Work Assignment - Phase 1

*

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*—For the first phase of the work assignment, we optimized a single-thread fluid simulator written in the language C/C++, starting by analysing and profiling the code, to the improvements in instruction-level parallelism, memory hierarchy, and vectorization.**

***Index Terms*—profiling, instruction-level parallelism, memory hierarchy, styling, vectorization**

I. Introduction

The first phase of this work assignment consists in optimizing the fluid simulator code. The metric to evaluate our results will be the final execution time of the program, but, more importantly, this phase aims at using analysis, the exploitation of instruction level parallelism, the memory hierarchy, and vectorization to reduce execution time, as well as the number of instructions, clock cycles, and L1 cache misses while maintaining the code legibility as much as possible.

II. Methods Of Execution And Code Analysis

The code was executed in the group´s personal machines, but was essentially measured within the the SeARCH cluster at the University of Minho. The runs on the cluster were done with *perf* to get the execution time, number of instructions, number of clock cycles, and the number of L1 cache misses. We opted to measure in a exclusive environment so that metrics like the cache misses were as close as possible to locally running the program.

For every optimization, the code was executed 10 times, so we could get the best execution time of all the executions, excluding outliers. We opted to select the best execution time that is within a maximum distance of 5% from at least two other execution times.

The code was profiled using the tools *perf* and *gprof*, and occasionally, we opted to use *gprof2dot* to get a visual graph of the profiling, known as a call graph.

The 11.2.0 version of GCC was used to compile the program, so all the techniques and flags should be considered for that exact version. The program was executed with the flags -pg (enables profiling for performance analysis using *gprof*), -Wall, (enables a comprehensive set of compiler warnings to catch potential issues), -O2 (to compile with optimizations, reduce execution time, and improve overall performance), but later on will be changed to a higher optimization level.

III. Analysis Of The Original Program

The program without any optimizations (this was a run without the flags mentioned prior), 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.

IV. Optimizations

A. 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.

B. 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.

C. 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.

D. 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. 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.

E. 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. Until now, we were testing based on the level two of optimization, but now we decided to take it to the next level. So, the first flag we introduce is the "-O3" flag. This flag improves the instruction-level parallelism, reduces branching, and improves cache locality through loop transformations. It generally maximizes the performance for most programs. Hence why some of the previous optimizations only enhanced the results on this level. The flag "-march=native" makes the compiler generate code optimized for the exact CPU architecture of the machine. This will cumulate in efficient use of instructions. The "-funroll-loops" flag enables loop unrolling explicitly. This technique gains in increasing instruction-level parallelism, and making it easier for the CPU to keep the instruction pipeline full. We also used "-ffast-math" that enables optimizations like reordering of floating-point operations and the use of hardware-specific instruction, trading of some accuracy in the process of the calculations. Finally, "-flto" improves the overall performance of large programs by optimizing function calls and memory access across different source files, which is great for our program, considering all the constants and multiple source files. The results of just applying these flags (no other optimizations) and everything combined can be found in the table I

F. 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.

G. 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.

V. Conclusion

Throughout this work, we applied several levels of optimization to a C++ fluid simulator, targeting performance improvements by reducing instruction-level parallelism, clock cycles, and L1 cache misses. By first analysing the program's *hotspot*, we were able to identify significant opportunities for optimizations in specific functions, especially the nested loops in the *lin_solve* function. Our efforts focused on techniques like separating independent computations, loop unrolling, loop interchange, and applying compiler flags, all of which contributed to a performance boost. In terms of overall speedup, we achieved a good performance enhancement, in our opinion. Although certain optimizations, such as vectorization, were not feasible, the combination of implemented strategies successfully led to a faster execution of the fluid simulator, cumulating in a 10.003 speedup, when comparing the final state of code with the first one.

VI. Apendix

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 Instruc-tions	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

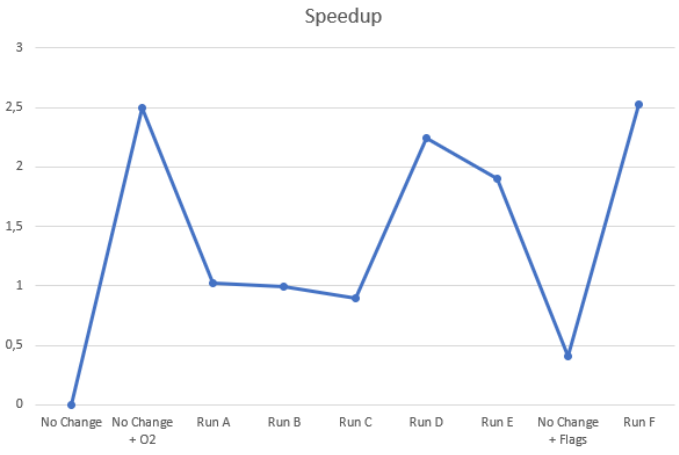

Fig. 1. Speedup, relative to previous state of code

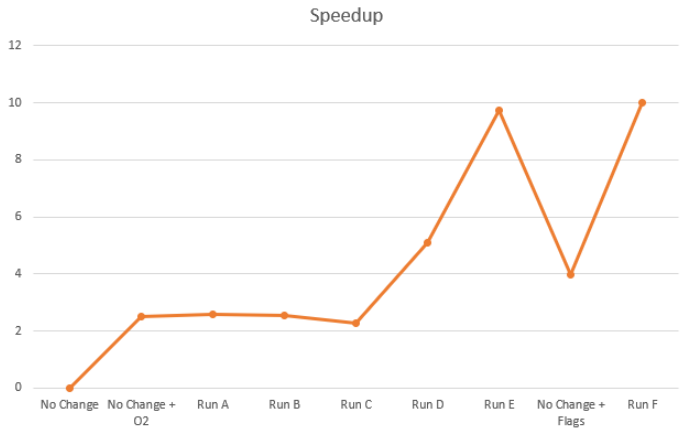

Fig. 2. Speedup, relative to FIRST state of code

B. Theoretical proof for results obtained

1) Separating Independent Computations (A): Considering the C code for the inner part of the nested loops, we can try to measure how much will be gaining from this optimization. Being "L" the number for LineSolver, and N^3 representing the $O \times N \times M$, then $L \times N^3$ equals to the number of iterations. Now analysing the inner code, we have six memory loads (for each *x* value), five instruction for the summation, one for the multiplication by the constant *a*, one for the load of *x0*, one for the sum with *x0*, one for the division by the constant *c* and finally the value is stored. Of these 16 instructions, we can only consider as independent the six memory loads, three summations (the summations of the 6 values broken down in pairs) and the load of *x0* that can happen at the same time. Thus, instead of 16 cycles for the serial execution, we could reach 7 cycles on a parallel execution. This could mean that the program, with the right level optimization, would go $2.29\times$ faster. Checking the results on table I, we can see just a small decrease on the number of clocks, due to the fact that the compiler was already dealing well with this section of the code.

2) Loop Interchange (C): The change from the original i-j-k loop order to k-j-i in the *lin_solve* function significantly improves performance by enhancing memory access patterns and cache efficiency. In C and C++, arrays are stored in row-major order, meaning that elements with the same first index are stored consecutively in memory. In the original i-j-k loop structure, where i is the outermost loop and k is the innermost, memory accesses jump across non-adjacent elements, leading to poor cache locality. This is because each time the innermost loop iterates over k, the elements being accessed are far apart in memory due to the varying i index, resulting in frequent cache misses. However, by changing the loop order to k-j-i, where i becomes the innermost loop, the iterations now access consecutive elements in memory, as data is stored contiguously along the i dimension. This greatly improves spatial locality, meaning that the CPU can fetch and process multiple consecutive memory locations in a single

cache line load, reducing cache misses and improving overall performance. Modern CPUs are designed to handle such sequential memory access efficiently, so this loop reordering helps make better use of the cache, particularly in large 3D grids like those used in fluid simulations. By ensuring that the innermost loop accesses adjacent memory locations, the new k-j-i loop order optimizes cache usage.

3) Loop Blocking (D2): An alternative for the Loop Unrolling technique, combined with the previous optimizations, is blocking the loops. The idea is to improve cache utilization by ensuring that data needed for computation stays in the CPU cache as long as possible. This technique works by breaking the grid (our *array* represents a grid) into smaller blocks that fit into the CPU cache. By processing small blocks that fit into the CPU cache, the neighbouring cells (which are needed for computation) are likely to stay in cache, leading to fewer cache misses and better memory locality. Due to the algorithm used in our code, by breaking the three nested *for* loops, the readability of the code and its complexity became more complicated for programmers unaware of the technique. For the original version we had the following equation:

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

To implement loop blocking, we need to add a new inner layer to the algorithm, that can be translated to mathematical notation by the following equation:

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

The results regarding this optimization can be found in table I. As shown, the number of instructions went up, but the number of cycles had a bigger fall. The cache misses had a slight increase, but not too much. From this, not only the memory locality was improved, but so was the parallelism.

4) Loop Blocking (D): Let's consider the original version of the *lin_solve* function, which has three nested loops:

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

The total number of iterations in this original version is:

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

Now, let's look at the optimized version with loop unrolling:

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

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

In this unrolled version, the innermost loops over i, j, and k are executed four times within each iteration, effectively reducing the total number of loop iterations. The total number of iterations in the unrolled version is:

$$\mathrm{L}\times\left\lceil\frac{O}{2}\right\rceil\times\left\lceil\frac{N}{2}\right\rceil\times M=\mathrm{L}\times\left\lceil\frac{N^3}{4}\right\rceil \tag{8}$$

Compared to the original version, the unrolled version reduces the total number of iterations by a factor of 2 in the k and j loops, while keeping the i loop unchanged. This theoretical speed up assumes that the unrolled code can be executed without any additional overhead, and that the compiler can effectively utilize the available CPU resources to execute the unrolled instructions in parallel. In practice, the actual speed up may be slightly lower due to factors such as cache effects, branch prediction, and potential limitations in the CPU's ability to exploit the increased instruction-level parallelism.

5) Flags (E): 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. Until now, we were testing based on the level two of optimization, but now we decided to take it to the next level. So, the first flag we introduce is the "-O3" flag. These techniques reduce the overhead associated with function calls and ensure that iterations inside loops run faster. Since the code contains numerous nested loops, especially in performance-critical functions like lin_solve, this flag showed substantial gains by optimizing how calculations are processed, resulting in faster execution. This flag improves the instruction-level parallelism, reduces branching, and improves cache locality through loop transformations. The flag "-march=native", directs the compiler to generate code specifically optimized for the CPU architecture of the machine. This leads to efficient use of hardware capabilities such as SIMD instructions (like AVX and SSE), which can execute operations on multiple data points simultaneously. In a program like ours, which involves extensive floating-point calculations and array operations, using the native architecture allows the processor to perform these tasks more efficiently, improving memory cache usage and register allocation. To further optimize the loops, we applied the "-funroll-loops" flag , which explicitly enables loop unrolling. This technique increases instruction-level parallelism by reducing the overhead caused by loop control instructions, such as branch checks and index increments. By unrolling the loops, the CPU is able to execute more iterations per cycle, keeping the instruction pipeline full and reducing the cost associated with jumping back and forth within loop structures. Given the heavy presence of nested loops in the program, unrolling them brought significant performance improvements. We also used the "-ffast-math" flag, which enables optimizations related to floating-point operations. This flag allows the compiler to reorder calculations and use hardware-specific instructions that prioritize speed over strict adherence to IEEE 754 floating-point standards. While this trades off a small degree of accuracy, it accelerates the computation of the numerous floating-point operations within the simulation, making it a beneficial tradeoff for this type of application. Finally, "-flto" improves the overall performance of large programs by optimizing function calls and memory accesses across

multiple source files. Our program, which consists of several source files and relies on shared constants, benefits greatly from this flag as it reduces the overhead involved in function interactions and data access. This was particularly useful for optimizing interactions between functions like lin_solve and others that share data. Additionally, although "-ftree-vectorize" was employed to enable automatic vectorization of loops, the dependencies between various data points in the program likely limited its effectiveness. This flag typically enables the compiler to convert scalar operations into SIMD instructions, but due to the data dependencies in the nested loops, we may not have seen the full potential of vectorization. The results of just applying these flags (no other optimizations) and everything combined can be found in the table I