Chalmers University of Technology

GPU-accelerated Computational Methods using Python and CUDA

Computational Fluid Dynamics

Author:

Pontus Malmsköld

Ritoban Biswas

Supervisor:

Prof. Lars Davidson

January 26th 2024

Abstract

This report describes the detailed study of Computational Fluid Dynamics(CFD) algorithms implemented to use NVIDIA GeForce RTX 3070 General Purpose GPU for performance improvement. Acceleration is achieved by using methods of CUDA implemented via python Cupy API's to offload the vectorized parallel processing into many core GPU architecture.

This paper focuses upon using different CFD solvers like Jacobi method & Gauss-Seidel and describes the benefits of GPU acceleration. By utilizing vectorization and the Jacobi solver to parallelize the computations on the GPU, a speedup of over 400 times was reached compared to an unoptimized code being run on the CPU.

Introduction

Graphics Processing Units (GPUs) are specialized hardware engineered to enhance the processing of graphics and visualizations. Beyond their traditional use, GPUs have gained traction in computational mechanics applications, serving tasks like scientific computing and data analysis. This shift is driven by the parallel processing capabilities of GPUs, significantly boosting the speed of specific computations.

Programming for the GPU typically involves languages such as CUDA C/C++, Python, Fortran, or C#, utilizing bindings to the CUDA API. In this report, we specifically delve into the use of GPU-accelerated Python code for computational fluid dynamics in non-compressible flow applications. The report provides a brief survey of the tools and libraries accessible for GPU programming in Python and addresses challenges encountered when incorporating CUDA code to expedite the performance of pre-existing Python CFD code within these specialized domains.

Background

Finite volume method (FVM)

As the basis for the project, a Finite Volume Method (FVM) code that solves the momentum equations using the SIMPLE algorithm for a co-located grid is used.

SIMPLE is a common algorithm used to solve the conservation of momentum equations. The conservation of momentum equations cannot be solved directly using an iterative solver due to the pressure gradient acting as a source term. The pressure field needs to be solved which in turn gives the pressure gradient. This is called pressure-velocity coupling. To do this an equation called the pressure correction equation is solved, which is an equation that is derived from conservation of mass. The SIMPLE algorithm works the following way:

1. Guess a pressure field
2. Solve velocity fields using some iterative solver, a direct solver cannot be used due to the system of equations being non-linear and coupled.
3. Calculate the pressure correction field using some iterative solver
4. Use the pressure correction field to correct the velocities and the pressure field
5. Use the corrected pressure field as new guess
6. Iterate from step 2 until convergence is reached

An issue with pressure-velocity coupling is a numerical phenomenon called checkerboarding in which unphysical oscilations of the pressure can be observed. This is due to the pressure in the odd and even cells being uncoupled. This is primarily solved in two different ways: using a staggered grid or by using Rhie & Chow interpolation. In a staggered grid the velocities are calculated on the cell faces instead of the centers, while pressure is still calculated at the cell centers. Rhie & Chow interpolation is an interpolation scheme specifically designed to mitigate checkerboarding. Rhie & Chow therefore makes it possible to have a co-located grid where all fields are calculated at the cell centers. This method is more often used in commercial solvers due to its ability to easily handle unstructured grids.

Iterative solvers

The two iterative solvers used in the project are: the Gauss-Seidel method and the Jacobi method. Through discretization the following equation for an arbitrary two-dimensional field is found:

$$\phi_P = \frac{1}{a_p}(a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + S_u)$$

Where the subscripts show in which cell the value is stored (except for S_u which is always stored in the current cell), P is the current cell, E is the east cell, W is the west cell and so on. For both methods, an initial field is guessed, and ϕ_P is calculated for each cell using the cells located right next to it. The difference between Gauss-Seidel and Jacobi is that in Gauss-Seidel ϕ_P is calculated sequentially for each cell and updated right after being calculated, which means that the value calculated in a cell will be dependent on the value calculated in the previous cell. The Jacobi method works by calculating all cells using the guessed field and only updating after all cells are calculated. This means that the Jacobi method converges more slowly than Gauss-Seidel. It is however significantly more suitable to parallelization due to Gauss-Seidel being fully sequential.

Computational Techniques & High Performance with GPU

NumPy and Numba are two great Python packages for matrix computations. Numpy is a Python library that provides a multidimensional array object, various derived objects, and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

CuPy is an open-source NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. CuPy acts as a drop-in replacement to run existing NumPy/SciPy code on NVIDIA CUDA or AMD ROCm platforms.
CuPy utilizes CUDA Toolkit libraries to make full use of the GPU architecture.

The foundation of the CUDA architecture revolves around a scalable array of multithreaded Streaming Multiprocessors (SMs). When a kernel grid is invoked by a CUDA program on the host CPU, the blocks within the grid are systematically enumerated and assigned to multiprocessors with available execution capacity. On each multiprocessor, the threads belonging to a specific thread block execute in parallel. Furthermore, multiple thread blocks can concurrently execute on a single multiprocessor. As thread blocks complete their tasks, new blocks are initiated on the now available multiprocessors.

Efficient data transfer between the CPU and GPU is essential. Minimizing unnecessary data movement is crucial for optimal performance.

Debugging

NVIDIA Nsight Compute is an interactive kernel profiler for CUDA applications. It provides detailed performance metrics and API debugging via a user interface and command line tool. Other debugging tool used is line_profiler in python giving a detailed view of the time spent running each line of code

Methodology

The code used for this project solves a two-dimensional lid-driven flow using SIMPLE and Rhie & Chow interpolation. The original code is taken from an assignment in the course *MTF073 Computational fluid dynamics (CFD)*. The original code uses Gauss-Seidel to solve all equations. The code also uses a large number of for-loops, making the entire code wholly sequential.

The initial part of the project consisted of vectorizing the code in order to make it more suitable for parallelization. Further, the Jacobi method was implemented which allowed Cupy to be used. Without vectorization and with Gauss-Seidel, Cupy is very inefficient due to all operations being sequential.

The following GPU was used in the project: NVIDIA GeForce RTX 3070 having CUDA cores 5888 and total memory of 8192 MB.

Listing out some of the Python libraries used in the project
- Linear algebra (cupy.linalg)
- Sparse matrices (cupyx.scipy.sparse)
- Matplotlib (Visualization with Python)

Results

Below is the code displaying the implementation of GPU acceleration through CUDA and CuPy. We highlight the utilization of a line profiler for debugging purposes.
https://github.com/ritobanbiswas/TRA220_GUP-accelerated_CFD/blob/d496067feb87a10999521dc5d9c9057c4f19c660/CFD_notebook_updated_solver.ipynb

Implementation of the Jacobi solver has been shown to be problematic for solving the pressure correction equation. As can be seen in the figure below, when solving the pressure correction equation using the Jacobi method, conservation of mass does not converge, it does however converge when using Gauss-Seidel for pressure correction.

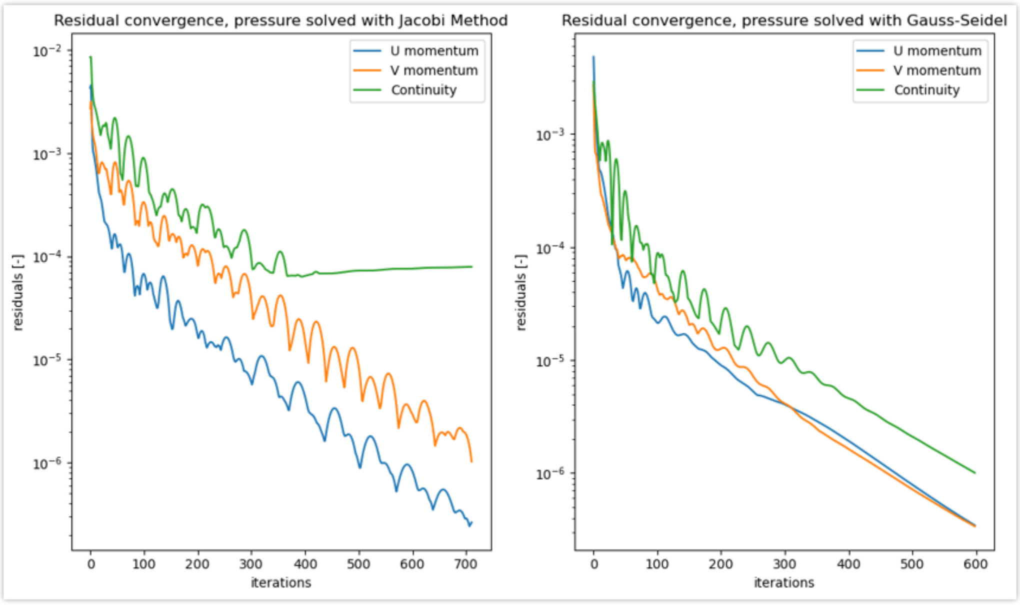

Figure 1. Conservation of Mass and Momentum, convergence

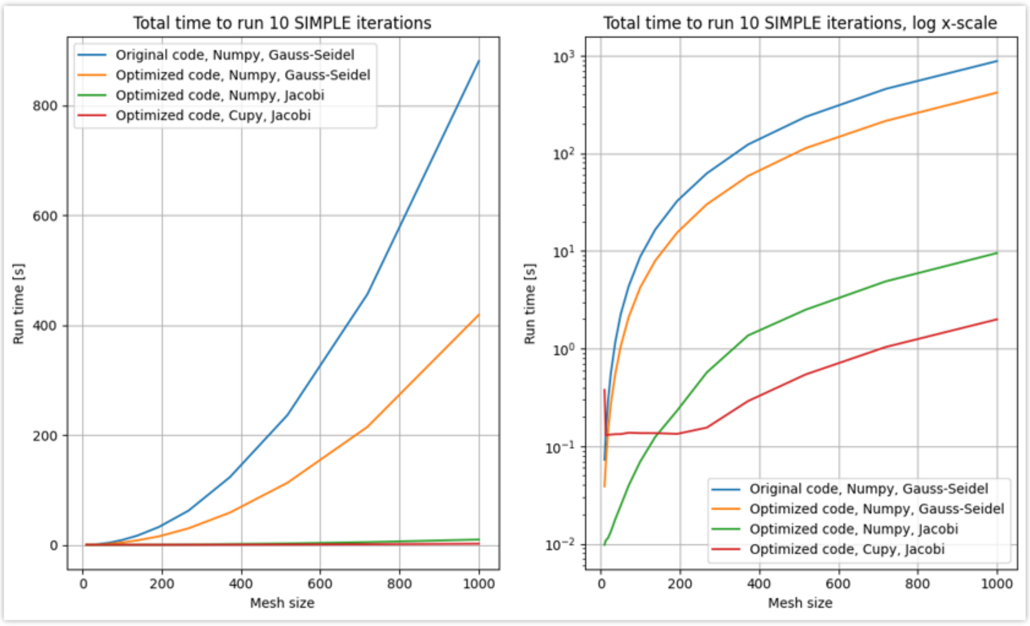

Figure 2. Run-time, Original Code vs Optimized code

The figure above shows four different simulations, one for the original unaltered code and three using the vectorized code (labeled as optimized) using different solvers and packages. The x-axis shows the number of nodes in each dimension of the mesh, while the y-axis shows total runtime for 10 SIMPLE iterations.

Conclusion

Figure 1 shows that using the Jacobi method to solve the pressure correction equation does not converge. The reason for this is unknown and a solution to this problem was not found. Therefore, it was decided that the results of the project should exclude convergence. Instead, 10 SIMPLE iterations were used for different mesh sizes in order to compare the different solvers and codes. As can be seen in Figure 2, the optimized code using Gauss-Seidel is roughly twice as fast as the original code, this is due to vectorization. It can also be seen that using the Jacobi method is significantly faster than using Gauss-Seidel. For a small mesh size, using Numpy is faster than Cupy, but around roughly 170 nodes in each dimension (around 30000 cells), the two packages perform the same with Cupy being more efficient for larger mesh sizes. Initial startup is slow for Cupy because Python code transformed into C language and compiled before started running on every startup, but this can be avoided by hooking compiled Cupy scripts.

The project has shown that harnessing the parallelization capabilities of a GPU for CFD simulations yields evident advantages. Substantial improvements in simulation time were reached by implementing a solver that parallelized well together with using Cupy.

In conclusion, this report explored the performance of a Computational Fluid Dynamics (CFD) solver implemented in Python, initially using the NumPy library and subsequently transitioning to CuPy for GPU acceleration. The advantages and challenges encountered in this transition underscored the impact of utilizing specialized hardware for parallel processing.

The initial implementation with NumPy provided a baseline for the CFD solver, highlighting the capabilities of a conventional CPU-based approach. While satisfactory for smaller-scale simulations, the need for enhanced computational speed prompted the exploration of GPU acceleration using CuPy.

Upon implementing CuPy, a distinct acceleration in the solver's performance was observed, particularly for computationally intensive tasks such as matrix operations. The parallel processing capabilities of the GPU significantly reduced simulation times, demonstrating the potential for substantial speedup in large-scale simulations. It can be seen in Figure 2 that a speedup of over 400 times was achieved for a very fine mesh, showing the potential in speedup using GPU acceleration.

Issues related to using the Jacobi method for the pressure correction equation were found but due to insufficient time a solution was not reached. Further improvements to the code could be made by making sure that the solution converges for solvers other than Gauss-Seidel. The boundary conditions in the code are set explicitly at nodes on the boundary, changing this implementation by including the effects of the boundary conditions in the source term and removing the nodes on the boundary should make it possible to structure the system of equations of the standard form given as: $Ax = b$
Doing this should make it possible to implement more sophisticated solvers included in Cupy that should significantly speed up the convergence of the solution