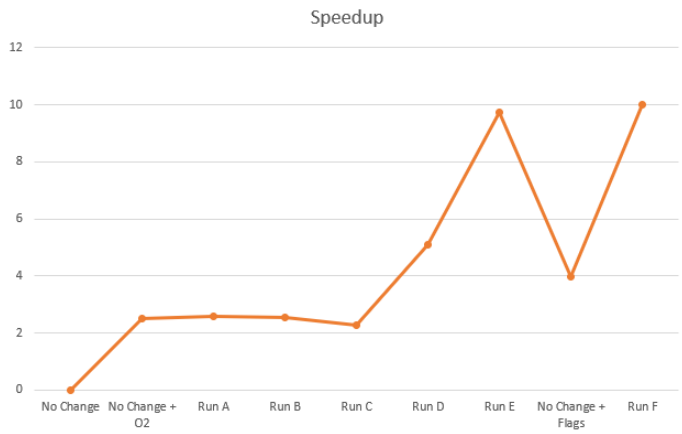

Fig. 3. Speedup, relative to FIRST state of code

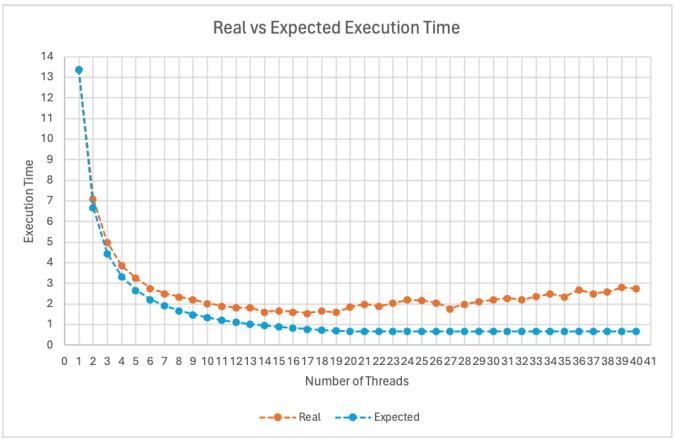

Fig. 4. Execution time of the program for the different number of threads with the expected ideal time.

B. Data Phase 2

TABLE III
PERFORMANCE METRICS BY NUMBER OF THREADS

Threads	$T_{Execution\ Time}$	CPI	Memory loads	L1-dcache-loads
1	13.37 s	0.50	0	31,036,051,605
2	7.10 s	0.50	0	31,109,765,135
3	4.99 s	0.50	0	31,216,676,735
4	3.87 s	0.60	0	31,236,658,879
5	3.26 s	0.60	0	31,314,091,465
6	2.77 s	0.60	0	31,133,993,718
7	2.52 s	0.60	0	31,405,856,180
8	2.34 s	0.60	1,483	31,498,863,595
9	2.20 s	0.60	53	31,722,649,670
10	2.02 s	0.60	0	31,719,390,833
11	1.90 s	0.70	0	31,829,716,735
12	1.81 s	0.70	0	31,749,132,688
13	1.83 s	0.70	1,294	31,996,448,207
14	1.61 s	0.70	1,216	31,937,539,835
15	1.68 s	0.80	0	32,042,985,451
16	1.61 s	0.80	0	32,343,212,555
17	1.55 s	0.80	0	32,330,879,189
18	1.67 s	0.90	0	32,377,803,830
19	1.60 s	0.90	279	32,747,018,030
20	1.86 s	1.00	16,583	33,537,125,622
21	2.00 s	1.10	29,129	33,914,882,225
22	1.89 s	1.10	215,127	34,303,545,381
23	2.04 s	1.20	249,779	34,353,370,282
24	2.21 s	1.40	302,383	34,581,431,947
25	2.18 s	1.40	207,044	34,471,252,798
26	2.06 s	1.40	412,112	34,564,771,613
27	1.77 s	1.30	112,917	34,458,149,491
28	2.00 s	1.50	627,363	34,128,070,365
29	2.11 s	1.60	616,128	34,341,498,082
30	2.21 s	1.70	625,951	34,794,249,225
31	2.28 s	1.80	646,231	34,723,560,926
32	2.20 s	1.80	766,961	34,793,075,859
33	2.37 s	2.00	653,783	34,848,826,103
34	2.49 s	2.10	784,301	35,266,469,346
35	2.35 s	2.00	1,761,849	35,369,211,147
36	2.68 s	2.40	1,447,214	35,313,665,275
37	2.51 s	2.30	1,370,538	35,442,166,975
38	2.59 s	2.40	1,854,607	35,129,440,223
39	2.81 s	2.70	1,376,675	35,721,842,820
40	2.75 s	2.60	3,431,356	35,983,817,724

C. Data Phase 3

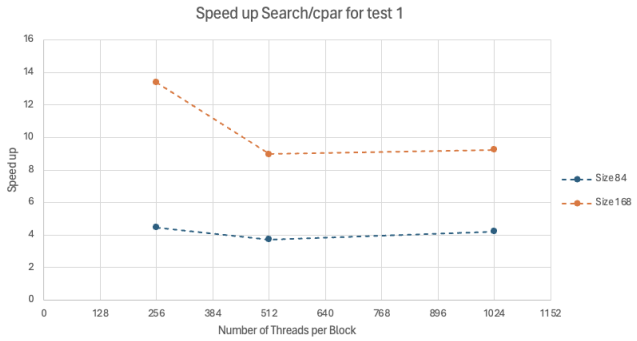

Fig. 5. Speed up for the cuda implementation in Search machine, test 1

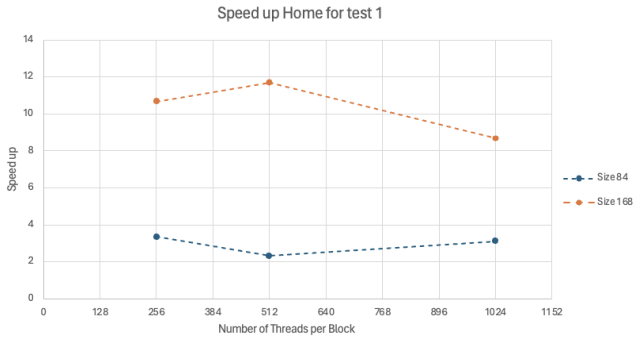

Fig. 6. Speed up for the cuda implementation in Home machine, test1

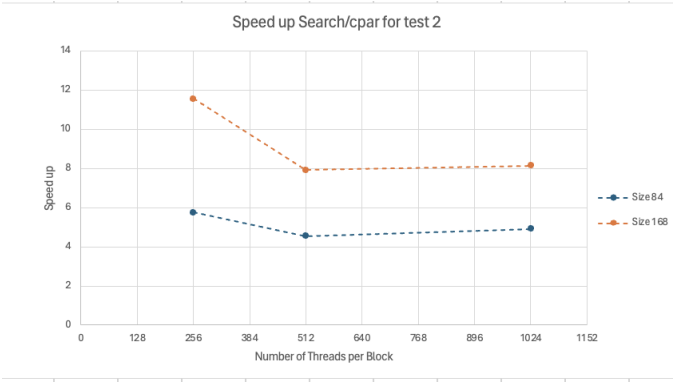

Fig. 7. Speed up for the cuda implementation in Search machine, test 2

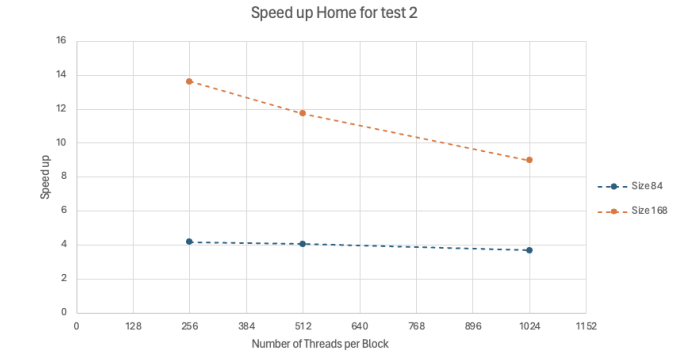

Fig. 8. Speed up for the cuda implementation in Home machine, test 2

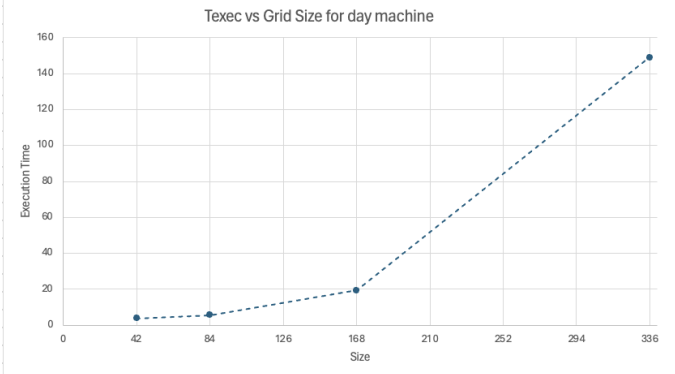

Fig. 9. Execution time for the cuda implementation in day machine

TABLE IV
PERFORMANCE METRICS FOR PARALLEL VERSION

Test	machine	size	Threads per Block	$T_{Execution\ Time}$
1	search/cpar	168	256	17.56 s
1	search/cpar	84	256	3.95 s
1	search/cpar	168	512	26.17 s
1	search/cpar	84	512	4.75 s
1	search/cpar	168	1024	25.40 s
1	search/cpar	84	1024	4.19 s
1	home1	168	256	6.16 s
1	home1	84	256	1.86 s
1	home1	168	512	5.62 s
1	home1	84	512	2.69 s
1	home1	168	1024	7.58 s
1	home1	84	1024	2.00 s
2	search/cpar	168	256	37.90 s
2	search/cpar	84	256	6.69 s
2	search/cpar	168	512	55.34 s
2	search/cpar	84	512	8.46 s
2	search/cpar	168	1024	53.71 s
2	search/cpar	84	1024	7.82 s
2	home1	168	256	10.23 s
2	home1	84	256	3.38 s
2	home1	168	512	11.87 s
2	home1	84	512	3.48 s
2	home1	168	1024	15.51 s
2	home1	84	1024	3.84 s

TABLE V
PERFORMANCE METRICS FOR SEQUENTIAL VERSION

Test	machine	size	$T_{Execution\ Time}$
1	search/day	168	235.14 s
1	search/cpar	84	17.68 s
1	home	168	65.65 s
1	home	84	6.24 s
2	search/day	168	437.90 s
2	search/cpar	84	38.44 s
2	home	168	139.31 s
2	home	84	14.12 s

TABLE VI
SCALING FOR PARALLEL VERSION

Test	machine	size	$T_{Execution\ Time}$
1	search/day	42	3.89 s
1	search/day	84	5.67 s
1	search/day	168	19.33 s
1	search/day	336	148.93 s