A typesetting system to untangle the scientific writing process

João Paulo Gomes
PG55960

Mariana Antunes Silva
PG55980

Rodrigo Casal Novo
PG56006

***Abstract*—This project focuses on optimizing an existing 3D fluid dynamics simulation that uses a Red-black algorithm. The optimization effort targets improving computational performance and parallel efficiency using different optimization approaches.**

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

I. Introduction

In this report we aim to talk about all of the important information of the previous assignments and explain everything about this third assignment. In order to do so, this report will be segmented in three different parts, where each part corresponds to one assignment.

II. Assignment I

A. Thoughts and reasoning

In this assignment we we're told to optimize the sequential version of the code so, after an analysis of the code utilizing a *gprof* call-graph and an analysis of the source code, we found the functions that were in need of optimizations. Those functions were: `lin_solver`, `advect`, `project`, and `set_bnd`.

B. 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 in 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 `i` would increase the result by one, each increase in `j` would increase the value by 44, and each increase in `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 caused a cache miss on each 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 better use of the cache, making a cache miss 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 in lin_solver:*

Reason In the original code, all arithmetic calculations are done in a single expression and stored into the same value. Because of this, dependencies in the expression made 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 single 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, storing each pair of sums in a temporary variable, then combining these variables at the end with the remaining calculations.

d) *Transforming the division into multiplication:*

Reason Division is the slowest arithmetic calculation that computers can perform. Removing divisions from `lin_solver` would reduce the CPI.

Optimization From math, any division by a number `N` is equivalent to multiplying by `1/N`. Since `a` and `c` are constants in `lin_solver`, we precomputed `1/c` and `a/c` before the loop starts, replacing divisions with multiplications.

e) *Tiling:*

Reason The algorithm always loads 5 distinct blocks of memory. However, even after correcting the loop order, the original implementation only exploited spatial locality for one block, making it 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 The algorithm structure requires the element before a matrix entry to be calculated first, while the next one remains unchanged until the current one is computed. This dependency prevents the compiler from unrolling loops or computing multiple indices simultaneously.

Optimization Although removing these dependencies entirely requires rewriting the algorithm, we moved them to allow partial parallelization. We incremented `i` by two instead of one, calculating `x[IX(i,j,k)]` and `x[IX(i+1,j,k)]` separately with different dependency handling. After storing intermediary sums, we added the skipped values to both indices, maintaining equation validity. 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. Assignment II

A. Thoughts and reasoning

In this assignment we were told to make a parallel version with *openMP* of a version of *lin_solve* based on the red-black algorithm, but all the other functions remained the same from our implementation in the first assignment. Creating and destroying threads is resource intensive, so to make an optimized *openMP* implementation we need to minimize the amount of times the threads are created/destroyed, to achieve this behavior we decided to parallelize every function in the file lin_solver.cpp that did calculations. To maintain the correctness of the parallel version we needed to first find and eliminate all of the data races in our code.

a) *Lin_solve analysis:*

The red-black algorithm allows "red" and "black" cells to be computed independently in parallel, but their interdependence prevents simultaneous computation. Also if we were to apply parallel for to each loop there would be a data race to the value of *max_c*. Additionally, the original loop order lacked memory locality optimization. Resolving these issues was key to improving performance.

b) *Vel_step and den_step analysis:*

Vel_step and *dens_step* only swap variables and call other functions. So the focus will be on the functions that are called by them: project, *add_source* and advect. Project and advect also call the function *set_bnd*, so *set_bnd* will also be a target for parallelization.

B. Optimizations and parallelization

a) *Sequential version changes:*

Reason Loops lacked memory locality, and arithmetic calculations could be optimized.

Optimization We applied the optimizations that were doing in the previous assignment except tiling to preserve parallelization efficiency.

b) *Replacing max_c with a local variable:*

Reason max_c caused a data race.

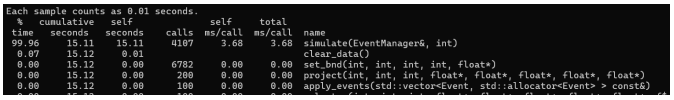
Solution Instead of marking a critical section for *max_c* updates (high overhead) or using reduction after each loop (unnecessary blocking), we assigned each thread a *local_max_c* that is a private variable. This local variable will preserve it's value when entering the second loop to preserve correctness. At the end of the second loop all threads will reduce their *max_c* into the global *max_c* in a critical section. This approach cutes synchronization overhead significantly compared to having a critical section in each loop or a reduction at the end of each loop.

c) *Utilizing collapse in* lin_solve*:*

Reason omp for applied only to the first loop, causing uneven workloads.

Solution Using collapse(2) allowed *omp* for to cover two loops, balancing workloads across threads and improving performance.

C. Callgraph *with* lin_solver *optimizations.*



a) *Utilizing* omp *for:*

Reason Target function loops can be parallelized without compromising correctness.

Optimization Adding *omp* for distributed workloads among threads, improving performance. For *lin_solve*, schedule(static) ensured balanced workloads with minimal thread communication. No schedule type was specified for smaller functions due to lower impact.

b) *Reducing recreation of threads:*

Reason Only parallelizing *lin_solve* required frequent thread recreation, a costly operation.

Solution Parallelizing other functions prevented thread destruction after *lin_solve*, improving overall performance. Even minor performance hits in small workloads were offset by the reduced thread management overhead.

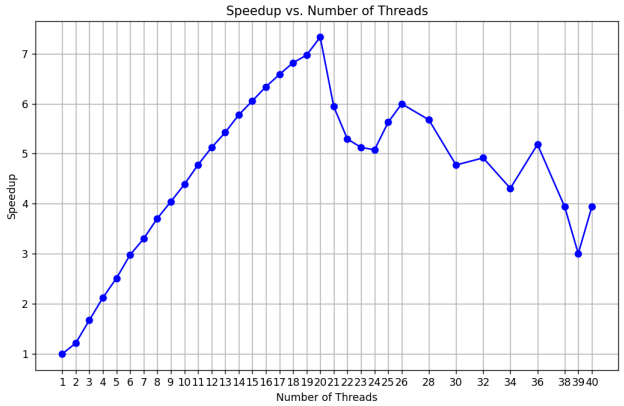
c) *Parallelization of the other functions:*

The parallelization of the other functions was made utilizing parallel for alongside collapse 2 in the cases where we had three nested loops and only parallel for when we had less than three. This distinction was made, because if there is only one loop then there is no loop to collapse and if there's only two loops, collapsing them would result in a worse use of spacial locality which decreased performance.

D. *gprof result with all optimizations*

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 52.25      7.22     7.22      300    24.07    24.07  lin_solve(int, int, int, int, float*, float*, float, float) [clone .constprop.11]
 36.83     12.31     5.09      100    50.91    78.34  vel_step(int, int, int, int, float*, float*, float*, float*, float*, float*, float, float)
  6.15     13.16     0.85      100     8.50    32.57  dens_step(int, int, int, int, float*, float*, float*, float*, float, float)
  4.20     13.74     0.58      200     2.90    27.13  project(int, int, int, float*, float*, float*, float*, float*)
```

E. SpeedUP *graph*



IV. Assignment III

A. *Introduction*

In this assignment we were asked to choose between doing a better version of *openMP*, created a version with *MPI* or a version with accelerators. We opted to do a version with accelerators, namely GPU with *CUDA*. We choose to do the *CUDA* version, because we wanted to learn how to work with GPU's and because *lin_solve* had enough complexity (N^3 * timestamps) to make a *CUDA* approach viable.

B. *Disadvantages and constraints of using GPU*

Even though the GPU can do calculations thousands of times faster than the CPU, the GPU isn't perfect and there a few things that we need to keep in mind to utilize it. The first thing to keep in mind is that each PU in a GPU is way more limited than a GPU, so in order to utilize them effectively, we shouldn't assign them complicated instructions such as divisions.

Another issue with GPU's is that all threads in the same warp have to be doing the same instruction at the same time, so when there are if statements even if only one thread in a warp enters the statement all of them will perform all the instructions inside the if statement (in this case the result of the calculations by these *PUs* would be discarded), so ideally the code shouldn't have if statements, or in the presence of them, the amount of instructions inside them should be minimal. Another implication of needing every thread in a warp to do the same instructions is that the warp can only execute when all the threads have loaded the values that they need, so code that doesn't explore spacial locality can take up to 32 times longer to execute.

The last issue is that the GPU utilizes a different memory than the CPU, so the values need to be loaded from the CPU into the GPU and later loaded back from the GPU into the CPU and, because this process is restricted by the memory, this process is very costly, which makes it so that algorithms with low complexity take longer with a GPU than without one, since just copying the values from the CPU into the GPU would take longer than just having the CPU compute the values.

C. *Our approach to* CUDA

In order to have decent performance with *CUDA* we needed to change our code in a way that minimizes the previously mentioned constraints. To reduce the number of data copies between the CPU and the GPU we changed our code to make all the functions that manipulated the data arrays into kernel functions so that we would never have to copy the arrays back to the CPU, due to this we had to change both the fluid_solver.cpp and the main.cpp files into .cu files.

Across all the kernel functions we always utilized 256 threads per block, except in the function add_source where it's 512, because those were the values that gave us better results for the size 168, we knew from the start that the size needed to be a multiple of 32 (because if it wasn't, there would be warps where only a few threads were doing useful work) and we knew that the size should be big enough to have more than one warp per block, so that the GPU could execute another warp, while the previous one waits to write or to read the values it needs, in order to reduce the downtime.

We also made it so that all functions were one dimensional, because this way we could better utilize the spacial locality in a warp, which would reduce the amount of loads necessary resulting in an improvement of performance.

The function project was split into three parts. A function with the same original name, that is responsible for calling the functions and executes in the cpu and two kernel functions that realize the calculations that the original version did in CPU in the GPU.

The function *lin_solve* was also split. This functions needs all the red nodes to be calculated before the blacks and can only attempt to proceed to the next iteration when all the black ones are calculated, so to achieve this behavior, we split the function into a function that makes the function calls and the kernel that calculates the values in the GPU and decides if it's black or red due to an argument called parity, that is called twice, the first for red and the second for black.

The other functions were all passed to *CUDA* by following the approach of splitting them into a kernel to do the calculations in *CUDA* by doing a direct translation of instructions. The only exception was the function *set_bnd* where, since all the *for's* shared the same conditions, we were able to aggregate them into one big if, where we have three smaller ifs to determine to what for it originally belonged.

As mentioned previously we had to turn the main.cpp file into main.cu, we had to do this to make the function simulate be the one responsible for allocating memory in the GPU for the data

arrays and to copy their corresponding values, before entering the *timesteps* loop.

D. What could be improved in our implementation

At the moment our implementation is doing the *sum_density* in the CPU instead of the GPU which leads to a slower computing time and makes it so that we need to retrieve the dens array from the GPU, so by putting this function in the GPU as a kernel we could instead of having to copy an array of size N we could copy instead a single float. The reason why we have it in the CPU is because, even with Kahan's algorithm we were getting above the acceptable margin.

Also at the moment our *lin_solve* kernel is only utilizing 50 % of GPU at a time due to half of the threads corresponding to the wrong color. We could solve this issue by separating the black and red tiles into two separated arrays, however this change would require a lot of changes in all functions and proving that they were equivalent would be hard so we didn't implement that version.

E. Expected behavior when scaling

As mentioned previously, copying data across GPU and CPU is extremely time consuming, so for small values the time for copying the array will be larger than the time it takes to compute. Due to this it is expected that an execution time graph for difference sizes shows a linear increase in time for a small size, slowly becoming more curvy, until it reaches a point where the time to do the calculations surpasses the time to copy where the time will begin to grow cubically.

V. Tests and scalability

To test if our *CUDA* approach was correct and scaling as expected we decided to execute our program with different sizes and time our runs in different machines (cpar, day and fortnight) while validating the output by comparing it to the sequential version of the second assignment. We also did a scalability graph on the *openMP* version developed in the previous assignment to compare it to our *CUDA* version.

An important thing to note is that the cluster was unstable so all the values in the graphs were the medium of five different tests eliminating outliers, however for values over 200 we had variance in the tens place so those values aren't completely accurate.

The *openMP* version was supposed to be a cubic curve, however, due to constraints in the cluster, the machine where we were testing would occasionally allow other people to run code between our tests which made the measurements for the 180 and 200 size be slightly inflated, which makes it seem that from 200 to 250 there is barely an increase.

A. *Scalability chart for* OpenMP *(Machine cpar)*

B. *Scalability chart for* CUDA *(Machine day)*

C. *Scalability chart for* CUDA *(Machine cpar)*

As we can see by analyzing the behavior described in the previous chapter is followed by all machines, since we can see a near straight line at the beginning and after 120 we can see the incline of the curve increasing for each following point.

When we compare it to the second assignment graph we can clearly see that at the beginning the *CUDA* version is very close in time to the *openMP*, but as the size grows larger the difference in time between them grows larger and for size 250 the *CUDA* version is almost twice as fast as the *openMP* version.

VI. Conclusion

During the process of writing this report, we identified that the previous assignment submission contained a version where the *add_source* function was not parallelized. Additionally, we acknowledge that certain functions in our current implementation have room for further optimization, as noted in the corresponding sections of this report. However, we believe our overall implementation is robust, achieving a performance improvement of over 25% compared to our previous *OpenMP* version for the target size of 168, which earned a score of 14/15 for execution time despite the absence of parallelization for *add_source*. Working on this assignment using *CUDA* significantly enhanced our understanding of how to navigate the constraints and limitations of GPU's to achieve superior performance.

A. NVPROF CUDA

```
Total density after 100 timesteps: 160834
==772== Profiling application: ./fluid_sim
==772== Profiling result:
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   81.58%  10.6853s     14234  750.69us  728.53us  7.0709ms  lin_solve_kernel_3d(int, int, int, float*, float const *, float, float, int, float*)
                   10.12%  1.32513s      8517  155.59us  152.39us  6.4861ms  set_bnd_kernel(int, int, int, int, float*)
                    3.06%  400.81ms       400  1.0020ms  989.11us  1.0399ms  advect_kernel(int, int, int, int, float*, float*, float*, float*, float*, float)
                    2.02%  263.93ms       200  1.3197ms  1.1954ms  2.1680ms  project3_kernel(int, int, int, float*, float*, float*, float*)
                    1.38%  180.84ms       200  904.21us  898.13us  911.09us  project2_kernel(int, int, int, float*, float*, float*, float*, float*, float)
                    1.29%  169.16ms       400  422.91us  420.33us  430.22us  add_source_kernel(int, int, int, float*, float*, float)
                    0.44%  57.164ms      7144  8.0010us     863ns  6.4575ms  [CUDA memcpy HtoD]
                    0.12%  15.640ms      7118  2.1970us  1.2480us  5.8005ms  [CUDA memcpy DtoH]
                    0.00%  87.777us        19  4.6190us  4.2890us  5.6960us  apply_events_kernel(Event*, int, float*, float*, float*, float*, int)
      API calls:   99.38%  126.794s     23351  5.4299ms  17.506us  113.017s  cudaDeviceSynchronize
                    0.21%  270.90ms       708  382.63us     253ns  188.75ms  cudaMalloc
                    0.18%  229.64ms     14343  16.010us     224ns  6.7682ms  cudaMemcpy
                    0.17%  210.84ms     23970  8.7950us  5.8910us  537.15us  cudaLaunchKernel
                    0.06%  77.826ms       708  109.92us     680ns  2.0636ms  cudaFree
                    0.00%  329.27us         1  329.27us  329.27us  329.27us  cuDeviceTotalMem
                    0.00%  185.88us       101  1.8400us     150ns  72.577us  cuDeviceGetAttribute
                    0.00%  35.449us         1  35.449us  35.449us  35.449us  cuDeviceGetName
                    0.00%  16.976us         1  16.976us  16.976us  16.976us  cuDeviceGetPCIBusId
                    0.00%  3.1390us         3  1.0460us     438ns  1.6990us  cuDeviceGetCount
                    0.00%  1.5590us         2     779ns     408ns  1.1510us  cuDeviceGet
                    0.00%     494ns         1     494ns     494ns     494ns  cuDeviceGetUuid
```