Chalmers University of Technology

GPU ACCELERATED COMPUTATIONAL METHODS USING PYTHON AND CUDA

Computational Fluid Dynamics

Author:
Bala Kumaresh Thileep Kumar
Wei Liu
Wuyang Hao

Supervisor:
Prof. Lars Davidson

February 24, 2025

Abstract

This report explores the implementation and evaluation of GPU-accelerated computational methods to enhance the performance of a 2D finite volume Computational Fluid Dynamics (CFD) solver, `pyCALC-RANS`. By leveraging CuPy and CUDA programming, the study aims to optimize critical components of the solver, including iterative linear algebra operations and sparse matrix computations. The profiling of the existing code identified major bottlenecks, which were subsequently targeted for GPU acceleration using solvers such as GMRES, Conjugate Gradient (CG), and PyAMGX. Comparative analysis highlights significant computational speedups, particularly for high-resolution mesh grids, demonstrating the advantages of GPU-based parallel processing in reducing execution time for large-scale turbulence modeling. Specifically, at intermediate mesh sizes, GPU acceleration achieved a speedup of up to 14.84× for GMRES and 14.36× for CG. However, as mesh sizes increase, GPU performance starts to degrade due to memory latency and bandwidth saturation.

Contents

1 Introduction

General-purpose Graphics Processing Units (GPUs) have become increasingly popular for their efficiency in handling massively parallel computations, outperforming Central Processing Units (CPUs) in tasks like scientific computing, machine learning, and CFD. Their parallelized architecture is well-suited for large-scale workloads. While programming GPUs often requires custom kernels, libraries such as CuPy simplify access to GPU acceleration through high-level functions. By leveraging these capabilities, this report focuses on extending the pyCALC-RANS CFD code [4] with GPU acceleration to significantly enhance performance and scalability, enabling faster and more efficient solutions for large-scale CFD problems.

1.1 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) is a branch of fluid mechanics that utilizes numerical methods and algorithms to analyze and solve problems involving fluid flows. It has become an essential tool in engineering and scientific research, enabling the simulation of complex physical phenomena such as turbulence, heat transfer, and multiphase flows. CFD applications span various industries, including aerospace, automotive, energy, and environmental sciences. The foundation of CFD lies in the discretization of the governing equations of fluid motion, namely the Navier-Stokes equations. These equations describe the conservation of mass, momentum, and energy within a fluid system. By discretizing these equations using finite volume method, CFD solvers compute approximate solutions for fluid flow problems on a computational grid. The pyCALC-RANS code, which forms the basis of this project, is a 2D finite volume CFD solver that employs the SIMPLEC algorithm for pressure-velocity coupling. It incorporates turbulence modeling using the standard k-omega model and has recently been extended to include an Explicit Algebraic Reynolds Stress Model (EARSM) improved with a Neural Network. By leveraging GPU acceleration, the goal is to enhance the computational efficiency of this solver, particularly for cases involving high grid resolutions and complex turbulence modeling.

1.2 Why GPU Computing

The increasing complexity of modern simulations, particularly in Computational Fluid Dynamics (CFD), demands significant computational resources for accurate and timely results. Traditionally run on CPUs, these simulations increasingly leverage GPUs due to their parallel architecture, which is optimized for tasks decomposable into smaller, independent sub-tasks. Unlike CPUs designed for serial processing, GPUs perform many computations simultaneously, delivering substantial performance gains for CFD applications involving large datasets and intensive calculations [1]. GPU computing enhances CFD solvers by accelerating time-consuming operations like matrix computations, finite difference calculations, and memory transfers, with advances in GPU hardware, such as increased core counts and higher memory bandwidth, further improving their suitability for solving complex partial differential equations (PDEs) on large grids [2]. High-level libraries and frameworks like CuPy, PyCUDA,

and CUDA simplify GPU utilization by abstracting parallel programming complexities, enabling efficient implementation of GPU-accelerated functions in Python and other languages, achieving substantial speedups with minimal codebase modifications [3]. For the pyCALC-RANS code, GPU acceleration offers significant performance improvements, particularly for large-scale simulations with fine mesh resolutions, allowing for efficient handling of higher detail levels, faster predictions, and reduced turnaround times in practical engineering applications.

1.3 CUDA Programming Model

CUDA (Compute Unified Device Architecture) by NVIDIA is a platform and programming model enabling GPUs to handle computationally intensive tasks. It uses a heterogeneous computing model where the CPU (host) manages program flow, data transfers, and kernel launches, while the GPU (device) executes parallel kernels. CUDA employs a thread hierarchy of threads, blocks, and grids to scale parallelism effectively. The GPU memory hierarchy includes global memory (large but slow), shared memory (faster and shared within blocks), and local memory (private to threads). Optimizing performance requires minimizing the significant overhead of host-device data transfers and maximizing the use of faster shared memory, which can be up to 100 times faster than global memory. NVIDIA also provides optimized libraries like cuBLAS, cuFFT, and cuDNN, reducing the need for low-level coding in common algorithm implementations.

1.4 CuPy

CuPy is an open-source, GPU-accelerated library designed as a NumPy equivalent, offering a Python API that wraps various CUDA library functions for high-performance computing on NVIDIA GPUs. Most NumPy operations are supported in CuPy, allowing developers to easily convert CPU-based programs to GPU-accelerated ones with minimal code changes. Built on CUDA, CuPy provides efficient implementations of operations like array manipulations, linear algebra, Fourier transforms, and sparse matrix solvers via the cupyx.scipy.sparse module, making it particularly suited for computationally intensive tasks like CFD. Its compatibility with CUDA libraries such as cuBLAS, cuFFT, and cuDNN further enhances performance and usability, enabling seamless integration into Python workflows.

2 Methodology

This section outlines the systematic approach adopted to optimize and evaluate the performance of the codebase. It begins with an overview of the codebase, followed by profiling techniques to identify computational bottlenecks using cProfile. The identified bottlenecks are analyzed to measure speedup, and GPU acceleration is implemented to enhance performance. Finally, the optimized code is tested with varying mesh sizes to evaluate its scalability and computational efficiency.

2.1 Codebase Utilized in This Report

This report focuses on accelerating the simulation of a fully-developed channel flow at $Re_\tau = 5200$ using **pyCALC-RANS**. The optimization will be performed using CuPy, in particular its sparse functionality of `cupyx.scipy.sparse`. The emphasis is on benchmarking the performance of various solvers across different hardware configurations to evaluate the effectiveness of GPU acceleration in improving computational efficiency.

2.2 Identifying Bottlenecks with cProfile

To optimize the performance of the `pyCALC-RANS` code, `cProfile`, a built-in Python library, was utilized (A.1) to identify computational bottlenecks. By profiling specific functions, `cProfile` provides detailed insights into execution times, helping pinpoint the most time-consuming sections of the code. The profiling highlighted three functions as the primary contributors to the computational overhead:

- `solve_2d`: This function is responsible for solving discretized linear algebraic equations, either using direct solvers or iterative methods. It is computationally demanding due to the manipulation of large sparse matrices and the need for multiple solver iterations to achieve convergence. The majority of its runtime is consumed by sparse matrix operations and the solver backend, particularly for large computational grids. The runtime of `solve_2d` scales significantly with the grid size and the number of iterations required to achieve convergence, making it the most computationally intensive component of the code.

- `conv`: Responsible for computing convection terms based on velocity and pressure fields (with Rhie and Chow interpolation), this function incurs a high computational cost due to numerical flux calculations performed at each cell face. Its frequent invocation during each iteration significantly impacts runtime.

- `coeff`: This function computes the coefficients for discretized equations, incorporating convection, diffusion, and source term contributions. It is resource-intensive due to operations like turbulent viscosity calculations, hybrid scheme applications and the computational overhead associated with large grid sizes and complex numerical schemes.

The profiling results provide a clear roadmap for optimization efforts by targeting these functions and their dependencies. In conclusion, optimizing `solve_2d`, `coeff`, and `conv` functions offers significant potential to reduce computational overhead and improve the overall performance of the `pyCALC-RANS` code. Future efforts can focus on streamlining sparse matrix operations, leveraging more efficient solvers, parallelizing the coefficient computations, and optimizing interpolation algorithms for convective terms. These insights from the profiling process, further corroborated by findings in the referenced report [5], serve as a foundation for targeted enhancements, ultimately enabling faster and more efficient simulations while maintaining accuracy.

2.3 Measuring Speedup Using cProfile Results

The performance of the `solve_2d`, `coeff`, and `conv` functions was analyzed using `cProfile` for the `120x120` grid. The profiling was conducted for **2000 iterations** instead of the original **20000 iterations**, with a convergence criterion of the residual dropping below 10^{-3}. Various solvers, including Direct, PyAMG, CGS, CG, GMRES, QMR, and LGMRES, were evaluated for efficiency. Figure 1 shows the cumulative execution times (in seconds) for these functions.

- `solve_2d` is the most time-consuming function across all solvers, with the Direct solver requiring 427.30 seconds compared to 30.50 seconds for CG.
- Iterative solvers like CG and CGS are more efficient than the Direct solver, making them preferable for larger grids.
- The `coeff` and `conv` functions exhibit consistent performance across solvers, with minimal variations in runtime.

These results highlight the computational burden of `solve_2d`, particularly for direct solvers, and underscore the advantages of iterative solvers for reducing computational time.

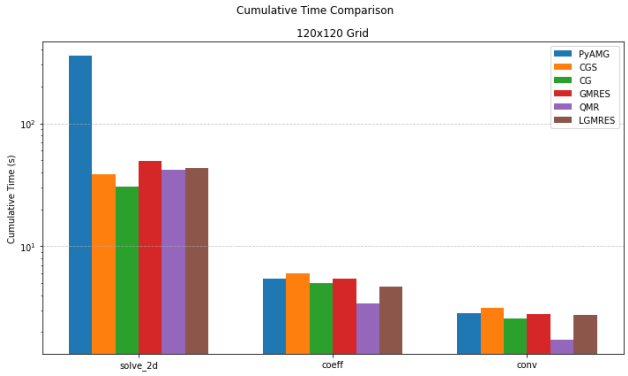


Figure 1: Cumulative Time Comparison for Different Solvers on a 120x120 Grid.

2.4 Implementation of GPU Acceleration using GMRES Solver

To accelerate the computational fluid dynamics (CFD) code, the entire codebase was converted to CuPy, replacing all NumPy and SciPy operations with their CuPy equivalents. By ensuring that all computations, including pre-processing, matrix assembly, solver execution, and post-processing, are executed on the GPU, this approach eliminates the memory overhead caused by CPU-GPU data transfers during iterations. Only the initial data transfer from the CPU to the GPU and the final transfer back are required, making this method significantly more efficient. To achieve this, all NumPy functions were replaced with their CuPy counterparts, such as using `cp.append` instead of `np.insert`, `cupy.ndarray.max` instead of `np.max()`, and `cupy.ndarray.flatten` instead of `np.matrix.flatten`. Unsupported operations were re-implemented using CuPy's array manipulation features to maintain the

functionality of the original code. Additionally, the solver was replaced with CuPy's GMRES solver, the only solver in CuPy suitable for CFD applications. This solver performed all matrix and vector operations on the GPU, ensuring full utilization of GPU resources. This approach minimizes memory overhead during iterations and fully leverages GPU resources, resulting in significantly improved performance and scalability for large-scale turbulence modeling simulations.

2.5 Implementation of GPU Acceleration using CG Solver

The Conjugate Gradient (CG) solver was employed within the GPU-accelerated CFD framework for solving symmetric positive-definite systems. Using CuPy, all computational data, including the sparse matrix A, solution vector x, and load vector b (i.e., $Ax = b$), were directly loaded into GPU memory, ensuring minimal CPU-GPU communication overhead. The CG solver from `cupyx.scipy.sparse.linalg` was utilized to perform iterative matrix-vector operations entirely on the GPU. While CG is computationally efficient for symmetric and well-conditioned matrices, it was observed that the solver required a higher maximum iteration limit (`nmax`) to achieve convergence compared to GMRES. This behavior is attributed to the solver's sensitivity to the condition number of the matrix, as poor eigenvalue clustering can slow convergence. To address this, preconditioning techniques can be employed to improve the spectral properties of the matrix. Despite requiring more iterations, CG proved effective in solving specific subsystems within the CFD code where the matrix characteristics aligned with the solver's strengths. For these systems, CG provided a viable alternative to GMRES with potentially lower memory requirements, as it avoids the need to store large Krylov subspaces. The solver parameters, including tolerances and iteration limits, were carefully configured to balance computational efficiency and accuracy.

2.6 Tests and Benchmarking

The `pyCALC-RANS` code, developed for simulating fully-developed channel flow at $Re_\tau = 5200$, spends most of its computational time in the `solve_2d` function, which solves finite volume equations $A\phi = b$. Profiling also identified `conv` and `coeff` as key contributors to overhead, especially for larger grids.

GPU-accelerated solvers (GMRES, CG, PyAMGX) will be compared with CPU-based SciPy solvers across mesh sizes of 120×120, 240×240, 480×480, and 960×960. These tests will evaluate scalability, speedup, and convergence, demonstrating the benefits of GPU acceleration for high Reynolds number simulations.

3 Results

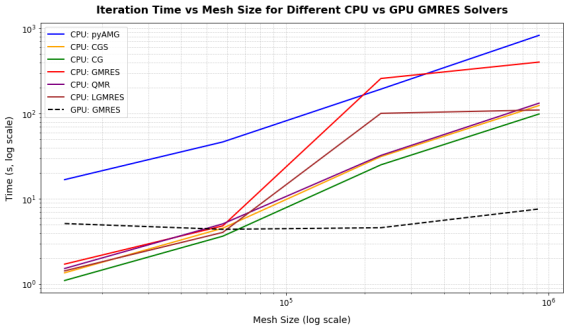
This section presents the outcomes of implementing GPU acceleration in the `pyCALC-RANS` code and evaluates its performance across various test cases. Key metrics, such as computational speedup, solver efficiency, and scalability, are analyzed for different solvers, including GMRES and CG, on varying mesh resolutions. The results provide insights into the effectiveness of GPU acceleration in reducing computational overhead, with a particular focus on large-scale turbulence modeling simulations. Through comparisons with CPU-based approaches, the benefits of GPU computing in enhancing the solver's efficiency for high-resolution CFD problems are demonstrated.

3.1 Performance Gains with GMRES and CG

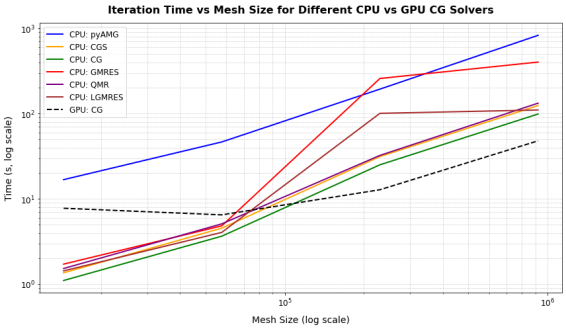
The comparative performance analysis of GMRES and CG solvers on CPU and GPU architectures highlights significant computational advantages, particularly for GPU-accelerated implementations. The GPU-based CG solver shows substantial reductions in computational time across all mesh sizes compared to its CPU counterpart, as illustrated in Figures 2a and 2b. This improvement is especially evident for larger meshes (960×960), where the GPU CG solver achieves up to an order-of-magnitude speedup. This performance gain stems from the GPU's parallel processing capabilities, optimized for iterative linear algebra operations involving sparse matrices. The maximum global iteration count was set to 50 to ensure residuals fell below the range of 10^{-3}.

On the CPU, GMRES exhibits stable convergence but suffers from disproportionate computational time growth with mesh size due to its reliance on large Krylov subspaces. This overhead becomes particularly significant at higher resolutions, as seen in Figure 2a. In contrast, CG demonstrates more consistent scaling across both CPU and GPU, remaining computationally efficient for symmetric positive definite matrices. However, the GPU CG solver occasionally triggered warnings such as: "Warning in module `solve_2d`: convergence in sparse matrix solver not reached." This indicates that the solver did not reach the specified residual tolerance within the iteration limit, potentially due to matrix conditioning, numerical instabilities, or insufficient iterations. Despite this, the GPU CG solver leveraged the GPU's high memory bandwidth and parallel processing capabilities, achieving improved runtimes even at finer grid resolutions (Figure 2b).

These results underscore the importance of solver selection for optimizing performance. The GPU-accelerated CG solver proves advantageous for turbulence simulations involving symmetric, well-conditioned matrices due to its scalability and efficiency. GMRES remains versatile for non-symmetric or ill-conditioned matrices but would benefit significantly from GPU acceleration and preconditioning to reduce computational overhead.



(a) Different CPU vs GPU GMRES solvers.

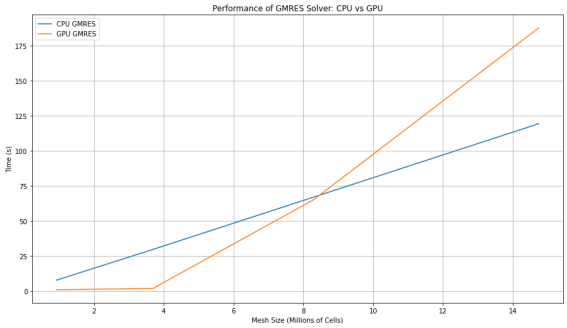


(b) Different CPU vs GPU CG solvers.

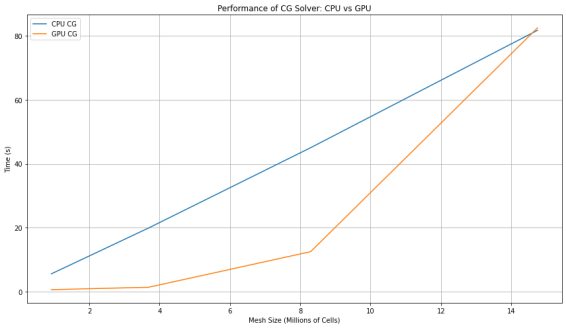
Figure 2: Comparative analysis of solver performance for various mesh resolutions.

3.2 Scalability Analysis: From 960×960 to 3840×3840 Mesh Resolutions

The scalability of GPU acceleration in the `pyCALC-RANS` code was evaluated for mesh resolutions ranging from 960×960 to 3840×3840, as depicted in Figures 3a and 3b. At smaller resolutions (960×960), GPU solvers, particularly CG, significantly outperform their CPU counterparts, benefiting from efficient parallelism and memory bandwidth. However, as mesh sizes increase, GPU performance begins to degrade due to memory latency and bandwidth saturation, especially for GMRES, as shown in Figure 3a. This is likely due to GMRES requiring larger Krylov subspaces, leading to increased memory overhead and reduced computational efficiency on GPUs at extreme resolutions. For the largest resolution (3840×3840), CPU solvers exhibit competitive or superior performance, with CG maintaining scalability due to better cache utilization and sparse matrix handling (Figure 3b). As shown in Table 1, GPU acceleration provided a speed-up of up to $14.84\times$ for GMRES and $14.36\times$ for CG at intermediate mesh sizes. However, at the largest mesh size (3840×3840), speed-up dropped to $0.64\times$ for GMRES and $0.99\times$ for CG, indicating limited GPU advantages at extreme resolutions. These results highlight the need for solver-specific GPU optimizations and adaptive load balancing strategies to ensure efficient scalability across varying mesh resolutions.



(a) GMRES Solver



(b) CG Solver

Figure 3: Performance comparison of CPU vs GPU across mesh resolutions.

Mesh Size (Millions of Cells)	Speed-up GMRES	Speed-up CG
0.9216	6.78	8.77
3.6864	14.84	14.36
8.2944	1.03	3.60
14.7456	0.64	0.99

Table 1: GPU speed-up over CPU for GMRES and CG across mesh sizes.

4 Conclusion

This study explored the implementation of GPU-accelerated computational methods for the `pyCALC-RANS` CFD solver using Python and CUDA, with a focus on improving computational efficiency in solving large-scale turbulence modeling problems. Profiling the code identified key computational bottlenecks, particularly within the `solve_2d`, `coeff`, and `conv` functions, which were targeted for optimization.

For the purpose of benchmarking GPU performance, the turbulence code was implemented using the Conjugate Gradient (CG) solver. Although the turbulence equations for the velocity components and turbulent quantities (u, v, k, ω) are inherently non-symmetric—and therefore not ideally suited for the CG method-the decision to use CG was driven by its simplicity, lower memory requirements, and potential for significant computational speedup. The GPU-accelerated CG solver provided notable acceleration for cases where the matrices behaved in a relatively well-conditioned manner, despite its sensitivity to eigenvalue clustering which often necessitated higher iteration limits.

Comparative assessments with GMRES indicated that while GMRES is better equipped to handle non-symmetric systems, it suffered from higher computational overhead due to the storage and manipulation of large Krylov subspaces. Despite its theoretical limitations for non-symmetric problems, the use of the CG solver in this study enabled a practical evaluation of GPU performance improvements over CPU-based implementations, particularly for moderate grid sizes. For very large grids (e.g., 3840×3840), memory bandwidth limitations and cache inefficiencies began to impact performance, highlighting the need for further optimization strategies.

In summary, while applying the CG solver to a non-ideal, non-symmetric turbulence problem presents certain challenges, its implementation in this study provided valuable insights into the benefits of GPU acceleration in CFD simulations. The results underscore the potential of parallel computing to significantly reduce computation time, thereby making large-scale turbulence simulations more feasible and efficient, and lay the groundwork for future research into solvers more tailored to non-symmetric systems.

5 Future Work

To further enhance the performance of GPU-accelerated CFD solvers, several key areas should be explored. Multi-GPU parallelization could help address memory bandwidth constraints and improve scalability for ultra-large grids, while a hybrid CPU-GPU computing strategy could optimize computational efficiency by distributing workloads based on hardware strengths. Additionally, preconditioning techniques should be investigated to improve the convergence behavior of the CG solver, reducing iteration counts and mitigating the effects of poor eigenvalue clustering. Exploring alternative sparse linear solvers specifically optimized for GPU architectures, including deep learning-based iterative solvers, could further enhance computational efficiency. By addressing these areas, the GPU-accelerated `pyCALC-RANS` solver can be further optimized for handling high-resolution CFD simulations more efficiently, enabling faster and more accurate predictions in turbulence modeling and other complex engineering applications.

References

[1] Nickolls, John and Dally, William J. *The GPU computing era*. IEEE Micro, vol. 30, no. 2, 2010, pp. 56–69.

[2] Owens, John D., Houston, Mike, Luebke, David, Green, Simon, Stone, John E., and Phillips, James C. *GPU computing*. Proceedings of the IEEE, vol. 96, no. 5, 2008, pp. 879–899.

[3] Borkar, Shekhar. *Thousand core chips: a technology perspective*. Proceedings of the 44th Annual Design Automation Conference, 2007, pp. 746–749.

[4] Davidson, Lars. *pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow*. 2023. URL: `https://www.tfd.chalmers.se/~lada/postscript_files/py-calc-rans.pdf`.

[5] Hasselwander, Erik, Cheng, Yuhua, and Gavras, Kyriakos. *GPU-accelerated computational methods using Python and CUDA*. 2025. URL: `https://www.tfd.chalmers.se/~lada/TRA220-erik-yuhua-kyriakos.pdf`.

A Appendix

A.1 Profiling Code

```
######################### Start of global iteration process ##############################

# Initialize the profiler
profiler = cProfile.Profile()
profiler.enable()

for iter in range(0, maxit):

      start_time_iter = time.time()
      # coefficients for velocities
      start_time = time.time()
      # compute inlet fluctuations
      if iter == 0:
         u_bc_west, v_bc_west, k_bc_west, om_bc_west, u2d_face_w, convw = modify_inlet()
      aw2d, ae2d, as2d, an2d, su2d, sp2d = coeff(convw, convs, vis2d, 1, scheme)

      # u2d
      # boundary conditions for u2d
      su2d, sp2d = bc(su2d, sp2d, u_bc_west, u_bc_east, u_bc_south, u_bc_north,
                      u_bc_west_type, u_bc_east_type, u_bc_south_type, u_bc_north_type)
      su2d, sp2d, ap2d = calcu(su2d, sp2d, p2d_face_w, p2d_face_s)

      u2d, residual_u = solve_2d(u2d, aw2d, ae2d, as2d, an2d, su2d, ap2d, convergence_limit_u, nsweep_vel, solver_vel)
      print(f"{'time u: '}{time.time()-start_time:.2e}")

      start_time = time.time()
      # v2d
      # boundary conditions for v2d
      su2d, sp2d = bc(su2d, sp2d, v_bc_west, v_bc_east, v_bc_south, v_bc_north,
                      v_bc_west_type, v_bc_east_type, v_bc_south_type, v_bc_north_type)
      su2d, sp2d, ap2d, ap2d_vel = calcv(su2d, sp2d, p2d_face_w, p2d_face_s)
      v2d, residual_v = solve_2d(v2d, aw2d, ae2d, as2d, an2d, su2d, ap2d, convergence_limit_v, nsweep_vel, solver_vel)
      print(f"{'time v: '}{time.time()-start_time:.2e}")

      start_time = time.time()
      # pp2d
```

```
        convw, convs = conv(u2d, v2d, p2d_face_w, p2d_face_s)
        convw = modify_outlet(convw)
        aw2d, ae2d, as2d, an2d, su2d, ap2d = calcp(pp2d, ap2d_vel)
        pp2d = np.zeros((ni, nj))
        pp2d, dummy = solve_2d(pp2d, aw2d, ae2d, as2d, an2d, su2d, ap2d,
        ↪  convergence_limit_pp, nsweep_pp, solver_pp)

        # correct u, v, w, p
        convw, convs, p2d, u2d, v2d, su2d = correct_u_v_p(u2d, v2d, p2d)
        convw = modify_outlet(convw)

        # continuity error
        su2d = convw[0:-1, :] - np.roll(convw[0:-1, :], -1, axis=0) + convs[:, 0:-1]
        ↪  - np.roll(convs[:, 0:-1], -1, axis=1)
        residual_pp = abs(np.sum(su2d))

        print(f"{'time pp: '}{time.time()-start_time:.2e}")

        u2d_face_w, u2d_face_s = compute_face_phi(u2d, u_bc_west, u_bc_east,
        ↪  u_bc_south, u_bc_north,
                                                  u_bc_west_type, u_bc_east_type,
                                                  ↪  u_bc_south_type,
                                                  ↪  u_bc_north_type)
        v2d_face_w, v2d_face_s = compute_face_phi(v2d, v_bc_west, v_bc_east,
        ↪  v_bc_south, v_bc_north,
                                                  v_bc_west_type, v_bc_east_type,
                                                  ↪  v_bc_south_type,
                                                  ↪  v_bc_north_type)
        p2d_face_w, p2d_face_s = compute_face_phi(p2d, p_bc_west, p_bc_east,
        ↪  p_bc_south, p_bc_north,
                                                  p_bc_west_type, p_bc_east_type,
                                                  ↪  p_bc_south_type,
                                                  ↪  p_bc_north_type)

        start_time = time.time()

        if kom:

            vis2d = vist_kom(vis2d, k2d, om2d)
            # coefficients
            start_time = time.time()
            aw2d, ae2d, as2d, an2d, su2d, sp2d = coeff(convw, convs, vis2d, prand_k,
            ↪  scheme_turb)
            # k
            # boundary conditions for k2d
            su2d, sp2d = bc(su2d, sp2d, k_bc_west, k_bc_east, k_bc_south, k_bc_north,
                            k_bc_west_type, k_bc_east_type, k_bc_south_type,
                            ↪  k_bc_north_type)
```

```
72          su2d, sp2d, gen, ap2d = calck(su2d, sp2d, k2d, om2d, vis2d, u2d_face_w,
            ↪  u2d_face_s, v2d_face_w, v2d_face_s)
73
74          k2d, residual_k = solve_2d(k2d, aw2d, ae2d, as2d, an2d, su2d, ap2d,
            ↪  convergence_limit_k, nsweep_kom, solver_turb)
75          k2d = np.maximum(k2d, 1e-10)
76          print(f"{'time k: '}{time.time()-start_time:.2e}")
77
78          start_time = time.time()
79          # omega
80          # boundary conditions for om2d
81          aw2d, ae2d, as2d, an2d, su2d, sp2d = coeff(convw, convs, vis2d,
            ↪  prand_omega, scheme_turb)
82          su2d, sp2d = bc(su2d, sp2d, om_bc_west, om_bc_east, om_bc_south,
            ↪  om_bc_north,
83                          om_bc_west_type, om_bc_east_type, om_bc_south_type,
                            ↪  om_bc_north_type)
84          su2d, sp2d, ap2d = calcom(su2d, sp2d, om2d, gen)
85
86          aw2d, ae2d, as2d, an2d, ap2d, su2d, sp2d = fix_omega()
87
88          om2d, residual_om = solve_2d(om2d, aw2d, ae2d, as2d, an2d, su2d, ap2d,
            ↪  convergence_limit_om, nsweep_kom, solver_turb)
89          om2d = np.maximum(om2d, 1e-10)
90
91          print(f"{'time omega: '}{time.time()-start_time:.2e}")
92
93       # scale residuals
94       residual_u = residual_u / resnorm_vel
95       residual_v = residual_v / resnorm_vel
96       residual_p = residual_p / resnorm_p
97       residual_k = residual_k / resnorm_vel**2
98       residual_om = residual_om / resnorm_vel
99
100      resmax = np.max([residual_u, residual_v, residual_p])
101
102      print(f"\n{'--iter:'}{iter:d}, {'max residul:'}{resmax:.2e},
         ↪  {'u:'}{residual_u:.2e}\
103   , {'v:'}{residual_v:.2e}, {'pp:'}{residual_pp:.2e}, {'k:'}{residual_k:.2e}\
104   , {'om:'}{residual_om:.2e}\n")
105
106      if resmax < sormax:
107         break
108
109   ######################### End of global iteration process
      ↪  ############################
110
111   # Disable the profiler
```

```
profiler.disable()

# Print profiling results
ps = pstats.Stats(profiler).sort_stats('cumtime')

# Filter the results for specific functions
filtered_functions = ["solve_2d", "conv", "coeff"]  # Add other function names if
    ↪ needed
for func in filtered_functions:
    ps.print_stats(func)
```