A typesetting system to untangle the scientific writing process

João Paulo Gomes
developer
PG55960

Mariana Antunes Silva
developer
PG55980

Rodrigo Casal Novo
developer
PG56006

***Abstract*—This project focuses on optimizing an existing 3D fluid dynamics simulation based on Jos Stam's stable fluid solver. The optimization effort targets improving computational performance and efficiency, with a particular emphasis on reducing complexity while maintaining accuracy. The primary goal is to maximize spatial and temporal locality in the simulation, enhancing memory access patterns and cache usage to improve data reuse.**

***Index terms*—Paralel computing, Analysis, Optimization**

I. Introduction

A. Code analysis

Before we can optimize the code, we must first understand what is worth optimizing. To achieve this understanding we used a call-graph with *gprof*. This call-graph gave us a list of the functions that were executed alongside the number of times they were called and more importantly, the amount of time spent in each one, as we can see at **Section IV.A**

Based on the analysis of the list, we came to the conclusion that the function *lin_solver* should be the main focus of our optimization process, since it accounts for 85% of the execution time, which corresponds to approximately 38 seconds. Meaning that optimizing this function could reduce the time it takes to execute by nearly 7 times.

Having acquired our target for optimization, we decided to do a further analysis on it, by looking at the source code of the function. After that analysis, we concluded that there were two main factors contributing to the slow execution time: no spacial locality between iterations, and dependencies between any element of the matrix and the one after him. The function also had complexity N^4, but there was nothing we could do to solve this issue, without changing the entire algorithm.

We proceeded to optimize *lin_solver* until we solved to some degree the problems presented, we redid the call-graph (**Section IV.B**). With this second call-graph, we noticed that now the functions *advect* (the function *advect* doesn't appear explicitly on the call-graph, because the compiler injected it's code in the function that calls it), *project* and *set_bnd* were responsible for a large percentage of the execution time (nearly 12%), so we decided to analyze them further. From this analysis, we noticed that this functions weren't exploring spacial locality, and that there was a quick to implement solution to improve them.

We also noticed that besides the code not being optimized, the compiler also wasn't being explored well enough, so we added some changes to the *Makefile*.

II. Optimizations

A. Compiler changes

Reason By looking at the *Makefile* that was given to us, it was easy to note that the compiler was creating the executable with no optimizations.

Prediction By adding some flags to the *Makefile* it is expected that the arithmetic expressions are calculated faster, with some added variance due to the compiler changing the order of the operations, and that the compiler better explores memory locality reducing the execution time significantly.

Optimization We replaced the compiling command on the *Makefile* to "g++ -Wall -O3 -ffast-math -funroll-loops".

final result By changing the *Makefile* we reduced the time of the original program from 28 seconds down to 12 seconds.

B. Switching the for loop order

Reason By analyzing the IX define, we noticed that each increase in the i would increase the result by one, each increase in j would increase the value by 44 and each increase in the k by 1936. With this in mind we noticed that the original order of the loops was i, j, k. Due to this, for every iteration the index of x would be increased by 1936 which made it so that there was always a cache miss on this access massively reducing the performance of the program.

Optimization To solve this issue we changed the order of the loops to k,j,i allowing for a better use of the cash, making it so that a cache miss would occur every 6 cycles (in the case of *lin_solver*) or 8 cycles (in the other functions). We applied this change to the functions *lin_solver*, *advect, project* and *set_bnd*.

C. *Changing the calculation on* lin_solver

Reason In the original code, all of the arithmetic calculations are done in a single expression and stored into the same value. Because of this, there are dependencies in the expression, which makes it hard for the compiler to vectorize the calculation of the x[IX(i,j,k)] value.

Prediction By separating the sums we should be able to calculate them all in a singular vectored sum reducing the number of instructions per iteration of the loop by at least 5.

Optimization To achieve this, we separated the expression into different parts, where we would store each pair of sums into a different temporary variable and later on reduce by summing up those temporary variables together and doing the remainder of the calculations at the end.

D. *Transforming the division into multiplication*

Reason Division is the slowest arithmetic calculation that computers can do, so if we could remove divisions from *lin_solver* our CPI would be reduced.

Optimization We know from math that any division by a number N is equal to doing a multiplication of $\frac{1}{N}$. Since a and c are constants in *lin_solver* we can calculate the value of $\frac{1}{c}$ and $\frac{a}{c}$ before the loop starts and store them into auxiliary values, which makes it so that we can replace all divisions in the expression with multiplications.

E. *Tiling*

Reason

Due to how the algorithm works, we always load 5 distinct blocks of memory, however the original implementation, even after correcting the order of the loops, only explores the spacial locality of one of them, which is inefficient.

Optimization

To make better use of the already loaded chunks of memory we implemented tiling in the matrix, by utilizing 6 as the block size. Since the cache on the machine is of 64 bytes, which means that it loads 8 floats at a time, however the algorithm requires always the index before and the one after the current one, which means that for the any size chosen we will always utilize two more values than the chosen size. Since $8 - 2 = 6$ and 42 (the value of O) is divisible by 6, 6 is the perfect block size for this machine.

F. *Delaying dependencies*

Reason Due to how the algorithm is structured, to calculate any element of the matrix we need the one before it to be calculated and for the one after it to remain unchanged until calculate the current one. Making it impossible for the compiler to unroll the loops or to calculate two or more indexes of the matrix at the same time.

Optimization This dependencies are impossible to remove without rewriting the entire algorithm, however the dependencies can be moved, so that we can compute most of the two indexes values at the same time. We can achieve this by increasing the i by two instead of one and making it so that for x[IX(i,j,k)] we calculate everything like we did before, but we don't add x[IX(i+1,j,k)] and for x[IX(i+1,j,k)] we calculate everything like before except we don't add x[IX(i,j,k)]. Then after both of the auxiliar sums are calculated and stored into local variables, we add the value that we skipped before to both of the indexes applying the multiplications necessary to maintain the equation valid. This approach alongside the other optimizations gives us the ability to vectorize all of the calculations with exception of the ones that were moved to the bottom.

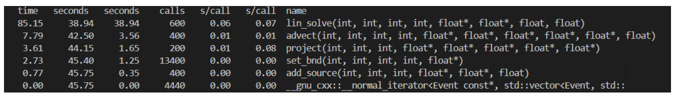
III. Conclusion

With all of our optimizations we managed to reduce the time from approximately 42 seconds down to 3.7 seconds. However it is important to mention, that if the value of O M or N changed to a number that was not multiple of 6, we would need to apply some changes to *lin_solver*, since tiling only works if the chosen block size is a divider of O, M and N.

The final time can be seen at **Section IV.C**

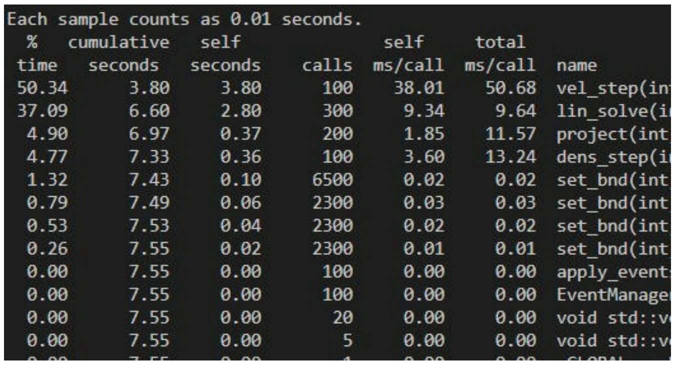
IV. Appendix

A. *gprof result.*



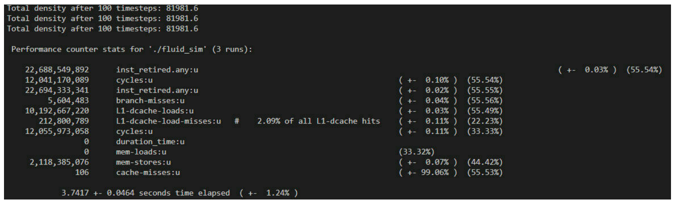
```
 time   seconds   seconds    calls   s/call   s/call  name
 85.15     38.94     38.94      600     0.06     0.07  lin_solve(int, int, int, int, float*, float*, float, float)
  7.79     42.50      3.56      400     0.01     0.01  advect(int, int, int, int, float*, float*, float*, float*, float*, float)
  3.61     44.15      1.65      200     0.01     0.08  project(int, int, int, float*, float*, float*, float*, float*)
  2.73     45.40      1.25    13400     0.00     0.00  set_bnd(int, int, int, int, float*)
  0.77     45.75      0.35      400     0.00     0.00  add_source(int, int, int, float*, float*, float)
  0.00     45.75      0.00     4440     0.00     0.00  __gnu_cxx::__normal_iterator<Event const*, std::vector<Event, std::
```

B. *Callgraph with lin_solver optimizations.*



```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 50.34      3.80     3.80      100    38.01    50.68  vel_step(in
 37.09      6.60     2.80      300     9.34     9.64  lin_solve(i
  4.90      6.97     0.37      200     1.85    11.57  project(int
  4.77      7.33     0.36      100     3.60    13.24  dens_step(i
  1.32      7.43     0.10     6500     0.02     0.02  set_bnd(int
  0.79      7.49     0.06     2300     0.03     0.03  set_bnd(int
  0.53      7.53     0.04     2300     0.02     0.02  set_bnd(int
  0.26      7.55     0.02     2300     0.01     0.01  set_bnd(int
  0.00      7.55     0.00      100     0.00     0.00  apply_event
  0.00      7.55     0.00      100     0.00     0.00  EventManage
  0.00      7.55     0.00       20     0.00     0.00  void std::v
  0.00      7.55     0.00        5     0.00     0.00  void std::v
```

C. *Final run time.*



```
Total density after 100 timesteps: 81981.6
Total density after 100 timesteps: 81981.6
Total density after 100 timesteps: 81981.6

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

    22,688,549,892      inst_retired.any:u                                                                ( +-  0.03% )  (55.54%)
    12,041,170,089      cycles:u                                                  ( +-  0.10% )  (55.54%)
    22,694,333,341      inst_retired.any:u                                        ( +-  0.02% )  (55.55%)
         5,604,483      branch-misses:u                                           ( +-  0.04% )  (55.56%)
    10,192,667,220      L1-dcache-loads:u                                         ( +-  0.03% )  (55.49%)
       212,800,789      L1-dcache-load-misses:u   #    2.09% of all L1-dcache hits ( +-  0.11% )  (22.23%)
    12,055,973,058      cycles:u                                                  ( +-  0.11% )  (33.33%)
                 0      duration_time:u
                 0      mem-loads:u                                               (33.32%)
     2,118,385,076      mem-stores:u                                              ( +-  0.07% )  (44.42%)
               106      cache-misses:u                                            ( +- 99.06% )  (55.53%)

            3.7417 +- 0.0464 seconds time elapsed  ( +-  1.24% )
```