# Optimizations of Jos Stam's Fluid Algorithm
## Parallel Computing - Phase I

Carlos Ribeiro
*PG55926*

Diogo Matos
*PG55934*

Júlio Pinto
*PG57883*

***Abstract*—This report explores the optimizations made to Jos Stam's Fluid Algorithm for calculating real-time fluid dynamics. The optimized code was written in C++.**

## I. Introduction

In the first phase of this parallel computing project, we were tasked with optimizing a C++ code simulation of real-time fluid dynamics. In this phase, we were asked to keep the optimizations and our code single-threaded and focus on algorithmic and mathematical improvements, as well as memory management and code organization. Our approach to optimization prioritizes runtime and maintainability. In this paper, we talk about all the ideas and optimizations we thought about or implemented.

## II. Analysis and Code Profiling

In the first step of this project, we focused on understanding the problem. For that, we searched for examples of Jos Stam's Fluid Algorithm applied as well as papers about it. After understanding the problem and how it uses certain equations like the Navier-Stokes Equations we understood that in terms of the equations itself, there wasn't a lot of room for improvement.

Afterwards, we decided it would be best to start profiling the code. To do that `gprof` and `perf` were instrumental. These programs provide capable interfaces that allow us to profile and analyze the code in the best way possible. Utilizing data collected during program execution, these tools allow us to identify the functions that are most intensive and time-consuming. These tools also allow us to record events such as CPU cycles, executed instructions and, without a doubt one of the most important things for us, cache misses.

Using these tools, we managed to obtain a graphic (Fig. 1) showing us the most intensive functions in the code, such as `lin_solve`, enabling us to understand the best route to optimize it.

## III. Compiler Optimizations

During our analysis of the code, we deducted that using simple flags like `-O0` would dramatically dampen performance. Knowing this, we decided to start the project by instantly using the `-O3` flag. Using this flag, we enable optimizations such as:

- **Loop Unrolling**
- **Vectorization**
- **Inline Functions**
- **Register Allocation**

Having established that these optimizations will be done, we decided to run the provided code using this flag, obtaining an average time of

$$11.636 \pm 0.540 \text{ seconds} \tag{1}$$

when running it 3 times in a row, using the default `event.txt` file, the result being the following value:

$$81981.3 \tag{2}$$

Knowing this, we decided to try using a combination of different flags to obtain better results in terms of run time.

After much research and discussion, we concluded that these flags were be the best ones in terms of performance:

`-Ofast -march=native -ftree-vectorize -mavx`

Using these flags, we were able to achieve better run times since each of these flags has the following effect:

- `-Ofast` allows for greater optimization when converting C++ code into machine code and enables some optimizations that break strict things like floating point arithmetic.
- `-march=native` allows for enhancements in respect to the computer you are running and compiling the code on.
- `-ftree-vectorize` allows the compiler to vectorize as many instructions as it can automatically.
- `-mvax` allows the usage of AVX intrinsics in the code.

With this in mind, we decided to run the project once more without changing the code. Time wise, we obtained:

$$7.7771 \pm 0.0823 \text{ seconds} \tag{3}$$

But due to the usage of `-Ofast`, we obtained a slightly different value:

$$81981.6 \tag{4}$$

## IV. Code Optimizations

After analyzing the possibilities with compiler flags, we decided to advance towards code optimizations.

### A. *Optimization of Divisions*

To begin with, we identified all instances where divisions were being performed. We noticed that many of these divisions occurred repeatedly within loops. Since the denominator values remained constant throughout the loop iterations, we optimized the process by precomputing the inverse of the denominator and storing it in a variable before entering the loop. We replaced the division operations inside the loop with multiplications, which are generally 3 to 5 times faster. This optimization was applied across all functions with multiple iterations to improve overall performance.

```
float invA = a / c, invC = 1 / c;
```

### B. Optimization of Conditionals

Another issue we identified was the use of conditionals in certain loop iterations. These conditions were based on constant values and only affected the multiplication factor. In these cases, we moved the condition checks outside the loop, determining the factor beforehand. This change improves performance by minimizing branch predictions, which reduces the likelihood of branch misses. While branch predictions are usually accurate, this adjustment provides a slight efficiency gain by eliminating unnecessary checks within the loop.

```
int loopMN = 1, loopNO = 1, loopMO = 1;

switch (b) {
  case 3: loopMN = -1; break;
  case 2: loopNO = -1; break;
  case 1: loopMO = -1; break;
}
```

### C. Linear Accesses

One thing we noticed in the macro `IX(i,j,k)` was the way the array was organized, being the outermost variable `k`, while `i` was the innermost. When analyzing the loops, we realized these were being accessed with `i` being the outermost, while `k` was the innermost loop. This affects time since it means we were jumping around the array instead of accessing this array linearly. We went from accessing the array like this: `[1,4,7,10,2,5,8,11,3,6,9,12]` to: `[1,2,3,4,5,6,7,8,9,10,11,12]`.

This way, we changed the way the arrays were being read and written, allowing us to achieve better times due to spatial locality.

```
for (int k = 0; k <= O + 1; ++k) {
  for (int j = 0; j <= N + 1; ++j) {
    for (int i = 0; i <= M + 1; ++i) {
```

### D. Memory Tiling

Our loops processed large data, so to optimize cache performance, we refactored them to iterate in blocks. To determine the optimal block size, we used a gradient-based approach and found that the best size, among powers of 2, was 4. This heavily impacted our performance, seeing how the ratio of cache misses decreased, therefore making the program faster.

```
float kc = std::min(kk + BLOCK_SIZE, O + 1);
float jc = std::min(jj + BLOCK_SIZE, N + 1);
float ji = std::min(ii + BLOCK_SIZE, M + 1);

for (int k = kk; k < kc; k++) {
  for (int j = jj; j < jc; j++) {
    for (int i = ii; i < ic; i++) {
```

### E. Pre-computation of values

One thing we did understand was some functions tend to use certain values that can be precomputed. This way we decided to compute these values before they are needed. This ended up being useful in terms of computing and automatically vectorizing certain values that are needed for loops. In the end, this didn't impact our time as much as we would expect.

```
for (int k = 0; k <= O + 1; ++k) {
  for (int j = 0; j <= N + 1; ++j) {
    for (int i = 0; i <= M + 1; ++i) {
      int idx = IX(i, j, k)
      precomp[idx] = x0[idx] * invC;
    } } }
```

### F. AVX Intrinsics

One thing we wanted to do in our project was use AVX intrinsics to vectorize the code. Observing functions such as `lin_solve` we understood that these functions can't be vectorized due to data dependencies. On another note we did understand that functions like `advect` could be optimized with AVX, for that, we decided to create the class `Vector` which allows us to create different operations to easily use intrinsics. In the end, we decided to end up not using intrinsics, but this way managed to achieve a clean and readable way to implement AVX intrinsics in the future.

### G. Minor Changes

To finish, we did some minor optimizations and changes, like storing some values in variables to keep them in the cache, removing smaller calculus inside a lot of loops, changing the way some of them work, as well as, changing some of the macros to use standardized functions. Most of these changes weren't relevant in terms of time but help keep the code cleaner and easier to understand and read.

## V. Conclusion and Results

As specified by the assignment, every test took place on the SEARCH cluster using gcc 11.2.0. With that in mind, with all these optimizations we obtained the following time:

$$3.1624 \pm 0.0161 \text{ seconds} \tag{5}$$

Maintaining the same value of 81981.6 and with a total of 2.95% misses of all L1-cache hits. We can better understand the way the code has been changed by analyzing the results given to us using `gprof` (Fig. 2), as well as understand how the time has progressed (Table I)

We believe that this task allowed us to better understand what kind of "easy" steps can be taken in order to minimize the necessary time needed by a program and are excited to learn more about what can be done next.

## VI. References


(1) Jom's Stam, Real-Time Fluid Dynamics for Games, 2003

## VII. Appendix

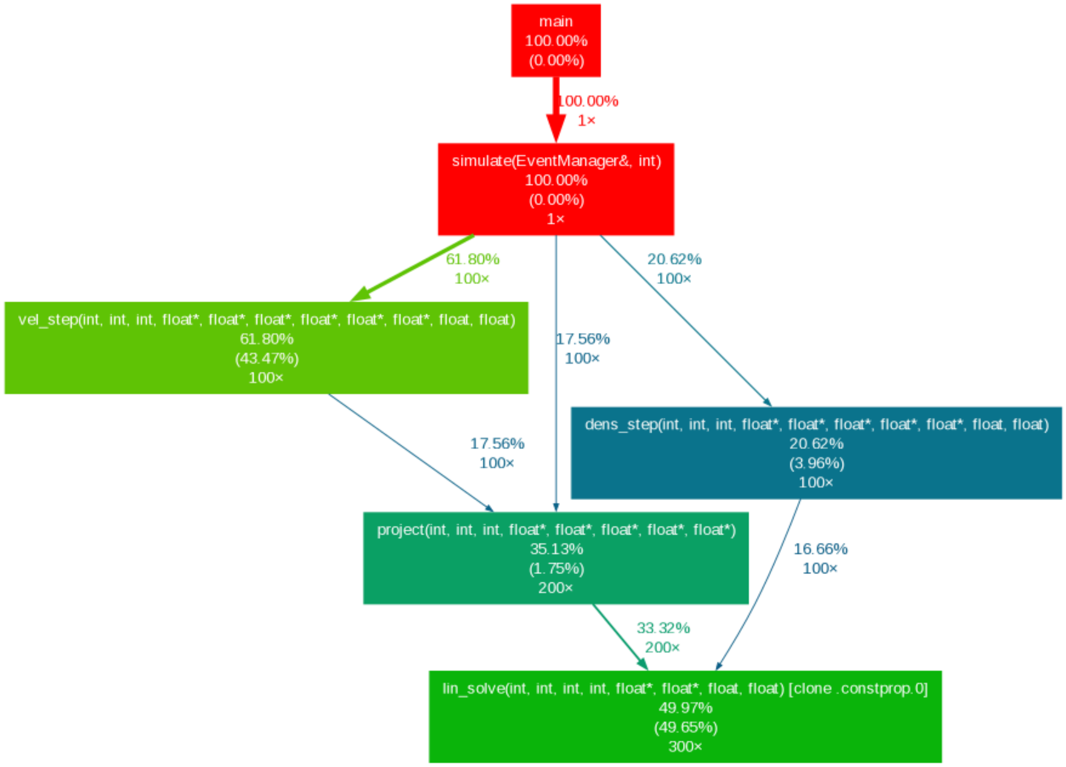


Fig. 1: First Execution Time Graph

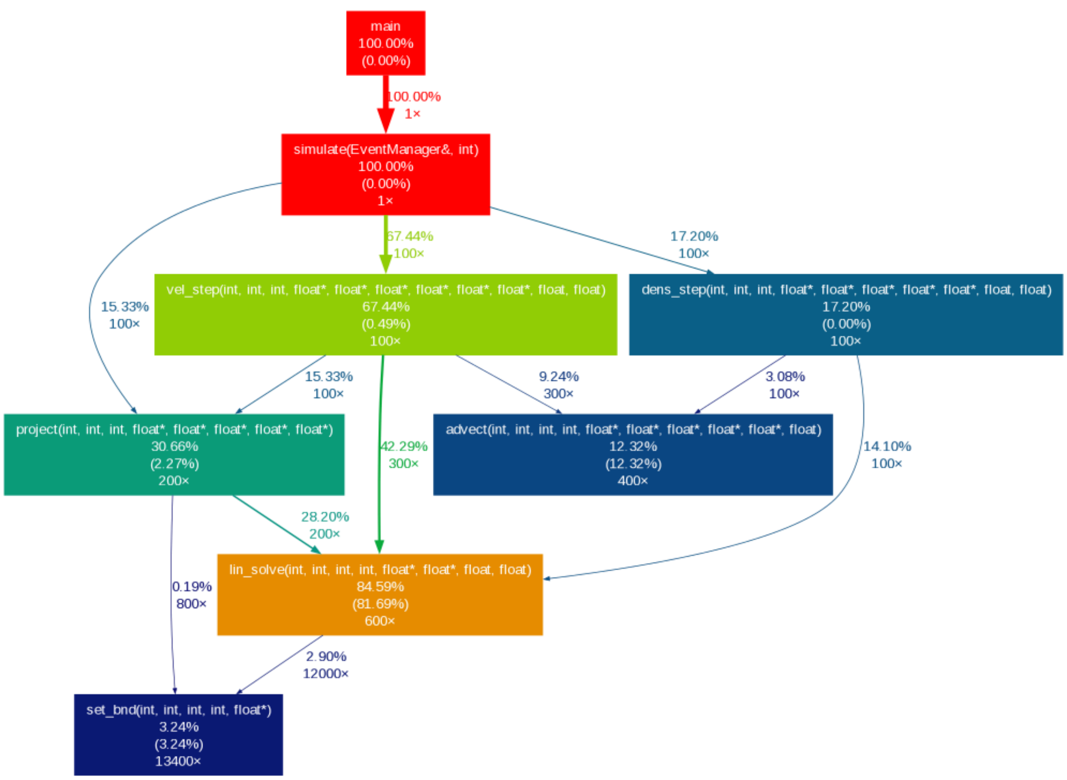


Fig. 2: Final Execution Time Graph

TABLE I: OVERALL TIME FOR VERSION

| **Version** | **Flags** | **Time** (s) | **Differnce** (s) |
|---|---|---|---|
| Teacher's Version | `-O0` | 61.53 | 20.68 |
| Teacher's Version | `-O3` | 11.636 | 0.540 |
| Teacher's Version | `-Ofast -march=native -ftree-vectorize -mavx` | 9.208 | 0.526 |
| Conditionals + Divisions | `-O3 -march=native -ftree-vectorize -mavx` | 5.9692 | 0.0578 |
| Conditionals + Divisions | `-Ofast -march=native -ftree-vectorize -mavx` | 4.0760 | 0.0536 |
| Conditionals + Divisions + Memory Tiling + Loop Orders | `-Ofast -march=native -ftree-vectorize -mavx` | 3.6055 | 0.0978 |
| Final Version | `-Ofast -march=native -ftree-vectorize -mavx` | 3.1624 | 0.0161 |