Optimizing a Fluid Dynamics Simulation Code with Grid Tiling

Pedro Azevedo
Universidade do Minho
pg57897

Ricardo Jesus
Universidade do Minho
pg57898

Rui Pinto
Universidade do Minho
pg56010

***Abstract*—This report presents the optimization of a 3D fluid simulation code, using techniques such as tiling to improve cache locality and loop reorganization. The improvements implemented resulted in a significant reduction in execution time.**

I. Introduction

This work aims to optimize the code of a 3D fluid dynamics solver provided for the Parallel Computing course. The simulation is based on Jos Stam's stable fluids solver. The main focus is to reduce the execution time using memory access optimization techniques while maintaining the accuracy of the simulation.

II. gprof

To identify the main areas for performance improvement in the code, we conducted a profiling analysis using the `gprof` tool. This tool generated a call graph, which provides a detailed view of the most CPU-intensive functions. Based on the generated graph, we identified two main functions that consumed the majority of computing resources: `lin_solve()` and `project()`.

The profiling analysis with `gprof` was essential for identifying the key code blocks that required optimization. The `lin_solve` and `project()` functions accounted for a significant portion of the program's execution time. With the optimizations applied, we achieved a substantial improvement in performance, especially in simulations involving large data volumes and multiple iterations.

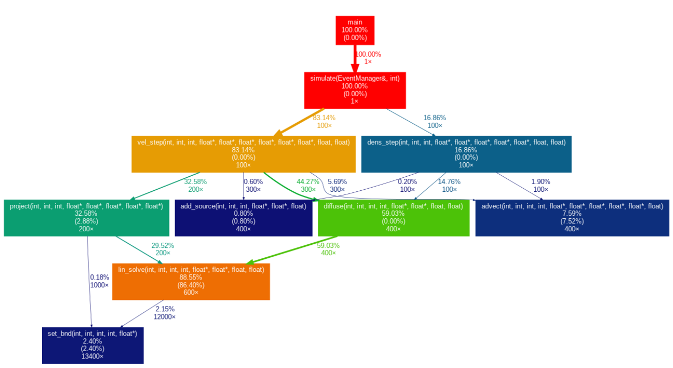

Fig. 1. Call graph of original code

III. Code optimization

The original fluid simulation code, with an execution time of **28.18 seconds**, was optimized to reduce this time to **3.62 seconds**, resulting in an improvement of approximately **86.98%** Below, we detail the main code modifications we developed for this significant improvement.

A. *Reorganization of loops*

In the original code, the loops traversed the grid dimensions sequentially. The order of iteration, however, can significantly impact the efficient use of cache memory. In the optimized code, the order of the loops was adjusted to improve data locality, allowing recently accessed data blocks to remain in the cache for longer, reducing the number of accessed to main memory, reducing the number of cache misses and increasing overall performance. With this said, for example in lin_solve, instead of having the cycles i,j,k respectively, we switch to k,j,i. You can see this example in image 5 of the attachments

B. *Application of the Tiling Technique (Block Division)*

The tiling technique was introduced in the optimized code to divide the large three-dimensional matrices into smaller blocks. This way, the data is processed in block that fit better into the cache, resulting in more efficient use of cache memory. Block processing ensures that data is reused before being replaced in the cache, reducing the time spent on intensive operations, such as diffusion and projection calculations. An example of this can be seen in figure 4 in the attachments.

C. *Simplifying mathematical calculations*

One of the optimizations applied to the code was the modification of the `diffuse` function, which performs the diffusion step in the fluid simulation. In the original code, the function used the predefined `MAX` function, which evaluates two variables at a time. In the optimized code, this function was replaced with a custom version that compares three dimensions simultaneously. Additionally, the calculation of the variable `c`, which was previously done inside the `lin_solve` funtion, was moved outisde, optimizing execution and avoiding unnecessary recalculations.

In the original code, the default `MAX` function compares two values at a time, which requires multiple comparisons when it comes to three dimensions (`M`, `N` and `O`). To simplify and improve code efficiency, we have implemented an optimized version that compares the three dimensions directly in a single line of code. This reduces the number of comparison operations, making the code more efficient in terms of CPU instructions.

D. *Optimization summary*

Optimizations made directly to the code, including the reorganization of loops, the application of the *tiling* technique and the simplification of mathematical calculations were fundamental in significantly reducing execution time. These modifications focused mainly on improving cache locality and reducing the number of memory accesses, resulting in more efficient code for processing large volumes of data, such as 3D fluid simulation.

IV. USED FLAGS

To optimize the performance of the simulation code, we used several advanced compilation options that allow the compiler to generate more efficient code for the hardware architecture in use. The main flags used in the compilation process are described below:

A. *-Ofast*

The `-Ofast` flag activates aggressive optimizations that prioritize performance over strict compliance with IEEE standards for floating point operations. This includes:

- Reordering and simplification of mathematical expressions, which can reduce precision in exchange for greater speed.
- All `-O3` optimizations, such as vectorization, aggressive inlining and loop elimination.
- Bypasses certain C-standard compliance checks, such as bounds checking.

B. *-funroll-all-loops*

This flag instructs the compiler to apply "loop unrolling" to all loops in the code, regardless of their size or structure. "Loop unrolling is a technique that expands loops, allowing multiple iterations to be executed at once. This reduces the control overhead of loops and can increase instruction-level parallelism, resulting in performance gains.

C. *-march=native*

The `-march=native` flag tells the compiler to generate code optimized for the specific CPU architecture of the machine on which it is being compiled. The compiler will automatically detect the processor's capabilities and generate code that takes full advantage of these instructions, resulting in superior performance.

D. *-flto (Link Time Optimization)*

The `-flto` flag enables link-time optimization (*Link Time Optimization*). This optimization allows the compiler to postpone certain optimization decisions until the linking phase, allowing for a global analysis of the program. With this, the compiler can perform more aggressive optimizations, such as removing dead code between different compilation units, reducing binary size and improving overall performance.

E. *Makefile Objectives and Commands*

The makefile uses the flags described above to compile the simulation code. The `all` command is responsible for generating the executable `fluid_sim`, using the source code files `main.cpp`, `fluid_solver.cpp`, and `EventManager.cpp`. The `clean` command removes the generated executable, cleaning up the working directory.

F. *Results*

The optimizations applied resulted in a significant reduction in the execution time of the fluid solver. Table I summarizes the results before and after the optimizations.

TABLE I
COMPARISON OF EXECUTION TIMES AND CACHE MISS

Version	Runtime (s)	Improvement (%)	Cache Miss (%)
Original	28.18	-	3.29%
Optimized	3.62	86.98%	3.19%

V. CONCLUSION AND FUTURE WORK

The optimization of the 3D fluid simulation code was successful, with a significant improvement in execution time. The techniques applied, such as tiling and reorganizing loops, were key to achieving these results.

Although the optimizations applied have generated a significant improvement in the performance of the simulation code, there are still some limitations and opportunities for future improvements.

One area that could be explored is parallelism for multi-core architectures. Although the current code has been optimized in terms of cache locality and memory access efficiency, exploring explicit parallelism using libraries such as OpenMP could enable faster execution on multi-core systems. By parallelizing critical parts of the code, such as diffusion and projection calculations, it would be possible to distribute the workload among several cores, further speeding up execution time.

Attachments

The original code and the optimized code are included, along with the performance profiles obtained during the optimization process.

```
Performance counter stats for './fluid_sim':

  166,525,674,935      inst_retired.any          # 166525674935.0 Instructions
                                                 #      0.5 CPI                         (62.50%)
   87,408,494,755      cycles                                                           (62.50%)
  166,677,431,675      inst_retired.any                                                 (62.50%)
       22,813,776      branch-misses                                                    (62.50%)
   70,455,069,156      L1-dcache-loads                                                  (62.50%)
    2,315,800,501      L1-dcache-load-misses     #    3.29% of all L1-dcache hits       (25.00%)
   87,493,077,725      cycles                                                           (37.50%)
                0      mem-loads                                                        (37.50%)
    2,745,116,723      mem-stores                                                       (50.00%)

     28.185508812 seconds time elapsed

     28.175424000 seconds user
      0.001999000 seconds sys
```

Fig. 2. Output without optimization

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

   18,990,228,504      inst_retired.any          # 18990228504.3 Instructions
                                                 #      0.6 CPI                         ( +-  0.02% )  (62.49%)
   11,833,490,519      cycles                                                           ( +-  2.94% )  (62.49%)
   18,997,682,983      inst_retired.any                                                 ( +-  0.01% )  (62.49%)
        1,694,614      branch-misses                                                    ( +-  0.26% )  (62.49%)
    7,027,063,949      L1-dcache-loads                                                  ( +-  0.02% )  (62.48%)
      225,328,205      L1-dcache-load-misses     #    3.21% of all L1-dcache hits       ( +-  0.22% )  (25.00%)
   11,844,600,398      cycles                                                           ( +-  2.94% )  (37.51%)
                0      mem-loads                                                        (37.50%)
    1,311,132,801      mem-stores                                                       ( +-  0.04% )  (49.99%)

           3.620 +- 0.104 seconds time elapsed  ( +-  2.88% )
```

Fig. 3. Output with optimization

```c
for (int l = 0; l < LINEARSOLVERTIMES; l++) {
  // Iterate over tiles in the k, j, i directions
  for (int bk = 1; bk <= O; bk += TILE_SIZE) {
    int bk_min = (bk + TILE_SIZE < O + 1) ? bk + TILE_SIZE : O + 1;

    for (int bj = 1; bj <= N; bj += TILE_SIZE) {
      int bj_min = (bj + TILE_SIZE < N + 1) ? bj + TILE_SIZE : N + 1;

      for (int bi = 1; bi <= M; bi += TILE_SIZE) {
        int bi_min = (bi + TILE_SIZE < M + 1) ? bi + TILE_SIZE : M + 1;
```

Fig. 4. Example of tiling in lin_solve function

```c
  // Iterate over elements within each tile
  for (int k = bk; k < bk_min; k++) {
    for (int j = bj; j < bj_min; j++) {
      for (int i = bi; i < bi_min; i++) {
```

Fig. 5. Reorganization of loops