Phase 1: Parallel Programming Project 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, a 3D version of Jos Stam's stable fluid solver.**

Keywords*— fluid_sim, lin_solve, CPI, number of instructions, clock cycles and cache misses.

I. Introduction

This project aims to analyze and optimize a 3D version of Stam's solver to reduce execution time. We will identify performance inefficiencies and apply optimization techniques to enhance computational efficiency while preserving output.

This report details the initial phase, including code analysis and performance metrics such as CPI (Cycles Per Instruction) and cache miss rates. In this phase, simple single-core optimizations will be implemented, with more advanced optimizations planned for future stages.

II. 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. 5 is presented, focusing on the most relevant aspects for the program analysis. This figure highlights the functions that consume the most time.

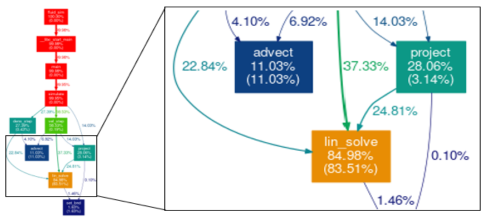


Fig. 1. Enlarged view of the most time-consuming functions from Fig. 5, highlighting critical areas for performance analysis.

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.

A. *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.

III. Analysis and Optimization of the lin_solve

A. *Complexity Analysis*

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.

B. *Analysis of Data Dependencies and Introduction of Optimizations*

In Fig. 6 (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. 6, specifically the part of the graph that demonstrates what prevents parallelism and increases the program's execution time.

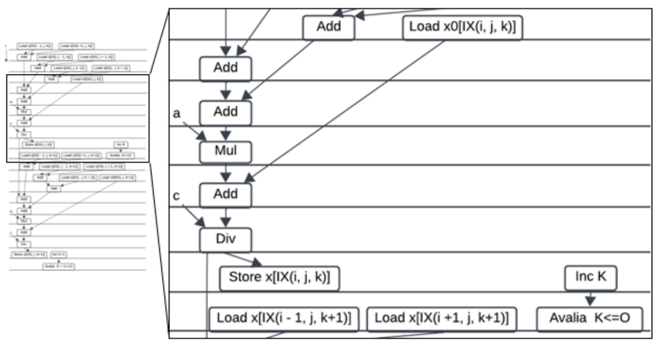


Fig. 2. Enlarged section of the instruction dependency graph from Fig. 6, demonstrating the factors preventing code parallelism.

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

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.

C. *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.

D. *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. Gave this, the vectorization of this function is not possible, as confirmed by compiling with the flag -fopt-info-vec-all.

IV. 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 FINAL CODE MEASUREMENTS

T(s)	#I	#CC	CPI	L1_Dmiss	Miss/#I
4.595	20,8G	15G	0,72	350M	0,017

V. 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. Figure 3 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.

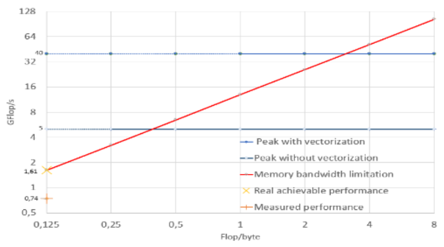
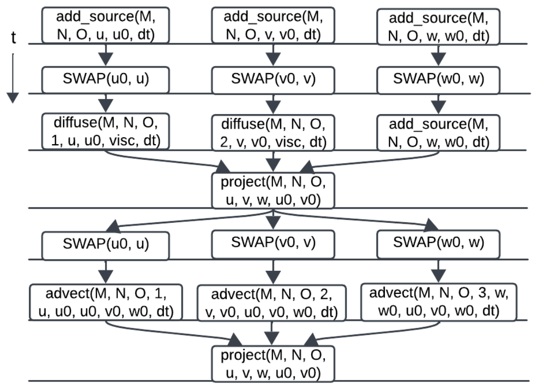


Fig. 3. Enlarged section of the roofline model graph from Fig. 7, demonstrating what is limiting the application's performance.

VI. Optimizations for the Next Phase

In Fig.4, there is an instruction dependency graph for the vel_step function, which indicates that multithreading can be applied to this function. Additionally, we plan to replace the lin_solve function with a more efficient alternative.



t
add_source(M, N, O, u, u0, dt)
add_source(M, N, O, v, v0, dt)
add_source(M, N, O, w, w0, dt)
SWAP(u0, u)
SWAP(v0, v)
SWAP(w0, w)
diffuse(M, N, O, 1, u, u0, visc, dt)
diffuse(M, N, O, 2, v, v0, visc, dt)
add_source(M, N, O, w, w0, dt)
project(M, N, O, u, v, w, u0, v0)
SWAP(u0, u)
SWAP(v0, v)
SWAP(w0, w)
advect(M, N, O, 1, u, u0, u0, v0, w0, dt)
advect(M, N, O, 2, v, v0, u0, v0, w0, dt)
advect(M, N, O, 3, w, w0, u0, v0, w0, dt)
project(M, N, O, u, v, w, u0, v0)

Fig. 4. Shows illustrates the data dependency graph for the vel_step function, with swap being the only instruction that doesn't call a function.

References

[1] Slides of theoritical parallel programming classes

ANNEXES

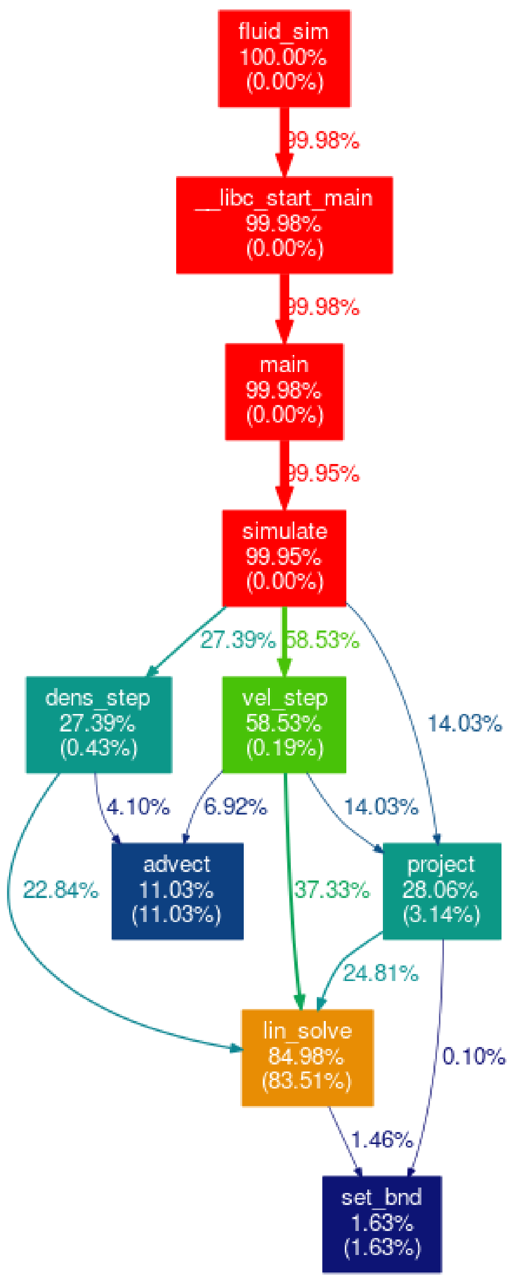


Fig. 5. Performance graph of the program divided by functions, obtained using gprof2dot.

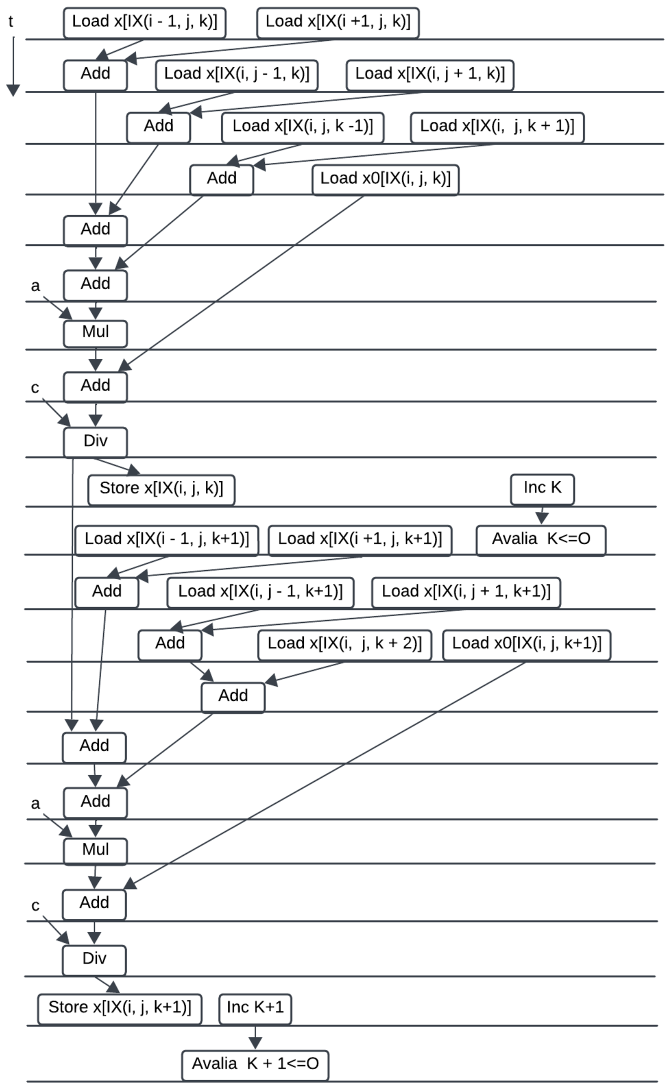


Fig. 6. A dependency graph of instructions for the function lin_solve, created based on the Ivy Bridge architecture.

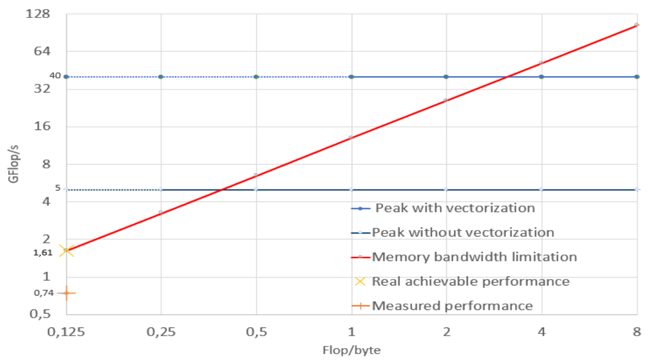


Fig. 7. Final RoofLine model of the application with the bandwidth bound and the compute bound calculated for the backend of the cluster used, where the maximum memory bandwidth was calculated through the STREAM benchmark