Parallel programming project final report

Diogo Gomes Rodrigues
Physical Engineering Student
Minho's University
Braga, Portugal
a103847@alunos.uminho.pt

Gonçalo Filipe Ribeiro Rodrigues
Physical Engineering Student
Minho's University
Braga, Portugal
a103849@alunos.uminho.pt

Vitor Fernandes Alves
Physical Engineering Student
Minho's University
Braga, Portugal
a103940@alunos.uminho.pt

***Abstract*— This document reports on the discussion, analysis, and optimization of one of the simplest fluid dynamics simulation codes, the 3D version of Jos Stam's stable fluid solver.**

Keywords*— *fluid_sim, lin_solve, CPI, number of instructions, clock cycles, cache misses, threads, GPU´s,

I. Introduction

In the first analysis we focus on simple single core/threaded optimizations with a 3D grid size of internal cells of 42, starting by profiling our application to see the critical areas in performance and then identifying performance inefficiencies including code analysis and performance metrics such as CPI (Cycles Per Instruction), cache miss rates, etc. then applying optimization techniques to enhance computational efficiency while preserving output precision.

After that we started to improve the parallelism with multithreading and multiple cores using OpenMP, following the same approach: identifying bottlenecks, restructuring the algorithm, and using OpenMP constructs. Additionally, we doubled the 3D grid size of internal cells to 82, increasing the problem size by eight times.

Finally, we raised the level of parallelism even further by programming the program to run on a GPU (NVIDIA Kepler K20), always following the same methodology. The 3D grid size of internal cells has been increased to 168 in each dimension, which means we have once again increased the size of the problem by 8 times

II. Simple single core/threaded optimizations

A. *Code profiling and performance analysis using GProf2Dot*

By using gprof2dot on the program, we obtained Fig. 5 in the annexes, which represents a performance graph of the fluid_sim program divided by functions. Each square in this figure displays the percentage of time spent in that function and in the functions it calls, along with the execution percentage of time for that function (in parentheses).

In Fig. 1, an enlarged view of Fig. 6 is presented, focusing on the most relevant aspects for the program analysis. This figure highlights the functions that consume the most time.

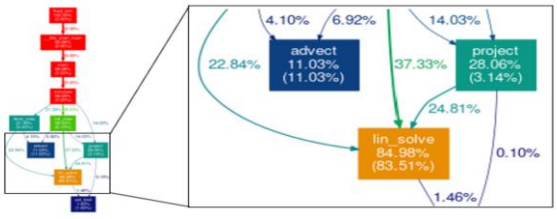


Fig. 1. Enlarged view of the most time-consuming functions from Fig. 5, highlighting critical areas for performance analysis.

Upon analyzing Fig. 1, it is concluded that the function consuming the most time is lin_solve, which accounts for 83.51% of the total execution time of the program. Therefore, this function requires a more careful and detailed analysis and is the one that needs optimization. Additionally, one of the optimizations performed by the compiler can be identified: inlining. This is evident in the advect function, which calls the set_bnd function in the code. However, the figure does not show a call to any other function. Since this function is the second most time-consuming, we will analyze it as well.

B. *Analysis of the Compiled Function with Different Optimization Levels*

The measurement results for different optimization levels are presented in Table I.

TABLE I. Results of the Measurements for the Original Code with Different Optimization Levels

	T(s)	#I	#CC	CPI	L1_Dmiss	Results
-O2	10.842	19G	35G	1,89	2,3G	81981.3
-O3	10.674	18G	35G	1,89	2,3G	81981.3
-Ofast	8.610	20G	28G	1,38	2,3G	81981.5

The compilation with -O3 shows a small improvement compared to -O2, due to increased inlining and loop unrolling. However, -Ofast shows a greater improvement because it introduces floating-point optimizations that break some of the IEEE 754 rules, causing slight inaccuracies in the results. We will use the -Ofast optimization.

C. *Analysis and optimization of the lin_solve*

The complexity of the function is $O(LMNO)$, where L is the variable LINEARSOLVERTIMES. As the grid size increases, the values of N, M, and O also increase, resulting in higher values of #I and #CC, which consequently leads to an increase in the function's execution time.

In Fig. 7 (annexes), a dependency graph of instructions based on the Ivy Bridge architecture is presented. Each box represents a C instruction, with arrows indicating data dependencies. Since the variables a and c are likely stored in registers, we use them directly in the operations.

Figure 2 shows an enlarged view of the most relevant part of the dependency graph from Fig. 7, specifically the part of the graph that demonstrates what prevents parallelism and increases the program's execution time.

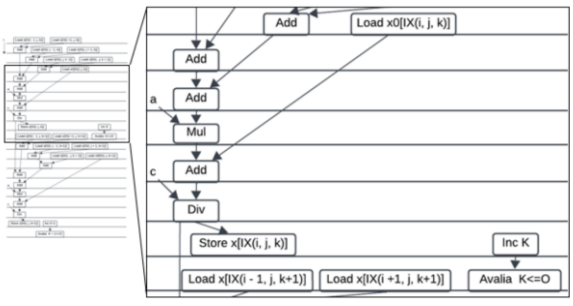


Fig. 2. Enlarged section of the instruction dependency graph from Fig. 6, demonstrating the factors preventing code parallelism.

Analysis of Fig. 2 concludes that significant data dependencies between instructions are the primary barrier to instruction-level parallelism. Additionally, this architecture only has one ALU for performing float additions and another for float divisions and multiplications. Furthermore, the value of x[IX(i, j, k)] must be stored at the end of each iteration of the k loop, after which k can be incremented.

To increase parallelism in the code and reduce execution time, we made two changes to the function. The first change was replacing the expression the original expression

x[IX(i,j,k)]=(x0[IX(i,j,k)]+a*(x[IX(i-1,j,k)]+x[IX(i+1,j,k)]+ x[IX(i,j-1,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k-1)]+x[IX(i,j,k+1)]))/c

with

x[IX(i,j,k)]=x0[IX(i,j,k)]*e+d*(x[IX(i-1,j,k)]+x[IX(i+1,j,k)] +x[IX(i,j-1,k)]+x[IX(i,j+1,k)]+x[IX(i,j,k-1)]+x[IX(i,j,k+1)]),

where d=a/c and e=1/c, and both are calculated only once to eliminate the costly division operation and enhance instruction-level parallelism. The second change was to apply loop unrolling by a factor of 4 to the loop over k, which increased instruction-level parallelism and better utilized the superscalar. This allowed the k+1 iteration to start before storing x[IX(i,j,k)] completed, while also reducing the increment and evaluation instructions for k.

In Table II, we present the results of the measurements for execution time (T), #I, #CC, CPI, number of L1 data cache misses, and Miss/#I for this optimization.

TABLE II. RESULTS OF THE MEASUREMENTS

T(s)	#I	#CC	CPI	L1_Dmiss	Miss/#I
5.976	22G	19G	0.86	2.5G	0.11

By analyzing the values, we conclude that that there has been a reduction in execution time and a decrease in CPI, indicating an increase in instruction-level parallelism.

D. *Impact of the memory hierarchy – Spatial and temporal locality*

The current loop implementation is inefficient, as it does not leverage the spatial and temporal locality of the code. It accesses memory cells that are far apart and consistently accesses different memory locations. To fix this, we reordered the loops in the lin_solve function to kji, as the array is stored contiguously along i. We also aligned the order of the sums with the loop order to facilitate the compiler's ability to introduce parallelism into the code. Below is a summary of the performance metrics:

TABLE III. RESULTS OF THE MEASUREMENTS

T(s)	#I	#CC	CPI	L1_Dmiss	Miss/#I
4.934	18.6G	16G	0.86	468M	0.025

By analyzing the values in the table, we conclude that there was a reduction in the function's execution time, as well as in the CPI and #CC, but the main improvement was the huge decrease in the L1_Dmiss.

E. *Vector Processing.*

In this function, there are no control dependencies. However, due to data dependencies arising from accessing non-contiguous elements of the array x, each iteration relies on the positions of x based on i, j, and k, where one of these variables is either decreased by 1 or increased by 1. Given this, the function can only be vectorized by exploiting the three-dimensional diagonals, as the elements within the same diagonal have no data dependencies between them. This enables efficient parallelism and vectorization, such as using SIMD instructions, to process the data of a single diagonal simultaneously. However, dependencies between different diagonals remain, limiting the vectorization to the level of each diagonal.

Since the compiler is not capable of vectorizing in this manner, as confirmed by compiling with the flag -fopt-info-vec-all, we did not vectorize this function.

F. *Analysis and optimization of the advect*

This function has a complexity of O(MNO). However, upon analyzing the code of this function, it is evident that there are significant data dependencies between instructions. In this function, because the order of the loops is I, j, k, it lacks spatial locality in data access. However, it is possible to change the loop order to k, j, i allowing for spatial locality in this function. This was the only optimization we applied to this function. Regarding data spatial locality, this function initially lacked it, as each iteration accesses different data in memory.

It has control dependencies due to the presence of if instructions within the for loops, as well as data dependencies because the indices i0, j0, k0 and their increments i1, j1, k1 are calculated based on the current indices i, j and k. This prevents the vectorization of the function.

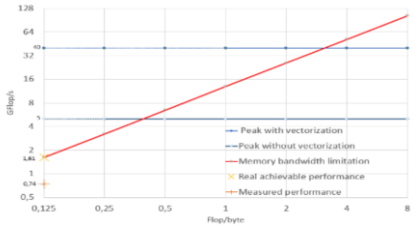
TABLE IV. RESULTS OF THE MEASUREMENTS

version	T(s)	#I	#CC	CPI	L1_Dmiss	Miss/#I
Initial	8.610	20G	28G	1,38	2,3G	8.610
Final	4.595	20.8G	15G	0.72	350M	0.017

Comparing the initial time with the final time after optimizations, we conclude that there was a reduction of approximately half, which was not as significant as expected. We observed a substantial decrease in L1 cache misses and a reduction in the number of clock cycles. Despite these improvements, the execution time remains high, primarily due to the lin_solve function, which has not been optimized efficiently enough.

G. *Final Roofline model*

The Roofline model allows to quickly identify whether an application's performance is limited by the processor's computational capacity (CPU/GPU) or by memory bandwidth. Graph I. shows the graph obtained where we can conclude that the application is below the memory bandwidth limitation line, which indicates that the application is limited by memory bandwidth for this Flop/Byte ratio and is well below the theoretical peak, suggesting that, the problem is being restricted by memory bandwidth, and not by the processor's calculation capacity.



Graph I. Enlarged section of the roofline model graph from Fig. 7, demonstrating what is limiting the application's performance.

III. SHARED MEMORY PARALLELISM (OPENMP-BASED)

Before we begin, the data size has been increased from N=42 to N=84 (8 fold-increase in data size) and a new solver, more efficient and suitable for parallel execution, has been implemented. This solver is a red-black implementation with convergence check which means you divide the grid cells into two subsets, which can be seen as "red" and "black", in the first loop you update only the "red" cells and in the second the "black" ones and instead of run a fixed number of iterations (LINEARSOLVERTIMES), the function checks for the largest "error" or "change" in each iteration (max_c) and stops when this error is less than a tolerance (tol), indicating that the system has converged, this eliminates direct dependencies between adjacent cells, is ideal for parallelism, especially on modern architectures like GPUs or multi-core CPUs, reduces resource contention and improves efficiency in using shared memory and caches, and allows fine-grained control of accuracy and performance.

We apply the observations made in the previous point to single core/threaded optimization and reordered the loops in the new lin_solve function to k, j, i and adjusted the summation order, replacing division with multiplication. We also enabled compiler flags: -O3 for optimization, -funroll-loops for loop unrolling, -ftree-vectorize and -mavx for vectorization, and -march=ivybridge for architecture-specific optimization.

A. *Identification of bottlenecks*

Using gprof, we identified that the functions consuming the most time were lin_solve with 46.77% and advect with 27.89%.

Parallelism was also applied to the other functions with OpenMP when possible and advantageous, for example, in the vel_step function, where in Fig.3, there is the instruction dependency graph, which indicates that multithreading can be applied to this function, but we also need to see what the function does to see if multithreading can really be applied, so this function will not be parallelized as we will explain later.

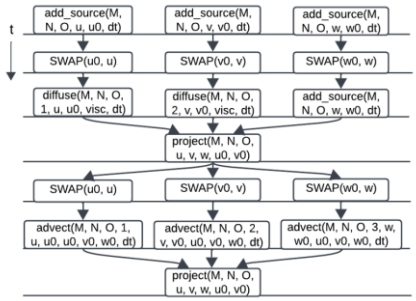


Fig. 3. Shows illustrates the data dependency graph for the vel_step function, with swap being the only instruction that doesn't call a function.

B. *Parallelization of the function lin_solve (par_l)*

The lin_solve function contains two sets of nested loops. A parallel region was created for both set with a barrier between them. While x array updates are independent, reduction(max:max_c) was used in max_**c** to ensure correctness, and old_x and change were declared private to avoid conflicts. The outermost loop (k) in each set was parallelized using #pragma omp for schedule(static) collapse(2) nowait, enabling efficient workload distribution by combining two outer loops into larger chunks and minimizing thread synchronization.

The results of the performance improvement achieved with this parallelization are presented in Table V.

C. *Parallelization of the function advect (par_l_a)*

The advect function contains three nested for loops with independent modifications to the d array across iterations. Parallelization was achieved using with #pragma omp and #pragma omp for schedule(static) collapse(3) nowait to efficiently distribute the workload. Additionally, _mm_prefetch was used to preload data from the d0 array.

The results of this parallelization are show in Table V. Gprof revealed that project takes 44.34% of execution time.

D. *Parallelization of the function project (par_l_a_p)*

This function includes two sets of three nested loops. To optimize, we reordered the loops to k, j, i for better spatial locality and precomputed MAX(M, MAX(N, O)). For parallelization, we created two parallel regions, one for each set of loops. The first region updates div and p, while the second handles u, v, and w, with independence between iterations. Both regions use #pragma omp parallel for schedule(static) to efficiently distribute work across threads.

The parallelization results are shown in Table V. Gprof revealed that the set_bnd with 25.23% is a new bottlenecks.

E. *Parallelization of the function set_bnd (par_l_a_p_b)*

In this function, we began by reordering the loops to improve spatial locality. Then, we created a parallel region to cover all three sets of nested loops. The positions of the array x that are modified are distinct, preventing data races. To distribute the workload across threads, we used #pragma omp for schedule(static), ensuring efficient load balancing with static scheduling. The results are presented in Table V.

F. *Parallelization of the function vel_step*

As mentioned in the earlier phase, while this function can be parallelized, doing so would involve parallelizing calls to already parallelized functions, worsening performance. Therefore, we chose not to parallelize it.

G. *Timing Measurement Results*

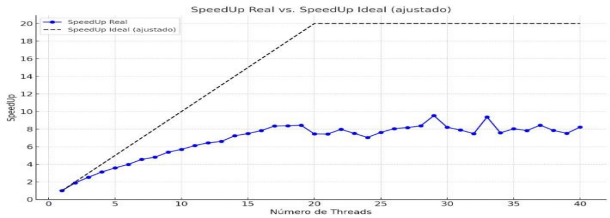
TABLE V. THE RESULTS OF THE TIME MEASUREMENTS WERE OBTAINED USING THE K-BEST MEASUREMENT HEURISTIC TO ENSURE REPLICABILITY, WITH K=5 AND 20 MEASUREMENTS, USING 18 THREADS.

	T(s)		T(s)
original	25.7±0.2	par_l_a_p	4.2±0.1
par_l	23.1±0.1	par_l_a_p_b	2.9±0.1
par_l_a	8.4±0.1	seq	21.7±0.1

Analyzing the table, we see that parallelization reduced execution time, but the final speedup of 7.5 falls short of the expected 18 (with 18 threads and cores). This is due to remaining sequential code and the lin_solve function, which, despite being parallelized, remains time-consuming due to extensive float operations.

H. *Performance speedup graph*

In order to analyze the ideal number of threads, we made a graph comparing the speedup time compared to the already optimized sequential version.



Graph II- Scalability Graph comparing the SpeedUp with the number of threads used.

We can clearly see that speedup is pratically linear up to 19 threads, wich is to be expected since the cluster has 20 cores, and from 20 threads there is an inconsistent increase and decrease in execution time, reaching a variable peak around 29 threads, wich can be explained by the processor´s support for Hyper-Threading. The real speedup is much lower than the theoretical one, wich may be due to the fact that there is an overhead in thread management and frequent context switching between threads, wich is quite costly when several threads compete for the same cores.

IV. PARALLEL VERSION USING ACCELERATORS (GPUS) USING CUDA

In this phase, GPUs were selected over MPI for their superior performance in handling highly parallel tasks with reduced communication overhead. The program was adapted to run on an NVIDIA Kepler K20 GPU, using a systematic approach to enhance performance. The 3D grid size of internal cells was increased to 168 per dimension, marking an 8-fold increase in problem size compared to the previous phase. This implementation takes advantage of the GPU's massive parallelism to efficiently manage the expanded computational workload.

A. Modifications in main.cu file.

In the main.cu file, we modified the allocate_data function to allocate arrays directly in GPU memory using cudaMalloc. The clear_data function now uses cudaMemset to initialize arrays to zero, and the free_data function was updated to release memory with cudaFree.

The apply_events function was modified to directly add data to the GPU memory. For the ADD_SOURCE event type, the density value is copied directly to the corresponding position in the d_dens array using cudaMemcpy. For the APPLY_FORCE event, the forces are copied to the respective GPU arrays (d_u, d_v, d_w). This design ensures that only the necessary values are transferred to the GPU, improving performance and reducing unnecessary memory operations.

In the simulate function, the dens_step and vel_step functions are called, passing the arrays allocated in the GPU as arguments. The sum_density function, since it is called only once and does not consume much time, is executed on the host. Performing the reduction on the GPU could result in slight differences due to the order of operations, making it harder to validate the results. Therefore, the values from the d_dens array are copied to dens.

B. Modifications in fluid_solver.cu.

In the fluid_solver.cu file, we modified almost all functions to be executed on the GPU, with the exception of the vel_step, dens_step, and diffuse functions. These functions remain functionally the same as before. The only difference is that they now call functions that launch CUDA kernels to perform the computations on the GPU.

C. CUDA Optimization for add_source function.

The add_source function was updated to launch a CUDA kernel with 256 threads per block and enough blocks to cover the array size. Each thread updates one element of the array x by adding the product of the corresponding value in s and the time step dt, with a boundary check ensuring memory safety.

D. CUDA Optimization for set_bnd function.

In the set_bnd function, we launch a 2D kernel with 32 threads per block in the x-dimension, ensuring that each warp consists of threads along this direction. The number of threads in the y-dimension is set to 8 threads per block to optimize performance. The total number of threads launched corresponds to the number of positions that need to be modified in each phase.

The set_bnd_kernel is designed such that each thread calculates the boundary positions for each phase, ensuring that threads do not go out of bounds with appropriate conditional checks. The indices are offset by 1, as positions where i = 0 or j = 0 are not modified.

As an optimization, to avoid redundant comparisons of b (such as checking for values 1, 2, and 3 multiple times), these values are passed directly as arguments to the kernel function, eliminating the need for repeated operations. This improves efficiency by preventing unnecessary branching within the kernel. Additionally, the corners of the grid, are handled by only a single thread—specifically, thread 0 in both the x and y directions. This ensures that boundary corner values are computed correctly without redundant calculations.

E. Optimized Kernel Launch Configuration in advect, lin_solve, and project.

The advect, lin_solve, and project functions launch 3D kernels with 32 threads per block in the x-dimension, ensuring that each warp consists of threads along this direction. Since the x-dimension corresponds to the i index, which represents positions that are closer together in memory, this configuration improves memory access patterns and boosts performance. The number of threads in the y- and z-dimensions is adjusted to optimize overall performance. Additionally, enough blocks are launched to cover the entire array size, ensuring efficient parallelization and workload distribution across the 3D grid.

F. CUDA Optimization for lin_solve function.

The lin_solve function has a dependency between calculations, where the red values must be calculated first, and only then can the black values be computed. To handle this, we create two separate kernels: one to calculate the red values and another to calculate the black values. This function executes a maximum of 20 iterations, or until the variable max_c is smaller than tol. The d_max_c variable is allocated in GPU memory, and in each iteration of the loop, it is initialized to 0. At the end of the execution of both kernels, the cudaMemcpy operation is used to copy the value back to the host.

The lin_solve_kernel_Red function is a CUDA kernel where each thread computes a "red" position in the x array. The indices i and j correspond to the x and y directions,

respectively, while the k index is calculated to ensure that only "red" positions are updated.

To compute the k index, threadIdx.z is multiplied by 2, (i + j) % 2 is added, and blockDim.z is also multiplied by 2. This ensures that the kernel exclusively operates on the "red" cells of the grid while avoiding the "black" cells. This approach helps prevent thread divergence, ensuring that all threads within a warp execute the same instructions, which leads to significant improvements in performance and efficiency. Additionally, to prevent computations outside the boundaries, the indices i, j, and k are offset by +1.

The expression for calculating x remains unchanged. The reduction of the value max_c is performed using shared memory. The differences between the old and new values of x are stored in the shared memory array local_max for each block.

The maximum change within a block is computed using sequential parallel reduction with the fmaxf function. Finally, the global maximum value is stored in the max_c variable using an atomic operation (atomicMax_float), ensuring the maximum value is correctly updated across all blocks.

The atomicMax_float function atomically finds and stores the maximum value between a provided value and the value stored in a shared variable. It uses the atomic operation atomicCAS to ensure that the modification occurs without interference from other threads, repeating the process until the swap is successful.

The lin_solve_kernel_black function is almost identical to the lin_solve_kernel_Red function, with the only difference being in the calculation of the index k. For accessing black positions, the expression (i+j+1)%2 is used instead of (i+j)%2.

An optimization that could improve performance would be to perform loop unrolling when reducing max_c. When s is less than or equal to 32, synchronization between threads wouldn't be necessary because there is only one warp. However, when we made this change, the result became incorrect when removing the __syncthreads() instruction. We were unable to identify the cause of this error.

G. CUDA Optimization for advect function.

The advect function calls the advect_kernel, which is designed so that each thread calculates one of the positions in the array d. The indices i, j, and k are derived from the x, y, and z directions, respectively, with an offset of 1, since the positions where i = 0, j = 0, and k = 0 are not modified.

Boundary clamping for the grid is now done using the fminf and fmaxf functions. The values of x, y, and z are first clamped using fminf to ensure they do not exceed the grid boundaries, and then the result is passed through fmaxf to ensure the lower boundary is not below 0.5.

Another optimization is in the calculation of d. The computation has been broken down into multiple variables, which helps better leverage the large number of registers available and reduces redundant calculations, thus making more efficient use of the available hardware resources.

H. CUDA Optimization for project function.

The project function calls two kernels. The first is compute_div_and_init_p, and the second is update_velocity, with calls to set_bnd and lin_solve between the two kernels, which modify the p and div arrays.

The compute_div_and_init_p kernel calculates the divergence (div) and initialize the pressure array (p) to 0. Each thread computes a position in the div array using velocity components (u, v, w) and applies a scaling factor (scale) to the calculation. The scale is precomputed as -0.5f / MAX(M, MAX(N, O)) to avoid redundant calculations. The indices i, j, and k are offset by 1 to ensure boundary positions are not modified.

The update_velocity kernel has each thread calculate the values of the u, v, and w arrays based on the values from the p array in each direction.

I. Analysis of Performance Gains and Optimization Opportunities.

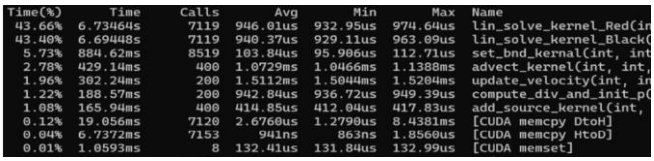
In the Table 3 presents the execution time measurements for the original, sequential, CUDA, and OpenMP versions, obtained using the K-Best Heuristic with K=5 and 15 measurements to ensure replicability. The results provide a comparison of performance across these different implementations. The original version uses the original solver, whereas the other three versions use the red-black solver.

TABLE VI. TIME MEASUREMENTS FOR ORIGINAL, SEQUENTIAL, CUDA, AND OPENMP VERSIONS WITH A SIZE OF 168

	T(s)
Original	2137 ± 0.5
Sequential	428.7 ± 0.3
OpenMp	58.3 ± 0.2
Cuda	17.4 ± 0.1

Upon analyzing the values in the table, we observed that the CUDA version is significantly faster than the sequential version, approximately 24 times faster, and also considerably faster than the original version, likely 122 times faster. However, when compared to the result from the previous phase, the OpenMP version, the performance improvement is only about 3.5 times faster. Given the expectations for GPU execution, we had anticipated a much more substantial efficiency gain compared to the OpenMP version. This discrepancy can be attributed to the fact that the lin_solve function was not well optimized, as seen by analyzing Figure 4, which shows that the kernels called by this function take the most execution time of all the kernels.

Figure 4 shows the total execution time for each kernel, as well as the cudaMemcpy and cudaMemset instructions, obtained using nvprof for a size of 168.



Time(%)	Time	Calls	Avg	Min	Max	Name
43.66%	6.73464s	7119	946.01us	932.95us	974.64us	lin_solve_kernel_Red(in
43.40%	6.69448s	7119	940.37us	929.11us	963.09us	lin_solve_kernel_Black(
5.73%	884.62ms	8519	103.84us	95.906us	112.71us	set_bnd_kernal(int, int
2.78%	429.14ms	400	1.0729ms	1.0466ms	1.1388ms	advect_kernel(int, int,
1.96%	302.24ms	200	1.5112ms	1.5044ms	1.5204ms	update_velocity(int, in
1.22%	188.57ms	200	942.84us	936.72us	949.39us	compute_div_and_init_p(
1.08%	165.94ms	400	414.85us	412.04us	417.83us	add_source_kernel(int,
0.12%	19.056ms	7120	2.6760us	1.2790us	8.4381ms	[CUDA memcpy DtoH]
0.04%	6.7372ms	7153	941ns	863ns	1.8560us	[CUDA memcpy HtoD]
0.01%	1.0593ms	8	132.41us	131.84us	132.99us	[CUDA memset]

Fig. 4. Total time spent by each kernel, cudaMemcpy and cudaMemset instructions for a size of 168.

Analyzing the data provided by nvprof, we can conclude that the two kernels invoked by the lin_solve function consume the most execution time, taking approximately 14 seconds to complete. As such, the lin_solve function currently requires further optimization to achieve better

performance. The remaining kernels consume significantly less time, accounting for only 13% of the total execution time, and are therefore not a bottleneck for the program's performance at this stage. However, as the simulation size increases, these kernels could become more significant and may require optimization in the future.

The main issues in the lin_solve function are frequent and redundant accesses to global memory for reading the values of x, resulting in high latency, as well as the inefficient implementation of the reduction used to find the local maximum. This inefficiency is further exacerbated by excessive synchronization, which leads to reduced performance.

To improve the performance of the lin_solve function, shared memory could be utilized to optimize memory access and reduce latency. Additionally, implementing a more efficient reduction technique for calculating the maximum value (max_c), as mentioned earlier, would significantly enhance performance. By addressing these inefficiencies and optimizing both memory access and the reduction process, the overall efficiency of the GPU implementation could be greatly improved.

In future optimizations, when the kernels of the advect and project functions become a bottleneck, one solution would be to use shared memory to minimize repeated accesses to the same global memory positions by the threads within the same warp.

J. Speedup Analysis for GPU and OpenMP Versions.

In Figure 5, a graph is presented that analyzes the speedup performance as a function of problem size (input size), comparing two approaches: GPU and OMP.

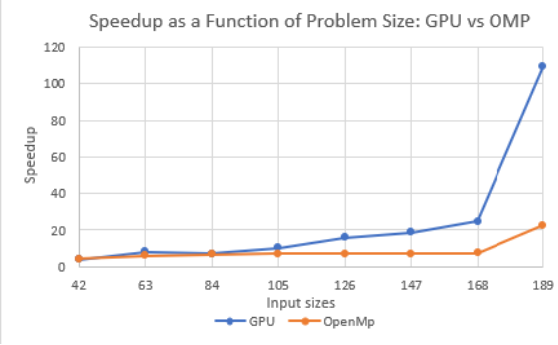


Fig. 5. Speedup comparison between the GPU and OpenMP versions as a function of input size.

It is observed that, for input sizes up to 84, the speedup is almost identical for both the GPU and OpenMP versions, indicating similar performance for smaller problem sizes. In these cases, the use of a GPU is not advantageous due to the small amount of data, which limits the ability to fully exploit the parallelism capabilities of the GPU and causes the initialization overhead to become a significant factor relative to the total execution time.

However, as the input size increases, the GPU's performance improves dramatically, with an exponential increase in speedup starting from an input size of 168. In contrast, the OpenMP approach maintains relatively consistent performance, which is notably lower than the GPU's performance.

Although this behavior is not directly observed in the current graph, it is likely that the GPU speedup will eventually stabilize as the problem size continues to increase. This phenomenon, known as diminishing returns, occurs when the overhead of managing parallelism, memory access, and synchronization limits the performance gains. While the GPU demonstrates significant speedup for medium-sized problems, at very large problem sizes, the speedup will likely plateau due to hardware constraints and memory bottlenecks.

From the results, we can conclude that as the problem size increases, the speedup for both GPU and OpenMP improves. However, the GPU version typically shows more significant speedup due to its ability to process many elements in parallel. While the OpenMP implementation is faster than the sequential version, it does not scale as efficiently for larger problem sizes. This analysis highlights the superiority of the GPU for larger problems, whereas OpenMP does not exhibit the same scalability and is more suitable for smaller problem sizes.

REFERENCES

[1] Slides of theoritical parallel programming classes

ANNEXES

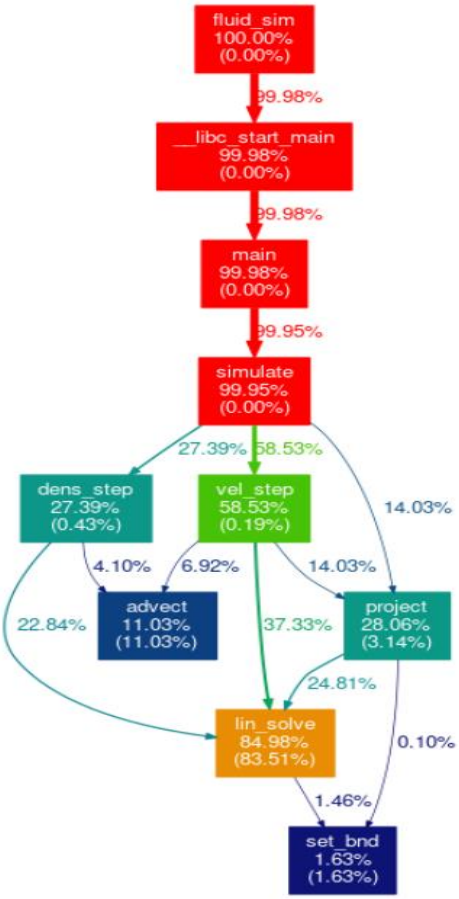


Fig. 6. Performance graph of the program divided by functions, obtained using gprof2dot.

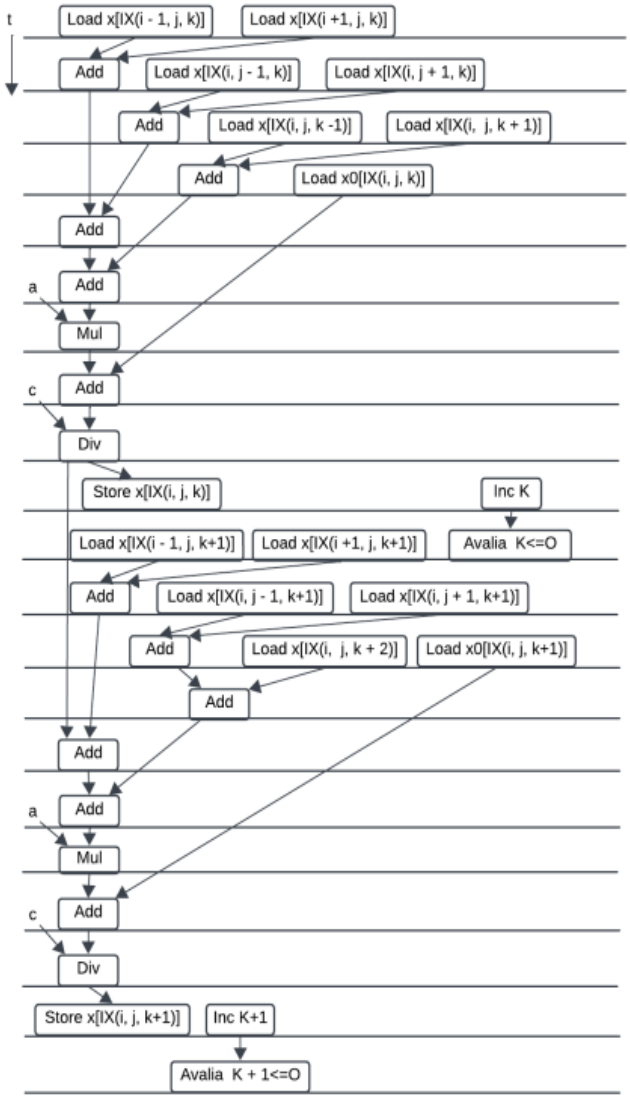


Fig. 7. A dependency graph of instructions for the function lin_solve, created based on the Ivy Bridge architecture.

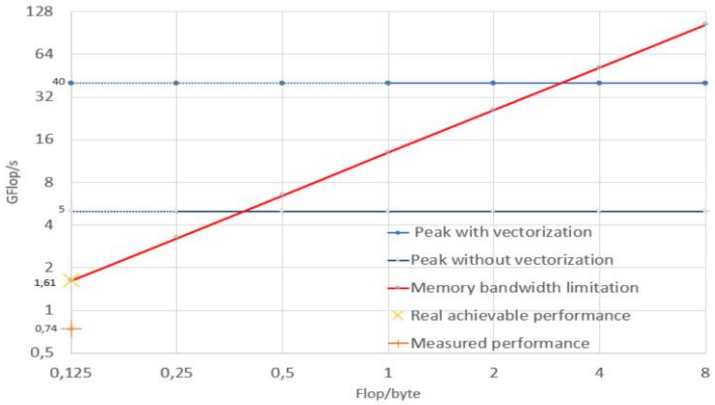


Fig. 8. Final RoofLine model of the application with the bandwidth bound and the compute bound calculated for the backend of the cluster used, where the maximum memory bandwidth was calculated through the STREAM benchmark

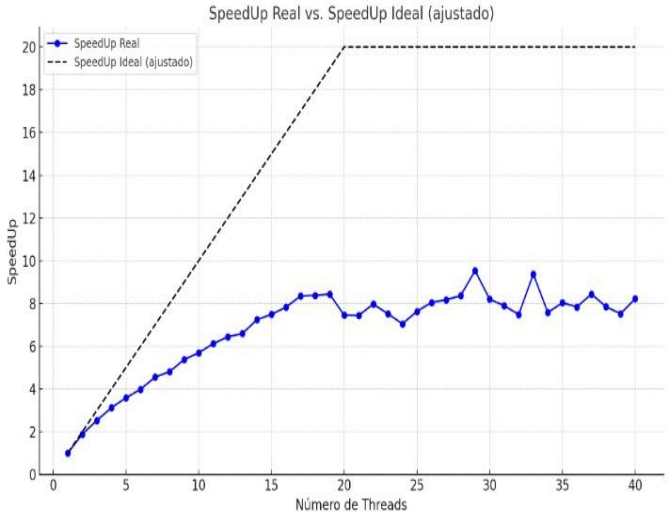


Fig. 9. Scalability Graph comparing the SpeedUp with the number of threads used