Parallel Computing (Phase 3)

Lucas Oliveira
PG57886

João Silva
PG55618

Rafael Gomes
PG56000

***Abstract*—This report details the systematic analyses and optimization of a non-interactive version of the fluid dynamics simulation code, a 3D version of the Jos Stam's stable fluid solver. This includes both a sequential and parallel implementation, the last one, using OpenMP and CUDA.**

***Index terms*—Execution Time, Vectorisation, Performance, Optimisations, Legibility**

I. Phase 1

The main focus of the first assignment was to explore optimization techniques applied to the given code (single threaded). The goal was to analyze and optimize the program, improving its execution time

A. Code Profiling

The first step was to analyze the provided code, then executing it, to measure its runtime. Following that, we used the *gprof* tool to identify the hotspots. The functions that contributed the most to the execution time of the programs. We found out the *linear_solve* was taking most of the execution time, so we shifted our focus to optimizing this function

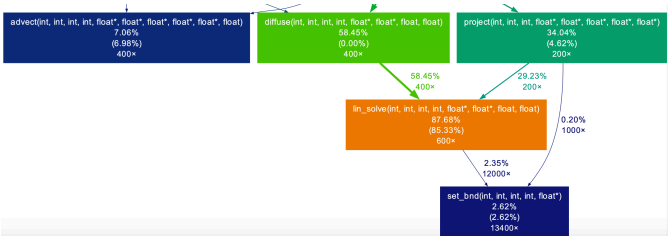

Fig. 1. Results from ***gprof2dot*** of the functions performance for the **non-optimised** code

B. Spacial Locality

We noticed that the way we are iterating through the vectors is not the most correct one. Let's say that in a given iteration we are looking at the element in the position *(i,j,k)*, in the way the *"for cycles"* are nested in the original code, in the next iteration we will be looking at the element in the position *(i,j,k+1)*, and if we look at the definition of the defined macro *"IX"*, this 2 positions have a distance of 44^2 from each other, meaning that between where they are stored in the memory, exist other 44^2 elements of the vector. This leads to a big number of cache misses, because when getting the value of the current element we are iterating through, it most likely won't be in the cache, because it wasn't loaded to the cache in the block fetched by the last iteration. We also note that given 2 elements at positions *(i,j,k)* and *(i,j+1,k)*, there are 44 elements between then in the memory, so we can also expect a big number of cache misses if we change the *"for cycles"* so that *j* is being iterated in the innermost cycle. All this results in very poor spacial locality, as each cache miss forces the program to fetch data from a higher lever of the memory hierarchy, such as L2-cache, L3-cache, or in the main memory wich is significantly slower, rendering the effectiveness the CPU utilization, making it spend more time waiting for data to be fetched, than actually performing computations. We conclude that the best way rearrange the *"for cycles"* is to make it so that the outermost cycle is iterating the variable *k*, and the innermost cycle if iterating the variable *i*, because an element at the position *(i,j,k)*, the very next one in the memory if the element at the position *(i+1,j,k)*, meaning that in this way, at any given point in the cycle, it is almost certain that the current element is already in the cache, being contained in the previous block fetched by some previous iteration.

C. Inlining

We declared our most costive function *lin_solve* with *"attribute((always_inline)) inline"*. The keyword *"inline"* suggests the compiler to inline the function, meaning that when it is called, instead of doing all the work associated with function calls, such has creating a new stack frame and pushing arguments, it will replace the function call with the code of the actual function being called. The keyword *"attribute((always_inline))"* brute forces the compiler to always inline this function, regarding other compiler choices or different optimizations.

D. New Functions

It was noted that at a certain points in the code, the functions numerous functions, are called consecutively 3 times, each time with a different pair of vectors. We can see that each vector pair doesn't interfere with the other, meaning there is no data dependency between them, so we decided to compress the various calls of these functions into a minimum. This way we expect a decrease in the number of instructions, hoping for a better runtime. In order to do this we created the functions *diffuse3*, *addvect3*, *add_source3*, *lin_solve3*, wich do the same computations, but for the 3 vector pairs.

E. Loop Unrolling

We tried the *"-funroll-loops"* flag, in order to unroll the various loops. By reducing the number of iterations in the cycles. Although it improved the runtime, we notice that by unrolling the loops manually in the code, we get even better results, this is because the compiler has limitations when unrolling the loops, it might not detect all the data independencies, so that's we we did.

F. Performance

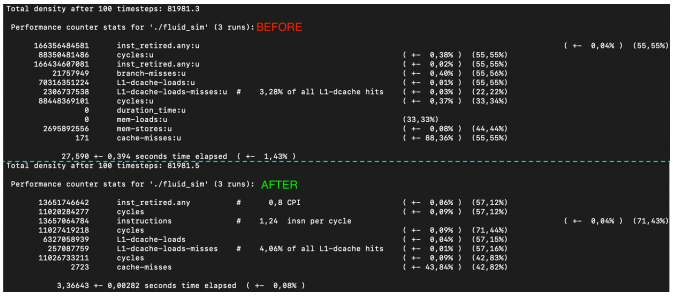

Fig. 2. Comparing the time between the original and the final program

Comparing now the execution data between the original program and the final one, the difference is vast.

Firstly, the final program has **2049649779** less cache misses than the original one. This difference comes from the improved spacial locality.

The number of instructions was also reduced by **152704737939**. This is because of the *"-Ofast"* flag, and because the loops were manually unrolled.

The CPI is approximately **0.8**, primarily due to the use of the *"-Ofast"* optimization flag.

Lastly, we managed a runtime of about **3.3 seconds**, a big difference from the original runtime of **28 seconds**.

II. Phase 2

In the second assignment, the goal was to analyze the code we had for the sequential version and try to find ways to improve the execution time of the program while parallelizing it efficiently

A. Analysis and Alternatives

Option 1: Parallelizing Outer Loop of Red/Black Phases This approach works well when the outer loop has a big number of iterations, to take advantage of the overhead in the spawning of threads. It's also a good option if the first loop processes a large contiguous block of data, which can improve cache performance by allowing threads to work on their own block. However, this method might not be the best if there's invariant workload in the innermost loops, potentially causing imbalance between the work of different threads. This isn't our case, since every iteration of the innermost loops has the same level of workload.

Option 2: Parallelizing Inner Loop of Red/Black Phases This method provides fine-grained parallelism, maximizing thread utilization for large grids with a significant inner dimension. However, frequently spawning and destroying thread adds significant overhead. This option is viable only if the first loops have a very small number of iterations, which isn't the case.

Given all this, we opted for parallelizing the Outer Loop for its balanced workload distribution, and minimum synchronization overhead.

B. Implementation and Optimizations

a) ***pragma omp parallel:*** Instructs the compiler to spawn multiple threads to execute the block of code following it, with each thread running the same code.

b) ***for:*** instructs the compiler to distribute loop iterations within the team of threads.

c) ***Collapse:*** We decided to collapse the two outermost loops, transforming these two in just a single loop, which ups the number of iterations that can be distributed between threads. By not collapsing the two loops, each thread would be given an iteration of the first loop, and then executing all iterations of the second loop sequently, this way we would not be taking full capacity of the threads, since the second loop doesn't have a lot of iterations.

d) reduction(max: *max_c*)*:* This indicates that the *max_c* variable should be handled as a private variable for each thread, and in the end, its final value will be the maximum value calculated by all threads, meaning that if on thread manages to reach the tolerance limit, the final value will also be greater than the limit. This allows us to have no data races without using atomic operations, wich can be inefficient.

C. More Parallelization

Given the number of cycles with big numbers of iterations and without data dependency, we decided to parallelize them, also collapsing them, so we can have more iterations to distribute for threads, as said before.

D. Measures and Conclusion

We made some tests with average times with multiple cores, and got the Fig.3

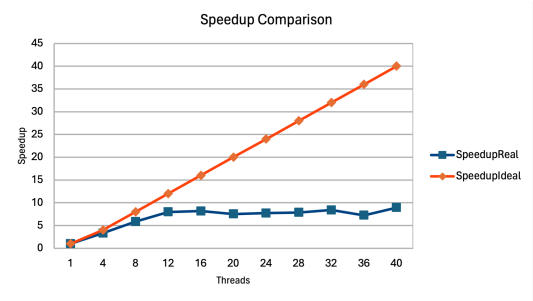

Fig.3. Graphic of the program's speedup with **N** threads

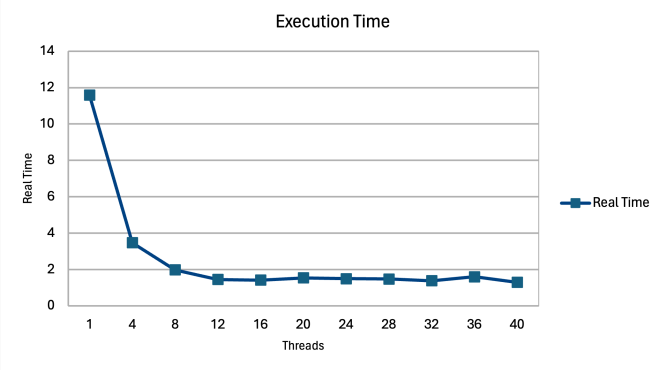

Fig.4. Graphic of the program's execution with **N** threads

We can see that when the number of threads increase, it fails to achieve the ideal speedup, this can be caused because there are some parts of the code that are sequential and cannot be parallelized, other cause that can be its synchronization between processes, where there can be a delay in the barriers that some processes are waiting for others.

III. Phase 3

For this final phase, we chose to use **CUDA** to optimize the previously achieved solution. Several tests were conducted both on the *cluster* and on our local machine (equipped with an **Nvidia GeForce GTX 1660**) to compare the results. Interestingly, the performance on our local machine was better compared to the *cluster*, which we assumed to be equipped with an **Nvidia Tesla K20**.

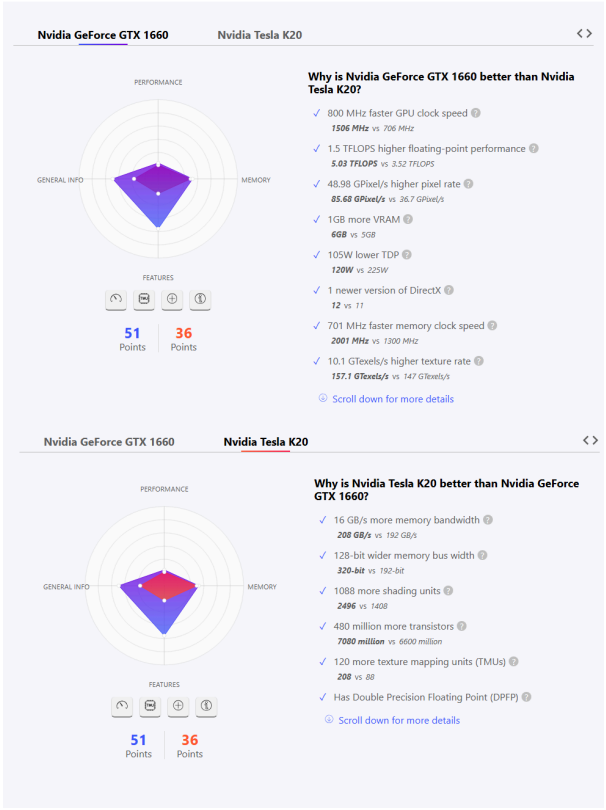

Fig. 5. Comparing between GPU from cluster and GPU from local computer

A. Implementation

The first step in converting the **fluid_solver** from *OpenMP* to *CUDA* was to ensure that all functions parallelized with *OpenMP* directives were transformed into *CUDA kernels*, enabling parallel execution on the *GPU*. This conversion required a significant restructuring of the existing functions, replacing the parallelization directives with explicit configurations of blocks and threads, as well as implementing kernels that could be directly executed on the *GPU*.

In the *OpenMP* implementation, the loops that traverse the three-dimensional grid (with indices i, j, and k) were parallelized using directives such as *#pragma omp parallel for collapse(3)*, which automatically divides the workload among the *CPU* cores. In *CUDA*, this approach had to be adapted to take advantage of the *GPU's* hierarchical architecture, where the work is distributed across grids composed of blocks, each containing multiple threads. As a result, each thread in the *GPU* processes a single cell of the grid.

The conversion of loops to *CUDA* involved correctly calculating the three-dimensional indices (i, j, and k) so that each thread knows which cell of the grid it should process. This calculation is performed, for example, within the *kernels* *lin_solve_red_kernel()* and *lin_solve_black_kernel()*, using the following code:

```
  int k = blockIdx.z * blockDim.z +
threadIdx.z + 1;
  int j = blockIdx.y * blockDim.y +
threadIdx.y + 1;
  int i = blockIdx.x * blockDim.x +
threadIdx.x + 1;
```

Where:

blockIdx: Indicates the index of the block within the grid. In *CUDA*, grids can be three-dimensional, so *blockIdx.x*, *blockIdx.y*, and *blockIdx.z* represent the block's position in each dimension.

blockDim: Represents the number of threads in each dimension of the block. For example, *blockDim.x* is the number of threads in the *x* dimension within a block.

threadIdx: Indicates the index of the thread within the block. Like blocks, threads within a block also have indices in each dimension (*threadIdx.x*, *threadIdx.y*, and *threadIdx.z*).

This allows each thread in the *GPU* to know exactly which cell of the three-dimensional grid it should process, ensuring that the entire grid domain is traversed in parallel.

In this implementation, each kernel is configured with a different configuration of threads and blocks. This configuration varies based on the number of dimensions and the size of the data processed by each kernel. By adapting the number of threads per block and blocks per grid to the specific requirements of each kernel, the program ensures optimal utilization of GPU computational resources, maximizing performance and efficiency.

After the complete conversion of the *fluid_solver* to *CUDA*, we noticed that each function was performing allocation (**cudaMalloc()**), data transfer (**cudaMemcpy()**), and memory deallocating (**cudaFree()**) operations repeatedly, as it needed the data. This behavior proved to be inefficient, as these operations are relatively expensive in terms of execution time, especially when performed multiple times throughout the program. Therefore, we decided to move all memory allocation, transfer, and deallocation operations to the **main file**, so that they were executed only once. With this approach, the necessary variables were allocated and copied to the *GPU* at the start of the program, before any *kernel* was launched, and the memory was only freed at the end of the program. This resulted in a significant improvement in the program's performance.

Regarding atomic operations and reductions, *OpenMP* simplifies these operations through the reduction directive, such as *reduction(max:max_c)*, which automatically computes the maximum value across multiple threads while ensuring proper synchronization and avoiding race conditions. In *CUDA*, these operations must be implemented manually, as there is no equivalent directive for the *GPU*. In the *lin_solve_red_kernel()* and *lin_solve_black_kernel() kernels*, the **atomicMax()** function is used to calculate the maximum value of change across all threads, ensuring that multiple threads do not simultaneously modify the same shared variable, which could lead to incorrect results. However, while atomic operations are safe, they can be slow, especially when used frequently. To mitigate this impact, the following techniques were applied:

The **warp-level reduction** technique is implemented using the **__shfl_down_sync()** function, which allows threads within a warp (a group of 32 threads) to directly share their values through the *GPU's* internal registers without accessing global memory. This significantly reduces the latency associated with thread communication. The use of this function enables each thread to access the value of another thread within the same warp, shifting the value by a specified number of positions, thus facilitating the implementation of parallel reductions. Like this:

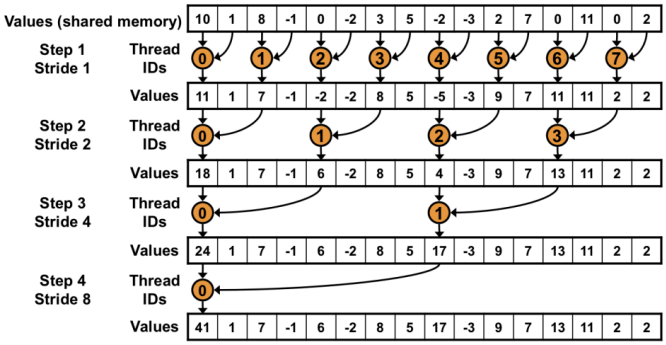

Fig. 6. Parallel reduction: Interleaved Addressing without divergence

The use of the **__ldg()** function, which is a specific instruction for read-only loads, takes advantage of the *GPU's* constant memory cache, which is optimized for repeated accesses to data that is not modified. Using that function instead of a simple global memory access (*x[idx]*) reduces load latency and alleviates pressure on global memory, resulting in a significant performance improvement when frequently accessed data remains unchanged.

Lastly, the use of the **__fmaf_rn()** function combines multiplication and addition operations in floating-point arithmetic in an optimized manner. This function performs the operation $a * b + c$ as a single hardware step, ensuring higher precision by avoiding rounding errors that can occur when multiplication and addition are performed separately. Furthermore, it is more efficient in terms of execution time, as it uses a single instruction instead of two, reducing the number of clock cycles required to complete the operation.

B. Results

After completing the full conversion and optimization of the program, we obtained the following results:

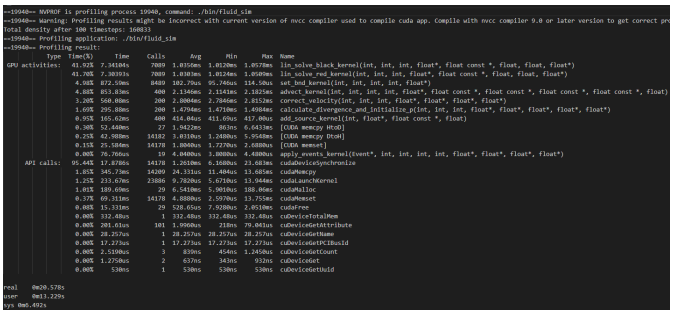

Fig. 7. Time execution of the CUDA version

Unfortunately, the *CUDA* implementation performed worse compared to the *OpenMP* version. The potential reasons for this outcome could include the overhead of memory allocations and data transfers between the *CPU* and *GPU*, the thread synchronization processes, the atomic operations previously mentioned, or even limitations of the *GPU* cores, which may have a low computational precision.

a) *Scalability Test:*

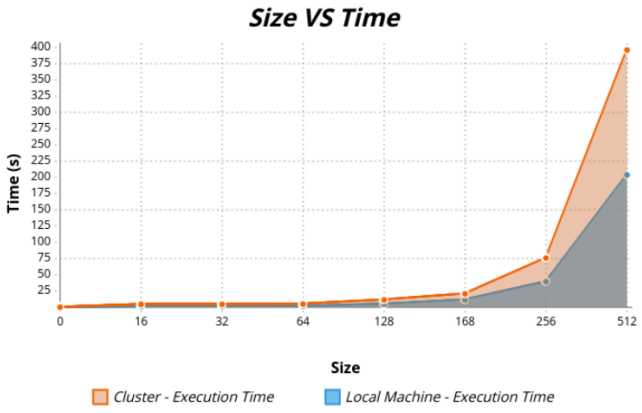

Fig. 8. Scalability comparation between cluster and local machine

The graph shows that the local machine performs better for larger input sizes, indicating that the cluster has worse scalability in the developed. The execution time on the cluster grows significantly beyond size 256, surpassing the local machine's time. This difference may be due to the Tesla K20 being an older GPU architecture, optimized for double-precision computations, while the GTX 1660 is a more modern GPU, better suited for general-purpose CUDA tasks. Additionally, the cluster likely suffers from communication overhead between nodes, which negatively impacts performance. Therefore, for this specific CUDA code, the local machine proves more efficient and scalable than the cluster. machine.

C. Future Improvements

To optimize the CUDA-based fluid solver and reduce execution time below 20 seconds, several improvements can be considered. Adjusting the block and thread configuration to better utilize the *GPU*, utilizing shared memory to reduce global memory access latency, and reducing thread divergence in *kernels* can improve performance. Implementing asynchronous memory transfers and using CUDA streams to enable concurrent execution of tasks can further enhance efficiency. Leveraging compiler optimization flags and targeting specific GPU architectures will also help. We could also Use texture memory for read-only data, tuning the linear solver iterations, and profiling the code to identify and optimize hotspots are important steps. Additionally, ensuring memory access patterns are optimized to be coalesced can maximize memory throughput. These strategies collectively can significantly reduce execution time and can be implemented in the future.

D. Conclusion

Throughout the three assignments completed this semester, we had the opportunity to analyze the impact of parallelism on code optimization. By applying various techniques we learned, we were able to observe significant improvements in the performance of the code initially provided by the professors.

In the first assignment, the implementation of techniques such as algorithm adjustments, *ILP*, memory hierarchy optimization, and vectorization allowed us to achieve a much better execution time compared to the original code.

In the second assignment, by integrating *OpenMP* into the code developed in the first phase, we managed to further reduce the execution time, achieving even better performance results.

Finally, in the third assignment, we explored the *CUDA* programming model. Although we weren't able to achieve a better execution time than the *OpenMP* version or even the optimized code from the first phase, this task proved to be a valuable learning experience. The complexity involved motivated us to further study this programming model in our pursuit of the best possible solutions.