Chalmers University of Technology

GPU-accelerated Computational Methods using Python and CUDA

GPU-Accelerated RANS: Scalable Python Implementation with CUDA-Aware MPI

Author:
Ahmed Watioui
Ali Shirzad
Henry Hudoyo
Yichang Chang

Supervisor:
Lars Davidson

February 13, 2026

Abstract

This project presents the development of, as well as comparisons for, a CuPy and CUDA implementation of a two-dimensional RANS solver using Computational Fluid Dynamics for simulation on Graphics Processing Units (GPU) with a Multi-GPU Domain Decomposition Method enabled by CUDA-aware MPI to bypass memory size restrictions on single Graphics Processing Units.

Performance results for the VERA computer cluster with NVIDIA A100 and H100 are concentrated on the scalability for preconditioned sparse solvers applied to the pressure correction equation. With the involved symmetric linear system, the preconditioned Conjugate Gradient algorithm is superior to the preconditioned GMRES algorithm for the solution process in terms of speed and computational cost. The interfacing with the NVIDIA AMGX library as a fully resident algebraic multigrid solver is an important enhancement. No data transfer from the CPU to the NVIDIA GPUs is involved.

For very high resolutions (12000×12000), scale analysis indicates that a high arithmetic intensity promotes near-ideal scalability on the H100 Multi-GPUs, even when synchronization overheads are considered for intermediate scale runs. A minor point of efficiency will occur at three GPUs because of the one-dimensional domain decomposition. In summary, the results demonstrate that Python-based GPU acceleration is well-suited for large-scale CFD when solver choices are carefully matched to hardware characteristics.

Contents

1 Introduction **1**
- 1.1 CPU and GPU Computing . . . 1
- 1.2 Why GPU? . . . 1
- 1.3 GPU Computing on VERA . . . 2
- 1.4 Project Objectives . . . 3

2 Methodology **4**
- 2.1 Grid Generation and Wall Treatment . . . 4
- 2.2 Project Variation . . . 4
- 2.3 Sparse Solver Configuration . . . 5
- 2.4 GPU Implementation . . . 5
- 2.5 Multiple GPU Implementation with MPI . . . 6
  - 2.5.1 Multi-GPU Code Implementation Details . . . 8
- 2.6 AMGX as Different Alternative for Sparse Matrix Solver . . . 9
  - 2.6.1 The Algebraic Multigrid Method . . . 9
  - 2.6.2 PyAMGX: Python Interface . . . 9
  - 2.6.3 Implementation Architecture . . . 9

3 Results and Discussion **11**
- 3.1 Profiling on CPU and GPU . . . 11
- 3.2 CPU vs. GPU Performance Comparison . . . 11
- 3.3 Impact of AMGX on GPU Pressure Solves . . . 13
- 3.4 GMRES and CG with their preconditioners (GMRES-PRE and CG-PRE) 13
- 3.5 GPU vs. Multi-GPU Performance Comparison . . . 14
- 3.6 Extremely Large Scale Multi-GPU Performance . . . 16

4 Conclusions **19**

5 Future Work **19**

References **21**

Appendix A Grid Generation Methodology **22**

Appendix B Sparse Iterative Solvers **23**
- B.1 Generalized Minimal Residual (GMRES) . . . 23
- B.2 Conjugate Gradient (CG) . . . 23
- B.3 Jacobi Preconditioning and Large-Scale Strategy . . . 23
- B.4 Computational Fluid Dynamics . . . 24

Appendix C AMGX **25**
- C.1 Figures . . . 25
- C.2 PyAMGX: Python Interface . . . 25
- C.3 AMGX implementation . . . 26
  - C.3.1 Mode String Format . . . 26
  - C.3.2 Modes Used in This Implementation . . . 26
  - C.3.3 Implementation with Automatic Fallback . . . 27

C.3.4 Data Type Requirements 27

1 Introduction

Modern engineering simulations require high spatial resolution, detailed physical models, and short turnaround times. As mesh sizes grow and additional physics are introduced, the computational cost increases rapidly. Classical Central Processing Units (CPUs) based solvers often struggle to handle these demands efficiently. This motivates the use of hardware architectures that can process very large amounts of data in parallel. Graphics Processing Units (GPUs) offer this capability and have become an important tool in accelerating scientific computing. Based on Prof. Lars Davidson's pyCALC-RANS[1] this study evaluates Multi-GPU acceleration using $Re_\tau = 5200$ channel flow as a benchmark we follow the single-GPU framework established by a previous group [2]. We aim to quantify the speedup and scalability of multi-device execution relative to single-GPU and CPU-based baselines.

1.1 CPU and GPU Computing

CPUs are designed for efficient execution of sequential instructions. They contain a small number of powerful cores that excel at tasks involving branching, decision making, and complex control logic. This structure is not ideal for workloads where the same numerical operations must be applied repeatedly across large arrays.

GPUs contain thousands of lightweight processing cores arranged for high throughput parallel execution. They are well suited for operations that can be decomposed into many independent tasks, such as stencil evaluations, vector updates, and sparse matrix multiplications. These patterns appear naturally in Computational Fluid Dynamics, which makes GPUs an efficient platform for accelerating large scale flow simulations.

1.2 Why GPU?

Computational Fluid Dynamics (CFD) simulates fluid flow by discretizing the governing equations of motion—namely the Navier–Stokes equations—onto a grid of control volumes. This process transforms the differential equations into a large, sparse algebraic system of the form $\mathbf{Ax} = \mathbf{b}$. Solving these systems, which can involve millions of unknowns for high-resolution grids, is the dominant computational cost in any CFD simulation.
The core of the solution process lies in iterative algorithms, such as the SIMPLEC method, which repeatedly perform a sequence of computationally intensive operations. Each iteration involves:

1. Coefficient Assembly (`coeff`): Assembling the sparse matrix $\mathbf{A}$ based on local cell-to-cell interactions.

2. Linear System Solving (`solve_2d`): Iteratively solving the $\mathbf{Ax} = \mathbf{b}$ system using methods like GMRES or CG.

3. Flux/Convection Calculation (`conv`): Evaluating terms that represent the transport of quantities across cell faces.

Crucially, these operations exhibit two characteristics that make them ideal for GPU acceleration:

1. Massive Data Parallelism: The same set of calculations (e.g., stencil operations in `coeff` or matrix-vector products in `solve_2d`) are performed independently on millions of grid cells simultaneously.

2. High Arithmetic Intensity: The core linear solve involves a vast number of floating-point operations relative to the amount of data transferred.

These patterns align perfectly with the GPU's architecture, which features thousands of lightweight cores designed for high-throughput parallel execution. By offloading these computational bottlenecks to the GPU, a dramatic reduction in simulation time can be achieved, particularly for the large-scale problems explored in this project. A detailed formulation of the governing equations and numerical methods is provided in Appendix B for reference.

1.3 GPU Computing on VERA

The VERA cluster offers several NVIDIA GPUs, notably the A40, A100, and H100. For CFD simulations, robust Double-Precision (FP64) performance is critical for numerical accuracy and stability. This requirement makes the A40, with its limited FP64 capabilities, unsuitable for this task.

The choice therefore narrows to the A100 and H100, both designed for high-performance computing. Among these, the H100 (Hopper architecture) is the optimal choice. It delivers substantially higher FP64 throughput and memory bandwidth than the A100, enabling dramatically reduced simulation times and superior efficiency for the large-scale models in this project, as illustrated in Figure 1. To empirically validate the superiority of the H100, our future experiments will include performance benchmarks on both the A100 and H100 hardware.

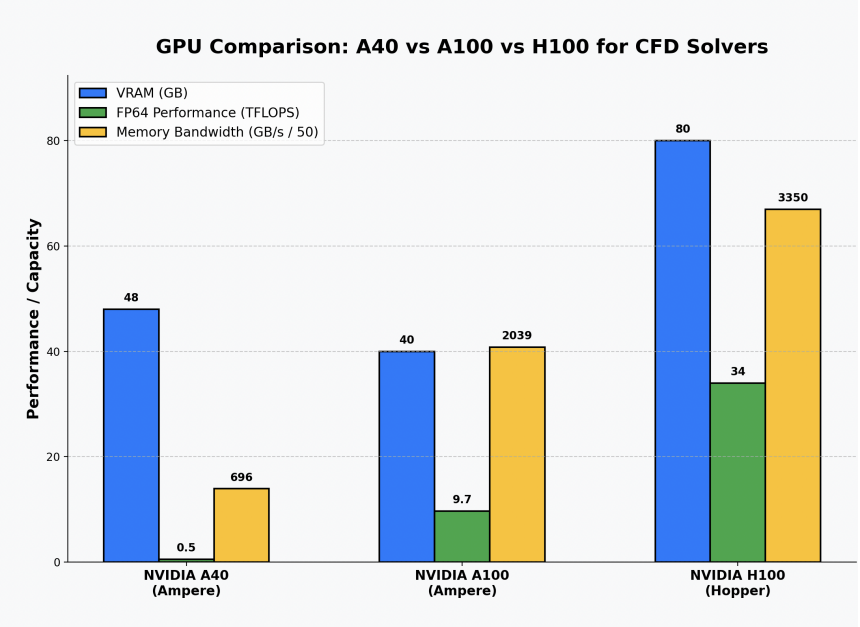


Figure 1: GPU Comparison.

To utilize these devices, VERA provides a software environment built around CUDA and GPU accelerated Python libraries. CuPy mirrors the NumPy interface but executes operations directly on the GPU, enabling vectorized routines to run efficiently without major code modifications. Additional libraries such as cuSPARSE, cuBLAS, and the sparse modules in CuPy provide fast implementations of matrix operations and iterative solvers. By combining these tools with the GPUs available on VERA, it becomes possible to investigate how different hardware generations affect execution time, memory usage, and overall solver performance.

1.4 Project Objectives

There are four main objectives that will be fulfilled in this project.

1. The 2D channel flow - RANS ($Re = 5200$) will be executed on the CPU and GPU.
2. The performance between CPU and GPU will be compared.
3. The code will be modified to work in multiple GPUs using MPI communication.
4. Different alternatives for the sparse matrix solver will be explored and compared with the current one.

2 Methodology

2.1 Grid Generation and Wall Treatment

The computational domain is a 2D channel with walls at $y = 0$ and $y = H = 2$. To satisfy the low-Reynolds number requirement ($y^+ < 1$) for the k-ω SST model, we employed a symmetric hyperbolic tangent stretching scheme.

The vertical coordinate y_j for the j-th grid point is given by:

$$y_j = \frac{H}{2}\left[1 + \frac{\tanh\left(\gamma\left(\frac{2j}{N_y} - 1\right)\right)}{\tanh(\gamma)}\right] \quad \text{for} \quad j = 0, 1, \ldots, N_y \tag{1}$$

where:

- H is the channel height ($H = 2.0$).
- N_y is the total number of grid intervals in the wall-normal direction.
- γ is the stretching parameter
- The term $\left(\frac{2j}{N_y} - 1\right)$ maps the index range to $[-1, 1]$, ensuring symmetry around the channel centerline.

To maintain $y^+ < 1$ without inducing numerical stiffness on finer meshes, γ was adapted based on the project scale categories defined in Table 2. Table 1 summarizes the selected parameters.

Table 1: Selection of stretching parameter γ based on grid scale.

Scale Category	Gamma (γ)	Rationale
Small scales	3.25	Aggressive clustering to ensure $y^+ < 1$ on coarse grids.
Large & Ext. Large	1.25	Relaxed stretching to prevent numerical stiffness.

Detailed mathematical formulations and sensitivity analyses are provided in Appendix A.

2.2 Project Variation

The performance of CPU, GPU, and multiple GPU configurations is evaluated across various grid sizes. To ensure numerical accuracy and a fair comparison across all hardware architectures, all simulations are performed using double-precision floating-point format (float64). The grid sizes are divided into three categories presented in Table 2 below.

Table 2: Project variation and hardware configuration

Category	Grid sizes	Cases	GPU
Small scales	$144^2, 300^2, 600^2, 960^2$	CPU / GPU	H100
Large scales	$2000^2, 4000^2, 6000^2, 8000^2, 10000^2$	Single / Multi	H100/A100
Extremely large	$10000^2, 11000^2, 12000^2$	Multi-GPU (2-4)	H100

2.3 Sparse Solver Configuration

The numerical solution involves three distinct linear systems, each with unique mathematical properties. We categorize the solver configurations as follows:

Table 3: Solver configurations and matrix properties.

Parameter	Equation	Matrix Properties
`SOLVER_VEL`	Momentum (u, v)	Non-symmetric (convection-dominated)
`SOLVER_PP`	Pressure correction	Symmetric, positive semidefinite
`SOLVER_TURB`	Turbulence (k, ω)	Non-symmetric, stiff source terms

The momentum and turbulence solvers are fixed to robust algorithms (typically GMRES) because their systems are non-symmetric and prone to instability.

In this study, the benchmarking variation focuses exclusively on the Pressure Correction solver (`SOLVER_PP`). Because this system results in a Symmetric Positive Definite (SPD) matrix, it allows us to test a wide range of algorithms, comparing specialized, highly efficient methods (such as Conjugate Gradient and AMG) against general purpose solvers. Consequently, only `SOLVER_PP` is varied in the performance results. Detailed mathematical formulations and preconditioner strategies are provided in Appendix B.

2.4 GPU Implementation

The GPU implementation, contained within the script `exec-pyCALC-RANS_GPU.py`, involves converting critical functions (`solve_2D`, `coeff`, and `conv`) from NumPy to CuPy. The overarching strategy is strict data locality: arrays are initialized on the CPU and transferred to the GPU once. They remain resident on the device throughout the entire iterative time-stepping process, eliminating expensive Host-to-Device transfers. Data is transferred back to the CPU only for final file saving or monitoring.

- Setup and Initialization: Imports are switched to `cupy` (as `cp`). Initial fields are allocated directly in VRAM using `cp.zeros`. This ensures that all subsequent operations occur within high-bandwidth GPU memory.
- Sparse Matrix Solver (`solve_2D`):
    1. Matrix Construction: `cupyx.scipy.sparse.diags` builds matrices directly on the GPU, avoiding CPU generation.
    2. Flattening: `cp.ravel()` replaces `np.flatten()` for optimized contiguous memory access.
    3. Solvers: CuPy wrappers (e.g., `cupyx.scipy.sparse.linalg.gmres`) interface directly with NVIDIA cuSOLVER.
    4. Synchronization: `cp.cuda.Stream.null.synchronize()` is enforced immediately after residual calculation to ensure the CPU receives correct values for convergence checks without stalling the pipeline unnecessarily.
- Coefficient Computation (`coeff`):
    1. Stencils: Stencil operations are vectorized using `cp.roll`, removing the need for explicit loops.

2. Math Operations: Standard math functions are replaced with CuPy equivalents (e.g., `cp.maximum`) to trigger automatic CUDA kernel compilation.

- Convection Term (`conv`):

1. Boundary Padding: Since `np.insert` is unsupported in CuPy, boundary padding is implemented using `cp.concatenate` with array slicing.
2. Interpolation: Face interpolation logic utilizes `cp.roll`, ensuring all index shifting remains on the device.

- Memory Management: `mempool.free_all_blocks()` is called explicitly after large matrix operations to prevent out-of-memory errors during large grid sweeps.

Testing Procedure on VERA: The code validation on the VERA cluster follows a specific workflow to ensure correct environment configuration and hardware utilization. This can be performed either in interactive mode, where a GPU node is borrowed for development and debugging, or via batch submission for long-running benchmarks.

- Environment Setup: Upon logging in via SSH, the environment is prepared by cleaning loaded modules and loading the specific GPU-accelerated libraries. This typically involves running `module purge` followed by `module load CuPy/13.6.0` and activating the dedicated virtual environment containing the requisite Python dependencies.
- Interactive Mode (`salloc`): For direct testing, an interactive session is requested using `salloc` to allocate a compute node with specific hardware (e.g., A100 or H100) for a set duration. Once the node is granted, the script is executed locally on that node.
- Batch Mode (`sbatch`): For production runs, a Slurm script is submitted via `sbatch`. This script automates the module loading and environment activation steps before executing the solver code.

2.5 Multiple GPU Implementation with MPI

In multiple GPU operation, the whole domain must be discretized according to the number of GPUs available. For this project, the domain is decomposed only in the i-direction (1D streamwise splitting) to simplify the implementation.

In order for several GPUs to work in parallel, they need to communicate and exchange information with their neighboring grids. The Message Passing Interface (MPI) is the standard protocol enabled here. Specifically, we utilize CUDA-Aware MPI, which allows CuPy arrays to be passed directly to MPI calls. This enables data to flow directly between GPU memories (via NVLink or PCIe) without the latency of staging data through the host CPU RAM.

Figure 2 represents how neighboring grids exchange information between their boundaries. In each GPU, a ghost cell is introduced as an extra layer of cells that mirrors the boundary conditions of the neighboring GPU. By combining ghost cells and MPI communication, the GPUs are expected to send data simultaneously to one neighbor and receive data from another. However, this communication must be avoided in the left (BC West) and right

(BC East) boundary conditions, since the value of the real physical boundary condition must be kept as it is.

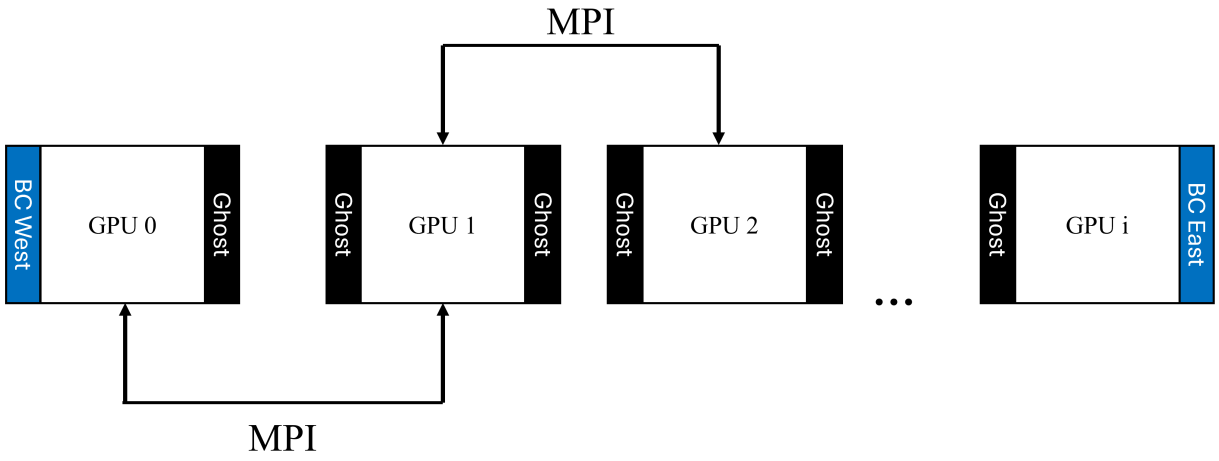


Figure 2: MPI communication scheme

In general, there are two types of iteration occurring in the code: inner (NSWEEP) and outer iteration (MAXIT). While the former iterates to solve each linearized system independently, the outer iteration couples these systems.

The coupling between subdomains follows an Additive Schwarz strategy. Since the coefficient matrix A is local to each GPU, the influence of the neighbor is treated explicitly. For a cell adjacent to a ghost boundary, the coefficient a_{nb} corresponding to the neighbor is set to zero in the matrix to decouple the systems. Its contribution is instead moved to the source term S_u, modifying the linear system $A\phi = S_u$ as:

$$S_u^{\text{new}} = S_u^{\text{old}} + a_{nb} \cdot \phi_{\text{ghost}} \tag{2}$$

where a_{nb} is the link coefficient to the neighbor and ϕ_{ghost} is the value received from the MPI exchange. This effectively enforces a Dirichlet boundary condition using the neighbor's data.

To ensure stability, we implement a **Dual Under-Relaxation Strategy**. First, standard physics under-relaxation is applied locally to stabilize the non-linear SIMPLE algorithm. Second, a specific *Schwarz Exchange Relaxation* (controlled by `URF_SCHWARZ`) is applied to the ghost cell values themselves. When a GPU receives new data from a neighbor, it mixes it with the previous iteration's value: $\phi_{ghost} = (1 - \alpha)\phi_{old} + \alpha\phi_{new}$. This dampening prevents numerical "ping-pong" oscillations between subdomains, which can occur even if the local physics are stable.

In the current configuration, the exchange frequency (N_SCHWARZ_EXCHANGE) is set to **1**. This indicates that a ghost cell exchange occurs exactly once per inner solver call. This is confirmed by the simulation logs, which consistently show exactly **350 total exchange calls**. With 50 outer iterations (MAXIT) performed, the calculation is consistent: 50 outer iterations $\times$ 7 exchanged variables = 350 total exchanges. Ideally, the exchange

ensures consistency across boundaries for the following 7 variables, which are packed into a single batched MPI message to minimize latency:

- **Primary Solved Variables:** u (Velocity X), v (Velocity Y), p (Pressure correction), k (Kinetic Energy), ω (Specific Dissipation).
- **Derived Variables:** ν_t (Turbulent Viscosity) and a_P^{vel} (Velocity Diagonal Coefficient). The exchange of a_P^{vel} is critical for ensuring the Rhie-Chow pressure correction remains valid at the inter-GPU boundaries.

2.5.1 Multi-GPU Code Implementation Details

The implementation, contained in `exec-pyCALC-RANS_GPU_MULTIPLE.py`, extends the single-GPU logic by introducing MPI management. The key practical steps are:

- Initialization and Mapping: The script uses `mpi4py` to initialize the environment. A strict 1-to-1 mapping is enforced where the MPI Rank determines the GPU device ID (e.g., `cp.cuda.Device(rank).use()`). This ensures each process controls exactly one GPU.
- Data Decomposition: The global grid is sliced along the x-axis. Each GPU allocates only its specific subdomain (plus ghost buffers) in VRAM using `cp.zeros`, ensuring memory usage scales efficiently with the number of devices.
- CUDA-Aware Halo Exchange: Unlike the `cp.concatenate` padding used in single-GPU, the multi-GPU code uses `comm.Sendrecv` to exchange boundary rows. Crucially, the send and receive buffers are CuPy arrays. By enabling UCX flags (e.g., `UCX_TLS=sm,cuda_copy,cuda_ipc`), the MPI library detects the GPU pointers and routes data directly over NVLink or PCIe, bypassing the CPU entirely.
- Additive Schwarz Loop: Inside the solver function, a loop runs for N_SCHWARZ_EXCHANGE iterations (currently set to 1):
  1. Local Solve: The local matrix system is solved using `cupyx.scipy.sparse.linalg.gmres` (here example for gmres).
  2. Communication: Ghost cells are updated via the Halo Exchange.
  3. Stabilization: The received ghost values are under-relaxed using `URF_SCHWARZ` to dampen inter-GPU oscillations.
  4. Coupling: The stabilized ghost values are multiplied by the boundary coefficients and added to the source term (S_u) of the boundary cells.
- Global Reduction: For convergence checking, the local residuals are aggregated across all GPUs using `comm.Allreduce` with the `MPI.SUM` operator. Synchronization via `cp.cuda.Stream.null.synchronize()` is performed before this step to ensure all GPU computations are finalized before the reduction.

2.6 AMGX as Different Alternative for Sparse Matrix Solver

AMGX It is an open-source library developed by NVIDIA for solving large sparse linear systems given as $Ax = b$. Unlike traditional CPU architectures, modern GPUs with thousands of computing cores offer massive parallelism. This feature enabled AMGX, an algebraic multi-grid solver on GPUs, to significantly accelerate iterative methods traditionally employed within CFD simulations [4].

2.6.1 The Algebraic Multigrid Method

AMGX is based on the Algebraic Multigrid (AMG) method, which is one of the most efficient numerical techniques to solve large-scale systems of linear equations arising from the discretization of partial differential equations [6]. AMG is designed to overcome the limitations of traditional iterative methods that are based on local relations, such as the Gauss–Seidel and Jacobi methods. These classical smoothers are very effective at damping the high-frequency components of the error, but they converge slowly for the low-frequency components on fine grids. The AMG method addresses this limitation by combining such smoothers with a hierarchy of coarse spaces that are constructed algebraically from the system matrix, so that low-frequency error components on the fine grid become high-frequency on coarse grids and can be efficiently reduced.

The AMG approach achieves nearly linear computational convergence ($O(N)$) with respect to the number of unknowns on large CFD meshes, making it highly suitable for scalable GPU-accelerated simulations. In this work, AMGX is used as a GPU-accelerated sparse linear solver for the pressure correction system, where it is often the most computationally expensive component in RANS simulations. AMG can reduce the required number of iterations to a few tens, whereas basic iterative methods may require orders of magnitude more iterations, as illustrated later in Figure 4.

2.6.2 PyAMGX: Python Interface

While AMGX is implemented in C++/CUDA, the `pyamgx` library provides Python bindings that allow direct integration with Python-based CFD solvers.[5] In this work, `pyamgx` acts as a thin wrapper between the pyCALC-RANS logic and the AMGX core solver, exposing configuration, matrix, and vector objects while keeping all heavy linear algebra on the GPU. The overall zero-copy data flow between the Python layer, CuPy arrays on the device, and the AMGX-based solver architecture is illustrated in Appendix C , Figure C.2.

2.6.3 Implementation Architecture

Prior to incorporating AMGX, the GPU code employed a *hybrid mode* approach, wherein the computation of flow and turbulence simulations were done on GPUs with CuPy, and solutions involving the Pressure Poisson equation were done on CPUs with PyAMG, an algebraic multigrid solver available exclusively on CPUs and independent of GPUs. However, this implementation introduced a large bottleneck with regard to solution times because PyAMG would have no direct access to data on the GPUs.

Solutions involving pressure would have needed transfers of the sparse matrix A and the vector b from GPUs to CPU main memory, with subsequent CuPy arrays converted into

NumPy/SciPy compatible data types on CPUs, solution for $Ax = b$ on CPUs, and subsequent transfers of solution vector x back into GPUs (see Appendix C , Figure C.3a).

To eliminate the bottleneck associated with data transfer from the host to devices, the AMGX integration contains a pure GPU implementation with all linear algebra data residing within device memory for the solve.The architecture, illustrated in (Appendix C , Figure C.3b) is organized into three layers. Additional details regarding the coding implementation and solver configuration are provided in Appendix C.3.

1. **CuPy layer:** The sparse matrix A and the related vectors b and x are created and stored immediately on the GPU memory as CuPy arrays, without requiring an intermediate representation on CPU memory as a linear problem.

2. **Pointer exchange**: The raw pointers to GPU memory containing CuPy arrays are then exchanged with the `pyamgx` matrix and vector objects so that no extra data copy needs to be done as the operation will be carried out on device allocations.

3. **Core solver with AMGX**: Once configured with AMGX, the solver computes the solution completely on the GPU without CPU intervention. Subsequent utilization of x and other CuPy arrays within Python CFD simulations isn't necessary after convergence because they haven't been modified. At this stage, convergence doesn't occur because x initially contains an approximate solution x_1, indicating it doesn't start from scratch but uses an approximate.

3 Results and Discussion

3.1 Profiling on CPU and GPU

We begin by profiling the solver on the CPU to identify the dominant computational bottlenecks. As shown in Fig. 3, the sparse linear solve performed in `solve_2d` clearly dominates the total runtime. This makes it the primary target for GPU acceleration. In comparison, the cost of coefficient assembly (`coeff`) and convection term evaluation (`conv`) is relatively small, although these routines are also ported to the GPU to assess their impact on overall performance.

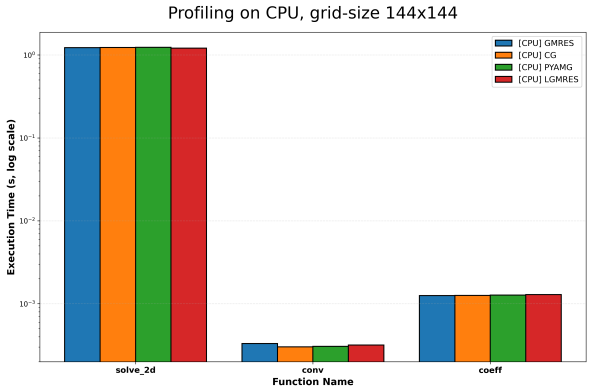


Figure 3: CPU profiling for a 144×144 *grid showing execution time (log scale) of* `solve_2d`*,* `conv`*, and* `coeff` *for different sparse solvers.*

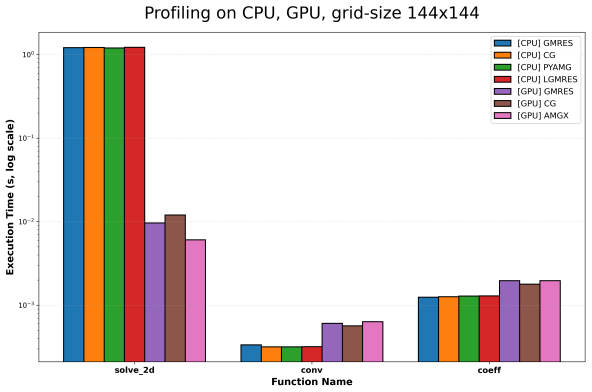


Figure 4: CPU–GPU profiling for a 144×144 *grid, highlighting the speedup of* `solve_2d` *when using GPU-accelerated solvers.*

Figure 4 compares CPU-only execution with a hybrid CPU–GPU configuration. Offloading the sparse solver to the GPU leads to a substantial reduction in runtime for `solve_2d`, resulting in a significant overall speedup. The execution time of `coeff` and `conv` increases slightly when executed on the GPU; however, their contribution to the total runtime remains negligible compared to the cost of the linear solve.

The modest increase in runtime for `coeff` and `conv` can be attributed to GPU kernel launch overhead and memory access latency. These routines involve relatively lightweight arithmetic and limited reuse of data, making them less favorable for GPU acceleration compared to the sparse linear solve, which exhibits high parallelism and arithmetic intensity.

3.2 CPU vs. GPU Performance Comparison

To assess the benefit of GPU acceleration, the sparse solvers are evaluated on a sequence of increasingly fine grids: 144×144, 300×300, 600×600, and 960×960, as listed in Table 2. In these tests, only the pressure-correction solver is varied, while all other components of the algorithm are kept unchanged. As a result, the different CPU solvers exhibit very similar performance, since the dominant computational workload outside the pressure solve is identical.

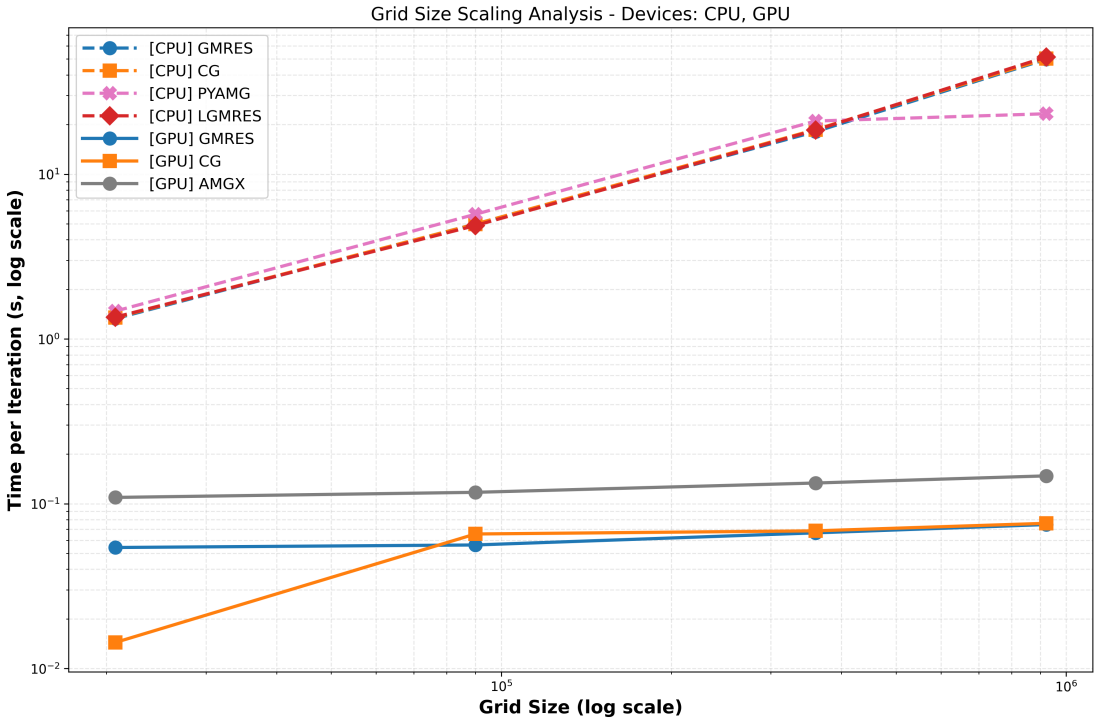


Figure 5: Grid size scaling analysis for CPU and GPU solvers. Iteration time (log scale) as a function of grid size (log scale) for GMRES, CG, LGMRES, and PyAMG on CPU, and GMRES, CG, and AMGx on GPU.

Figure 5 shows the iteration time as a function of grid size for both CPU and GPU configurations. The results demonstrate that the GPU-accelerated solvers outperform their CPU counterparts across all tested grid sizes, including the coarsest 144×144 mesh. While the speedup is evident even at this smallest scale, it becomes increasingly pronounced as the grid is refined.

It is important to acknowledge, however, that for grid sizes significantly smaller than those tested here, the CPU would theoretically outperform the GPU. This crossover occurs because GPUs incur fixed setup overheads, such as kernel launching, driver interactions, and PCIe latency that exist regardless of the problem size. On extremely small or lightweight problems, these fixed costs can dominate the total execution time, making the CPU's low-latency architecture more efficient. In our benchmarks, the 144×144 grid already provides sufficient arithmetic intensity to amortize these startup costs, allowing the GPU to maintain its performance advantage throughout the entire testing range.

For the grid sizes considered here, the simulations remain well within the memory capacity of the NVIDIA H100 GPU. Consequently, performance is not limited by memory capacity or bandwidth saturation, and the observed scaling primarily reflects increased arithmetic workload rather than memory constraints. This explains why the runtime does not degrade sharply with increasing grid size and highlights the suitability of GPUs for large-scale CFD simulations.

3.3 Impact of AMGX on GPU Pressure Solves

Figure 4 and Figure 6 summarize the performance impact of the AMGX integration for a 144×144 grid. The figure 6 compares the GPU implementation using PyAMG in hybrid mode with the pure-GPU AMGX integration, while the second figure places AMGX in the context of several CPU- and GPU-based solvers.

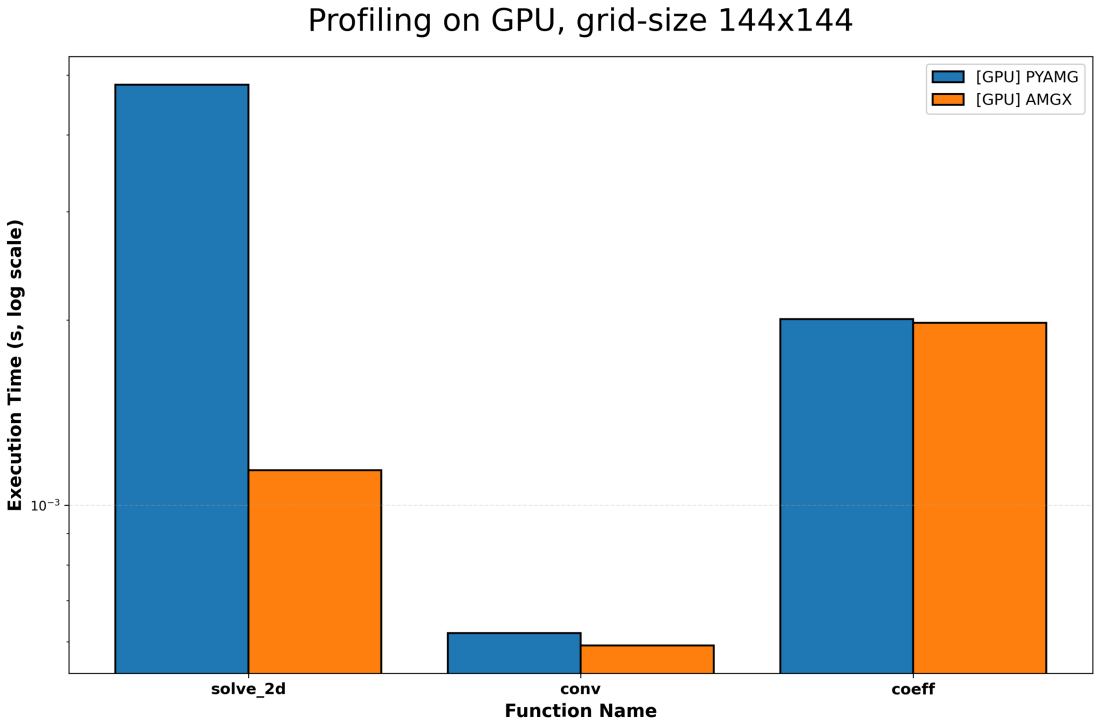


Figure 6: GPU profiling for a 144×144 grid comparing the hybrid PyAMG implementation with the pure-GPU AMGX integration. Execution times for the main kernels (`solve_2d`, `conv`, `coeff`) are shown on a logarithmic scale.

3.4 GMRES and CG with their preconditioners (GMRES-PRE and CG-PRE)

To assess the impact of preconditioning on GPU performance, we compare GMRES and CG with their Jacobi-preconditioned counterparts (GMRES-PRE and CG-PRE) over increasing grid sizes, performed on 300×300, 600×600, and 960×960 grids. The results are shown in Figure 7.

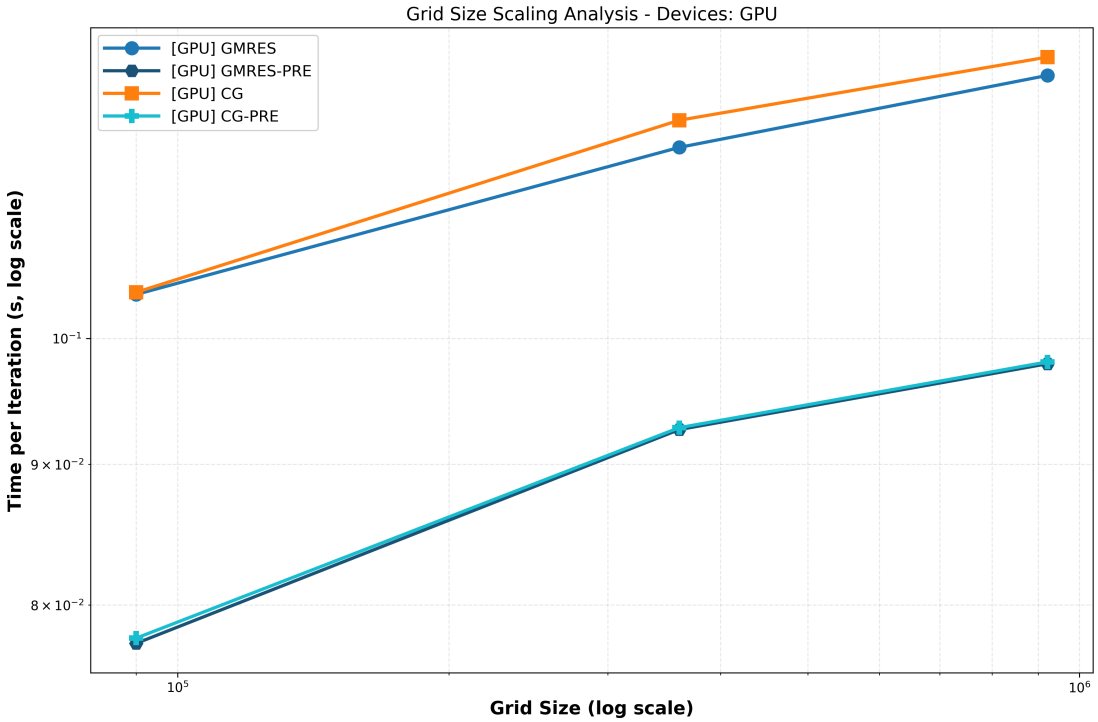


Figure 7: Grid-size scaling on GPU for GMRES and CG with and without Jacobi preconditioning. Time per iteration (log scale) is shown as a function of grid size (log scale).

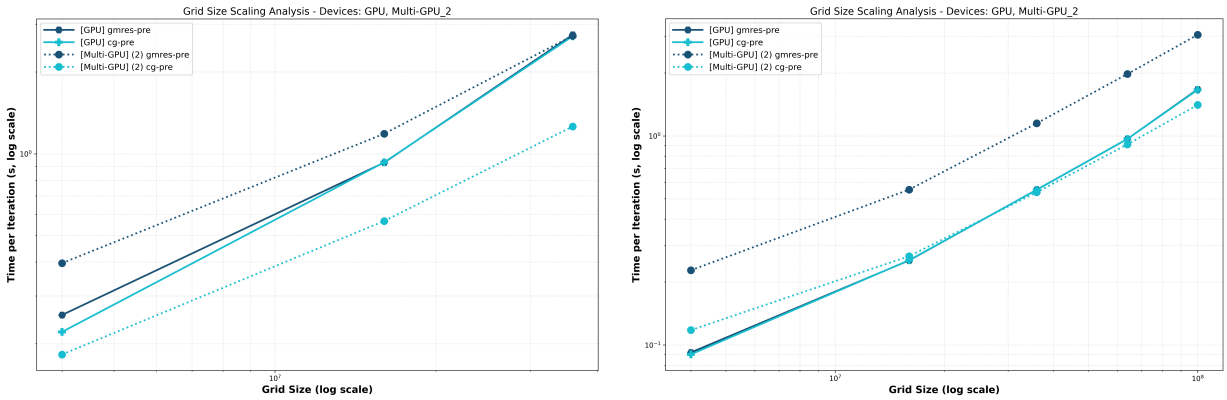
Figure 7 clearly demonstrates the effectiveness of preconditioning. For both GMRES and CG, the preconditioned variants exhibit a consistently lower time per iteration compared to their unpreconditioned counterparts, with the performance gap becoming more pronounced as the grid size increases. This indicates that preconditioning significantly improves the numerical properties of the linear systems, leading to fewer operations per effective iteration and better utilization of the Krylov subspace.

In particular, CG-PRE shows the strongest performance, benefiting from both the symmetry of the pressure-correction system and the improved conditioning introduced by the Jacobi preconditioner. The reduced iteration cost reflects the fact that the solver converges with smoother search directions and requires fewer corrective updates. From a hardware perspective, this means that less computational work is needed to achieve the same residual reduction, effectively lowering the time spent per iteration on the GPU.

Due to this clear and robust performance improvement, the preconditioned solvers are adopted as the default choice in subsequent experiments. In the Multi-GPU case, the same preconditioning strategy is retained, allowing a clean and fair comparison between single-GPU and Multi-GPU performance without introducing additional solver-side variability.

3.5 GPU vs. Multi-GPU Performance Comparison

This section evaluates the performance scaling from a single GPU to a two-GPU configuration across the "Large" and "Extremely large" grid categories defined in Table 2. The scaling behavior is presented for both A100 (40GB) and H100 (94GB NVL) hardware in Figure 8.



(a) A100: Single vs. Two GPUs

(b) H100: Single vs. Two GPUs

Figure 8: Performance scaling on A100 vs H100. On the A100 (a), the Multi-GPU configuration outperforms Single-GPU for CG-PRE across all tested grid sizes. On the H100 (b), the superior bandwidth allows the Single-GPU to remain dominant until the largest scales.

On the A100 (40GB), we observe a distinct performance divergence. For CG-PRE, the Multi-GPU configuration consistently outperforms the single-GPU setup. Notably, at the 6000×6000 resolution, we observe a speedup of 2.15x (see Table 5). This unusual efficiency gain indicates that the single A100 was heavily saturated by memory bandwidth requirements; distributing the problem across two GPUs doubled the available aggregate bandwidth, effectively uncapping the solver's performance.

Conversely, GMRES-PRE on the A100 shows negligible speedup (1.01x) despite low communication overhead. As shown in Table 4, the communication overhead decreases significantly as the grid size increases for both hardware architectures. However, a direct comparison reveals that the H100 consistently exhibits higher relative overhead percentages than the A100 (e.g., 1.2% vs 0.8% for CG-PRE). This reflects the massive computational throughput of the H100; because the solver computes the solution so rapidly, the fixed latency of MPI communication occupies a larger fraction of the total iteration time compared to the slower A100.

Table 4: Communication overhead trends across grid sizes on A100 vs. H100.

Solver	**A100 (40GB)**			**H100 (94GB)**		
	2000^2	4000^2	6000^2	2000^2	4000^2	6000^2
GMRES-PRE	1.1%	0.5%	0.3%	1.4%	0.8%	0.6%
CG-PRE	2.3%	1.0%	0.8%	2.5%	1.6%	1.2%

It is critical to interpret the low communication overhead values in Table 4 (e.g., 0.6% for GMRES-PRE) with caution. This metric primarily captures inter-GPU communication (ghost cell bandwidth and exchange synchronization). However, it completely excludes the intra-GPU synchronization latency (the frequent thread barriers during orthogonalization) which accounts for the majority of the GMRES execution time.

Unlike CG, where every inner iteration performs an identical set of operations (constant work), GMRES iterations become progressively more expensive as the orthogonalization subspace grows. Although a restart parameter (tuned to 20) is used to periodically reset

memory and compute costs, the solver still requires an increasing number of inner products within each cycle to maintain orthogonality. Each inner product triggers a local GPU reduction, forcing thread synchronization; over 50 iterations, this results in approximately 5× more reductions than CG. This heavy administrative overhead significantly prolongs the local solve on each GPU, delaying the global ghost cell exchange in the Additive Schwarz framework. On the ultra-fast H100 at 6000^2, the pure physics calculation is virtually instant, making this fixed orthogonalization latency the dominant bottleneck. Consequently, GMRES-PRE suffers a slowdown (0.48× speedup) because its high per-iteration overhead overwhelms the gains from parallelization at this scale.

Consequently, the H100 behavior differs fundamentally from the A100. At 6000^2, the single H100 is not yet memory-bound. As a result, CG-PRE shows only a marginal speedup (1.03x), while GMRES-PRE suffers the severe slowdown described above. It is only at the extremely large scale (10000×10000) that the computational workload becomes heavy enough to hide these latencies, allowing the Multi-GPU configuration to show clear benefits.

Table 5 summarizes the quantitative results at the 6000^2 resolution. The low overheads confirm that the poor scaling on the H100 is due to a lack of computational intensity relative to the hardware's capacity, rather than network bottlenecks.

Table 5: Speedup ($T_{\text{single}}/T_{\text{multi}}$) and Communication Overhead at 6000^2 resolution.

Solver	A100 (40GB)		H100 (94GB)	
	Speedup	Overhead	Speedup	Overhead
GMRES-PRE	1.01	0.3%	0.48	0.6%
CG-PRE	2.15	0.8%	1.03	1.2%

3.6 Extremely Large Scale Multi-GPU Performance

To evaluate performance at the limits of the hardware capabilities, we benchmarked the "Extremely Large" grid category (10000×10000, 11000×11000, and 12000×12000) using 2, 3, and 4 H100 GPUs. The scaling trends are presented in Figure 9. The results follow a similar trend to the single-vs-multi comparison: the benefit of increasing the number of GPUs becomes most pronounced at the largest grid sizes.

At 10000^2, the difference between 2, 3, and 4 GPUs is visible but relatively small. However, as the grid expands to 12000^2, the 3-GPU configuration appears to sit in a "sweet spot," maintaining slightly higher efficiency than the 4-GPU setup (see Table 7). This behavior can be attributed to the 1D domain decomposition topology. The 3-GPU setup contains only one "interior" domain (Rank 1) that performs double halo exchanges (Left+Right), whereas the 4-GPU setup contains two such domains (Ranks 1 and 2), effectively doubling the number of nodes under maximum communication load.

To quantify this, we analyzed the specific communication overheads for the CG-PRE solver, as detailed in Table 6. The data confirms that the 3-GPU configuration consistently exhibits the lowest relative overhead (e.g., 0.9% at 11000^2). In contrast, the 4-GPU configuration shows a slight regression, peaking at 1.5% overhead.

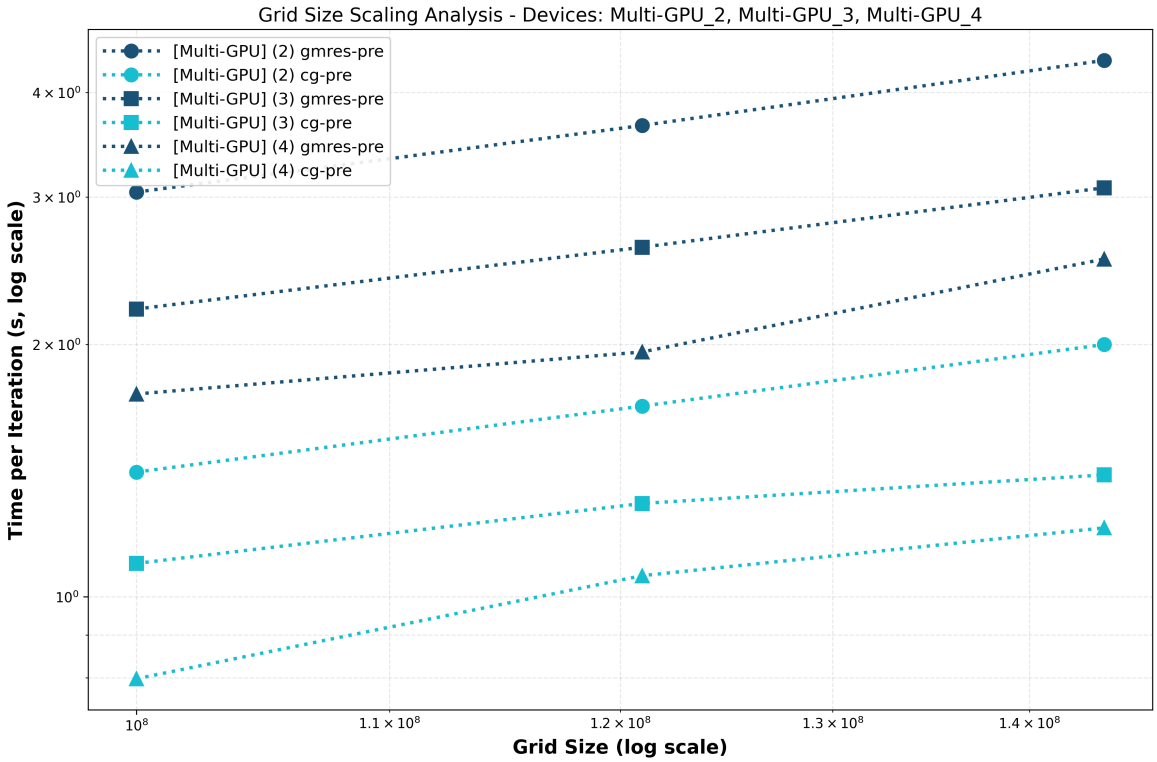


Figure 9: Performance scaling for extremely large grids on H100 (2, 3, and 4 GPUs). As the grid size increases, the gap between the configurations widens, indicating that the communication overhead of adding more GPUs is increasingly amortized by the computational load.

Table 6: Communication Overhead for CG-PRE on Extremely Large Grids.

Grid Size	2 GPUs	3 GPUs	4 GPUs
10000×10000	1.2%	1.0%	1.5%
11000×11000	1.3%	0.9%	1.4%
12000×12000	1.2%	1.1%	1.3%

Notably, unlike the A100 benchmarks where overhead dropped sharply from 2.3% to 0.8% as grids grew (Table 4), the H100 overheads in Table 6 remain relatively flat, plateauing around the 1% mark. This stability is consistent with our earlier observation at 6000^2, where the H100 also exhibited a 1.2% overhead. This confirms that the H100 operates in a different scaling regime compared to the A100. While the A100 transitions from a latency-bound state to a bandwidth-bound one as the grid grows ($2000^2 \rightarrow 6000^2$), the H100 enters the saturated "Asymptotic" regime much earlier. By 6000^2, the system has already hit a highly efficient floor where communication costs scale linearly with the grid boundaries. Consequently, further increasing the grid size to 12000^2 does not yield lower relative overheads, as the system is already operating at maximum efficiency with the communication cost stabilized at $\approx 1\%$.

Despite the 3-GPU setup showing the lowest relative overhead, the 2-GPU configuration is noticeably more constrained at these resolutions, likely operating closer to its memory bandwidth and capacity limits. While the 3-GPU setup currently offers the optimal balance, it is expected that challenging the system with even larger grids (beyond 12000^2) would eventually shift the advantage back to the 4-GPU configuration. At such hypo-

thetical scales, the 3-GPU setup would approach its own saturation points, whereas the 4-GPU system would continue to benefit from its larger aggregate memory and compute resources, effectively "out-scaling" the slight topological overhead observed here.

While CG-PRE remains the faster solver in absolute terms, GMRES-PRE demonstrates significantly superior scalability (1.95x vs 1.73x speedup). This behavior stands in sharp contrast to the results at 6000^2 (Table 5), where transitioning from Single-GPU to Multi-GPU caused GMRES-PRE performance to collapse (0.48x speedup). This reversal illustrates the critical difference between *introducing* communication and *scaling* an existing workload.

At the 12000^2 scale, the Multi-GPU synchronization penalty is already present (Table 7). In this regime, the scalability is determined by the ratio of computation to communication. GMRES-PRE, being a computationally "heavy" algorithm with extensive orthogonalization workload, requires substantial execution time per iteration (4.48s on 2 GPUs). This extended computation interval effectively masks the fixed latency of the MPI synchronization, allowing the solver to utilize the additional GPUs with near-perfect efficiency (1.95x).

Table 7: Iteration times and Speedup analysis at max resolution (12000^2).

Solver	Iteration Time (s)			Speedup	
	2 GPUs	**3 GPUs**	**4 GPUs**	**2 → 4**	**3 → 4**
GMRES-PRE	4.48	2.98	2.30	1.95x	1.30x
CG-PRE	2.11	1.50	1.22	1.73x	1.23x

Conversely, the lightweight CG-PRE solver finishes its local computation much faster (2.11s). As a result, it begins to hit the strong-scaling limit: the fixed time required for communication and synchronization becomes a larger percentage of the total iteration cycle, restricting the potential speedup to 1.73x.

4 Conclusions

The results of this work are consistent with prior Multi-GPU CFD studies that identify data communication efficiency as the dominant factors that limits scalability. As shown by Xue and Roy [7], solver performance on multiple GPUs is strongly affected by domain decomposition strategy and synchronization behavior rather than computational time itself. In particular, solvers with frequent global synchronization, such as GMRES, exhibit limited scaling efficiency, while CG-based solvers benefit more from bandwidth-dominated execution and reduced communication overhead. The trends observed in this study fit closely with these findings.

In addition, the observed scaling behavior aligns well with the Data-Centric MPI–CUDA framework proposed by Li et al. [3]. Their work demonstrates that restructuring CFD solvers into data-centric code can significantly reduce communication overhead and improve multi-GPU scalability. The results shown in Table 4 also shows how the contribution of communication overhead decreases as grid size increases. This result confirms their observation by showing that meaningful multi-GPU speedup is achieved only when the computational workload is large enough to overwhelm the ghost-cell exchange costs.

Finally, being able to integrate the NVIDIA AMGX solution toolkit seamlessly into our RANS solver reinforces our findings above by avoiding the expense of PCIe transfers and allowing for a completely GPU-resident pressure solution step. By eliminating the prevalent communication expense in the pressure Poisson equation step, AMGX enables efficient scalability and solves for higher-rate flows on multiple GPUs.

5 Future Work

Based on the findings of Xue and Roy [7] together with the result of this project, a natural extension of this work is to move from the current 1D domain decomposition to a 2D decomposition strategy. Their results show that 1D decomposition scales poorly on GPUs due to large ghost-cell surface areas and inefficient memory access. Therefore, implementing 2D or 3D decomposition is expected to reduce communication overhead and improve solver scalability, particularly for large grid sizes.

Crucially, future investigations must also rigorously validate the global convergence behavior of the Multi-GPU implementation to ensure a strictly fair comparison with Single-GPU baselines. In the current Additive Schwarz approach, ghost cells are exchanged only once per inner iteration. While this minimizes communication costs, it relaxes the global coupling of the system, potentially delaying the propagation of information across subdomains. Although initial tests show that residuals drop more rapidly in the early stages compared to the Single-GPU case, it is vital to verify that this does not lead to stagnation or instability over long simulation times. Verifying that the Multi-GPU configuration achieves the exact same physical accuracy within a comparable number of total iterations is essential to confirm that the observed speedups are genuine performance gains rather than numerical artifacts.

Another future improvement can be done by implementing the Data-Centric approach suggested by Li et al. [3]. Future work should focus on improving how data is moved and

transferred inside the solver. This includes merging ghost-cell data into fewer MPI messages, reducing the number of MPI calls made in each iteration, and optimized overlapping communication with computation so that GPUs spend less time waiting. In addition, testing the solver with more GPUs and larger grid sizes would help determine when these data-centric optimizations become most beneficial.

Lastly, extending the present integration between AMGX and the ability to run on multiple GPUs is a promising future direction. Utilizing the Multi-GPU solutions provided by AMGX may help to even further alleviate the expense of communication for the solution of the pressure solve.

References

[1] L. Davidson. *"pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow"*. Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, Göteborg. https://doi.org/10.1137/140980260. 2021.

[2] Bala Kumaresh Thileeep Kumar, Wei Liu, and Wuyang Hao. *GPU Accelerated Computational Methods Using Python and CUDA: Computational Fluid Dynamics*. Tech. rep. CFD Group, Linné Flow Center, KTH Royal Institute of Technology, 2024. URL: https://www.cfd-sweden.se/lada/Final_Report_CFD_Group_1.pdf.

[3] Ruitian Li et al. "A Data-Centric Approach for Efficient and Scalable CFD Implementation on Multi-GPUs Clusters". In: *PDCAT 2023*. 2023.

[4] Maxim Naumov et al. "AmgX: A Library for GPU Accelerated Algebraic Multigrid and Preconditioned Iterative Methods". In: *SIAM Journal on Scientific Computing* 37.5 (2015), S602–S626. DOI: 10.1137/140980260. URL: https://developer.nvidia.com/amgx.

[5] Shwina. *pyamgx: Python interface to NVIDIA's AMGX library*. https://github.com/shwina/pyamgx. GitHub repository. 2024.

[6] Jinchao Xu and Ludmil Zikatanov. "Algebraic multigrid methods". In: *Acta Numerica* 26 (2017), pp. 591–721.

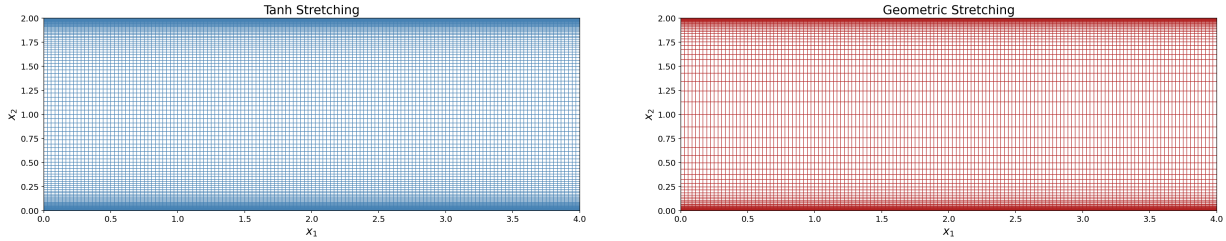
[7] Weicheng Xue and Christopher J. Roy. "Multi-GPU Performance Optimization of a CFD Code using OpenACC on Different Platforms". In: *arXiv preprint arXiv:2006.02602* (2020).

Appendix A Grid Generation Methodology

To resolve the turbulent boundary layer at $Re_\tau = 5200$, the computational domain requires a grid distribution that satisfies $y^+ < 1$ at the wall while managing computational costs in the bulk flow.

The original grid generation framework was based on a geometric stretching method provided by Lars Davidson (LADA). While geometric stretching is effective for specific configurations, we found that scaling the expansion ratio (y_{fac}) for significantly larger grid sizes was cumbersome. Consequently, we implemented a symmetric hyperbolic tangent stretching function. This method allows for easier control of wall clustering via a single parameter, γ, which proved more convenient than adjusting geometric expansion factors across varying resolutions.

Figure A.1 compares these methods for a representative coarse grid (144×144). The Tanh mesh uses a stretch factor of $\gamma = 3.25$, while the geometric mesh utilizes an expansion ratio of 1.15.



(a) Tanh Stretching ($\gamma = 3.25$) *(b) Geometric Stretching ($y_{fac} = 1.15$)*

Figure A.1: Visual comparison of grid distribution strategies for a 144×144 grid. The Tanh method (a) was selected as it allows for easier parameter tuning across large variations in grid density compared to the geometric approach (b).

To ensure the first cell height satisfies the low-Re criterion $y^+ < 1$, we analyzed the sensitivity of y^+ to grid density and the stretching parameter, γ. Table A.1 presents the results.

Table A.1: Dependence of first cell y^+ on grid density (n_j) and stretch factor (γ).

Stretch (γ)	$n_j = 144$	$n_j = 480$	$n_j = 2000$	$n_j = 5000$	$n_j = 10000$
1.25	15.14	4.50	**1.08**	0.43	0.21
1.5	11.02	3.26	0.78	0.31	0.16
2.0	5.44	1.60	0.38	0.15	0.08
2.5	2.52	0.74	0.18	0.07	0.04
3.0	1.12	0.33	0.08	0.03	0.02
3.25	**0.74**	0.21	0.05	0.02	0.01
3.5	0.48	0.14	0.03	0.01	0.01

Based on this analysis, we adopted an adaptive strategy: for coarse grids ($n_j \approx 144$), an aggressive stretch of $\gamma = 3.25$ is used to achieve $y^+ \approx 0.74$. Conversely, for fine grids ($n_j \geq 5000$), the stretch is relaxed to $\gamma = 1.25$ to avoid numerical stiffness while maintaining $y^+ \ll 1$.

Appendix B Sparse Iterative Solvers

For solving the large-scale sparse linear systems $\mathbf{Ax} = \mathbf{b}$ arising in this project, we initially benchmarked a variety of algorithms, including LGMRES and algebraic multigrid methods. However, the following detailed descriptions focus exclusively on GMRES and CG. These two solvers represent the fundamental baselines for non-symmetric and symmetric systems, respectively, and form the basis for the preconditioned variants (`GMRES-PRE` and `CG-PRE`) analyzed in the high-performance multi-GPU benchmarks.

B.1 Generalized Minimal Residual (GMRES)

GMRES is an extremely robust algorithm applicable to general non-singular linear systems. Its primary advantage is its monotonically decreasing residual, which ensures stable convergence even for difficult, non-symmetric systems dominated by convection.

In this project, GMRES is the mandatory choice for Momentum (`SOLVER_VEL`) and Turbulence (`SOLVER_TURB`) equations, where matrix asymmetry precludes the use of simpler solvers. Its main disadvantage is that memory consumption and computational cost grow linearly with the number of iterations, necessitating "restarts" to manage resources. Note, when we do multi-GPU benchmarks we set GMRES-PRE to these solvers instead of GMRES, as we then do only preconditioners.

B.2 Conjugate Gradient (CG)

CG is a highly efficient algorithm specifically designed for symmetric positive definite (SPD) matrices. When the matrix satisfies the SPD condition, as is the case for the Pressure Correction equation (`SOLVER_PP`), CG typically outperforms GMRES due to its low, fixed memory footprint and rapid convergence. Its main drawback is that it cannot be applied to the non-symmetric momentum or turbulence systems.

B.3 Jacobi Preconditioning and Large-Scale Strategy

As grid resolution increases, the condition number $\kappa(\mathbf{A})$ of the linear system grows rapidly, causing standard iterative solvers to stagnate. To enable convergence on the "Large" and "Extremely Large" grid categories (Multi-GPU), we employ a Jacobi preconditioner, transforming the system into:

$$\mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}, \quad \text{where} \quad \mathbf{M} = \text{diag}(\mathbf{A})$$

Here, $\mathbf{M}$ corresponds directly to the central coefficient array, a_P, derived from the discretized transport equations.

The action of the preconditioner reduces to a simple element-wise scaling, $\mathbf{z} = \mathbf{M}^{-1}\mathbf{r}$. This operation is computationally inexpensive and maps efficiently onto GPU architectures because it requires no additional sparse matrix-vector products or global reductions.

Consequently, for all large-scale Multi-GPU benchmarks, the focus shifts exclusively to the preconditioned variants: `GMRES-PRE` and `CG-PRE`. While standard solvers may suffice for small grids, the use of preconditioners is critical at scale to reduce the effective condition number and prevent the solver from hitting the maximum iteration limit.

B.4 Computational Fluid Dynamics

Computational Fluid Dynamics (CFD) approximates fluid motion by numerically solving the conservation equations for mass and momentum. For incompressible flow, these are given by the continuity equation in Eq. B.1 and the Navier–Stokes momentum equation in Eq. B.2. The computational domain is discretized into a mesh of control volumes, and the governing equations are integrated over each volume using the finite volume method. Fluxes through control-volume faces are approximated using interpolation schemes, resulting in a set of coupled algebraic equations for the discrete flow variables.

Assuming constant viscosity, the incompressible governing equations can be written in compact tensor form as

$$\frac{\partial v_i}{\partial x_i} = 0, \tag{B.1}$$

$$\frac{\partial v_i}{\partial t} + v_j \frac{\partial v_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \frac{\partial^2 v_i}{\partial x_j^2} + g_i. \tag{B.2}$$

After spatial discretization, each transport equation leads to a linear system of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b},$$

where $\mathbf{x}$ contains the unknown cell-center values (e.g. velocity components, pressure correction, or turbulence quantities), $\mathbf{A}$ is a large sparse coefficient matrix arising from convection and diffusion couplings, and $\mathbf{b}$ collects source terms and boundary condition contributions. For structured two-dimensional grids, $\mathbf{A}$ typically exhibits a five-point stencil sparsity pattern, with only a few non-zero entries per row.

In our case, the focus is on Reynolds-Averaged Navier–Stokes (RANS), which model all turbulent fluctuations to make simulations at practical Reynolds numbers computationally feasible. More expensive approaches such as DNS and LES are only considered conceptually, as their resolution requirements render them impractical for the large grids studied here.

Regardless of the turbulence model, near-wall regions require strong mesh refinement to resolve steep velocity gradients in the boundary layer. This rapidly increases the number of control volumes and leads to large, ill-conditioned linear systems. As the grid is refined, iterative solvers require more iterations to converge, further increasing computational cost.

For incompressible flow, pressure is obtained through a pressure–velocity coupling procedure, such as the SIMPLEC algorithm. Each SIMPLEC iteration requires assembling coefficient matrices and solving multiple sparse linear systems for momentum, pressure correction, and turbulence equations. Consequently, the repeated solution of systems of the form $\mathbf{A}\mathbf{x} = \mathbf{b}$ is expected to dominate the total runtime.

Sparse linear algebra therefore constitutes the primary computational bottleneck. GPU acceleration is targeted at iterative solver operations such as sparse matrix–vector products, inner products, and residual evaluations, where massive parallelism can be exploited to efficiently handle the large grids considered in this course.

Appendix C AMGX

C.1 Figures

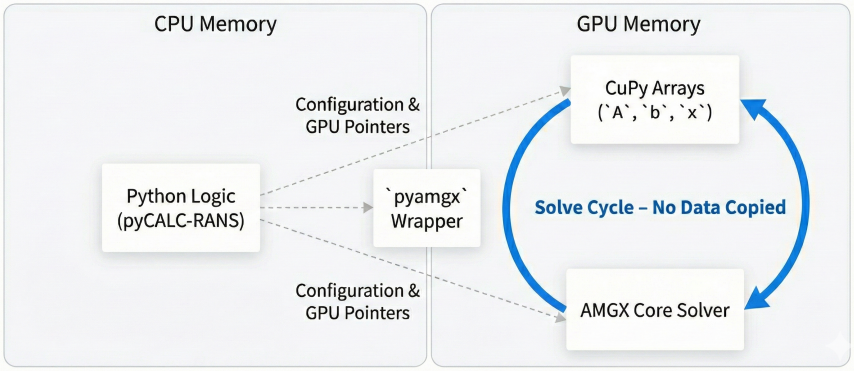
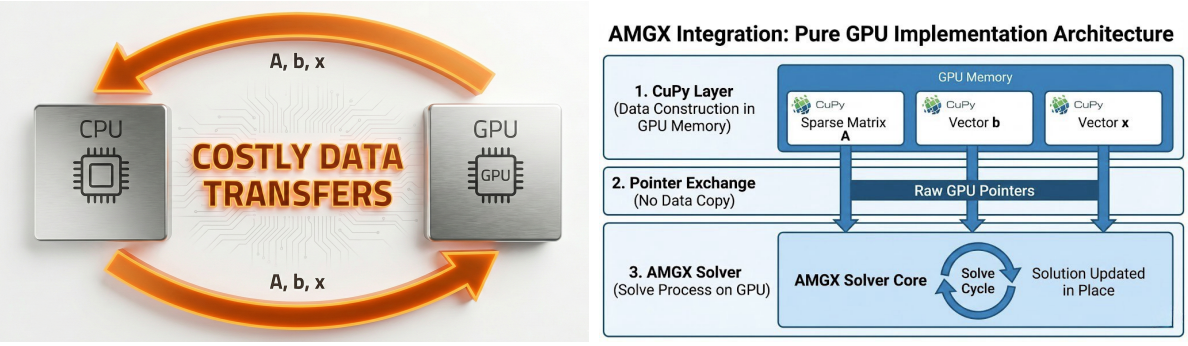


Figure C.2: System architecture of the pyCALC-RANS–AMGX integration showing a zero-copy data flow between CuPy arrays and the AMGX core solver on the GPU.



(a) Hybrid CPU–GPU pressure solve with repeated host–device transfers of A, b, and x.

(b) Pure GPU AMGX integration where the solve cycle operates directly on CuPy arrays without data copies.

Figure C.3: Shifting from a hybrid CPU–GPU architecture (a) to the proposed pure GPU AMGX implementation (b), illustrating the removal of costly PCIe data transfers.

C.2 PyAMGX: Python Interface

While AMGX is written in C++/CUDA, the `pyamgx` library provides Python bindings that enable seamless integration with Python-based CFD codes [5]. The key components of the pyamgx interface include:

Listing 1: AMGX Solver Initialization

```
# Example of Zero-Copy Data Transfer
import pyamgx

# Create resources
cfg = pyamgx.Config()
cfg.create(json.dumps(solver_config))
rsrc = pyamgx.Resources(cfg)

```

```
# Upload CSR matrix directly from GPU memory
d_A = pyamgx.Matrix(rsrc)
d_A.upload_CSR(A_cupy.tocsr())

# Solve on GPU
d_x = pyamgx.Vector(rsrc)
solver.solve(d_b, d_x)
```

C.3 AMGX implementation

AMGX uses a 4-character "mode string" to specify the data types for all internal computations. Understanding these modes is **critical** for successful integration, as mismatched types will cause runtime errors.

C.3.1 Mode String Format

The mode string follows the pattern: `[h/d][D/F][D/F][I/I64]`

- **Position 1 - Memory Location:**
  - `h` = Host (CPU memory)
  - `d` = Device (GPU memory)
- **Position 2 - Matrix Precision:**
  - `D` = Double precision (64-bit floating point, `float64`)
  - `F` = Single precision (32-bit floating point, `float32`)
- **Position 3 - Vector Precision:**
  - `D` = Double precision (`float64`)
  - `F` = Single precision (`float32`)
- **Position 4 - Index Type:**
  - `I` = 32-bit integers (`int32`)
  - `I64` = 64-bit integers (`int64`)

C.3.2 Modes Used in This Implementation

Our implementation uses two modes with automatic fallback:

1. **Primary Mode:** `dDDI`
   - `d` = Data resides on GPU (Device)
   - `D` = Matrix values are `float64` (double precision)
   - `D` = Vector values are `float64` (double precision)
   - `I` = Indices are `int32` (32-bit integers)

   This is the standard mode for scientific computing where double precision is required for accuracy.

2. **Fallback Mode:** `dDFI`

   - `d` = Data resides on GPU (Device)
   - `D` = Matrix values are `float64` (double precision)
   - `F` = Vector values are `float32` (single precision)
   - `I` = Indices are `int32` (32-bit integers)

   Some AMGX builds may not support full double precision vectors. The fallback mode uses single precision vectors while keeping double precision for the matrix.

C.3.3 Implementation with Automatic Fallback

The implementation includes automatic mode detection and fallback to handle different AMGX builds:

```
# Try dDDI mode first (full double precision)
try:
    pyamgx.initialize()
    cfg = pyamgx.Config()
    cfg.create_from_dict(amgx_config)
    resources = pyamgx.Resources()
    resources.create_simple(cfg)
    mode = pyamgx.Mode.dDDI  # Double matrix, Double vector, Int32 indices
    print("Using AMGX mode: dDDI (double precision)")
except Exception as e:
    if "Incorrect C API mode" in str(e):
        # Fallback to dDFI if dDDI not supported
        pyamgx.initialize()
        cfg = pyamgx.Config()
        cfg.create_from_dict(amgx_config)
        resources = pyamgx.Resources()
        resources.create_simple(cfg)
        mode = pyamgx.Mode.dDFI  # Double matrix, Float vector, Int32 indices
        print("Using AMGX mode: dDFI (mixed precision fallback)")
```

C.3.4 Data Type Requirements

Based on the mode string, all data passed to AMGX must match the expected types exactly. For `dDDI` mode:

Component	Required Type	NumPy/CuPy dtype
Matrix row pointers	32-bit integer	`int32`
Matrix column indices	32-bit integer	`int32`
Matrix values	64-bit float	`float64`
RHS vector b	64-bit float	`float64`
Solution vector x	64-bit float	`float64`

Table C.2: Data type requirements for AMGX dDDI mode

The implementation enforces these types explicitly:

```
# Ensure correct data types for AMGX
A_csr = A.tocsr()
A_csr.indptr = A_csr.indptr.astype(cp.int32)    # Row pointers
A_csr.indices = A_csr.indices.astype(cp.int32)  # Column indices
A_csr.data = A_csr.data.astype(cp.float64)      # Matrix values

b = b.astype(cp.float64)  # Right-hand side vector
x = cp.zeros(n, dtype=cp.float64)  # Solution vector
```