Analysis and Optimization of a 3D fluid solver

Afonso Magalhães
PG53598

Diana Teixeira
PG53766

Artur Leite
A97027

I. Introduction

Through this report, we will look to present an analysis and optimization of a 3D fluid dynamics simulation based on Jos Stam's stable fluid solver, using the original code provided in a GitHub repository. Our primary objective with this work will be to reduce the program's execution time while preserving the simulation output. To achieve this, we will first identify performance bottlenecks and explore potential optimizations, such as improving data locality and supporting vector processing. By applying these optimizations correctly, we can ensure that performance will be enhanced without compromising code readability. All changes and their impact on execution time are documented and evaluated.

II. Code Analysis

A. *Identifying the Bottleneck*

In order to identify the code segment that takes the most execution time, we used the gprof2dot tool to generate the output in fig.1.

As we can see by looking at the graph, the lin_solve function takes up the majority of execution time, so that's where we should focus our efforts. Despite this, we will also attempt to modify some other functions, as long as changing them proves to wield a significant performance boost.

B. *Assembly code*

We analyzed the original code and its corresponding assembly version and managed to identify the snippet that details the instructions executed by each internal cycle of the lin_solve function, present on fig.2 where, as we can see, there are 14 instructions per cycle.

C. *Dependencies*

After analyzing the Assembly snippet, we constructed the dependency graph in fig.3.

We identified 8 Write after Read dependencies and 11 Read after Write dependencies in total. Taking a closer look at the original code, we believe these dependencies are related to several characteristics that the function has, which we will look to highlight further ahead.

D. *Initial Execution Profile*

In order to establish an initial baseline that allows us to analyze whether our implementations are improving performance or not, we used the perf tool when executing the unmodified code. The results obtained are shown in fig.4.

III. Preliminary analysis

With the important data we've gathered about the problem, we can now thoroughly analyze the code's characteristics. This will allow us to identify optimizations that can yield significant performance improvements based on those characteristics. We will be focusing primarily on the **lin_solve** function.

A. *Function Characteristics and Breakdown*

a) ***Triple nested loops**:*

- The function iterates over a 3D grid
- This type of loop structure is **O(M x N x O)** in complexity, which can become expensive for large grids.

b) ***Memory Access Pattern**:*

- The most significant performance factor here is the access to the array **x** and **x0** inside the loops.
- The array accesses depend on neighboring values, suggesting this is a finite-difference method.
- We can also notice that memory access patterns imply that neighbouring grid points are accessed. However, depending on the storage format of **x**, these could result in cache misses.

c) ***Floating Point Arithmetics**:*

- There are multiple floating point additions (**addss**) as well as a division (**divss**).
- Floating point divisions are particularly more expensive in comparison to multiplications.

d) ***Boundary Condition Updates**:*

- After each full iteration of the 3D grid, a boundary condition is updated using **set_bnd()**. This ensures that the solution satisfies physical constraints at the edges, but it adds extra overhead.

B. *Potential Optimizations*

Given the functions characteristics, here are some potential optimizations that we believe could work in regards to bettering performance. We don't discard the possibility that some of these might not improve performance as much as expected.

a) ***Loop Unrolling**:*

Loop unrolling involves increasing the number of operations per iteration, essentially "unrolling" the loops manually, thus reducing the number of loop operations.

This reduces the overhead of incrementing and checking loop indices (**edx, r10d, etc.**) and can increase instruction-level parallelism.

Taking **lin_solve** into account, the nested loops have a significant overhead. By unrolling the innermost loop, we can reduce the number of iterations and improve performance.

b) ***Cache Blocking**:*

Cache Blocking involves restructuring the computation in order to operate with blocks of data that better fit the CPU Cache, reducing cache misses.

The function accesses **x[IX(…)]** and **x0[IX(…)]** in a strided fashion, meaning that consecutive accesses in the loop are not necessarily accessing consecutive memory locations. This can cause cache misses, leading to performance degradation.

By dividing the 3D grid into smaller blocks and processing one block at a time, the working set can fit into the CPU's cache. This would reduce cache misses and improve memory access efficiency.

c) ***Recomputing Divisions**:*

Given that divisions are significantly slower than multiplications, we can precompute **1/c** and use multiplication instead.

In the **lin_solve** function, **c** is a constant throughout the whole loop, so replacing it with **1/c** and switching division with multiplication should be relatively straightforward.

d) ***SIMD Vectorization**:*

Vectorizing the code, allows a CPU to perform the same operation on multiple data points simultaneously by using special vector instructions.

Modern CPUs can operate on vectors of 4 or more floating-point values in parallel. This can greatly reduce the time spent in the inner loop.

The operations inside the innermost loop are highly parallelizable. Vectorizing these operations would allow the CPU to process multiple grid points at once, increasing throughput. We can apply vectorization manually or use certain flags. We will analyze which of these increases performance more.

e) ***Spatial Locality**:*

Taking into account the IX macro, we observed that the i variable has minimal influence. Specifically, incrementing i by one only results in incrementing the output by one, whereas changes in k lead to much more significant variations

Based on this, we ranked the variables by their influence:$i < j < k$. Since we know that the memory model is row-major, we can restructure the for loop to optimize memory access, thereby improving cache locality.

f) ***Reducing Branch misses**:*

Upon further analysis of fig.4, we identified a significant number of branch misses. The issue appears to stem from certain **if conditions** inside the **set_bnd** function, which are part of the loop cycles. These conditions can be calculated outside the loop to reduce unpredictability. The branch misses likely occur because the **set_bnd** function is invoked with varying b values, causing the compiler to mispredict the condition too frequently.

To address this, one solution is to move the **if condition** outside the loop, limiting its evaluation to just once. This approach can also enhance the **Vectorization** process, as the compiler will have a clearer understanding of the operations to execute. Additionally, applying **Loop Unrolling**, as previously discussed, could further reduce branch misses.

IV. Results

All the potential optimizations, except for **f)**, were applied to the **lin_solve** function. Each approach had varying levels of impact, with some providing more significant gains than others, but none were negligible. As a result, we decided to implement every single one.

Following this, we applied the same optimizations to other functions where applicable, evaluating their effectiveness in each case.

After applying the various optimizations we mentioned above, we got the results seen in fig.5. Analyzing the results:

- A reduction in execution time by 8.5421 seconds;
- 24,528,944 fewer branch misses compared to the original program;
- 29.16% fewer cache misses than the initial version.

Given these results, we are highly satisfied with the outcome and believe we have made significant progress in optimizing the program.

V. Annexes

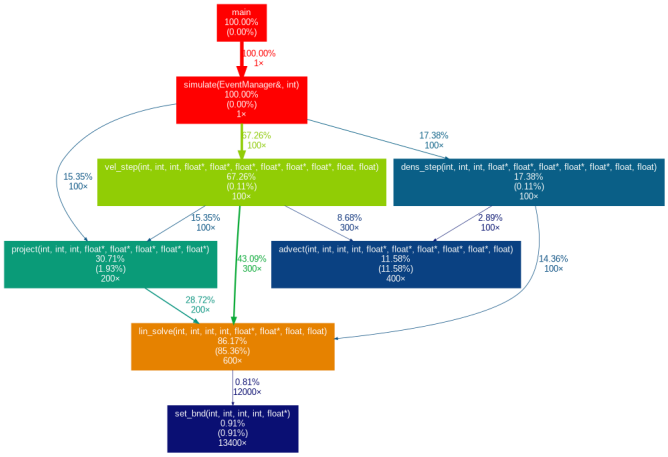

Fig. 1: Call graph generated by gprof2dot

```
        .p2align 3
.L43:
        movss   (%rax,%r9,4), %xmm0
        addss   (%rax,%r8,4), %xmm0
        addl    $1, %edx
        addss   (%rax,%rdi,4), %xmm0
        addss   (%rax,%rsi,4), %xmm0
        addss   (%rax,%r15), %xmm0
        addss   (%rax,%r14,4), %xmm0
        mulss   %xmm3, %xmm0
        addss   (%rcx), %xmm0
        addq    %rbx, %rcx
        divss   %xmm2, %xmm0
        movss   %xmm0, (%rax)
        addq    %rbx, %rax
        cmpl    %r10d, %edx
        jne     .L43
```

Fig. 2: Assembly snippet of each internal cycle in lin_solve

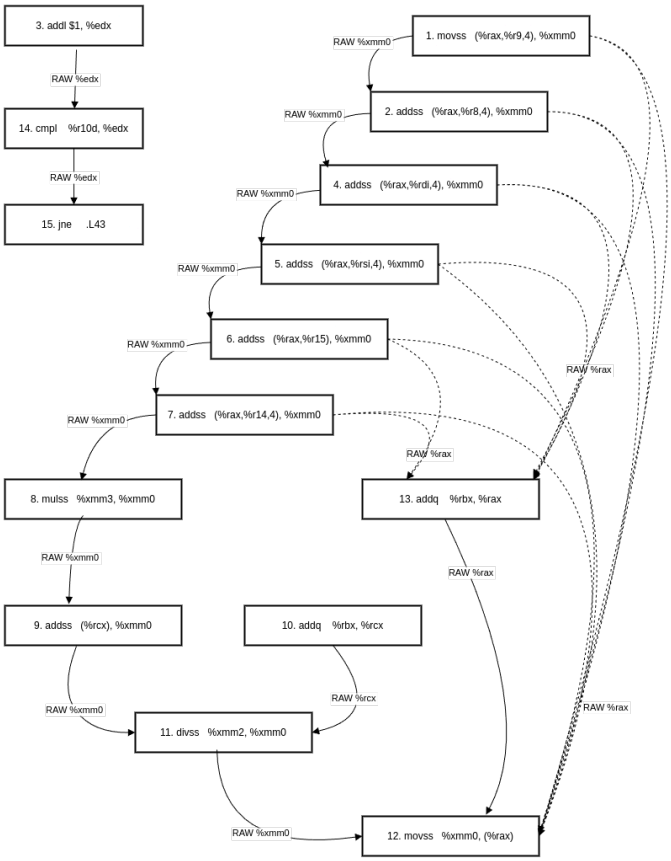

Fig. 3: Dependency Graph corresponding to the Assembly snippet

```
Performance counter stats for './fluid_sim' (3 runs):

     18764104197      inst_retired.any:u                                          ( +-  0.07% )  (50.00%)
     35254573152      cycles:u                                                    ( +-  0.04% )  (50.00%)
     18773638254      inst_retired.any:u                                          ( +-  0.00% )  (66.67%)
        24791736      branch-misses:u                                             ( +-  0.03% )  (83.34%)
      7297448493      L1-dcache-loads:u                                           ( +-  0.02% )  (83.33%)
       231549160      L1-dcache-load-misses:u   #   31.73% of all L1-dcache hits  ( +-  0.13% )  (33.33%)
     35276118080      cycles:u                                                    ( +-  0.04% )  (49.99%)
               0      duration_time:u

          11.7521 +- 8.0966 seconds time elapsed  ( +-  0.82% )
```

Fig. 4: Original Version's execution profile

```
Performance counter stats for './fluid_sim' (3 runs):

     24168567325      inst_retired.any          # 24168567324.7 Instructions
                                                #      0.4 CPI                    ( +-  0.04% )  (49.99%)
     10055690775      cycles                                                      ( +-  1.02% )  (49.99%)
     24181792141      inst_retired.any                                            ( +-  0.03% )  (66.67%)
        2627972       branch-misses                                               ( +-  3.82% )  (83.34%)
      8605490715      L1-dcache-loads                                             ( +-  0.02% )  (83.32%)
       220894998      L1-dcache-load-misses     #    2.57% of all L1-dcache hits  ( +-  0.08% )  (33.31%)
     10063210060      cycles                                                      ( +-  1.02% )  (49.97%)

          3.2100 +- 0.0932 seconds time elapsed  ( +-  2.90% )
```

Fig. 5: Final Version's execution profile