Optimization of a Fluid Solver

Rita Lino
School of Engineering
University of Minho
Braga, Portugal
pg54056@alunos.uminho.pt

Diogo Barros
School of Engineering
University of Minho
Braga, Portugal
a100600@alunos.uminho.pt

Pedro Silva
School of Engineering
University of Minho
Braga, Portugal
a100745@alunos.uminho.pt

***Abstract*—This assignment focuses on optimizing a single-threaded fluid dynamics simulation program using various performance improvement techniques. After trying several methods to improve its run-time, conclusions are reached as to why each method was more or less efficient.**

***Index Terms*—Parallel Computing, Code Optimization, Performance Analysis.**

I. Introduction

The primary objective of this assignment is to analyze and optimize a fluid dynamics simulation program — in order to achieve that goal, various optimization techniques were applied, such as loop rearranging, processing loops as blocks and minimizing function calls. Furthermore, ***gprof*** and ***gprof*** were used to pinpoint the functions that were causing the most bottleneck.

II. gprof

After using ***gprof***, we came to the conclusion that the functions that required more processing power were `lin\_solve` and `project`. Therefore, we focused our efforts on the previously mentioned functions, leaving the other function to only be altered if they had unoptimized loops.

III. Optimizations

In this section, we explain the reason why and how we changed the source code in order to improve the program's performance. To do so, we focused on improving time and space locality, as well as reducing the number of cache misses.

A. Optimization Flags

Optimization flags are crucial tools that allow programmers to improve the performance of their programs without changing the source code itself. Modern compilers can transform the way code is executed, making it faster and more efficient by taking advantage of advanced CPU features, reducing unnecessary instructions, and improving memory access patterns. In our Makefile, we used the following flags:

- `-O2`: enables a set of compiler optimizations that improve runtime performance without increasing compilation time significantly. The optimizations include instruction scheduling, loop optimizations, inlining of small functions, and code motion;
- `-ffast-math`: allows the compiler to make aggressive optimizations on floating-point calculations;
- `-ftree-vectorize`: enables the compiler to attempt automatic vectorization of loops, meaning that it tries to convert scalar operations into vector operations that process multiple elements in parallel;
- `-march=native`: tells the compiler to optimize the code for the specific architecture of the machine it is being compiled on;
- `-funroll-loops`: the compiler automatically unrolls loops, which can significantly improve performance in certain scenarios by reducing the number of iterations and allowing better use of the CPU's instruction pipeline;
- `-O3`: enables all optimizations in -O2 and adds more aggressive optimizations, such as function inlining, vectorization, and loop transformations;
- `-Ofast`: disables checks for things like signed overflow or strict aliasing, allowing more aggressive code transformations;
- `-falign-functions=16`: aligns functions in memory at 16-byte boundaries. Proper alignment can reduce cache misses and allow more efficient instruction fetching, especially on processors that benefit from aligned memory access;
- `-falign-loops=16`: similar to function alignment, this flag aligns the beginning of loops to 16-byte boundaries;
- `-flto`: enables link-time optimization (LTO), allowing the compiler to optimize across file boundaries at the time of linking. Normally, each source file is optimized individually, but with LTO, the compiler can see the entire program at once and apply more global optimizations.

B. Function Optimizations

Once we've found the found our optimization targets, we started focusing on applying optimizations to each of these functions:

- `set_bnd(int M, int N, int O, int b, float *x)`:
  - The original code calculates the index IX(...) inside the array access multiple times in each loop iteration. In the optimized version, these indices are computed once and stored in local variables like IX1, IX2, etc., which are then reused within the loop.
  - The local variables (IX1, IX2, etc.) store indexes and helps to reduce repeated index calculations and

simplifying the code. This lowers cognitive load and removes redundancy, while highlighting the symmetrical structure of the operations for better clarity
  - To sum up, these changes end up to boost both performance and code legibility.

- `lin_solve(int M, int N, int O, int b, etc...)`:
  - Initially, the sequential processing of the entire grid could have led to inefficient memory access, causing data to be evicted from the cache before reuse. So to solve the problem, our version processes the grid in smaller blocks, improving CPU cache utilization. This enhances memory locality and reduces cache misses.
  - The use of blocking also enhances scalability, making the code more efficient, where memory access patterns and cache efficiency become increasingly important.
  - The new code also uses local variables to store neighboring cell values, reducing cache misses since local variables are stored in registers for faster access.
  - Despite these optimizations, the code maintains clarity and remains easy to follow, facilitating future adjustments or enhancements.
- `advect(int M, int N, int O, int b, etc...)`:
  - In this function, the index `IX(i, j, k)` is calculated once and stored in the variable IX1 instead of being recalculated multiple times within the loop, to reduce the redundant function calls.
  - The boundary checks are slightly optimized by storing constants like `M + 0.5f, N + 0.5f, and O + 0.5f` in local variables `if1, if2, if3`. This reduces repeated calculations of the same values during the boundary clamping process.
  - The indexes for neighboring points (i0, i1, j0, j1, k0, k1) are precomputed and stored in IX2 through IX9, to avoid repeated recalculations and to improve both code clarity and performance.
  - Local variables are reused, avoiding redundant calculations. This optimization helps reduce overall computation and improves memory access patterns.
- `project(int M, int N, int O, float *u, etc...)`:
  - The indexes for neighboring grid points (IX1, IX2, IX3, etc.) are precomputed before each velocity update, reducing redundant calculations, within the loops.
  - The maximum grid size, used to normalize the divergence, is computed once at the beginning `(maxSize)` instead of within the loop. This reduces redundant calls to the `MAX()` function, improving efficiency.
  - Local variables (u_left, u_right, v_up, v_down, etc.) are introduced to hold neighboring values of the velocity components. This improves cache locality, as data is stored in registers rather than repeatedly accessing array elements from memory.
  - Precomputing and reusing indices and local variables reduce the number of memory accesses and improve data locality, which is crucial for performance when working with large datasets.

C. Results

After applying all the optimizations mentioned in the previous chapters, these were the results after running each program version on the SERACH cluster:

Program version	**Results**	
	Time (s)	***Cache misses*** (%)
o2	5,7638	4,08
o2_vec	5.849	4.20
o2_loop	5.55	4.04
o2_ofast	4.60	4.11
o2_all	4.50	4.92
o3	4.5150	4.12
o3_vec	4.5644	4.11
o3_loop	4.4907	4.99
o3_ofast	4.6357	4.13
o3_all	4.4990	4.93
ofast	4.6503	4.11
ofast_vec	4.5250	4.10
ofast_loop	4.4984	5.00
ofast_all	4.499	4.95

IV. Conclusion

The optimization of the fluid dynamics simulation successfully enhanced performance through targeted adjustments. The changes included efficient memory management, local variable usage, and loop restructuring, which resulted in noticeable improvements in runtime and cache efficiency. While each optimization contributed differently to performance, the combination of all techniques provided the most significant speedup. Future work could explore parallelization or further fine-tuning for specific hardware architectures.

APPENDIX A

This image presents the results after running the command **srun –partition=cpar –exclusive perf stat etc....**.

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

     17274973938      inst_retired.any:u
     14491078769      cycles:u
     17280024632      inst_retired.any:u
        3263982      branch-misses:u
      6834254995      L1-dcache-loads:u
       335698188      L1-dcache-load-misses:u   #    4,91% of all L1-dcache hits
     14499005915      cycles:u
               0      duration_time:u
               0      mem-loads:u
      1338863071      mem-stores:u

          4,5114 +- 0,0663 seconds time elapsed  ( +-  1,47% )
```

Fig. 1. Results extracted from the new and optimized code

APPENDIX B

This shell script provided by the teachers, **validate.sh**, is executed with the command , **./validate.sh project.zip**, to validate our work and display the results, including execution time and the calculations performed by the code.

```
Output validation passed
Contents of run1.txt:
Total density after 100 timesteps: 81981.5

 Performance counter stats for './fluid_sim':

       14270779072      cycles:u

       4,493924210 seconds time elapsed

       4,489018000 seconds user
       0,001000000 seconds sys
```

Fig. 2. Output validation with the **validate.sh**

APPENDIX C

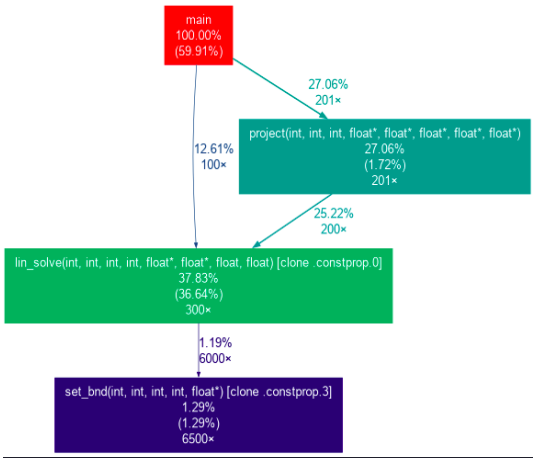

Fig. 3. Results from gprof2dot for the optimised code