Parallel Computing
Practical Assignment Phase 3

António Silva
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57867

Duarte Leitão
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57872

Vasco Faria
Mestrado em Engenharia Informática
Universidade do Minho
Braga, Portugal
PG57905

***Abstract*—In this phase, the objective was to optimise the same problem addressed in Phase 2 increasing the problem's data size (N=168). The focus was on leveraging shared memory techniques, utilizing CUDA, to achieve a reduction in execution time.**

***Index Terms*—CUDA, Parallelism, Optimization**

I. Introduction

The primary objective of this project is to explore and evaluate various strategies and tools for optimizing an algorithm. This initiative offered an opportunity to examine multiple parallelization models, with the ultimate goal of enhancing the algorithm's performance. This document details the project's progression through its key stages, starting with the initial development of the algorithm in its sequential form, followed by its subsequent refinements, and culminating in the adoption of parallelism in later phases. Additionally, we analyze the performance of the applied techniques, assessing their effectiveness and comparative impact.

II. WA1 - Optimizations in sequential version

In order to identify possible optimizations, we recured to **gprof**. One critical discovery during our performance analysis was that the **lin_solve** function represented the most significant area for optimization. Profiling results revealed that this function was responsible for a substantial portion of the total execution time.

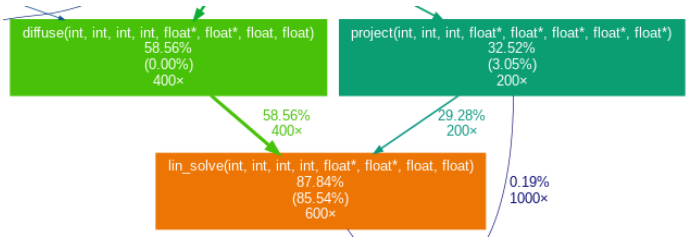

Fig. 1. Execution flow visualization from gprof

In the lin_solve function, several optimizations were implemented to enhance performance and efficiency. Initially, the indices for neighboring grid points were computed repeatedly within the nested loops, leading to unnecessary redundancy. By computing these indices (leftIndex, rightIndex, belowIndex, aboveIndex, backIndex, frontIndex) at the start of each iteration and storing them in variables, we significantly reduced redundant calculations. This change not only minimized the number of arithmetic operations but also improved cache efficiency by reducing memory access latency. Additionally, the neighboring values were accessed multiple times directly from the array, which was inefficient. By storing these values in temporary variables (left, right, below, above, back, front), we minimized memory accesses, leading to a noticeable improvement in execution speed due to fewer memory read operations.

The advect function also underwent several optimizations to enhance its performance. Previously, indices and values for the velocity components (u, v, w) and density (d, d0) were computed and accessed multiple times within the loop. By pre-computing these indices and storing the velocity components in variables (u val, v val, w val), we reduced the number of times the program needed to access and compute the same values. This change saved computational resources and sped up execution. The clamping of coordinates, which was originally done using multiple conditional checks, was simplified using ternary operators. This not only streamlined the clamping logic but also reduced the number of conditional checks, improving execution speed. Furthermore, the bilinear interpolation calculation was optimized by using precomputed indices (i0, i1, j0, j1, k0, k1) and temporarily storing the density values, which reduced the number of memory accesses required.

Changing the loop order in the project function enhances performance by improving cache locality and reducing L1 cache misses. By iterating over the depth dimension (k) first, followed by the height (j), and then the width (i), the modified code accesses all relevant data within the same k slice before proceeding to the next. This sequential access pattern allows the CPU to leverage cache line fetching effectively, ensuring that once a cache line containing related data is loaded, subsequent accesses to that data are much faster. Consequently, this optimization leads to fewer cache misses, as the inner loops access data that is already in the cache, resulting in shorter execution times and improved overall efficiency in the fluid simulation process.

III. WA2 - OpenMP Parallelism

In this phase, our objective was to explore parallelization techniques to enhance performance. Through detailed analysis, several computational hot spots were identified. One prominent bottleneck is the add_source function, which iterates over the entire data array to update each element. Another significant bottleneck is the set_bnd function, which enforces boundary conditions using nested loops across all grid dimensions. However, the most computationally intensive part of the code is the lin_solve function, which employs the red-black Gauss-Seidel solver. This function performs triple nested loops over the 3D grid, iteratively updating values based on neighboring cells until a convergence criterion is met. The combination of complex computations, multiple iterations, and a reduction operation to assess convergence makes lin_solve the most time-consuming section of the code.

A. Parallelism exploration

To enhance the performance of the fluid solver, shared-memory parallelism was implemented using OpenMP to distribute computational tasks across multiple threads. In the add_source function, the loop responsible for updating the array was parallelized with the directive #pragma omp parallel for simd, as each iteration operates independently and can execute concurrently. Similarly, in the set_bnd function, separate loops were parallelized using the #pragma omp parallel directive, enabling simultaneous updates and improving computational efficiency.

In the project function, shared variables were utilized to ensure all threads operated on the same global data structures, promoting efficient memory usage while maintaining consistency in updates. Parallelism was also applied to the lin_solve function to accelerate the Gauss-Seidel method. Both phases of the solver were parallelized using #pragma omp parallel for collapse(2), flattening the two outer loops to enhance load balancing across threads. A reduction(max: c) operation was employed to ensure consistent computation of the maximum residual across threads, a critical factor for convergence.

Thread safety was maintained by declaring variables such as old_x and change as private, providing each thread with independent copies, while global structures like M, N, O, x, x0, a, and inverso_c were shared to allow collective data access. The choice of scheduling strategies was also analyzed. The schedule(static) strategy demonstrated superior performance due to its low runtime overhead, whereas schedule(dynamic) incurred higher execution times, particularly at higher thread counts, due to increased scheduling overhead.

B. Performance and Scalability analysis

Building on the results from Phase 1, the implementation of OpenMP led to a substantial improvement in performance. This enhancement was evident in the consistently reduced execution time, which remained under the 3-second mark. To determine the optimal configuration, we tested a range of thread counts and evaluated various scheduling strategies.

The results, illustrated in the following graph, indicate that using 20 threads with a static scheduling strategy provided the best performance. This configuration achieved an ideal balance between workload distribution and minimal overhead, ensuring efficient execution of the program.

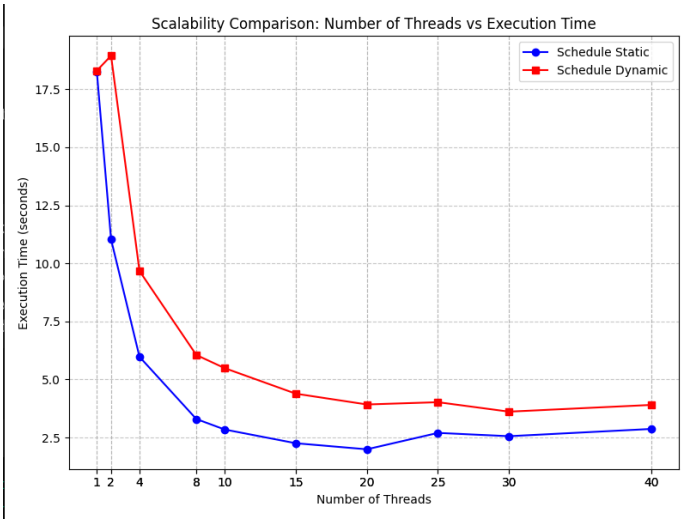

Fig. 2. Scalability comparison schedule Dynamic and Static

We also analysed how thread count impacts the CPI. Performance initially improves as threads distribute tasks efficiently. However, diminishing returns occur at higher thread counts due to synchronization overhead and memory bandwidth saturation, causing CPI to plateau or increase slightly.

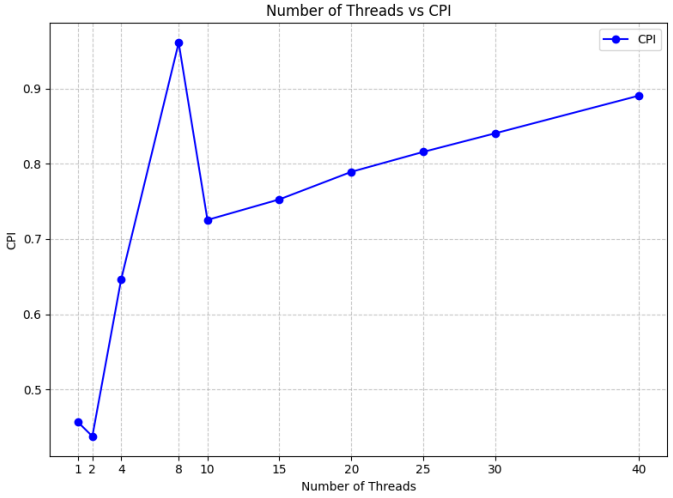

Fig. 3. CPI of different number of threads.

IV. WA3 - CUDA Parallelism

CUDA, developed by NVIDIA, is a parallel computing architecture designed to accelerate processing on GPUs. It enables programmers to leverage the massive parallel processing power of GPUs to perform complex computations efficiently,

making it particularly valuable for tasks such as graphics rendering, physics simulations, and scientific computations.

In Phase 3, we began with the sequential implementation from Phase 2 as our starting point. This version included several nested loops, which presented significant performance bottlenecks and were prime candidates for optimization using CUDA.

A. Implementation of CUDA code

The first sign of optimizations with CUDA is the use of unified memory through **cudaMallocManaged**, enabling seamless memory sharing between the CPU and the GPU. This simplifies data management and reduces memory transfer overhead. Key variables, including velocity components (v_u, v_v, v_w) and density fields (d_x, d_x0), are allocated in GPU memory to optimize computation. Additionally, the memory is carefully initialized, and data is transferred between the host and device only when necessary to maintain performance. These key variables are transferred to the GPU at the start of the simulate function in main, and their memory is released at the end.

Other optimizations consist in the **introduction of kernels** for most functions. The first one is add_sour_kernel. Even though add_source is not a major block to performance, the introduction of the kernel allowed us to slightly reduce the execution time. By launching various threads, the function reduces its execution time.

The same logic was carried to other functions. The implementation was slightly different, given that the other functions contained 3 nested loops. In these cases, we specified the number of blocks required in each dimension to cover the full problem size. Adding to that, we also specified the amount of threads that blocks will have in each dimension, which later required some fine tuning.

Regarding the logic implemented in the kernels, it did not change that much. The base algorithm was keep the same. The set_bnd_kernel replaces OpenMP's nested loops and thread scheduling with a single kernel, where each GPU thread independently computes boundary conditions using its unique (i, j, k) index. Conditional checks ensure only relevant threads execute, and corner cases are handled within the kernel. The implementation of lin_solve uses two separate kernels, lin_solve_red_kernel and lin_solve_black_kernel, to update the solution x. Each thread computes the updated value of a grid point based on its neighbors, and the maximum change is tracked. The lin_solve function manages kernel launches for both steps. The advect_kernel, besides using the same CUDA approach as other functions, kept its calculations fairly the same, besides now pre-calculating some values to reduce the number of operetions. The project function followed a similar approach to lin_solve by creating two separate kernels: one to compute div and init p, and another to update the velocities.

Some changes where also made in the main of the program. Since some variables are in the GPU, we updated the functions in main to directly update those variables. Because the GPU variables are defined in the fluid_solver file, we had to include the necessary variables in its header file using **extern** so they could be directly accessed in main. This allows us to avoid data transfers between the host and the GPU.

B. Fine tuning tests

After implementing our CUDA code, we conducted tests using various combinations of the number of **threads per block**. This fine-tuning process enabled us to identify the configurations that deliver the best performance.

Threads per block	Execution Time (seconds)
(64, 8, 2)	20.8
(32, 32, 1)	21.6
(16, 16, 4)	22.2
(8, 16, 8)	28.9
(8, 8, 8)	44.8

TABLE I
EXECUTION TIME WITH DIFFERENT NUMBERS OF THREADS PER BLOCK

All tests were conducted on the Cpar partition of the SeARCH cluster. It's important to note that the results presented here are based on successful runs, where the output was deemed acceptable.

During the tests, the number of threads per block of the add_source_kernel was kept at 256. Later in testing, the number of thread per block of the set_bnd_kernel was changed to (16,8,8) while keeping the rest the same. **The execution time dropped to 17.9 seconds**

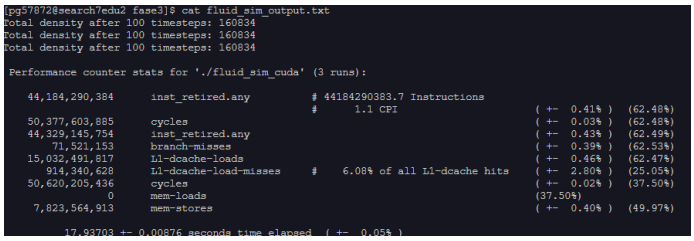

Fig. 4. Best result

Considering that the sequential version of the code (with SIZE=168) took over 200 seconds to execute, the CUDA implementation achieved significant performance improvements, resulting in a substantial reduction in execution time.

C. Heaviest operations analysis

In order to further analyze the program's performance, we used **nvprof** to find out which operations took the longest, and we obtained the following results:

Operation	Total time	Calls	Average
cudaDeviceSynchronize	15.3200s	23670	647.23us
cudaLaunchKernel	1.09449s	23870	45.851us
cudaMallocManaged	208.65ms	9	23.183ms
cudaMemcpy	87.724ms	8	10.965ms
cudaFree	20.306ms	9	2.2562ms

TABLE II
PROFILLING RESULTS

Based on the information about the API calls, we can draw several conclusions. Synchronization operations account for a significant portion of the execution time. Although we experimented with removing these operations, the program failed to function correctly without them. The time spent on kernel launches aligns with the expectations of our algorithm's design. Memory-related operations, on the other hand, represent a very small fraction of the total execution time and involve relatively few calls. Achieving this outcome was one of the primary objectives of our implementation.

D. Weak scaling analysis

For the weak scalability analysis, we evaluated the program's performance across various SIZE values.

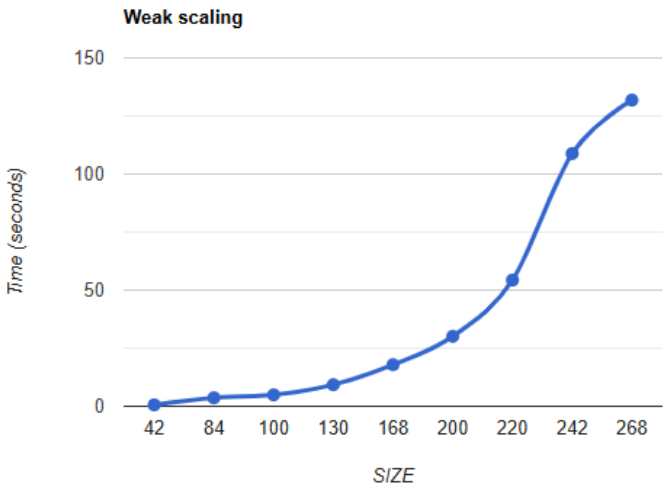

Fig. 5. Weak scaling analysis

The execution times vary significantly, ranging from just under one second to 131 seconds. Notably, the execution times increase exponentially as the SIZE parameter grows.

The range of values tested is constrained by the job time limit imposed on the Cpar partition. It is worth mentioning that the partition was under a lot of load during testing. Some of the results might be slightly higher than expected. The execution time for SIZE=168 stayed consistently in the 17 seconds mark. Since most of the development was made in this partition, we took the decision to not use the day partition.

E. Possible improvements

Looking at our implementation, we believe more performance can be extracted by introducing shared memory. We this it would be possible we could have memory regions that are shared among the threads of a block in a CUDA kernel. We expect this improvement to cut a few seconds in execution time, which could allow the CUDA version of the solver to better match its OpenMP version.

V. Performance comparison

Throughout the three phases of this work assignment, we explored various techniques to enhance algorithm performance. Among the implementations, the OpenMP version proved to be the most efficient, followed by the CUDA implementation, and lastly, the optimized sequential version.

With a problem size of SIZE=84, the OpenMP implementation completed execution in under 3 seconds, whereas the CUDA implementation took approximately 3.5 seconds. This result was unexpected, as we anticipated the CUDA implementation to match or surpass the performance of the OpenMP version. The discrepancy might stem from limitations inherent to the base algorithm or from optimization opportunities that we overlooked.

A key observation was that the CUDA implementation is significantly impacted by data transfers between the host and the GPU, particularly during **cudaMemcpy** operations. Despite efforts to minimize these transfers, they still contribute to a noticeable performance gap compared to the OpenMP implementation. Additionally, operations like **cudaDeviceSynchronize** also affect execution time. However, removing these synchronization points negatively impacts the program's results and consistency, making them indispensable for correctness.

The sequential version submitted in Phase 1 recorded execution times of approximately 8 seconds for SIZE=42, demonstrating significantly poor performance. Since this implementation lacks any form of parallelism, it is unlikely to achieve better performance when scaled to SIZE=84 or even match the OpenMP and CUDA versions.

Regarding the parallel versions, besides providing a bit more performance, we consider the OpenMP version was more straight forward to implement and didn't add much complexity to the code.

VI. Conclusion

In summary, the assignments throughout this semester emphasized the critical role of parallelism in code optimization and its transformative impact on computational efficiency. By progressively introducing and refining parallel programming techniques, we observed dramatic improvements in performance compared to the initial sequential implementation.

In the first assignment, the sequential version established a baseline, demonstrating the inherent limitations of single-threaded execution. The sequential approach, while straightforward to implement, struggled with larger problem sizes due to the lack of scalability and the inability to leverage modern multi-core processors.

The second assignment introduced OpenMP, a powerful tool for exploiting multi-core parallelism on CPUs. By distributing

workloads across multiple threads, OpenMP enabled us to achieve substantial reductions in execution time. The simplicity of its integration into existing code and the immediate performance gains highlighted the benefits of shared-memory parallelism, especially for compute-intensive tasks.

In the third assignment, we delved into GPU parallelism with CUDA, which unlocked a different paradigm of acceleration by leveraging thousands of lightweight threads running in parallel on specialized hardware. Although the CUDA implementation did not outperform the OpenMP version for our specific problem, it underscored the immense potential of GPU acceleration. GPUs excel at tasks that can exploit massive parallelism, such as matrix operations and data-parallel algorithms, making them an indispensable tool for modern high-performance computing.

Another critical insight from these assignments was the impact of compiler flags on program behavior. Aggressive optimization flags can significantly enhance execution speed by fine-tuning memory usage, loop unrolling, and other low-level operations tailored to the target architecture. However, such optimizations may introduce subtle variations in results due to floating-point precision issues or hardware-specific behaviors. Balancing these trade-offs required careful testing to ensure that performance improvements did not compromise the program's accuracy or reliability.

These assignments collectively demonstrated that parallelism is not merely an optional enhancement but a necessity for optimizing performance in computationally demanding tasks. They provided a hands-on understanding of how parallel programming paradigms, from multi-core CPUs to GPUs, can unlock the full potential of modern hardware, leading to transformative gains in execution efficiency and scalability. Moreover, they highlighted the importance of a nuanced approach to optimization, balancing speed with accuracy to achieve robust and reliable performance improvements.