Numerically Solving Navier Stokes equation with GPUs

Gašper Golob

May 22, 2024

1 Introduction

The goal of the project is to use the GPU to accelerate the execution of two methods for solving the Navier Stokes equation. Specifically the two methods that are being accelerated are the implicit and explicit variant of the artificial compressibility method.

The performance and accuracy of the versions, that use the GPU, are then tested on the lid driven cavity problem. The lid driven cavity problem is the case where the fluid is in a square case, where one of the sides induces some movement. The case is schematically presented in the figure 1, from [2].

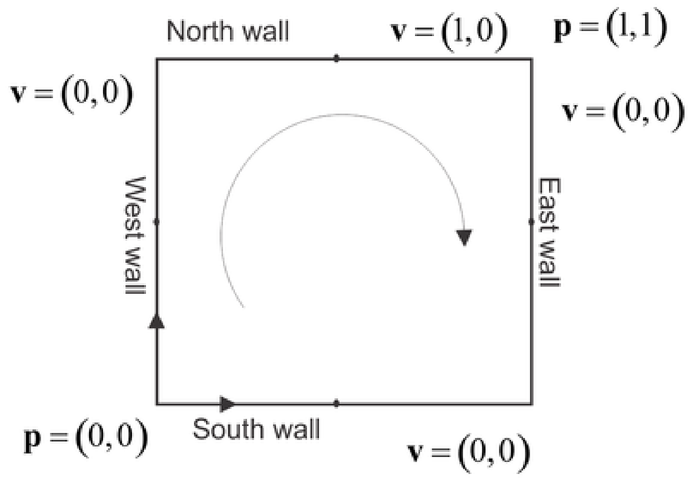

Figure 1: The schematic of the lid driven cavity problem.

The methods work iteratively, where at each time step, they compute new velocities using the Navier Stokes equation. Afterwards they correct the pressure using an assumed pressure volume invariant.

For the purposes of this project the details of solving the partial differential equations are not as important, since the project is mostly concerned with accelerating the solving of systems and various matrix operations.

The GPU versions are also compared to their base versions to gauge the level of improvement and to see if they remain accurate.

The project is made using C++, where the discretization of the differential equations is done using the medusa library [1]. To solve linear systems and work with matrices on the CPU the Eigen library is used. Similarly, to work with matrices and vectors on the GPU side, CUDA libraries such as cuSolver, cuSparse, cuDSS and Thrust are used.

2 Implicit method

For the implicit version of the method we are mostly concerned with solving two linear systems. The first system involves solving a system of the form $M_u x = b$. The matrix M_u changes at every time step, so we need a solver, that can quickly solve a different system on each step.

The other system $M_p x = b$ remains unchanged between different iterations. It is solved up to 100 times per time step, so it makes sense to preprocess the matrix for faster solving.

To try and accelerate the system with the matrix M_u several solvers from different CUDA libraries were tried. To solve the system with the unchanging matrix M_p only one other method was attempted as CUDA currently doesn't have too many linear system solvers, that allow for some preprocessing.

2.1 Solvers

All solvers are initialized using the Eigen `SparseMatrix` type for sparse matrices. To solve a system they receive the right-hand side vector which has the Eigen type `VectorXd`, which is a dense vector whose elements have the type `double`. Ultimately they return a vector of the same type.

2.1.1 SparseLU solver

The base CPU version uses the Eigen library to solve the linear systems. For both of the systems the SparseLU solver is used, which implements the supernodal LU factorization of the matrix, and then solves the system using the computed triangular matrices.

2.1.2 QR solver

The QR solver uses the function `cusolverSpDcsrlsvqr`, which given a sparse matrix A in the compressed sparse rows format alongside two dense vectors x and b solves the system $Ax = b$ on the GPU. In the background the method computes a sparse QR factorization of the matrix A, while also providing some reordering, to minimize zero fill in. For the purposes of this project, the `symrcm` reordering scheme is used.

2.1.3 cuDSS solver

The cuDSS solver is the main solver currently implemented in the cuDSS library. Since the matrix doesn't have any extra properties it uses the LDU decomposition. Same as the previous solver it takes the matrix in the CSR format, while the right-hand side and solution vectors are dense. All the matrices and vectors are transferred to the GPU where the decomposition is computed and the systems are solved.

2.1.4 RF solver

The refactorization solver is meant to sequentially solve multiple systems of the form $Ax_i = b_i$ for multiple vectors x_i, b_i and a fixed matrix A.

The comparison of times can be seen on figure 2. On the graph we are comparing the base CPU implementation denoted with lid driven cavity with other variants. The first time it has to solve a system it uses the `cusolverSP_lowlevel_preview` library. This library first computes the sparse LU decomposition (with some reordering) of the matrix and then solves the first system. Afterwards the sparse LU decomposition is extracted and passed to the `cuSolverRF` library, which can then use it to solve later systems.

The first iteration is done completely on the CPU, as the `cusolverSP_lowlevel_preview` library currently lacks GPU versions of the functions. Later, the systems are solved using the functions from `cuSolverRF` which works on the GPU.

2.2 Benchmarks

To reiterate we are solving a system of the form $M_u x = b$, which has to be solved once per time step and $M_p x = b$ which has to be solver multiple times per time step. The matrix M_u changes every iteration while the matrix M_p remains fixed.

The system with the matrix M_u is accelerated using the QR and cuDSS solvers. The second system is only accelerated using the RF solver, as this is the only one that allows preprocessing.

The execution time of the programs, when using different solvers can be seen in the figure 2.

We can see that the lid driven cavity version which is the reference CPU implementation is the slowest, except possibly right at the beginning, where there is still not that much parallelism.

The second-best case is when we use the QR solver to solve the system with the matrix M_u. It starts of a bit slower than the CPU version but quickly becomes faster.

The best case is when using the cuDSS solver to solve the system with the matrix M_u. For large enough N the performance increase seem to be about 100x, which is far better than even the QR solver.

The last case uses the cuDSS solver for the system with the matrix M_u and the RF solver for the system with the matrix M_p. Unfortunately it seems that solving the LU system on the GPU slows down the execution, which makes some sense as solving triangular systems is a very sequential operation.

Regardless it might prove useful to run the benchmarks on even larger cases, as there might be some performance improvements for very large N. The limiting factor here might however be, that the LU decomposition is done on the CPU, so the setup time might take too long.

3 Explicit method

The explicit method starts by constructing some matrices using the Medusa library. Afterwards it only requires matrix multiplications and some element wise operations. This means we can transfer all matrices and vectors to the GPU and do most of the work there.

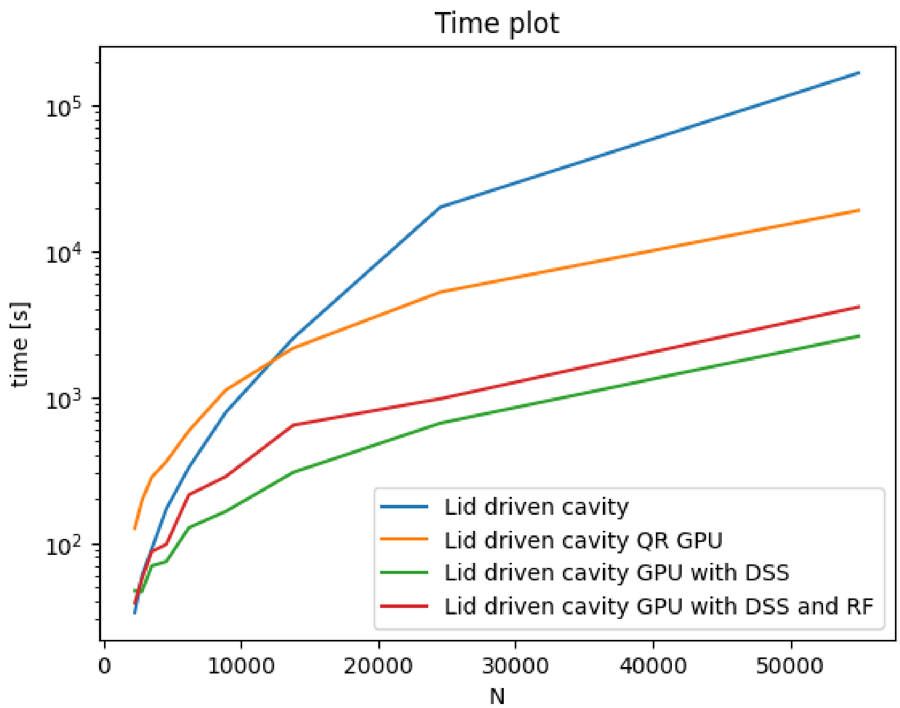

Figure 2: Graph of times to simulate 50 seconds of the lid driven cavity problem using the implicit method, for different problem sizes.

3.1 Basic GPU operations and types

For all operations sparse matrices are given in the compressed sparse row format and the vectors are the usual dense vectors given as simply arrays on the GPU.

3.1.1 VectorGPU and MatrixGPU

These are some basic wrapper classes that convert to and from their Eigen counterparts. Data that is passed to them is transferred to the GPU, where it can be used with other functions.

3.1.2 Matrix multiplication

To multiply sparse matrices with vectors the cuSparse library is used. More specifically the `cusparseSpMV` function is used which takes a matrix in the CSR format, a dense vector and return a dense solution vector. Some multiplication data is initialized before the simulation as it speeds up the later multiplications.

3.1.3 Transforms and reduces

For element-wise operations on dense vectors thrust's `transform` and `transform_reduce` functions are used.

The first use case is to allow element-wise vector multiplication, for which we can use the `transform` function and the built-in `multiplies` operator.

The second use case is to define a new operator `axpy_functor` which is initialized using some value α and then for given vectors x and y computes $y = \alpha x + y$. The operator is again applied using the `transform` function.

Another case is the definition of the operator `abs_functor` which for a given vector computes the element-wise absolute value.

The last two operators that are defined are `u_tuple_functor` and `tuple_max_functor`, which are then used with the `transform_reduce` function. The `u_tuple_functor` takes an array as an input and for a given set of indices returns an array of pairs of indices and values of the array at some indices relative to the given ones. Then the `tuple_max_functor` finds the maximum value among the values which have corresponding indices. This is used to find the maximum velocity in the y direction in the middle height of the lid driven cavity case.

Another function from Thrust that is often used is `max_element` which simply return the maximum element of a vector.

3.2 Benchmarks

We use the previously described functions to port the CPU code to the GPU. Additionally, we also use CUDA streams to execute multiple operations on the graphics card at once, where this is possible.

We can see the execution times of the base CPU version in comparison the GPU version on figure 3.

Once again the CPU version is better at the smallest values of N, but it quickly becomes better as N grows larger.

4 Further work

While the project is nearing completion there is some more work to be done. For an instance it would be good to benchmark the performance for some even larger N, to see if the trends continue as the amount of data on the GPU increases. It might also make sense to do some more detailed benchmarks to see how fast particular parts of the code are.

Besides that, the repository still needs to be cleaned up, and the code should be made easier to run as it is currently setup specifically for the server, which was used for the project's development.

References

[1] Jure Slak and Gregor Kosec. 2021. Medusa: A C++ Library for Solving PDEs Using Strong Form Mesh-free Methods. ACM Trans. Math. Softw. 47, 3, Article 28 (September 2021), 25 pages. https://doi.org/10.1145/3450966

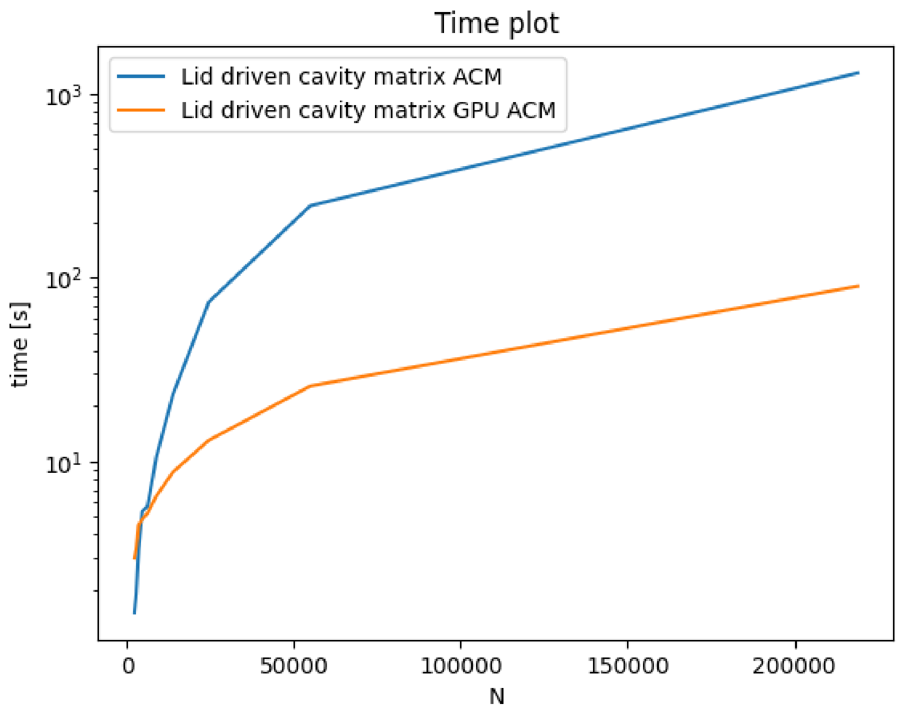

Figure 3: Graph of times to simulate 50 seconds of the lid driven cavity problem using the explicit method, for different problem sizes.

[2] G. Kosec; A local numerical solution of a fluid-flow problem on an irregular domain, Advances in engineering software, vol. 120, 2018 DOI: 10.1016/j.advengsoft.2016.05.010