iscte INSTITUTO UNIVERSITÁRIO DE LISBOA

PROJECT REPORT

HIGH-PERFORMANCE GPU-ACCELERATED FINITE ELEMENT ANALYSIS

António Cruz (140129), Bruno Santos (140586), Pedro Miranda (129268), Ricardo Kayseller (95813)

Table of Contents

1. Introduction - Finite Element Method

The Finite Element Method (FEM) is a numerical technique widely used to approximate solutions of partial differential equations arising in engineering and scientific problems. Its main strength lies in its ability to handle complex geometries, heterogeneous materials, and general boundary conditions, which are often intractable using analytical approaches.

The fundamental idea of FEM is to replace a continuous problem by a discrete one. The physical domain is subdivided into a finite number of smaller regions, called elements, over which the unknown field is approximated using interpolation functions. By assembling the contributions of all elements, the original continuous problem is transformed into a system of algebraic equations that can be solved numerically.

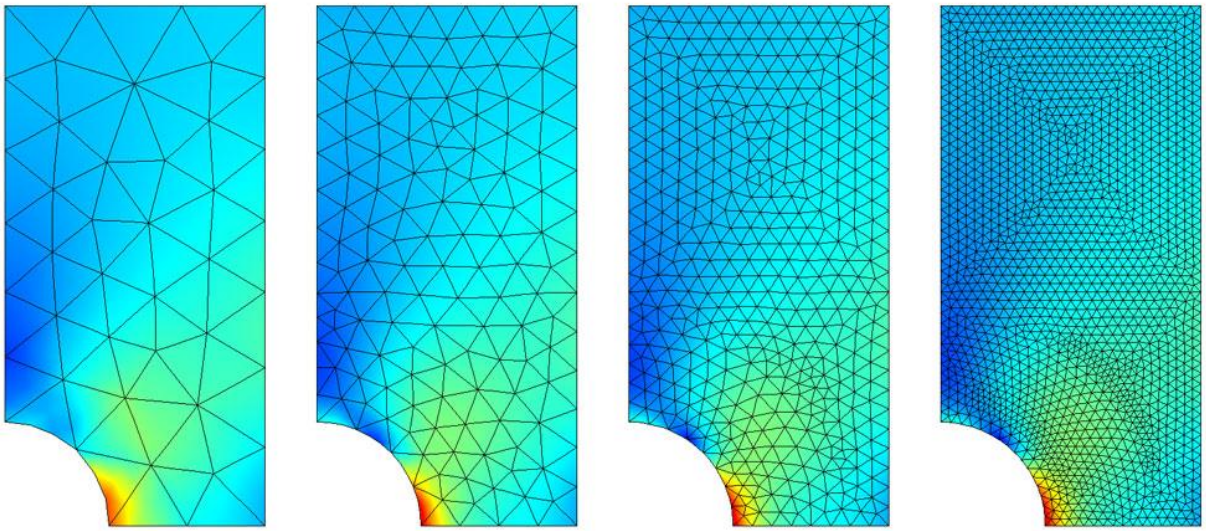

Figure 1: *Finite Element Method (FEM) workflow illustration: discretization of the domain into finite elements and transformation of the continuous problem into a discrete algebraic system.*

Because of this formulation, FEM naturally maps to linear algebra operations and therefore constitutes an ideal candidate for high-performance computing and parallel execution.

1.1. Classes of Problems Addressed by FEM

From a mathematical standpoint, FEM can be applied to several classes of partial differential equations, each associated with different physical phenomena and computational characteristics.

Elliptical problems describe steady state systems in which no time dependence exists. Typical examples include heat conduction, electrostatics, diffusion, and potential flow. These problems lead to well-conditioned linear systems that are symmetric and positive definite, making them particularly suitable for iterative solvers.

Parabolic problems introduce time dependence and describe transient diffusion processes, such as heat propagation. Their numerical solution requires both spatial discretization and time integration, increasing computational complexity.

Hyperbolic problems arise in wave propagation and dynamic systems, such as structural vibrations and acoustics. These problems are often dominated by stability constraints and time-stepping considerations.

The present work focuses exclusively on elliptical problems, more specifically on the Laplace equation:

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

where $u(x, y)$ represents the velocity potential field over a bounded domain $\Omega \subset \mathbb{R}^2$ with boundary $\Gamma = \partial\Omega$.

This equation governs a wide range of physical phenomena including steady-state heat transfer, electrostatics, mass diffusion, and incompressible potential flow.

Application Domain	Physical Interpretation of u
Incompressible irrotational flow	Velocity potential
Steady-state heat conduction	Temperature field
Electrostatics	Electric potential
Diffusion (steady state)	Concentration field

The choice of Laplace's equation as the benchmark problem provides several advantages for performance analysis:

- **Mathematical well-posedness**: Unique solution guaranteed under appropriate boundary conditions
- **Symmetric positive-definite system**: Enables use of efficient iterative solvers (CG)
- **Predictable convergence**: Facilitates consistent timing measurements
- **Representative workload**: Assembly and solve patterns typical of broader FEM applications

This equation captures all the essential computational challenges of FEM and is therefore well suited for performance-oriented studies.

1.2. Mathematical Formulation

1.2.1. Spatial Discretization and Element Choice

In FEM, the continuous domain is discretized into a finite number of elements connected at nodes. Within each element, the unknown field is approximated using shape functions defined over the element's geometry.

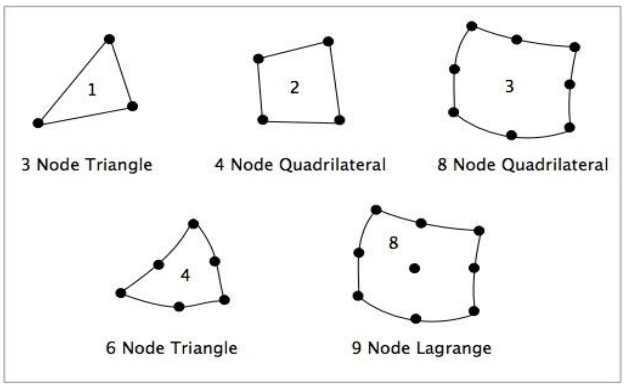

Figure 2: *Eight-node quadrilateral (Quad-8) element: geometry, corner and mid-edge nodes, and the counterclockwise node numbering convention.*

Several element types exist, depending on dimensionality and interpolation order. In two dimensions, common choices include triangular and quadrilateral elements, with either linear or higher-order interpolation.

In this work, eight-node quadrilateral elements (Quad-8) are used. These elements employ quadratic interpolation functions, allowing higher accuracy compared to linear elements while preserving numerical stability.

Each element comprises 8 nodes: 4 corner nodes and 4 mid-edge nodes. Node numbering follows the standard convention with counterclockwise ordering starting from the bottom-left corner.

Node	Type	Parametric Coordinates (ξ, η)
1	Corner	$(-1,-1)$
2	Corner	$(+1,-1)$
3	Corner	$(+1,+1)$
4	Corner	$(-1,+1)$
5	Mid-edge	$(0,-1)$
6	Mid-edge	$(+1,0)$
7	Mid-edge	$(0,+1)$
8	Mid-edge	$(-1,0)$

The increased number of nodes per element leads to larger element matrices and higher arithmetic intensity during numerical integration, making them particularly suitable for performance evaluation on modern hardware.

The use of Quad-8 elements also provides a realistic representation of engineering-grade FEM simulations, where higher-order elements are commonly employed to improve solution accuracy.

1.2.2. Variational Formulation and Algebraic Representation

The FEM formulation begins by expressing the governing differential equation in weak form. For the Laplace equation, this leads to the variational problem,

$$\int_{\Omega} \nabla v \cdot \nabla \phi \, d\Omega = \int_{\Gamma} v \; q \, d\Gamma$$

where ϕ is the unknown scalar field, v is a test function, and q represents prescribed boundary fluxes. After discretization using shape functions, the weak formulation results in a linear system of equations:

$$Ku = f$$

where: - $\mathbf{K} \in \mathbb{R}^{N_{dof} \times N_{dof}}$ is the global stiffness matrix (sparse, symmetric, positive-definite) - $\mathbf{u} \in \mathbb{R}^{N_{dof}}$ is the vector of nodal unknowns - $\mathbf{f} \in \mathbb{R}^{N_{dof}}$ is the global load vector

The global stiffness matrix is then assembled from element-level contributions of the form:

$$\mathbf{K}^{(e)} = \int_{\Omega_e} (\nabla \mathbf{N})^T \, \mathbf{D} (\nabla \mathbf{N}) \, d\Omega$$

where N denotes the shape functions and D represents the material or conductivity matrix. The resulting global matrix is sparse, symmetric, and positive definite, which strongly influences solver choice and performance behavior.

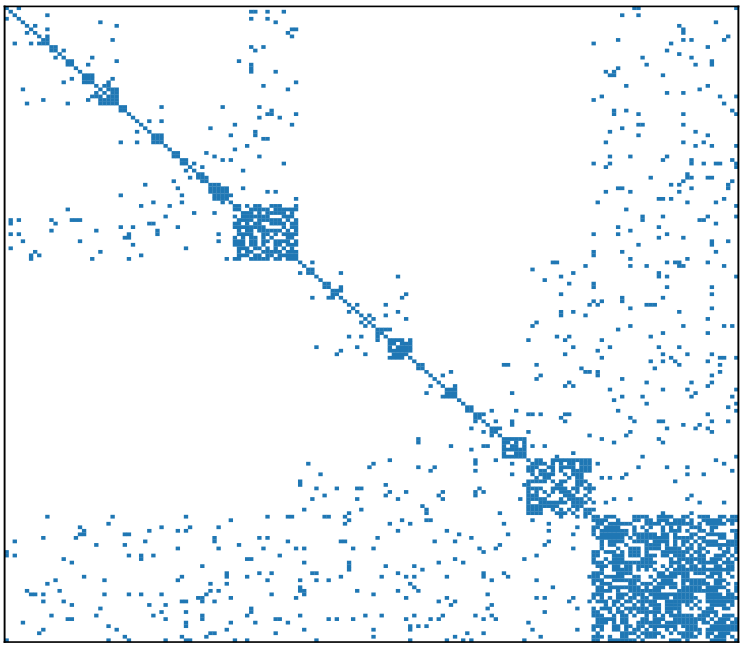

Figure 3: *Example of a sparse FEM global stiffness matrix assembled with FEM: nonzero entries reflect element connectivity, yielding a banded sparse global stiffness matrix structure.*

1.2.3. Boundary Conditions

Boundary conditions (BCs) specify the constraints and interactions imposed on the boundaries of a numerical model and are fundamental to obtaining a well-posed and solvable problem. They define how the system responds at its limits and ensure that the mathematical formulation admits a unique and physically consistent solution. In practical applications, boundary conditions are selected to reflect the real physical support, loads, or environmental interactions acting on the domain. The most identified categories of boundary conditions are essential (Dirichlet), natural (Neumann), and mixed (Robin) boundary conditions, and an appropriate combination of these is required to accurately represent the problem being analyzed.

1.2.3.1. Dirichlet Boundary Conditions

Dirichlet boundary conditions specify fixed potential values at designated boundary nodes:

$$u = \bar{u} \quad \text{on } \Gamma_D$$

These are implemented using row/column elimination: for each constrained degree of freedom i with prescribed value $\bar{u}_i$:

1. Set $K_{ii} = 1$ and $K_{ij} = K_{ji} = 0$ for $j \neq i$
2. Set $f_i = \bar{u}_i$
3. Modify $f_j \leftarrow f_j - K_{ji}\bar{u}_i$ for all $j \neq i$ (to preserve symmetry)

In the project context, Dirichlet conditions are applied at outlet boundaries where the potential is fixed.

1.2.3.2. Robin Boundary Conditions

Robin boundary conditions combine flux and potential contributions at inlet boundaries:

$$p \cdot u + \frac{\partial u}{\partial n} = \gamma \quad \text{on } \Gamma_R$$

where p is a coefficient and γ represents the prescribed combination of flux and potential.

1.2.3.3. Boundary Detection

Boundary nodes are identified geometrically based on coordinate tolerance. The implementation detects:

- **Inlet boundary**: Left edge of domain (minimum x coordinate)
- **Outlet boundary**: Right edge of domain (maximum x coordinate)

A tolerance parameter (bc_tolerance = 1e-9) handles floating-point precision in coordinate comparisons.

1.3. Linear Solver Strategy

1.3.1. Conjugate Gradient Method

All implementations use the Conjugate Gradient (CG) method for solving the linear system $\mathbf{Ku} = \mathbf{f}$. CG is particularly suitable for this application because:

1. **Symmetric positive-definite system**: The stiffness matrix $\mathbf{K}$ from elliptic PDEs satisfies the SPD requirement
2. **Memory efficiency**: Only matrix-vector products required, no explicit factorization
3. **Predictable convergence**: Error reduction bounded by condition number
4. **Parallelization potential**: Core operations (SpMV, dot products, axpy) are data-parallel

The CG algorithm generates a sequence of iterates $\mathbf{u}^{(k)}$ that minimize the $\mathbf{K} - norm$ Ajuof the error over a Krylov subspace of increasing dimension.

1.3.2. Jacobi Preconditioning

All implementations apply Jacobi (diagonal) preconditioning to accelerate convergence:

$$\mathbf{M} = \text{diag}(\mathbf{K})$$

The preconditioned system becomes:

$$\mathbf{M}^{-1/2}\mathbf{K}\mathbf{M}^{-1/2}\tilde{\mathbf{u}} = \mathbf{M}^{-1/2}\mathbf{f}$$

The Jacobi preconditioner was chosen deliberately for this performance study:

Characteristic	Benefit
Element-wise operations	Trivially parallelizable across all execution models
No fill-in	Memory footprint identical to diagonal extraction
No factorization	Setup cost $\mathcal{O}(n)$
Implementation-independent	Does not favor any particular parallelization strategy

More sophisticated preconditioners (ILU, AMG) might provide faster convergence but would introduce implementation-dependent performance variations that complicate fair comparison.

1.3.3. Solver Configuration

The following parameters are held constant across all implementations:

Parameter	Value	Rationale
Method	Conjugate Gradient	Optimal for SPD systems
Preconditioner	Jacobi (diagonal)	Parallelizes uniformly
Relative tolerance	10^{-8}	Engineering accuracy
Absolute tolerance	0	Rely on relative criterion
Maximum iterations	No limit	Sufficient in order to simulate all test problems
Progress reporting	Every 50 iterations	Balance monitoring vs. overhead

1.3.4. Convergence Monitoring

The solver monitors convergence using the relative residual norm:

$$\text{rel\_res} = \frac{\| \mathbf{r}^{(k)} \|_2}{\| \mathbf{b} \|_2} = \frac{\| \mathbf{f} - \mathbf{K}\mathbf{u}^{(k)} \|_2}{\| \mathbf{f} \|_2}$$

Convergence is declared when $\text{rel\_res} < 10^{-8}$ or the iteration count exceeds the maximum.

1.4. Post-Processing: Derived Fields

1.4.1. Velocity Field Computation

The velocity field is computed as the negative gradient of the potential:

$$\mathbf{v} = -\nabla u = -\begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix}$$

For each element, the gradient is evaluated at 4 Gauss points and averaged:

$$\mathbf{v}_e = \frac{1}{4}\sum_{p=1}^{4} \left(-\mathbf{B}_p^T \mathbf{u}_e\right)$$

where $\mathbf{u}_e$ is the vector of nodal solution values for element e.

1.4.2. Velocity Magnitude

The velocity magnitude per element:

$$|\mathbf{v}|_e = \frac{1}{4}\sum_{p=1}^{4} \sqrt{v_{x,p}^2 + v_{y,p}^2}$$

1.4.3. Pressure Field

Pressure is computed from Bernoulli's equation for incompressible flow:

$$p = p_0 - \frac{1}{2}\rho|\mathbf{v}|^2$$

where: - $p_0 = 101328.8$ Pa (reference pressure) - $\rho = 0.6125$ kg/m³ (fluid density)

These constants are configurable parameters in the solver constructor.

1.5. Computational Pipeline of the Finite Element Method

From a computational perspective, the FEM workflow can be decomposed into a sequence of well-defined stages, each exhibiting distinct performance characteristics.

1. Load mesh data

2. Initialize global stiffness matrix K ← 0

3. Initialize global load vector f ← 0

4. **for** each element e in mesh **do**
   Compute element stiffness matrix Ke
   Compute element load vector fe
   Assemble Ke into K
   Assemble fe into f
   end **for**

5. Apply boundary conditions to K and f

6. Solve linear system:
   K * u = f

7. Compute Derived

8. Export Results

The process begins with mesh loading, where nodal coordinates, element connectivity, and boundary information are read into memory. Although this stage is not computationally intensive, it defines data layout and memory access patterns for all subsequent steps.

Element-level assembly follows, during which local stiffness matrices and load vectors are computed using numerical integration. This stage involves a large number of floating-point operations and is inherently parallel, as each element can be processed independently. As such, it represents one of the most computationally intensive parts of the FEM pipeline.

```
for each element e do
   Retrieve node coordinates
   Compute Jacobian and its determinant

   for each Gauss integration point gp do
      Evaluate shape functions N
      Evaluate derivatives ∇N
      Compute local stiffness contribution:
         Ke += (∇Nᵀ · D · ∇N) * det(J) * w_gp
   end for
end for
```

The local contributions are then assembled into the global sparse matrix. This step involves indirect memory accesses and accumulation of values at shared locations, making it sensitive to memory bandwidth and synchronization overheads. Efficient implementation of this phase is crucial for overall performance, particularly on GPU architectures.

```
for each element e do
  for i = 1 to n_nodes_per_element do
    for j = 1 to n_nodes_per_element do
      I = global_index(e, i)
      J = global_index(e, j)
      K[I, J] += Ke[i, j]
    end for
  end for
end for
```

Boundary conditions are subsequently applied. Dirichlet conditions enforce prescribed values by modifying the system matrix and right-hand side, while Neumann conditions introduce additional contributions to the load vector. Although conceptually simple, this step must be carefully implemented to preserve numerical correctness.

```
for each prescribed node i do
  K[i, :] = 0
  K[i, i] = 1
  f[i] = prescribed_value
end for
```

Once the system is fully assembled, the resulting linear system is solved using an iterative solver. This stage usually dominates execution time, as it involves repeated sparse matrix-vector multiplications and vector operations.

```
Initialize u_o
r_o = f - K u_o
p_o = r_o

for k = 0 until convergence do
  α = (r^T r) / (p^T K p)
  u = u + α p
  r_new = r - α K p

  if ||r_new|| < tolerance then
    break
  end if

  β = (r_new^T r_new) / (r^T r)
  p = r_new + β p
  r = r_new
end for
```

Finally, post-processing is performed to reconstruct the solution field, compute derived quantities, and generate visualizations. While less computationally demanding, this step is essential for validating results and analyzing physical behavior.

1.5.1. Parallelization Targets

The assembly stage exhibits the highest parallelization potential because each element's stiffness matrix can be computed independently. The solve stage benefits from parallel SpMV but faces memory bandwidth constraints characteristic of sparse computations. Post-processing mirrors assembly in its parallel structure.

Stage	Computational Pattern	Parallelization Opportunity
Load Mesh	I/O bound	Limited (disk/memory bandwidth)
Assemble System	Element-independent loops	**High** (embarrassingly parallel)
Apply BCs	Sequential modifications	Low (small fraction of runtime)
Solve System	Sparse matrix-vector products	**Medium** (memory bandwidth limited)
Compute Derived	Element-independent loops	**High** (embarrassingly parallel)
Export Results	I/O bound	Limited (disk bandwidth)

2. Software Architecture

2.1 Solver Interface Contract

All solver classes implement a consistent interface, enabling the unified SolverWrapper to instantiate any implementation interchangeably:

```
class Quad8FEMSolver:
    """Base interface implemented by all solver variants."""

    def __init__(
        self,
        mesh_file: Path | str,
        p0: float = 101328.8,       # Reference pressure
        rho: float = 0.6125,        # Fluid density
        gamma: float = 2.5,         # Robin BC coefficient
        rtol: float = 1e-8,         # CG relative tolerance
        maxiter: int = 15000,       # Maximum CG iterations
        bc_tolerance: float = 1e-9,  # BC detection tolerance
        cg_print_every: int = 50,    # Progress interval
        verbose: bool = True,       # Console output
        progress_callback = None     # Real-time monitoring
    ):
        """Initialize solver with mesh and parameters."""
        ...

    def run(
        self,
        output_dir: Path | str = None,
        export_file: Path | str = None
    ) -> Dict[str, Any]:
        """Execute complete FEM workflow and return results."""
        ...
```

This interface contract ensures that switching between CPU, GPU, Numba, and CUDA implementations requires only changing the solver type parameter, with no modifications to calling code.

2.2 SolverWrapper: Unified Factory

The SolverWrapper class provides a unified factory interface for solver instantiation:

```
class SolverWrapper:
    """Unified interface for all solver implementations."""

    SOLVER_TYPES = [
        "cpu",          # NumPy/SciPy baseline
        "cpu_threaded",  # ThreadPoolExecutor
        "cpu_multiprocess", # ProcessPoolExecutor
        "numba",        # Numba JIT CPU
        "numba_cuda",    # Numba CUDA kernels
        "gpu",          # CuPy with RawKernel
        "auto"          # Auto-detect best available
    ]

    def __init__(self, solver_type: str, params: dict, progress_callback=None):
        # Instantiate appropriate solver based on type
        ...

    def run(self) -> Dict[str, Any]:
        # Execute solver with memory tracking
        ...

    @staticmethod
    def get_available_solvers() -> list:
        # Detect available implementations based on installed packages
        ...
```

The auto mode detects the best available solver by checking for GPU availability (CuPy import success).

2.3 Progress Callback System

Real-time monitoring is provided through a callback interface that all solvers invoke at consistent pipeline points:

```
class ProgressCallback:
    """Interface for real-time solver monitoring."""

    def on_stage_start(self, stage: str) -> None:
        """Called when a pipeline stage begins."""
        ...

    def on_stage_complete(self, stage: str, duration: float) -> None:
        """Called when a pipeline stage completes."""
        ...

    def on_mesh_loaded(
```

```
        self, nodes: int, elements: int,
        coordinates: dict, connectivity: list
    ) -> None:
        """Called after mesh loading with mesh metadata."""
        ...

    def on_iteration(
        self, iteration: int, max_iterations: int,
        residual: float, relative_residual: float,
        elapsed_time: float, etr_seconds: float
    ) -> None:
        """Called during CG iterations with convergence data."""
        ...

    def on_solution_increment(
        self, iteration: int, solution: ndarray
    ) -> None:
        """Called periodically with partial solution for visualization."""
        ...

    def on_error(self, stage: str, message: str) -> None:
        """Called when an error occurs."""
        ...
```

This callback system enables the web interface to display live progress, convergence curves, and intermediate solution fields regardless of which solver implementation is executing.

2.4 Timing Instrumentation

Each solver records per-stage wall-clock time using high-resolution timers (time.perf_counter()). The timing dictionary structure is consistent across all implementations:

```
timing_metrics = {
    'load_mesh': float,       # Mesh loading time (seconds)
    'assemble_system': float,  # Global assembly time
    'apply_bc': float,        # Boundary condition application
    'solve_system': float,     # Linear solver time
    'compute_derived': float,  # Post-processing time
    'total_workflow': float,   # Sum of above stages
    'total_program_time': float # Wall-clock from initialization
}
```

This granular timing enables identification of which stages benefit most from each parallelization strategy.

2.5 Result Format

All solvers return a standardized dictionary structure containing solution fields, convergence status, timing metrics, and metadata:

```
results = {
    # Solution fields
    'u': ndarray,          # Nodal potential (Nnodes,)
    'vel': ndarray,        # Velocity vectors (Nelements, 2)
```

```
    'abs_vel': ndarray,    # Velocity magnitude (Nelements,)
    'pressure': ndarray,   # Pressure field (Nelements,)

    # Convergence status
    'converged': bool,     # True if tolerance achieved
    'iterations': int,     # CG iterations performed

    # Performance metrics
    'timing_metrics': {
        'load_mesh': float,
        'assemble_system': float,
        'apply_bc': float,
        'solve_system': float,
        'compute_derived': float,
        'total_workflow': float,
        'total_program_time': float,
    },

    # Solution statistics
    'solution_stats': {
        'u_range': [float, float],  # [min, max]
        'u_mean': float,
        'u_std': float,
        'final_residual': float,
        'relative_residual': float,
    },

    # Problem metadata
    'mesh_info': {
        'nodes': int,
        'elements': int,
        'matrix_nnz': int,
        'element_type': 'quad8',
        'nodes_per_element': 8,
    },

    # Solver configuration
    'solver_config': {
        'linear_solver': 'cg',
        'tolerance': float,
        'max_iterations': int,
        'preconditioner': 'jacobi',
    },
}
```

The timing_metrics dictionary is essential for performance analysis, providing per-stage timing that reveals which computational phases benefit most from each parallelization strategy.

2.6. Shared Computational Modules

2.6.1. Module Organization

Each implementation variant includes adapted versions of four core computational modules. While the mathematical operations are identical, each version is optimized for its execution model:

Module	Purpose	CPU (NumPy)	Numba JIT	CuPy GPU	CUDA Kernel
shape_n_der8	Shape functions, derivatives, Jacobian	np.zeros, np.linalg	@njit, explicit loops	cp.zeros, cp.linalg	Inlined in kernel
genip2dq	Gauss point coordinates and weights	np.array constants	@njit, return arrays	cp.array constants	Helper function
elem_quad8	Element stiffness matrix	np.outer, matrix ops	@njit, nested loops	cp.outer, matrix ops	Full kernel
robin_quadr	Robin BC edge integration	NumPy loops	@njit loops	CuPy loops	CPU fallback

2.6.2. Implementation Adaptations

NumPy (CPU Baseline)

Uses vectorized operations and BLAS/LAPACK routines through NumPy:

```
# Jacobian computation
jaco = XN.T @ Dpsi  # Matrix multiplication
Detj = np.linalg.det(jaco)
Invj = np.linalg.inv(jaco)
B = Dpsi @ Invj

# Stiffness accumulation
Ke += wip * (B @ B.T)  # Outer product
```

Numba JIT

Replaces NumPy operations with explicit loops for LLVM optimization:

```
@njit(cache=True)
def shape_n_der8(XN, csi, eta):
   # Explicit Jacobian computation
   jaco = np.zeros((2, 2), dtype=np.float64)
   for i in range(8):
      jaco[0, 0] += XN[i, 0] * Dpsi[i, 0]
      jaco[0, 1] += XN[i, 0] * Dpsi[i, 1]
      jaco[1, 0] += XN[i, 1] * Dpsi[i, 0]
      jaco[1, 1] += XN[i, 1] * Dpsi[i, 1]

   # Explicit determinant
   Detj = jaco[0, 0] * jaco[1, 1] - jaco[0, 1] * jaco[1, 0]
   ...
```

CuPy GPU

Mirrors NumPy API but executes on GPU memory:

```
import cupy as cp

def Shape_N_Der8(XN, csi, eta):
    psi = cp.zeros(8)
    Dpsi = cp.zeros((8, 2))
    # ... same structure as NumPy
    jaco = XN.T @ Dpsi
    Detj = cp.linalg.det(jaco)
    ...
```

CUDA Kernels (RawKernel and Numba CUDA)

Inline all computations within the kernel to minimize memory transactions:

```
// CuPy RawKernel (CUDA C)
__global__ void quad8_assembly_kernel(...) {
    int e = blockIdx.x * blockDim.x + threadIdx.x;

    // Local arrays in registers/local memory
    double Ke[8][8] = {{0.0}};
    double XN[8][2];

    // All shape function, Jacobian, stiffness computation inlined
    ...
}
```

2.6.3. Mathematical Equivalence

Despite implementation differences, all versions compute mathematically identical results (within floating-point precision). This is verified by:

1. Comparing solution vectors across implementations
2. Checking that relative differences are within machine epsilon ($\approx 10^{-15}$)
3. Ensuring identical iteration counts for CG convergence

This equivalence is essential for valid performance comparisons: timing differences reflect execution model efficiency, not algorithmic variations.

2.7. Mesh Format and I/O

2.7.1. HDF5 Mesh Format

Meshes are stored in HDF5 format for efficient I/O operations:

```
mesh.h5
 ├── coordinates/
|    ├── x    (float64, shape: Nnodes)
|    └── y    (float64, shape: Nnodes)
└── connectivity/
    └── quad8 (int32, shape: Nelements × 8)
```

2.7.2. Format Advantages

HDF5 provides several advantages for this application:

Feature	Benefit
Binary format	Faster I/O than text formats
Compression support	Reduced storage for large meshes
Memory mapping	Efficient access patterns
Platform independence	Cross-platform compatibility
Hierarchical structure	Organized data layout
Partial reads	Future extensibility for distributed computing

2.7.3. Legacy Format Support

For compatibility, the solver also supports:

- **NPZ** (NumPy compressed archive): Fast binary format
- **Excel (.xlsx)**: Human-readable, useful for small test cases

All formats are converted to the internal NumPy array representation upon loading.

2.8. Summary

The common foundation described in this section ensures that all six solver implementations operate on identical mathematical and algorithmic ground. Key design decisions supporting fair performance comparison include:

1. **Identical FEM formulation**: Quad-8 elements, 9-point quadrature, Robin/Dirichlet BCs
2. **Uniform solver strategy**: Jacobi-preconditioned CG with fixed tolerance
3. **Consistent interfaces**: Same constructor signature, result format, callback system
4. **Equivalent computational modules**: Mathematically identical, adapted for each execution model
5. **Standardized timing**: Per-stage instrumentation with identical granularity

With this foundation established, the following sections examine how each implementation variant exploits parallelism within this common framework, and how the resulting performance characteristics differ across problem sizes and computational stages.

3. Implementations

3.1. Execution Models

This section presents multiple implementations of the same FEM problem using different execution models on CPU and GPU. All implementations share an identical numerical formulation, discretization, boundary conditions, and solver configuration; observed differences arise exclusively from the execution strategy and computational backend.

The implementations cover sequential CPU execution, shared-memory and process-based CPU parallelism, just-in-time compiled CPU execution using Numba, and GPU-based execution using Numba CUDA and CuPy with custom raw kernels. Together, these approaches span execution

models ranging from interpreter-driven execution to compiled and accelerator-based computation.

Numerical equivalence is preserved across all implementations, enabling direct and fair comparison of execution behavior, performance, and scalability under consistent numerical conditions.

3.1.1. Pre-Implementation Phase

Before the development of the CPU and GPU execution models presented in this section, a dedicated pre-implementation phase was carried out to migrate an existing Finite Element Method (FEM) solver, previously developed in MATLAB, to the Python programming language.

The primary objective of this transition was to ensure that the original numerical formulation was fully preserved. In particular, the following aspects were maintained:

- The element types used (eight-node quadrilateral elements - Quad-8)
- The assembly procedures for stiffness matrices and load vectors
- The treatment of boundary conditions (Dirichlet and Robin conditions)
- The configuration of the linear solver and the corresponding convergence criteria

This phase was exclusively focused on functional and numerical validation of the Python implementation, and no performance optimization was performed. All computational kernels were rewritten using scientific Python libraries appropriate to the project objectives, thereby enabling, in a subsequent phase, the implementation of both CPU- and GPU-based solutions.

3.2. Solver Implementation 1: CPU Baseline

3.2.1. Overview

The CPU baseline implementation serves as the reference against which all other CPU and GPU implementations are evaluated. It prioritizes correctness, algorithmic clarity, and reproducibility over performance, establishing both the functional specification and the performance floor for the project.

Attribute	Description
Technology	Python (NumPy, SciPy)
Execution Model	Sequential, single process
Role	Correctness reference and performance baseline
Dependencies	NumPy, SciPy, pandas, h5py

3.2.2. Technology Background

The baseline implementation is built on Python’s scientific computing ecosystem and executes on a sequential CPU model.

Software ecosystem:

- **NumPy** provides N-dimensional arrays and vectorized operations backed by optimized BLAS/LAPACK libraries.
- **SciPy** supplies sparse matrix data structures and iterative solvers for large linear systems.

- **h5py and pandas** support efficient binary input/output for mesh and result data.
- This stack enables concise algorithm expression while delegating computationally intensive kernels to compiled numerical libraries.

Execution characteristics:

- Execution is performed within the **CPython interpreter** and is therefore subject to the **Global Interpreter Lock (GIL)**.
- While NumPy and SciPy release the GIL during computational kernels, Python-level control flow remains serialized.
- For FEM workloads, this results in a mixed execution model:
    - **Element loops** execute sequentially at the Python level with the GIL held.
    - **Dense linear algebra operations** are executed in optimized BLAS/LAPACK routines with the GIL released.
    - **Sparse iterative solvers** execute predominantly in compiled SciPy code, also releasing the GIL during major operations.

Relevance for FEM:

- Provides a clear and traceable mapping between mathematical formulation and implementation.
- Serves as a correctness reference for validating parallel implementations.
- Enables early identification of computational bottlenecks through profiling.
- Establishes a minimum performance bound for speedup evaluation.

3.2.3. Implementation Strategy

The FEM workflow is organized into sequential stages to ensure correctness and consistent performance evaluation.

- **Mesh loading**: mesh data is loaded from binary HDF5 files to minimize parsing overhead and keep I/O negligible.
- **System assembly** (element-by-element):
    - initialize the global sparse matrix in an insertion-friendly format
    - compute per element an 8×8 local stiffness matrix + load terms using numerical quadrature
    - scatter local contributions into the global sparse system
    - convert the matrix to a compressed sparse format optimized for sparse matrix-vector products during solving
- **Boundary conditions**:
    - Robin (inlet) enforced via numerical integration of boundary terms
    - Dirichlet (outlet) imposed using a penalty method
    - overall cost is small compared to assembly/solve
- **Linear system solution**: solved using SciPy Conjugate Gradient (CG) with:
    - diagonal equilibration
    - Jacobi (diagonal) preconditioning
    - identical solver configuration across implementations for consistent convergence behavior

- **Post-processing**: derived fields (e.g., velocity, pressure) computed via additional element-level loops; not dominant, but measurable for large meshes.

3.2.4. Optimization Techniques Applied

Several optimizations are applied to improve performance while preserving numerical equivalence and implementation simplicity.

- **Sparse matrix format selection**: different sparse formats are used depending on the computation stage, balancing assembly efficiency and solver performance:

Format	Insertion	SpMV	Memory	Usage
LIL (List of Lists)	O(1) amortized	O(nnz)	Higher	Assembly
CSR (Compressed Sparse Row)	O(n)	O(nnz) optimal	Lower	Solve

- The LIL → CSR strategy minimizes insertion overhead during assembly while ensuring optimal sparse matrix-vector products during iterative solving.

- **Diagonal equilibration**:
  - The linear system is diagonally equilibrated before solving to improve conditioning
  - Reduces sensitivity to element size variation and improves convergence, especially for large/heterogeneous meshes
- **Preconditioning strategy**:
  - A Jacobi (diagonal) preconditioner is applied in the CG solver
  - Provides a good trade-off between simplicity and convergence robustness, ensuring stable iteration counts
- **Vectorized inner operations**:
  - Element-level dense operations are expressed using NumPy vectorized kernels
  - This delegates inner computations to optimized compiled BLAS/LAPACK routines, mitigating Python interpreter overhead

3.2.5. Challenges and Limitations

The sequential CPU baseline is mainly limited by Python interpreter overhead and sparse assembly costs, which restrict scalability for large meshes.

Limitation	Impact
Sequential Python element loop	Assembly becomes interpreter-bound and scales linearly with mesh size
GIL serialization	Limits any benefit from multi-threading at Python-level control flow
Incremental sparse insertion	High overhead from dynamic allocation and indirect indexing
Scattered memory access	Poor cache locality and increased memory traffic

Additional observed behavior:

- **Sparse format trade-off**: assembly uses an insertion-friendly sparse format and converts to a solver-efficient compressed format; conversion adds overhead but is amortized over CG iterations.
- **Preconditioning sensitivity**: CG convergence is highly dependent on preconditioning; Jacobi preconditioning stabilizes convergence with negligible cost.

- **Residual monitoring**: convergence is evaluated using the true residual norm, ensuring consistent diagnostics across implementations with minimal overhead.

3.2.6. Performance Characteristics and Baseline Role

The sequential CPU implementation defines the reference performance profile used to evaluate all parallel CPU and GPU approaches.

Expected scaling

Stage	Complexity	Dominant Factor
Mesh loading	O(N_nodes)	I/O bandwidth
Assembly	O(N_elements)	Python loop + sparse insertion overhead
Linear system solution	O(iterations × nnz)	SpMV memory bandwidth
Post-processing	O(N_elements)	Python loop overhead

Profiling observations (large meshes)

- **Assembly:** ~50-70% of total runtime
- **Solve:** ~20-40% (dominated by SpMV + iteration count)
- Post-processing: ~5-15%
- Mesh I/O + boundary conditions: typically, <5%

Baseline role

- **Correctness reference:** alternative implementations must match numerical results.
- **Performance floor:** parallel methods must outperform this runtime.
- **Solver reference:** convergence behavior and iteration counts should remain consistent.

This baseline defines the reference execution profile for speedup and scalability analysis.

3.2.7. Summary

The CPU baseline provides a clear, correct, and reproducible reference for all subsequent implementations. While intentionally limited in scalability, it establishes a shared algorithmic foundation, a correctness benchmark, and a performance floor for comparative evaluation.

Key observations include:

- Assembly is interpreter-bound and dominates runtime.
- Python-level overhead outweighs arithmetic cost for element-level operations.
- The iterative solver is primarily memory-bound.

Subsequent implementations address these limitations through parallel execution models, JIT compilation, and GPU offloading, while preserving numerical equivalence with this baseline.

3.3. Solver Implementation 2: CPU Threaded

3.3.1. Overview

The CPU Threaded implementation extends the CPU baseline by introducing parallelism through Python's concurrent.futures.ThreadPoolExecutor. The objective is to evaluate whether

multi-threading can accelerate FEM assembly and post-processing despite the presence of Python's Global Interpreter Lock (GIL).

Unlike the baseline, which executes all element-level operations sequentially, this implementation partitions the mesh into batches processed concurrently by multiple threads. The approach relies on the fact that NumPy releases the GIL during computational kernels, allowing partial overlap of execution across threads.

Attribute	Description
Technology	Python ThreadPoolExecutor (concurrent.futures)
Execution Model	Multi-threaded with GIL constraints
Role	Evaluate benefits and limits of threading on CPU
Dependencies	NumPy, SciPy, concurrent.futures (stdlib)

3.3.2. Technology Background

Python threading is limited by the Global Interpreter Lock (GIL), which serializes execution of Python bytecode and prevents true parallelism for CPU-bound workloads at the Python level. However, many NumPy kernels release the GIL, allowing partial concurrency.

GIL and NumPy behavior

- The GIL blocks parallel execution of Python-level code across threads.
- NumPy releases the GIL in several operations:
  - Vectorized arithmetic
  - BLAS/LAPACK dense kernels
  - Element-wise math kernels

ThreadPoolExecutor model

- ThreadPoolExecutor provides reusable worker threads and a future-based execution model.
- Key advantages:
  - Low overhead due to persistent threads
  - Asynchronous submission via Future
  - Dynamic scheduling (basic load balancing)
  - Shared-memory access to NumPy arrays

Implications for FEM workloads

Operation	GIL Released	Expected Benefit
Python loop iteration	No	None
Sparse matrix indexing	No	None
NumPy dense kernels	Yes	Moderate
Elementwise NumPy ops	Yes	Moderate

- Speedup depends on maximizing time spent in GIL-free NumPy kernels and minimizing Python coordination.

3.3.3. Implementation Strategy

3.3.3.1 Batch-Based Parallelization

To amortize threading overhead and reduce GIL contention, elements are grouped into fixed-size batches. Each batch is processed by a single thread, enabling coarse-grained parallelism:

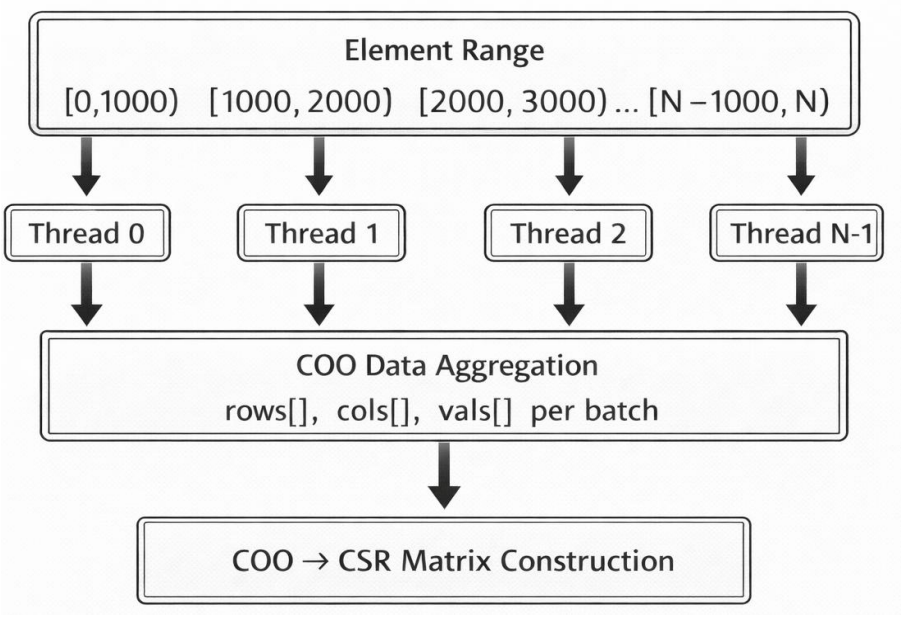

Figure 4: *CPU multithreading approach (ThreadPoolExecutor): the mesh is partitioned into batches, each processed by a thread to compute element contributions and assemble the global system.*

Each thread operates independently on a contiguous range of elements, computing local stiffness contributions and storing results in thread-local buffers.

3.3.3.2 Element Batch Processing

Each batch computes stiffness matrices and load contributions for a subset of elements and stores results in pre-allocated arrays using COO (Coordinate) format.

Key steps include:

1. Pre-allocation of output arrays for rows, columns, and values
2. Sequential processing of elements within the batch
3. Computation of local stiffness matrices using NumPy operations
4. Storage of local contributions in thread-local COO arrays

This design avoids shared writings during assembly and minimizes synchronization.

3.3.3.3 Parallel Assembly Orchestration

The main assembly routine dispatches batches to worker threads using a thread pool. Results are collected asynchronously, allowing faster threads to return without blocking on slower batches. After all threads are completed, individual COO arrays are concatenated and converted to CSR format.

3.3.3.4 COO-Based Global Assembly

Unlike the baseline implementation, which performs incremental insertion into a LIL matrix, this implementation assembles the global stiffness matrix using COO format:

Aspect	Baseline (LIL)	Threaded (COO)
Thread safety	Not thread-safe	Naturally thread-safe
Insertion pattern	Incremental	Batched
Duplicate handling	Explicit	Automatic on CSR conversion
Parallel suitability	Poor	High

The final COO → CSR conversion automatically merges duplicate entries arising from shared nodes between elements.

3.3.3.5 Post-Processing Parallelization

Derived field computation (velocity and magnitude) follows the same batch-based threading strategy. Each thread processes a disjoint subset of elements and writes results into non-overlapping regions of the output arrays, avoiding data races.

3.3.3.6 Linear System Solution

The linear solver is identical to the CPU baseline. SciPy's Conjugate Gradient solver is used with the same preconditioning and convergence criteria. No Python-level threading is applied to the solver phase, as SciPy internally manages optimized numerical kernels and threading via BLAS libraries.

3.3.4. Optimization Techniques Applied

Several optimizations are applied to improve threaded performance by reducing overhead and maximizing time spent in GIL-free NumPy kernels.

- Batch size selection:
  - Batch size controls the trade-off between scheduling overhead and load balance
  - Empirical testing shows best results for ~500-2000 elements per batch
- Pre-allocation of thread-local buffers:
  - Fixed-size arrays are allocated once per batch/thread invocation
  - Avoid repeated dynamic allocations inside inner loops, improving cache locality
- Inlined element computation:
  - Stiffness computation is implemented directly inside the batch function
  - Minimizes function call overhead and increases time spent in GIL-released NumPy kernels
- Shared read-only data:
  - Mesh coordinates, connectivity, and quadrature data are shared across threads as read-only arrays
  - Avoids memory duplication while ensuring thread safety

3.3.5. Challenges and Limitations

The threaded implementation improves assembly throughput but remains fundamentally limited by GIL-bound coordination and shared-memory resource contention.

Limitation	Impact
GIL contention	Python loops, indexing, and sparse manipulation remain serialized, limiting scalability
Memory bandwidth saturation	Threads contend for the same memory subsystem, causing diminishing returns beyond a few threads
Thread management overhead	Task submission/scheduling/result aggregation becomes significant, especially for small meshes
Limited solver parallelism	Solver remains effectively sequential at Python level; BLAS threading offers limited gains due to memory-bound behavior

3.3.6. Performance Characteristics and Role

Thread-level parallelism provides sub-linear speedup constrained by Amdahl's Law, since only parts of assembly and post-processing can benefit from concurrency.

- Expected scaling:
  - speedup is limited by GIL-bound coordination and sequential solver phases
  - only a fraction of total runtime is effectively parallelizable
- Practical speedup regime (empirical):
  - modest gains with 2-4 threads
  - diminishing returns beyond 4-8 threads
  - possible slowdowns when contention/overhead dominates
- Role in the implementation suite:
  - serves as an intermediate step between the sequential baseline and stronger parallel approaches
  - makes explicit the structural limitations imposed by the GIL, motivating designs that bypass it (multiprocessing or GPU)

3.3.7. Summary

The CPU Threaded implementation demonstrates both the potential and limitations of Python threading for numerical computation:

Achievements:

- Introduced parallelism without external dependencies
- Developed batch processing pattern reusable in other implementations
- Identified COO assembly as thread-safe alternative to LIL
- Established baseline for comparing more aggressive parallelization

Limitations:

- GIL contention limits achievable speedup
- Memory bandwidth shared across threads
- Python-level overhead remains significant
- Scaling plateaus at modest thread counts

Key Insight: For FEM workloads with significant per-element Python overhead, threading provides limited benefit. True parallelism requires either bypassing the GIL (multiprocessing, Numba) or offloading to hardware with native parallelism (GPU).

The batch processing architecture developed here, however, establishes a pattern that transfers to more effective parallelization strategies in subsequent implementations.

3.4. Solver Implementation 3: CPU Multiprocess

3.4.1. Overview

The CPU Multiprocess implementation achieves true parallelism by using process-based parallel execution. Unlike threading, multiprocessing bypasses the Global Interpreter Lock (GIL) entirely, enabling genuine concurrent execution across CPU cores. This comes at the cost of increased inter-process communication (IPC) and memory duplication.

Attribute	Description
Technology	multiprocessing.Pool (Python stdlib)
Execution Model	Multi-process, separate memory spaces
Role	True CPU parallelism and GIL bypass demonstration
Dependencies	NumPy, SciPy, multiprocessing (stdlib)

3.4.2. Technology Background

Python multiprocessing achieves parallelism by spawning multiple independent worker processes. Each process runs its own Python interpreter with an isolated memory space and its own GIL, avoiding GIL contention and enabling true CPU parallelism.

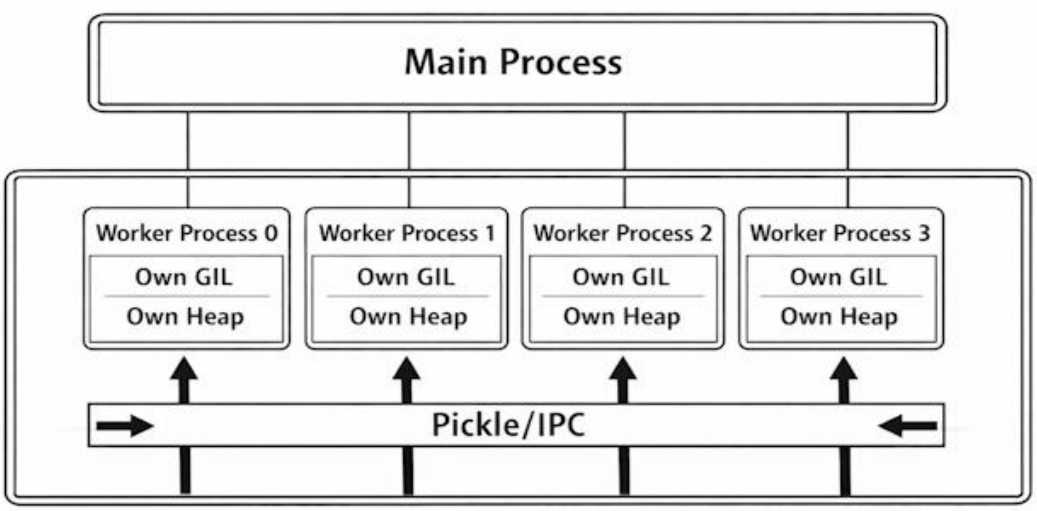

Figure 5: *CPU multiprocessing model: element batches are distributed across independent worker processes, bypassing the GIL at the cost of higher coordination and memory overhead.*

Multiprocessing model

- Separate memory: each process has an isolated address space
- Independent GIL: no GIL contention between processes
- IPC required: data must be transferred via serialization
- Higher overhead: process creation and coordination are more expensive than threads

multiprocessing.Pool execution

- Pool manages a fixed number of worker processes and distributes work using mapping primitives:

Method	Behavior	Ordering
map()	Blocking, returns list	Preserved
map_async()	Non-blocking	Preserved
imap()	Lazy iterator	Preserved
imap_unordered()	Lazy iterator	Arbitrary

Pickle serialization (IPC)

- Data transfer relies on pickle serialization:
  - input arguments are serialized and sent to workers
  - return values are serialized and returned to the main process
- Large NumPy arrays can introduce significant overhead.

Relevance for FEM workloads

Aspect	Threading	Multiprocessing
GIL impact	Serializes Python bytecode	None
Memory	Shared	Duplicated per process
Startup cost	Low	High
Communication	Direct memory access	Pickle serialization
Scalability	Limited by GIL	Limited by cores and IPC

For element-independent FEM assembly, multiprocessing can provide near-linear speedup if IPC costs are amortized.

3.4.3. Implementation Strategy

The multiprocessing implementation follows a batch-parallel execution model, where independent element batches are processed by separate worker processes. This enables true CPU parallelism but introduces additional constraints and IPC overhead.

- Module-level function requirement:
  - worker logic must be defined at module scope to be serializable (picklable)
  - computational kernels and batch-processing logic are therefore implemented at top-level scope
- Batch processing architecture:
  - the global element set is partitioned into contiguous batches
  - each batch is processed independently by a worker process
  - batches include the element range and required FEM data (coordinates, connectivity, quadrature)
  - batching amortizes IPC overhead and reduces scheduling frequency
- Data serialization implications (IPC overhead):
  - unlike threading, multiprocessing requires explicit data transfer per batch

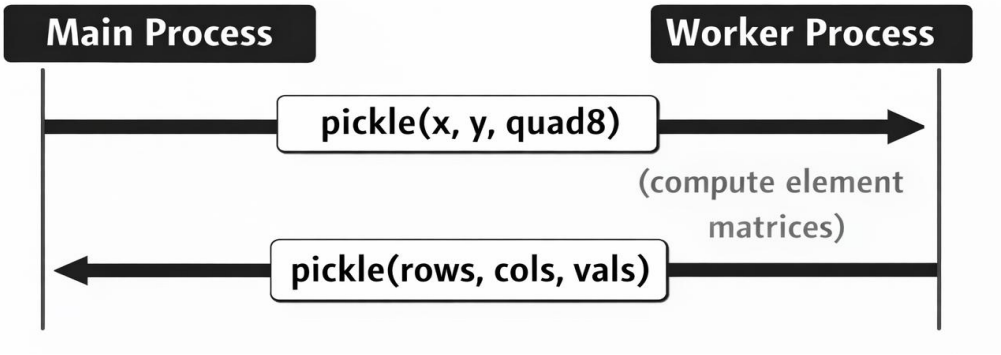

Figure 6: *Data serialization in multiprocessing: input mesh data and batch results must be transferred between processes (pickle/IPC), which can become a major overhead for large meshes.*

For large meshes, serialization volume and frequency become major performance constraints

- **COO assembly strategy**:
    - workers produce thread/process-independent COO contributions
    - the main process concatenates partial COO results
    - COO → CSR conversion automatically merges duplicates
    - avoids concurrent updates to shared sparse structures
- **Post-processing**:
    - derived field computation uses the same batching strategy
    - the solution field must also be serialized to workers, increasing IPC overhead
- **Linear system solution**:
    - executed in the main process using the same solver configuration as other implementations
    - ensures consistent convergence behavior and numerical equivalence

3.4.4. Optimization Techniques Applied

The multiprocessing implementation focuses on reducing IPC overhead and ensuring safe parallel sparse assembly.

- Batch size for IPC amortization:
    - larger batches reduce IPC frequency but reduce load-balancing flexibility

Batch Size	Batches (100K elements)	IPC Transfers	IPC Overhead
100	1000	2000	Very High
1000	100	200	Medium
5000	20	40	Low
10000	10	20	Very Low

- Tuple-based argument packing:
    - all required batch data is packed and transferred together
    - simplifies orchestration but increases serialization cost per task
- COO assembly for parallel safety:
    - each worker generates independent COO outputs (no shared-state writes)
    - duplicate handling is deferred to the final sparse matrix conversion
- Worker count configuration:

- worker count typically matches the number of CPU cores
- maximizes parallelism but increases memory duplication and IPC traffic ### 3.4.5. Challenges and Limitations

The multiprocessing implementation enables true CPU parallelism but is heavily constrained by IPC, serialization, and memory duplication overhead.

Limitation	Impact
Serialization overhead (pickle)	Dominates runtime, as input/output must be serialized per batch (small batches worsen this)
Memory duplication	Each worker holds private copies of mesh data, significantly increasing total memory footprint
Process startup cost	High fixed overhead from spawning processes, interpreter initialization, and pool creation
Limited shared state	Results must be merged in the main process, introducing a sequential aggregation phase
Pickle constraints	Worker functions must be serializable, restricting structure and increasing implementation complexity

Memory duplication model

Each worker process holds a private copy of input data:

Total Memory ≈ Main Process + N_workers × (coord arrays + connectivity)

For a 100K node mesh with 8 workers:

- Main process: ~10 MB
- Workers: ~80 MB
- Total: ~90 MB (vs. ~10 MB for threading)

Startup overhead example

Component	Typical Time
Fork/spawn	10-50 ms per process
Interpreter initialization	50-100 ms per process
Pool creation (4 workers)	200-500 ms

3.4.6. Performance Characteristics

Multiprocessing provides true CPU parallelism, but overall speedup depends on whether computation is large enough to amortize IPC and process management overhead.

Scaling model

[T_{parallel} = + T_{overhead}]

- (T_{serial}): sequential computation time
- (N): number of worker processes
- (T_{overhead}): IPC + process management overhead

Break-even behavior

Elements	Computation Time	Overhead (8 workers)	Benefit
1,000	~0.1 s	~0.5 s	Negative
10,000	~1 s	~0.5 s	Marginal
50,000	~5 s	~0.6 s	Good
100,000	~10 s	~0.7 s	Excellent

Practical limitations

- all processes share the same memory subsystem, so bandwidth saturation/NUMA effects can limit scaling
- compared to threading:
  - better scalability for large problems
  - worse performance for small problems
  - higher memory consumption

3.4.7. Summary

The CPU Multiprocess implementation demonstrates true parallel execution by bypassing Python's GIL through process-based parallelism:

Achievements:

- Genuine concurrent execution across CPU cores
- Near-linear speedup for large problems
- Validated COO assembly pattern for parallel safety
- Identified serialization as the primary overhead

Limitations:

- High memory usage from data duplication
- Significant IPC overhead for small/medium problems
- Code constraints from pickle requirements
- Process startup latency

Key Insight: Multiprocessing trades memory and communication overhead for true parallelism. It excels for large, compute-bound problems where the element loop dominates, but the overhead makes it less suitable for smaller problems or memory-constrained environments.

Comparison with Threading:

Criterion	Winner
Small problems (<10K elements)	Threading
Large problems (>50K elements)	Multiprocessing
Memory efficiency	Threading
Maximum speedup potential	Multiprocessing

The next implementation (Numba JIT) explores an alternative approach: instead of working around the GIL through separate processes, it compiles Python to native code that releases the GIL during execution, combining the benefits of shared memory with true parallelism.

3.5. Solver Implementation 4: Numba JIT CPU

3.5.1. Overview

The Numba JIT CPU implementation leverages Just-In-Time compilation to translate Python code into optimized native machine code at runtime. By compiling element-level FEM kernels and enabling parallel execution through Numba's prange construct, this approach achieves true multi-threaded execution while preserving shared-memory semantics.

This implementation combines the low memory overhead of shared-memory execution with performance characteristics close to compiled languages, eliminating Python interpreter overhead from the dominant FEM assembly and post-processing phases.

Attribute	Description
Technology	Numba JIT compiler with LLVM backend
Execution Model	JIT-compiled, multi-threaded shared memory
Role	High-performance CPU parallel execution
Dependencies	NumPy, SciPy, Numba

3.5.2. Technology Background

Numba provides Just-In-Time (JIT) compilation by translating Python functions into optimized machine code using the LLVM infrastructure. This removes Python interpreter overhead and enables near-native CPU execution.

Just-In-Time (JIT) compilation with Numba

- Eliminates Python interpreter overhead
- Native performance comparable to C/Fortran
- Enables compiler optimizations (inlining, loop optimizations, SIMD via LLVM)
- Executes without typical GIL constraints in compiled code

@njit compilation model

- The implementation uses @njit to enforce *nopython* mode:
  - Python bytecode is bypassed
  - Types are inferred at compile time
  - Unsupported Python/NumPy features are disallowed
- Compilation caching is enabled to amortize compilation cost across executions.

Parallel execution with **prange**

- Loop parallelism is implemented using prange:
  - Execution occurs without GIL limitations
  - Threads operate in shared memory
  - OpenMP-style work distribution
- Near-linear speedup is possible for independent iterations.

Relevance for FEM workloads

- JIT compilation targets key FEM bottlenecks:

  - Element stiffness matrix computation
  - Element-level assembly loops
  - Derived field computation
- Sparse matrix construction and solvers remain in SciPy, preserving numerical equivalence with previous approaches.

3.5.3. Implementation Strategy

The Numba implementation moves all element-level FEM computation into JIT-compiled kernels, minimizing interpreter overhead and enabling parallel execution through prange.

- Function-level JIT compilation:
  - all computational kernels are compiled with Numba in *nopython* mode
  - stiffness computation, boundary contributions, and post-processing are implemented using loop-based formulations
  - ensures element-level computation runs fully in compiled code (no Python interpreter overhead)
- Parallel element assembly (**prange**):
  - global assembly is performed as a parallel loop over elements
  - each iteration:
    1. gathers element nodal coordinates
    2. computes local stiffness matrix + load vector
    3. writes contributions into pre-allocated COO arrays
  - element independence enables safe parallelism and near-linear scaling
- Explicit loop-based kernels:
  - operations are written as explicit loops (not vectorized NumPy) to maximize LLVM optimizations:
    - loop unrolling for small fixed-size loops
    - inlining and reduced overhead
    - fewer temporary allocations
    - SIMD vectorization of inner loops
- Parallel post-processing:
  - derived field computation follows the same compiled-parallel pattern
  - each element evaluates gradients and stores results in element-wise output arrays
- Solver integration:
  - Numba generates COO-format data, while SciPy performs sparse matrix construction and solution (CG)
  - the JIT boundary is placed at the array level to preserve numerical equivalence with previous implementations

3.5.4. Optimization Techniques Applied

The Numba JIT implementation improves performance by eliminating interpreter overhead and enabling compiler-level optimizations.

- Interpreter elimination:
  - Python interpreter overhead is removed from element-level computation
  - inner loops execute as native machine code

- Loop unrolling and inlining:
  - small fixed-size loops are unrolled by LLVM
  - nested function calls in compiled code are typically inlined, reducing call overhead
- SIMD vectorization:
  - LLVM applies SIMD vectorization to inner arithmetic loops when possible
  - enables multiple operations per CPU cycle
- Memory access optimization:
  - COO output is written sequentially in element-major order
  - improves cache locality and reduces write overhead
- Shared-memory parallelism:
  - parallel execution uses shared memory without data duplication
  - preserves memory efficiency compared to multiprocessing approaches

3.5.5. Challenges and Limitations

While Numba JIT significantly accelerates element-level FEM computation, overall performance and usability are constrained by compilation cost, language limitations, and solver dominance at scale.

Limitation	Impact
JIT compilation overhead	First execution incurs hundreds of milliseconds of compilation time; amortized for large runs and reduced by caching
Limited NumPy/SciPy support	Only a subset of NumPy works in *nopython* mode; unsupported features must be rewritten using explicit loops
Debugging complexity	Debugging compiled code is harder; stack traces are less informative and debuggers cannot step into JIT regions
Allocations inside parallel loops	Memory allocation in parallel regions adds overhead; performance improves by minimizing allocations
Solver dominance at scale	As assembly accelerates, the sparse solver becomes the main runtime bottleneck, limiting further speedup

3.5.6. Performance Characteristics and Role

The Numba JIT CPU implementation greatly accelerates element-level computation, shifting the runtime bottleneck toward the sparse solver.

Expected scaling

Stage	Scaling Behavior	Dominant Factor
Assembly	O(N_elements)	Compiled arithmetic
Post-processing	O(N_elements)	Compiled arithmetic
Linear system solution	O(iterations × nnz)	Sparse memory bandwidth
Boundary conditions	O(N_boundary)	Minor relative cost

Profiling observations (large meshes)

- assembly and post-processing are reduced by **1-2 orders of magnitude**
- the solver becomes dominant in total runtime
- parallel efficiency remains high until **memory bandwidth saturation**

Role in the implementation suite

- highest-performing CPU-based solution in this study
- defines the practical upper bound for shared-memory CPU execution
- serves as the main CPU reference when evaluating GPU implementations

3.5.7. Summary

The Numba JIT CPU implementation eliminates Python interpreter overhead and enables true shared-memory parallelism for FEM assembly and post-processing.

Key observations include:

- Explicit loop-based kernels outperform vectorized NumPy formulations
- True parallel execution is achieved without GIL constraints
- Memory efficiency is preserved relative to multiprocessing
- Sparse solver performance ultimately limits end-to-end speedup

This implementation provides the most efficient CPU-based execution model in the study and forms a natural transition toward GPU-based acceleration.

3.6. Solver Implementation 5: Numba CUDA

3.6.1. Overview

The Numba CUDA implementation extends the FEM solver to GPU execution using Numba's @cuda.jit decorator, enabling the definition of GPU kernels using Python syntax. This approach provides access to massive GPU parallelism while avoiding direct CUDA C/C++ development, offering a balance between development productivity and performance.

Element-level FEM computations are offloaded to the GPU using a one-thread-per-element mapping, while sparse linear system solution is performed on the GPU using CuPy's sparse solvers.

Attribute	Description
Technology	Numba CUDA (@cuda.jit)
Execution Model	GPU SIMT execution
Role	GPU acceleration with Python-native kernels
Dependencies	NumPy, SciPy, Numba, CuPy

3.6.2. Technology Background

Numba extends its JIT compilation framework to NVIDIA GPUs through the @cuda.jit decorator. CUDA kernels are compiled to PTX and executed on the GPU, enabling massive parallelism using the CUDA SIMT model (many lightweight threads executing the same kernel concurrently).

Numba CUDA programming model

- @cuda.jit compiles Python functions into GPU kernels (PTX code).
- Execution follows the CUDA SIMT model, suited for thousands of parallel threads.

CUDA execution hierarchy

- GPU kernels launch threads using a hierarchical structure:
  - Grid: all threads launched by a kernel
  - Block: group of threads with cooperation via shared memory
  - Thread: smallest execution unit
  - Warp: 32 threads executing in lockstep
- Threads are indexed with cuda.grid(1), enabling direct mapping:
  - thread index ↔ FEM element index**

GPU memory hierarchy

- Memory is organized in tiers:
  - Registers (fastest, thread-private)
  - Local memory (thread-private, may spill)
  - Shared memory (fast, block-shared)
  - Global memory (large, high latency)
- The implementation typically uses:
  - registers/local memory for element-level arrays
  - global memory for mesh input data and assembled outputs

Relevance for FEM workloads

- GPUs are effective for FEM with many independent elements.
- Element stiffness computation has high arithmetic intensity and low dependency, making it well-suited for SIMT execution.

3.6.3. Implementation Strategy

The GPU implementation offloads FEM assembly and post-processing to CUDA kernels, using a one-thread-per-element mapping to exploit massive parallelism while avoiding inter-thread dependencies.

- Kernel-based element assembly:
  - assembly is implemented as a GPU kernel where each thread processes one element
  - per element, each thread:
    1. loads nodal indices and coordinates
    2. computes shape functions, Jacobians, and gradients
    3. assembles the local stiffness matrix and load vector
    4. writes results to global memory
  - computations use explicit loops compatible with Numba CUDA
- Thread-to-element mapping:
  - 1D grid launch, one thread per element
  - extra threads exit early when the element index exceeds mesh size
  - enables uniform work distribution without synchronization during element evaluation
- Local memory usage:
  - per-thread temporary arrays are stored using cuda.local.array:
    - DOF indices, coordinates
    - local stiffness matrix and load vector

        - shape functions and derivatives
    - thread-private memory avoids race conditions and synchronization overhead
- Force vector assembly (atomics):
    - shared nodes require thread-safe accumulation
    - global force vector is assembled using cuda.atomic.add to ensure correctness
- Post-processing on GPU:
    - derived fields are computed in a separate GPU kernel
    - each thread evaluates gradients and stores element-wise averaged results
- Solver integration:
    - the linear system is solved on the GPU using CuPy sparse Conjugate Gradient (CG)
    - sparse matrices are converted to CuPy formats and the solution phase runs fully on GPU before copying results back to CPU memory

3.6.4. Optimization Techniques Applied

The Numba CUDA implementation applies GPU-focused optimizations to maximize throughput and reduce memory/control-flow inefficiencies.

- Massive parallelism:
    - GPU executes tens of thousands of threads concurrently
    - enables element-level parallelism far beyond CPU core counts
- Block size tuning:
    - kernel launch configuration is tuned for occupancy vs. register pressure
    - 128 threads per block provides good performance for register-heavy FEM kernels
- Memory coalescing:
    - memory access patterns are structured so consecutive threads access contiguous memory
    - improves global memory bandwidth utilization
- Register and local memory management:
    - small per-thread arrays are kept in registers when possible
    - larger arrays may spill to local memory but remain thread-private and cached efficiently
- Warp divergence minimization:
    - control flow minimizes conditional branches
    - aside from bounds checks, threads follow identical execution paths

3.6.5. Challenges and Limitations

The Numba CUDA implementation provides high element-level parallelism but introduces GPU-specific constraints related to debugging, limited language support, atomic overhead, and CPU-GPU data movement.

Limitation	Impact
Debugging complexity	GPU kernel debugging is difficult; Python debuggers cannot be used and failures may be silent or cause kernel crashes
Limited NumPy support	Only a restricted subset is supported inside kernels; unsupported operations must be rewritten with explicit loops/arithmetic
Atomic operation	atomicAdd introduces serialization in force vector accumulation and may

overhead	become a bottleneck for higher connectivity
Memory transfer overhead	Host-device transfers (PCIe) add overhead; for small meshes transfer cost can dominate runtime
Partial CPU-GPU workflow	Some steps remain on CPU (e.g., COO index generation, boundary conditions), reducing end-to-end GPU acceleration

3.6.6. Performance Characteristics and Role

The Numba CUDA implementation accelerates element-level FEM computation on GPU, with end-to-end performance increasingly dominated by solver bandwidth and host-device transfers.

Expected scaling

Stage	Scaling Behavior	Dominant Factor
Element assembly	O(N_elements)	GPU throughput
Post-processing	O(N_elements)	GPU throughput
Linear system solution	O(iterations × nnz)	GPU memory bandwidth
Data transfer	O(N)	PCIe bandwidth

Profiling observations (large meshes)

- GPU occupancy typically reaches ~50-75%
- assembly achieves substantial speedup relative to CPU JIT
- sparse solver becomes memory-bandwidth bound
- end-to-end speedup improves as problem size increases

Role in the implementation suite

- first GPU-based implementation in the study
- validates GPU acceleration using Python-native kernels (Numba CUDA)
- serves as reference for comparison with raw CUDA (CuPy RawKernel) approaches

3.6.7. Summary

The Numba CUDA implementation enables GPU acceleration of FEM assembly and post-processing using Python syntax:

Key observations include:

- Thousands of GPU threads execute element computations concurrently
- Python-based kernel development significantly reduces development effort
- Performance approaches that of hand-written CUDA kernels
- Atomic operations and memory transfers limit scalability for smaller problems

This implementation represents a practical and accessible entry point for GPU acceleration, bridging the gap between CPU-based JIT execution and fully optimized raw CUDA implementations.

3.7. Solver Implementation 6: GPU (CuPy)

3.7.1. Overview

The GPU CuPy implementation represents the most performance-oriented approach, using CuPy's RawKernel to execute hand-written CUDA C kernels directly on the GPU. This provides maximum control over GPU execution while leveraging CuPy's ecosystem for sparse matrix operations and iterative solvers.

Attribute	Description
Technology	CuPy RawKernel (CUDA C), CuPy sparse
Execution Model	GPU SIMT with native CUDA C kernels
Role	Maximum GPU performance, production-quality implementation
Dependencies	NumPy, SciPy, CuPy

3.7.2. Technology Background

CuPy is a NumPy-compatible GPU array library that enables accelerated numerical computing using NVIDIA GPUs. It provides GPU-resident arrays, sparse matrix support, and iterative solvers running directly on the GPU.

CuPy overview

- Drop-in NumPy replacement (import cupy as cp) with a similar API
- GPU arrays stored in GPU memory (VRAM)
- Sparse matrices in CSR/CSC/COO formats on GPU
- GPU iterative solvers (e.g., CG, GMRES)
- RawKernel interface for custom CUDA C/C++ kernels

RawKernel execution model

- RawKernel embeds CUDA C/C++ code directly in Python, enabling:
  - full CUDA feature set
  - maximum performance (no Python overhead in the kernel)
  - explicit control over memory, synchronization, and shared memory
- Kernels are compiled once and cached for reuse.

Comparison with Numba CUDA

Aspect	Numba CUDA	CuPy RawKernel
Kernel language	Python	CUDA C/C++
Performance	~90-95% of peak	~100% of peak
Shared memory	Basic support	Full control
Warp primitives	Limited	Full access
Learning curve	Lower	Higher

GPU memory model

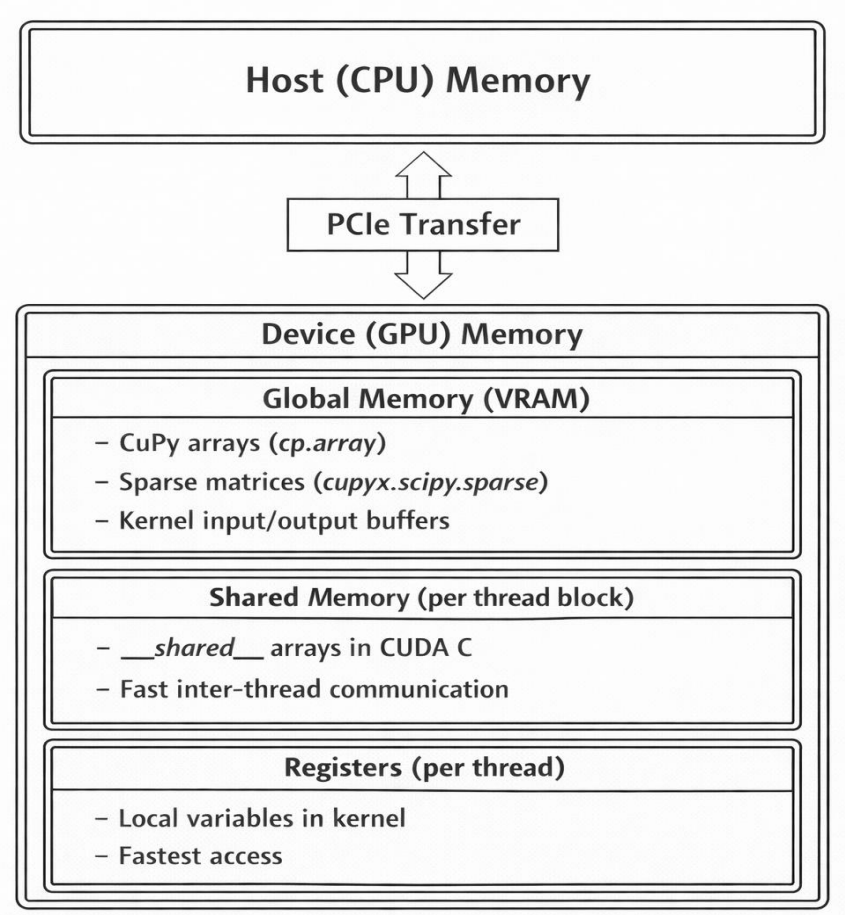

Figure 7: *GPU memory hierarchy: registers, shared memory, and global memory influence kernel performance through latency, bandwidth, and access patterns in FEM assembly and post-processing.*

3.7.3. Implementation Strategy

This implementation uses CuPy RawKernel to execute custom CUDA C kernels while keeping the full FEM pipeline GPU-resident, including assembly, sparse matrix construction, solving, and post-processing.

- CUDA C kernel architecture (RawKernel):
  - two primary kernels are embedded as CUDA C string literals:
    - **Assembly kernel** (quad8_assembly_kernel)
      - one thread per element
      - computes 8×8 stiffness matrix (64 values)
      - writes values to global COO value array
      - atomic accumulation into the global force vector
    - **Post-processing kernel** (quad8_postprocess_kernel)
      - one thread per element
      - evaluates velocity gradient at 4 Gauss points
      - averages to centroid velocity
      - writes velocity components and magnitude
- Kernel source structure (assembly):
  - thread index computed from blockIdx, blockDim, threadIdx
  - thread-local arrays for element data and local matrices

  - fixed quadrature/integration loops matching CPU formulation
  - scatter step writes flattened 8×8 values
  - force vector assembled via atomic updates
- GPU-accelerated COO index generation:
  - COO row/column indices are generated on GPU using vectorized CuPy ops:
    - creates all (N_{el}) indices in parallel
    - avoids CPU-side index generation and CPU-GPU synchronization
  - CUDA kernel computes only the COO values
- Sparse matrix construction on GPU:
  - build COO matrix with CuPy sparse
  - convert COO → CSR on GPU (duplicates merged automatically)
  - sparse matrix remains GPU-resident
- GPU sparse solver:
  - system solved fully on GPU using CuPy sparse solvers:
    - diagonal equilibration on GPU
    - Jacobi preconditioning via linear operator
    - Conjugate Gradient (CG) fully on GPU
    - de-equilibration on GPU

3.7.4. Optimization Techniques Applied

The CuPy RawKernel implementation applies CUDA C-level optimizations to maximize kernel efficiency and ensure solver robustness.

- Inline CUDA C shape function derivatives:
  - derivatives are computed directly inside the kernel using explicit CUDA C expressions
  - avoids function call overhead and enables compiler optimization
- Explicit Jacobian and inverse computation:
  - Jacobian, determinant, and inverse are computed inside the kernel using:
    - explicit loops over the 8 nodes
    - fixed-size operations suitable for compiler unrolling
    - direct 2×2 determinant and inverse evaluation
- Atomic force vector update:
  - nodal force accumulation uses CUDA atomics (atomicAdd)
  - ensures correctness when multiple elements contribute to shared nodes
- Solver fallback strategy:
  - attempts CG first
  - falls back to GMRES if CG fails
  - improves robustness under numerically difficult cases

3.7.5. Challenges and Limitations

The CuPy RawKernel approach delivers high performance but increases implementation complexity and introduces GPU-specific constraints.

Limitation	Impact

CUDA C complexity	Requires CUDA expertise (thread/memory hierarchy, occupancy, register pressure, divergence), increasing development cost
Debugging challenges	Harder than Python/Numba; limited tooling, silent crashes, and race conditions are difficult to diagnose
Kernel compilation overhead	First-run JIT compilation adds latency but is largely eliminated by CuPy kernel caching
GPU memory constraints	Large meshes may exceed VRAM due to sparse CSR storage and working buffers, limiting problem size
CuPy sparse solver limitations	Fewer/more limited solver and preconditioner options than SciPy; possible numerical differences mitigated via GMRES fallback

Kernel compilation cost (first use)

Kernel	Compilation Time
Assembly	~200-500 ms
Post-processing	~100-200 ms

GPU memory usage example (per 100K nodes)

Component	Memory
Coordinates (x, y)	~1.6 MB
Connectivity	~3.2 MB
Sparse matrix (CSR)	~50-100 MB
Solution vectors	~0.8 MB
Working memory	Variable

3.7.6. Performance Characteristics and Role

The CuPy RawKernel implementation achieves the highest GPU performance by combining custom CUDA C kernels with a fully GPU-resident sparse solve pipeline.

GPU utilization (typical)

Metric	Typical Value	Main Limiter
Occupancy	50–75%	Register pressure
Memory throughput	70–85% peak	Coalescing
Computing utilization	60–80%	Kernel efficiency

Performance breakdown (large problems)

Stage	Time Fraction	Notes
Mesh loading	<5%	I/O bound
Assembly kernel	5–15%	Highly parallel
Matrix construction	5–10%	CuPy sparse ops
Linear solve	60–80%	Memory-bandwidth bound
Post-processing	5–10%	Highly parallel
Data transfer	<5%	PCIe overhead

Scaling characteristics

Problem Size	GPU Advantage	Notes
<10K elements	Minimal	Transfer overhead dominates
10K–100K	Significant (5–20×)	Good GPU utilization
100K–1M	Maximum (20–100×)	Full GPU saturation
>1M	Memory limited	May require multi-GPU

Comparison with CPU approaches (high level)

Aspect	CPU Baseline	Numba CPU	GPU CuPy
Parallelism	1 core	multi-core	1000s of threads
Bandwidth	~50–100 GB/s	~50–100 GB/s	~500–900 GB/s
Latency	Low	Low	Higher (PCIe)
Throughput	Moderate	Good	Excellent

3.7.7. Summary

The GPU CuPy implementation with RawKernel represents the most performance-optimized endpoint of this implementation spectrum:

Key observations include:

- Native CUDA C kernels provide maximum GPU performance
- Full GPU-resident pipeline (assembly, solve, post-processing) minimizes PCIe transfers
- GPU-based COO index generation avoids CPU bottlenecks present in Numba CUDA
- Sparse solver dominates runtime once assembly is accelerated
- Development and debugging complexity is significantly higher than Numba CUDA

This implementation establishes the upper bound for single-GPU performance in this project and provides a production-quality reference design combining custom CUDA kernels with CuPy's sparse linear algebra ecosystem.

4. Performance Evaluation

4.1 Motivation and Scope

This section presents a systematic benchmark study of the finite element solver developed in this work, with the objective of **quantifying performance gains across execution models**, from conventional CPU-based implementations to fully GPU-resident solvers.

Rather than restricting the analysis to a single machine, the benchmark was designed as a **cross-hardware evaluation**, where identical solver implementations were executed on multiple systems equipped with different NVIDIA GPUs. This approach enables a clear separation between:

- algorithmic effects (assembly strategy, solver configuration), and
- hardware effects (CPU vs GPU, GPU architecture, memory bandwidth, VRAM capacity).

All implementations solve the *same mathematical problem* using the *same FEM formulation*, ensuring that observed differences arise exclusively from the execution model and underlying hardware.

4.2 Benchmark Objectives

The benchmark addresses the following key questions:

1. **CPU scaling limits**
   How far can performance be improved on CPU using:
   – threading,
   – multiprocessing, and
   – JIT compilation with Numba, before memory bandwidth and Python overhead become dominant?
2. **GPU acceleration impact**
   What is the performance gain when offloading:
   – element-level assembly,
   – sparse linear system solution, and
   – post-processing to the GPU using Numba CUDA and CuPy RawKernel?
3. **Cross-GPU scalability**
   How does solver performance scale across GPUs with different compute capabilities, memory bandwidth, and VRAM capacity?

4.3 Solver Variants Under Test

All benchmark runs use the same mesh, boundary conditions, numerical parameters, and convergence criteria. Only the execution backend changes.

Solver Variant	Execution Target	Description	Primary Role
CPU Baseline	CPU	Sequential NumPy/SciPy	Correctness reference
CPU Threaded	CPU	ThreadPool-based batching	Evaluate GIL-limited threading
CPU Multiprocess	CPU	Process-level parallelism	True CPU parallelism
Numba JIT (CPU)	CPU	@njit + prange	High-performance shared-memory CPU
Numba CUDA	GPU	Python CUDA kernels	GPU acceleration with Python kernels
GPU CuPy (RawKernel)	GPU	CUDA C kernels + CuPy sparse	Maximum single-GPU performance

This progression reflects a deliberate transition from interpreter-driven execution to compiled and accelerator-based computation.

4.4 Testing Environment

The experimental evaluation presented in this section constitutes the final performance assessment of the finite element solver implementations developed in this work. Benchmarks were conducted on a carefully selected set of computational servers and problem sizes, designed to capture the performance characteristics of CPU and GPU-based execution models across a representative range of hardware capabilities.

The selected systems span mid-range, high-end, and upper-bound GPU configurations, enabling a robust and comparative analysis of scalability, execution efficiency, and

architectural sensitivity. All experiments were performed using identical solver configurations, numerical parameters, and convergence criteria, ensuring that observed performance differences arise exclusively from the execution model and underlying hardware.

Participating Servers

The benchmark dataset was generated using the following computational servers:

#	Hostname	CPU	Cores	RAM	GPU	VRAM	Records
1	DESKTOP-B968RT3	AMD64 Family 25 Model 97 St…	12	-	NVIDIA GeForce RT…	15.9 GB	432
2	KRATOS	Intel64 Family 6 Model 183 …	28	-	NVIDIA GeForce RT…	12.0 GB	432
3	MERCURY	13th Gen Intel(R) Core(TM) …	20	94.3 GB	NVIDIA GeForce RT…	24.0 GB	432
4	RICKYROG700	Intel64 Family 6 Model 198 …	24	-	NVIDIA GeForce RT…	31.8 GB	432

Test Meshes

Model	Size	Nodes	Elements	Matrix NNZ
Backward-Facing Step	XS	287	82	3,873
Backward-Facing Step	M	195,362	64,713	3,042,302
Backward-Facing Step	L	766,088	254,551	11,965,814
Backward-Facing Step	XL	1,283,215	426,686	20,056,653
Elbow 90°	XS	411	111	5,029
Elbow 90°	M	161,984	53,344	2,502,276
Elbow 90°	L	623,153	206,435	9,692,925
Elbow 90°	XL	1,044,857	346,621	16,278,553
S-Bend	XS	387	222	4,031
S-Bend	M	196,078	64,787	3,048,716
S-Bend	L	765,441	254,034	11,947,139
S-Bend	XL	1,286,039	427,244	20,090,265
T-Junction	XS	393	102	5,357
T-Junction	M	196,420	64,987	3,057,464
T-Junction	L	768,898	255,333	12,012,244
T-Junction	XL	1,291,289	429,176	20,178,849
Venturi	XS	341	86	4,023
Venturi	M	194,325	64,334	3,023,465
Venturi	L	763,707	253,704	11,923,621
Venturi	XL	1,284,412	427,017	20,069,214
Y-Shaped	XS	201	52	2,571
Y-Shaped	M	195,853	48,607	2,287,756
Y-Shaped	L	772,069	192,308	9,044,929
Y-Shaped	XL	1,357,953	338,544	15,920,215

Solver Configuration

Parameter	Value
Problem Type	2D Potential Flow (Laplace)
Element Type	Quad-8 (8-node serendipity quadrilateral)
Linear Solver	CG
Tolerance	1e-08
Max Iterations	15,000,000
Preconditioner	Jacobi

Implementations Tested

#	Implementation	File	Parallelism Strategy
1	CPU Baseline	quad8_cpu_v3.py	Sequential Python loops
2	CPU Threaded	quad8_cpu_threaded.py	ThreadPoolExecutor (GIL-limited)
3	CPU Multiprocess	quad8_cpu_multiprocess.py	multiprocessing.Pool
4	Numba CPU	quad8_numba.py	@njit + prange
5	Numba CUDA	quad8_numba_cuda.py	@cuda.jit kernels
6	CuPy GPU	quad8_gpu_v3.py	CUDA C RawKernels

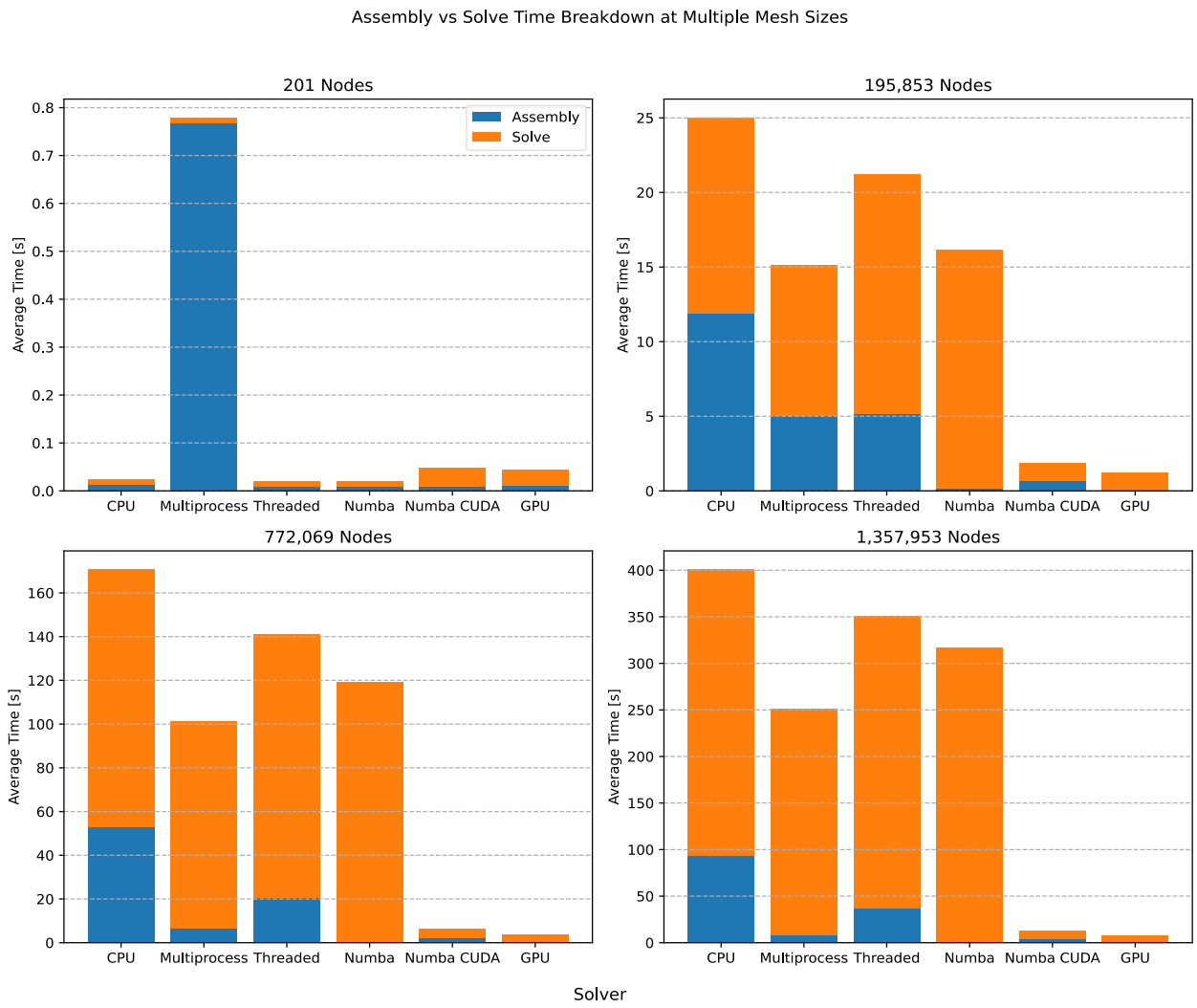

Figure 8: *Assembly vs. solve time breakdown across multiple mesh sizes and solver backends, highlighting how computational bottlenecks shift with problem scale.*

4.4.1 Assembly vs. Solve Time Breakdown Across Mesh Sizes

Figure 8 presents a detailed breakdown of total execution time into assembly and solve phases for all solver implementations, evaluated across four increasingly large mesh sizes. This decomposition is essential to understand not only which implementation is faster, but why performance changes with scale, revealing the underlying computational bottlenecks that dominate each regime.

For the smallest mesh (201 nodes), total runtimes are extremely short for all implementations, and performance is governed almost entirely by fixed overheads rather than sustained computation. The CPU baseline and lightweight threaded execution perform efficiently due to minimal setup costs, while multiprocessing exhibits a severe assembly penalty, clearly visible in the figure, caused by process spawning and inter-process communication overhead. GPU-based implementations (Numba CUDA and CuPy GPU) show relatively larger solve fractions despite low absolute runtimes, reflecting kernel launch latency and synchronization costs that cannot be amortized at this scale. These results confirm that accelerator-based execution is structurally inefficient for very small FEM problems, regardless of hardware capability.

At the intermediate mesh size (194,325 nodes), the performance profile enters a transition regime. Assembly time increases substantially for CPU-based implementations, especially for

the baseline solver, while the solve phase becomes the dominant contributor for most execution models. Threaded and multiprocessing approaches reduce assembly time relative to the baseline, but this primarily exposes the sparse solver as the new bottleneck rather than eliminating it.

Numba JIT significantly compresses assembly cost, making the solve phase overwhelmingly dominant. GPU-based solvers show a pronounced reduction in assembly time compared to CPU approaches; however, the solve phase remains substantial, indicating that performance is increasingly constrained by sparse linear algebra and memory access patterns rather than element-level computation.

For the large mesh (766,088 nodes), solver dominance becomes unequivocal. All CPU-based implementations spend most of their execution time in the iterative solver, with assembly contributing only a secondary fraction of the total cost—even when JIT compilation is employed. This reflects the inherently memory-bandwidth-bound nature of sparse matrix-vector operations on CPUs. In contrast, GPU implementations dramatically reduce assembly time to near-negligible levels and significantly lower overall solve time. The figure shows that GPU parallelism is highly effective at eliminating element-level bottlenecks; nevertheless, the solve phase remains the largest contributor even on the GPU.

This behavior is further reinforced for the largest mesh (1,357,953 nodes). Across all CPU execution models, runtime is almost entirely dominated by the solve phase, rendering additional assembly optimizations largely irrelevant. GPU-based solvers maintain minimal assembly costs and comparatively moderate solve times, but the solver still accounts for most of the execution time. The convergence of assembly times between GPU and Numba CUDA at this scale indicates that performance is governed primarily by memory bandwidth and sparse access patterns, rather than kernel-level computational throughput.

This analysis highlights that FEM performance optimization is inherently scale dependent. While CPU-level parallelism and JIT compilation provide meaningful gains at moderate sizes, they are insufficient to overcome the fundamental limitations of sparse linear algebra on CPUs. GPU acceleration effectively removes assembly as a bottleneck and substantially mitigates solver cost, making it the only viable strategy for large-scale problems. However, even on GPUs, further performance improvements must focus on solver algorithms, preconditioning strategies, and memory efficiency, rather than kernel-level optimizations alone.

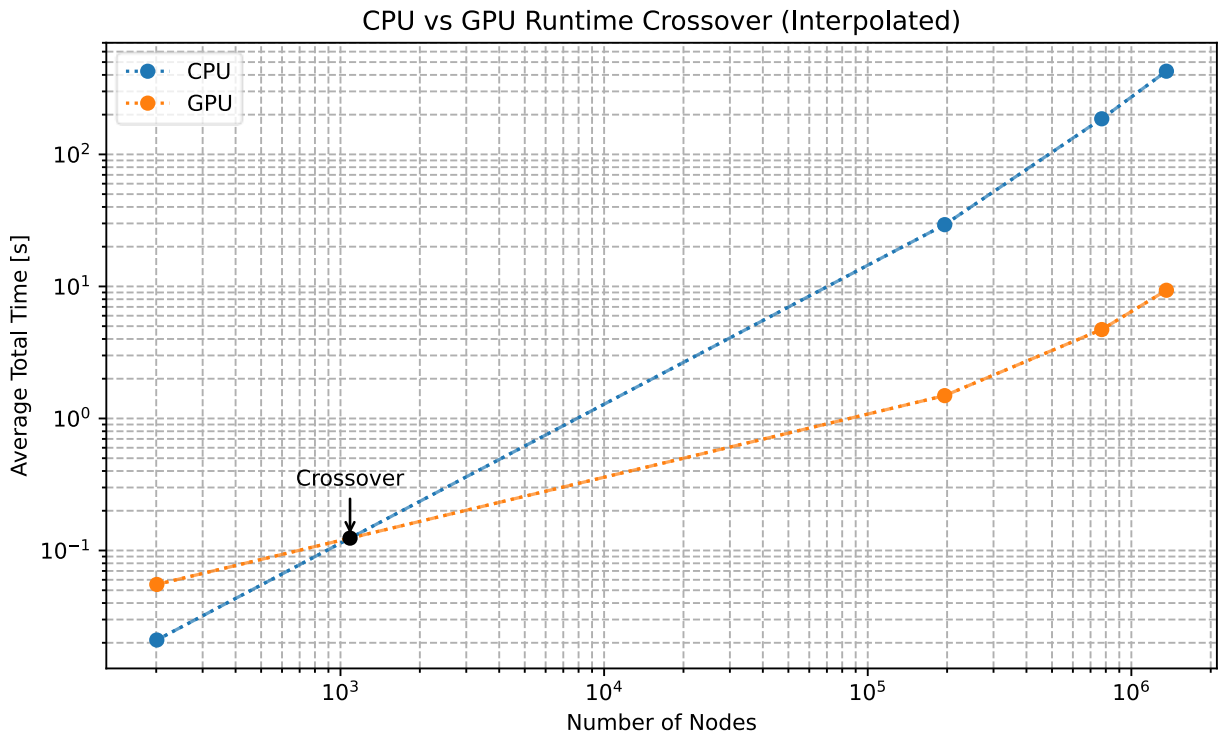

Figure 9: *Interpolated CPU–GPU runtime crossover as a function of problem size.*

4.4.2 CPU-GPU Runtime Crossover Analysis

Figure 9 presents the interpolated runtime crossover between CPU-based and GPU-based solver executions as a function of problem size. This analysis aims to identify the **break-even point** at which GPU acceleration becomes consistently advantageous over CPU execution, providing a quantitative criterion for hardware-aware solver selection.

At small problem sizes, the CPU implementation exhibits lower total runtime, which is primarily explained by its minimal startup overhead. GPU-based execution, while massively parallel, incurs fixed costs related to kernel launch, device synchronization, and data movement between host and device memory. In this regime, these overheads dominate total execution time, rendering GPU acceleration inefficient despite its superior theoretical throughput.

As the number of nodes increases, CPU runtime grows approximately linearly, reflecting the combined cost of element assembly and iterative sparse linear solves executed in a memory-bound environment. In contrast, the GPU runtime curve exhibits a much flatter slope. Once the problem size exceeds a critical threshold, the GPU is able to amortize its fixed overheads and exploit fine-grained parallelism across thousands of threads, leading to substantially better scalability.

The intersection point of the two curves defines the **CPU-GPU crossover region**, beyond which GPU execution consistently outperforms CPU execution. This crossover is not a single fixed value but rather a narrow interval, influenced by factors such as solver configuration, sparsity pattern, and memory access behavior. Importantly, this transition occurs well below the largest mesh sizes considered in this study, indicating that GPU acceleration is not merely beneficial for extreme-scale problems, but is already advantageous at moderately large FEM models.

Beyond the crossover point, the divergence between CPU and GPU runtimes increases rapidly. This behavior confirms that CPU-based solvers become increasingly constrained by memory

bandwidth and cache inefficiency, while GPU-based solvers sustain higher effective throughput due to wider memory interfaces and higher concurrency. The gap widens further as problem size grows, reinforcing the conclusion that CPUs do not scale favorably for large sparse FEM systems, even when augmented with threading or JIT compilation.

This crossover analysis provides a clear and actionable performance guideline: CPU execution is preferable only for small-scale problems, where overhead dominates, whereas GPU execution becomes the superior choice once the problem size exceeds the crossover threshold. This result complements the assembly-versus-solve breakdown by offering a global, hardware-agnostic perspective on performance scalability, and directly motivates the cross-GPU comparisons presented in the subsequent sections.

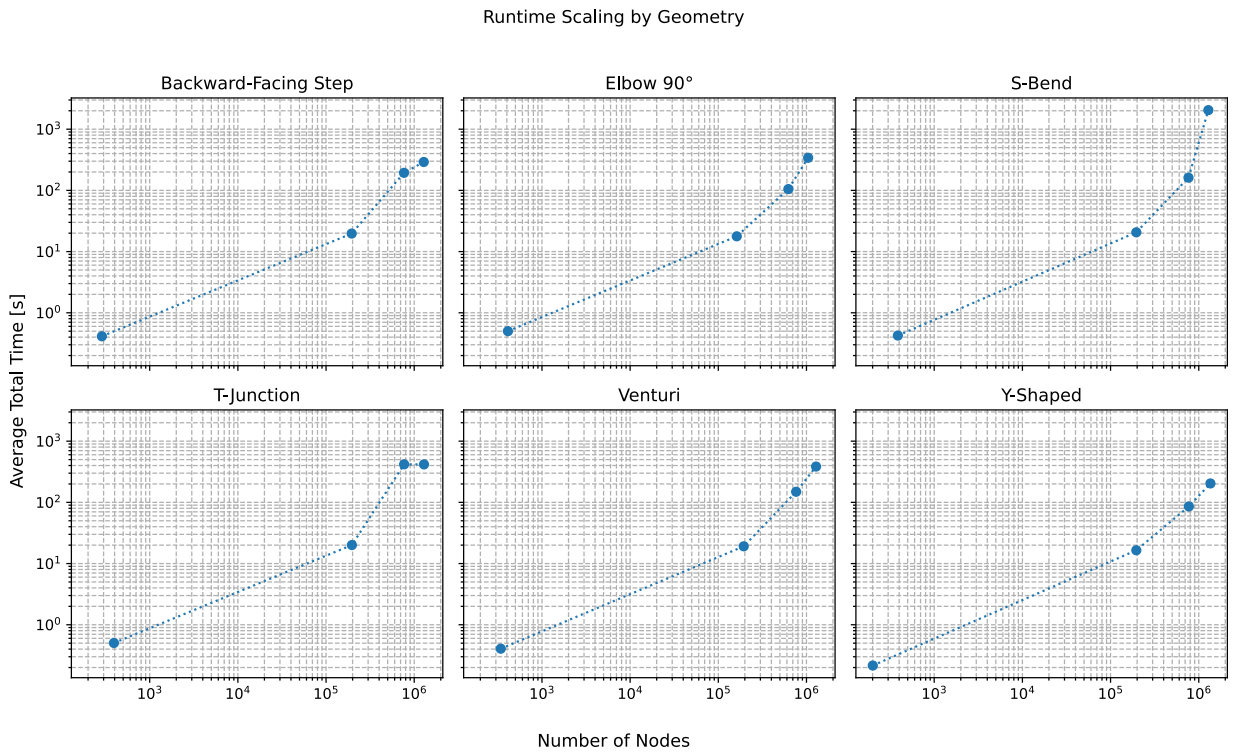

Figure 10: *Assembly and solve time breakdown for different solver strategies across multiple mesh sizes and geometries.*

4.4.3 Critical Analysis of Runtime Scaling and CPU-GPU Transition

The results presented in the previous figures reveal a clear and consistent transition in performance behaviour as problem size increases, highlighting the distinct computational regimes in which CPU-based and GPU-accelerated solvers operate.

For small-scale problems, CPU solvers—both sequential and parallel—exhibit competitive performance due to their low execution overhead and efficient handling of limited workloads. In this regime, the total runtime is dominated by fixed costs such as setup, memory allocation, and solver initialization, which reduces the relative benefit of parallel execution. Consequently, GPU-based solvers do not provide a measurable advantage for coarse meshes, as kernel launch overheads and data transfer costs outweigh the benefits of massive parallelism.

As the number of nodes increases, a progressive shift in computational dominance becomes evident. Assembly time grows approximately linearly with mesh size, while solver time increases more rapidly due to the expanding sparse linear system and its associated memory

access patterns. CPU-based solvers, including multithreaded and Numba JIT implementations, begin to exhibit limited scalability in this regime. Although parallelism mitigates some of the computational burden, performance becomes increasingly constrained by memory bandwidth and cache efficiency rather than raw compute capability.

Beyond an intermediate problem size, a distinct CPU-GPU crossover point is observed. At this stage, GPU-based solvers consistently outperform all CPU variants, with total execution time scaling more favourably as mesh resolution increases. This behaviour is primarily driven by the solver phase, where the GPU's high memory bandwidth and massive thread-level parallelism enable more efficient sparse matrix-vector operations. The assembly phase, while still relevant, becomes secondary in determining overall performance for large-scale simulations.

Importantly, the crossover point is not purely hardware-dependent but emerges from the interaction between problem size, algorithmic structure, and architectural characteristics. The results demonstrate that GPU acceleration becomes increasingly advantageous once the solver phase dominates runtime and parallel workload granularity is sufficient to amortize GPU overheads.

This analysis confirms that CPU-based approaches remain suitable for small and moderately sized problems, while GPU acceleration is essential for maintaining scalability in large-scale finite element simulations. The findings reinforce the importance of selecting solver strategies based on both problem size and computational architecture, rather than relying on a one-size-fits-all execution model.

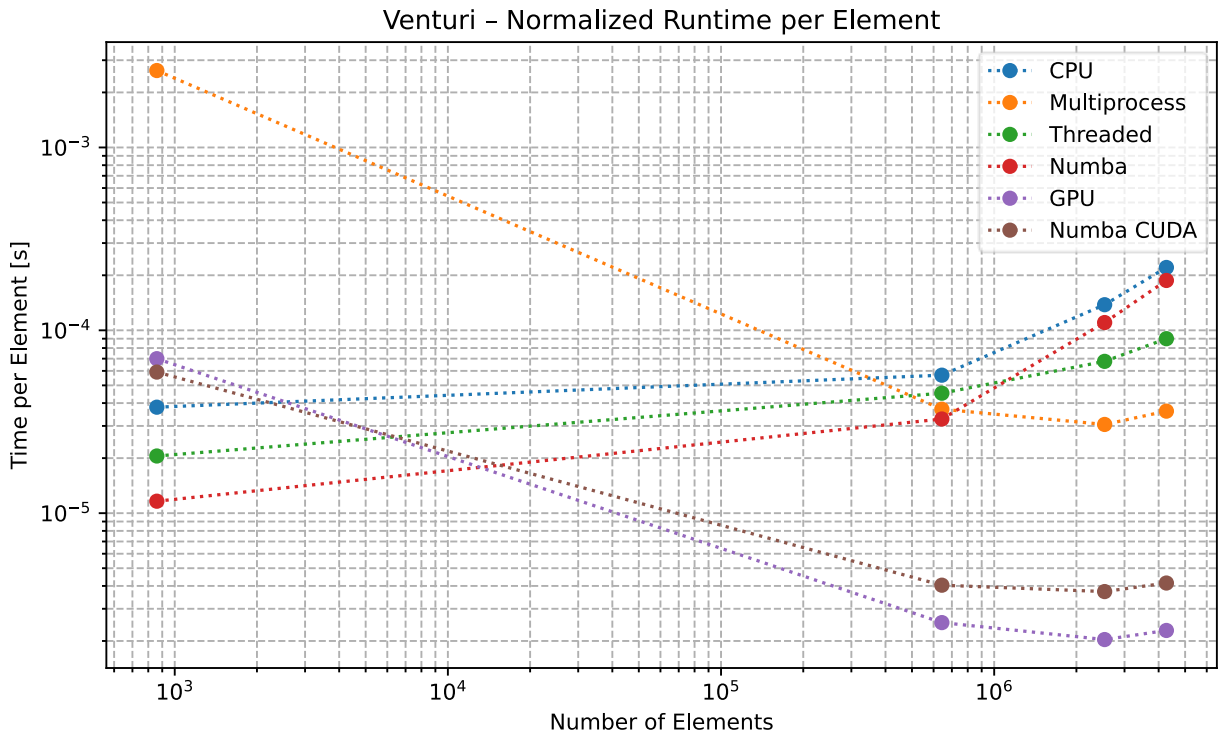

Figure 11: *Normalized runtime per element as a function of problem size (Venturi), highlighting scaling efficiency and the CPU–GPU crossover regime.*

4.4.3 Runtime Scaling and CPU-GPU Crossover Analysis

Figure 11 shows the normalized runtime per element as a function of the number of elements for the Venturi geometry, using logarithmic scales on both axes. This representation isolates

scaling efficiency from absolute runtime and provides a clearer view of how each execution model behaves asymptotically as problem size increases.

For small meshes (≈10^3 elements), the normalized runtime per element is relatively high and scattered across implementations. In this regime, fixed overheads dominate execution, and GPU-based solvers (Numba CUDA and CuPy GPU) exhibit significantly worse efficiency per element than CPU-based approaches. This behavior reflects kernel launch latency, memory transfer overhead, and GPU context management costs, which cannot be amortized when the computational workload per element is small. Lightweight CPU approaches, particularly Numba JIT, achieve the lowest per-element cost in this range due to minimal overhead and efficient compiled execution.

As the number of elements increases toward the mid-scale regime (≈10^5-10^6 elements), a clear change in scaling behavior emerges. CPU-based implementations show an increasing runtime per element, indicating deteriorating efficiency as sparse solver costs and memory bandwidth limitations begin to dominate. In contrast, GPU-based solvers exhibit a decreasing runtime per element, demonstrating improved amortization of overheads and more effective utilization of parallel hardware resources. This region corresponds to the CPU-GPU crossover, where GPU execution transitions from being overhead-bound to throughput-efficient.

Beyond the crossover point (≈10^6 elements), GPU implementations clearly dominate. Both CuPy GPU and Numba CUDA show the lowest and flattest curves, indicating near-optimal scaling where additional elements incur only marginal increases in per-element cost. This behavior highlights the advantage of massive thread-level parallelism and high memory bandwidth when handling large sparse systems. Among GPU approaches, CuPy consistently achieves slightly lower per-element runtimes than Numba CUDA, reflecting lower kernel abstraction overhead and more optimized execution paths.

CPU-based solvers, including multiprocessing and threading, display the opposite trend: their per-element runtime increases steadily with mesh size. This confirms that CPU execution becomes increasingly constrained by memory access patterns and sparse linear algebra operations, which do not scale favorably with core count alone. Multiprocessing shows particularly poor efficiency at small scales and only moderate improvement at larger sizes, underscoring the cost of inter-process communication.

Overall, the figure provides strong empirical evidence that GPU acceleration is essential for achieving scalable FEM performance at large problem sizes. While CPU-based solvers remain efficient and competitive for small meshes, their asymptotic behavior is fundamentally limited. GPU-based approaches, by contrast, demonstrate improving efficiency with scale and clearly superior asymptotic performance, making them the preferred execution model for high-resolution, production-scale finite element simulations.

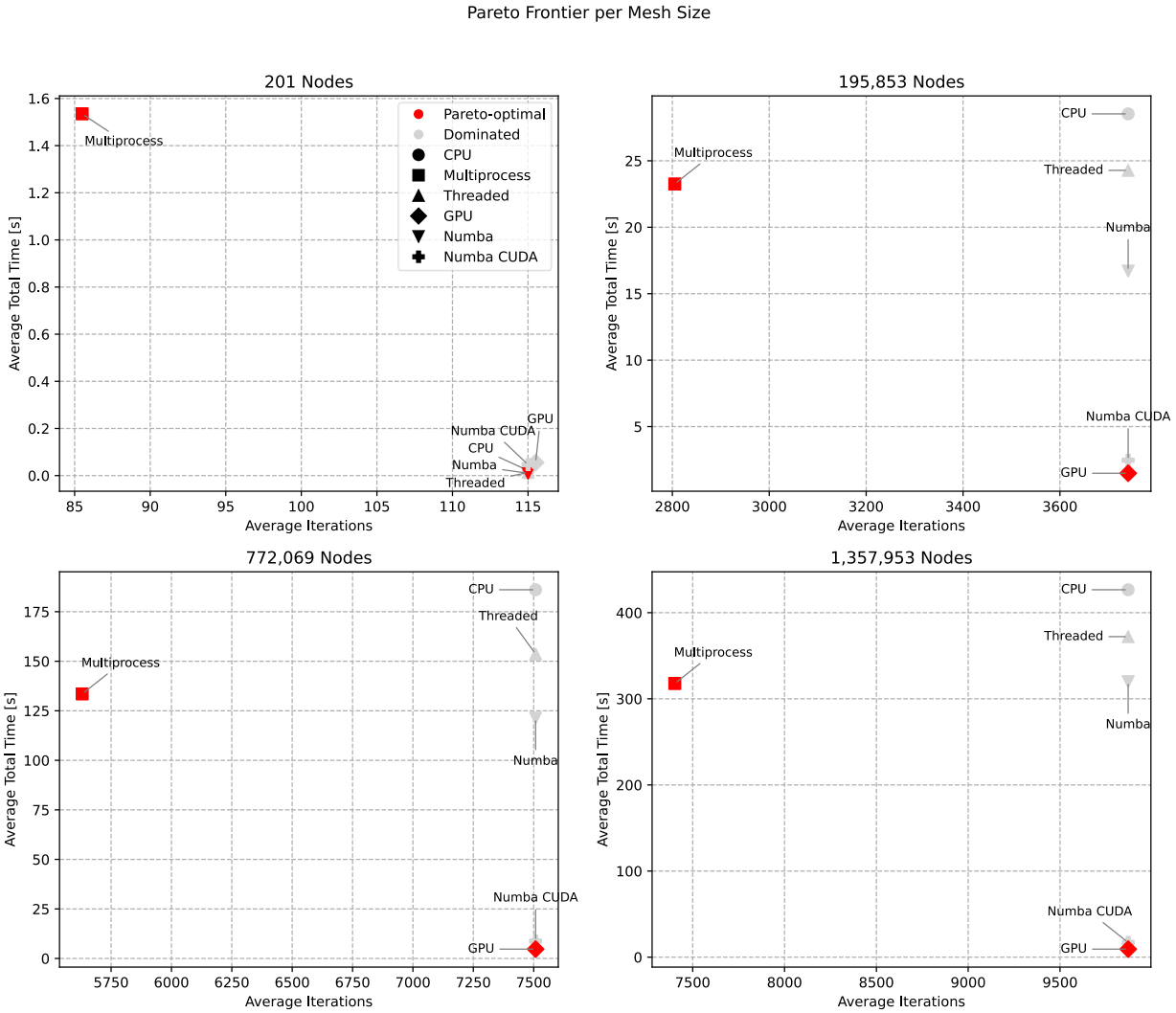

Figure 12: *Pareto frontier of average total runtime versus average solver iterations for different mesh sizes and execution models.*

4.4.4 Pareto-Based Performance Trade-off Analysis

Figure 12 presents a Pareto-based analysis of solver performance, explicitly relating average total runtime to average solver iteration count across four increasing mesh sizes. This representation provides a multidimensional view of efficiency, allowing runtime performance to be evaluated jointly with numerical effort, rather than in isolation.

For the smallest mesh (201 nodes), all implementations exhibit very similar iteration counts, confirming that convergence behavior is independent of the execution backend at this scale. Performance differences are therefore entirely driven by execution overhead. In this regime, the Pareto frontier is defined by CPU-based approaches, particularly the baseline and threaded CPU implementations, which achieve minimal runtime with no GPU-related initialization or data transfer costs. Multiprocessing is clearly Pareto-dominated, exhibiting both higher runtime and no numerical advantage. GPU and Numba CUDA solutions also lie off the Pareto frontier, as fixed GPU overheads outweigh any benefit from parallel execution for such small systems.

At approximately 200k nodes (195,853 nodes), a clear transition occurs. While iteration counts remain clustered across all solvers—indicating preserved numerical equivalence—the runtime dimension separates sharply by architecture. GPU-based solvers (GPU and Numba CUDA) move

decisively toward the Pareto frontier, achieving substantially lower runtimes for iteration counts comparable to CPU-based methods. In contrast, baseline CPU, threaded, and Numba CPU implementations become Pareto-dominated due to rapidly increasing wall-clock time, despite similar convergence behavior. This confirms that the performance divergence is architectural rather than algorithmic.

For larger meshes (≈772k nodes), the Pareto structure becomes even more pronounced. GPU and Numba CUDA implementations clearly define the Pareto frontier, combining low runtime with iteration counts indistinguishable from CPU solvers. CPU and threaded implementations occupy the upper-right region of the plots, reflecting both higher runtime and no numerical benefit. Multiprocessing, while improving over baseline CPU in absolute time, remains Pareto-dominated due to its limited scalability and overhead costs. Numba CPU retains acceptable iteration efficiency but become increasingly dominated in runtime as sparse solver and memory bandwidth limitations saturate CPU resources.

At the largest mesh size (1,357,953 nodes), GPU dominance is unequivocal. GPU-based solvers achieve order-of-magnitude reductions in runtime while maintaining iteration counts consistent with all other implementations. The Pareto frontier is exclusively defined by GPU and Numba CUDA approaches, demonstrating that no CPU-based execution model offers a competitive trade-off at this scale. The vertical alignment of iteration counts across all solvers further reinforces that numerical behavior is invariant, and that the Pareto advantage arises solely from superior execution efficiency.

This Pareto analysis leads to three key conclusions:

1. For small-scale problems, CPU-based solvers are Pareto-optimal, as minimal overhead outweighs any benefit from accelerator hardware.
2. For medium-scale problems, the Pareto frontier begins to shift toward GPU-based execution, marking the onset of the CPU-GPU crossover.
3. For large-scale problems, GPU-based solvers fully dominate the Pareto frontier, delivering the best achievable balance between runtime and numerical effort.

This analysis reinforces the central finding of the performance study: GPU acceleration is not merely faster in absolute terms, but becomes structurally superior as problem size increases, while fully preserving numerical consistency across all execution models.

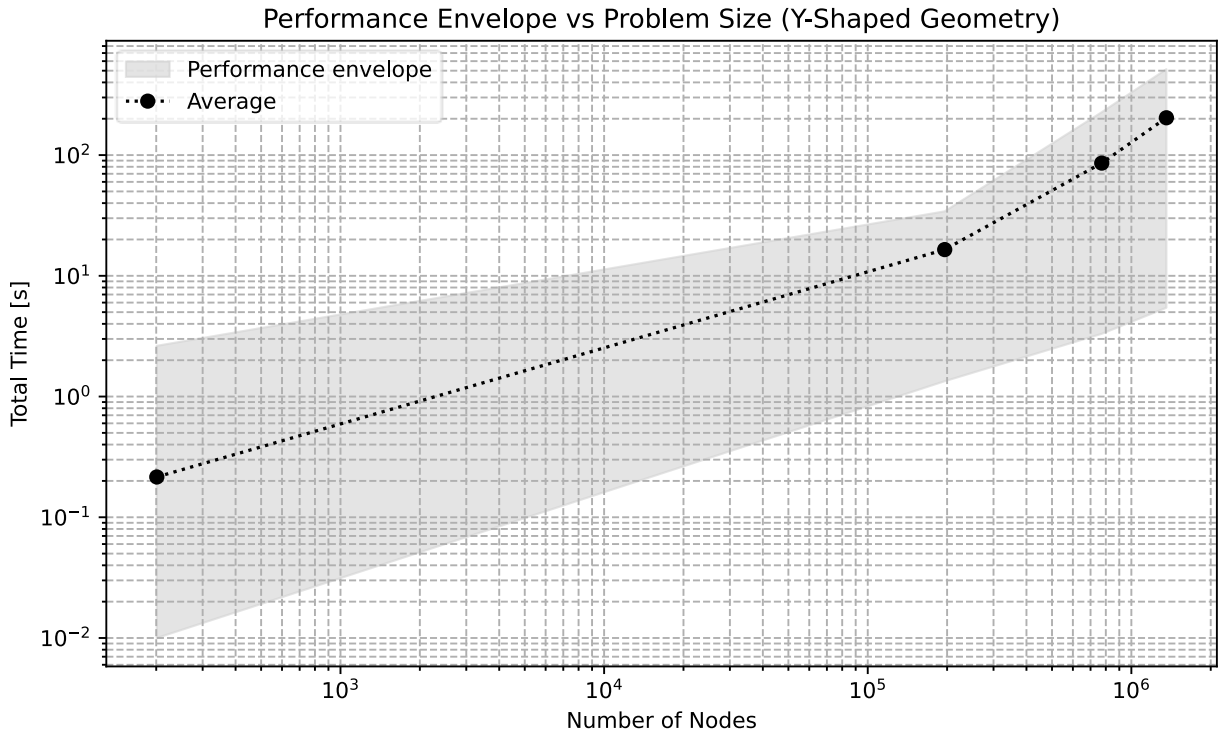

Figure 13: *Performance envelope across execution models for the Y-shaped geometry.*

4.4.5. Performance Envelope Analysis for the Y-Shaped Geometry

The performance envelope clearly reveals distinct computational regimes as the problem size increases. For small meshes, CPU-based solvers define the lower envelope, achieving the shortest runtimes due to minimal overhead and immediate execution. In this regime, GPU implementations are penalized by kernel launch latency, memory allocation, and host-device data transfer costs, which outweigh the benefits of massive parallelism.

As mesh complexity increases, a clear crossover point emerges where GPU-based solvers begin to outperform all CPU alternatives. Beyond this threshold, the envelope shifts decisively toward GPU execution, indicating superior scalability and throughput. The widening gap between GPU and CPU curves highlights the asymptotic advantage of GPU architecture for element-level parallel workloads characteristic of FEM assembly and post-processing.

An important observation is that Numba CUDA and CuPy-based implementations form the lower bound of the envelope for large meshes, confirming that once overheads are amortized, execution efficiency is primarily governed by available parallelism and memory bandwidth rather than interpreter or compilation strategy.

This figure demonstrates that solver optimality is strongly mesh-dependent. While CPU execution remains appropriate for small-scale problems, GPU acceleration defines the optimal performance envelope for medium to large meshes, justifying its use as the default strategy in high-resolution FEM simulations.

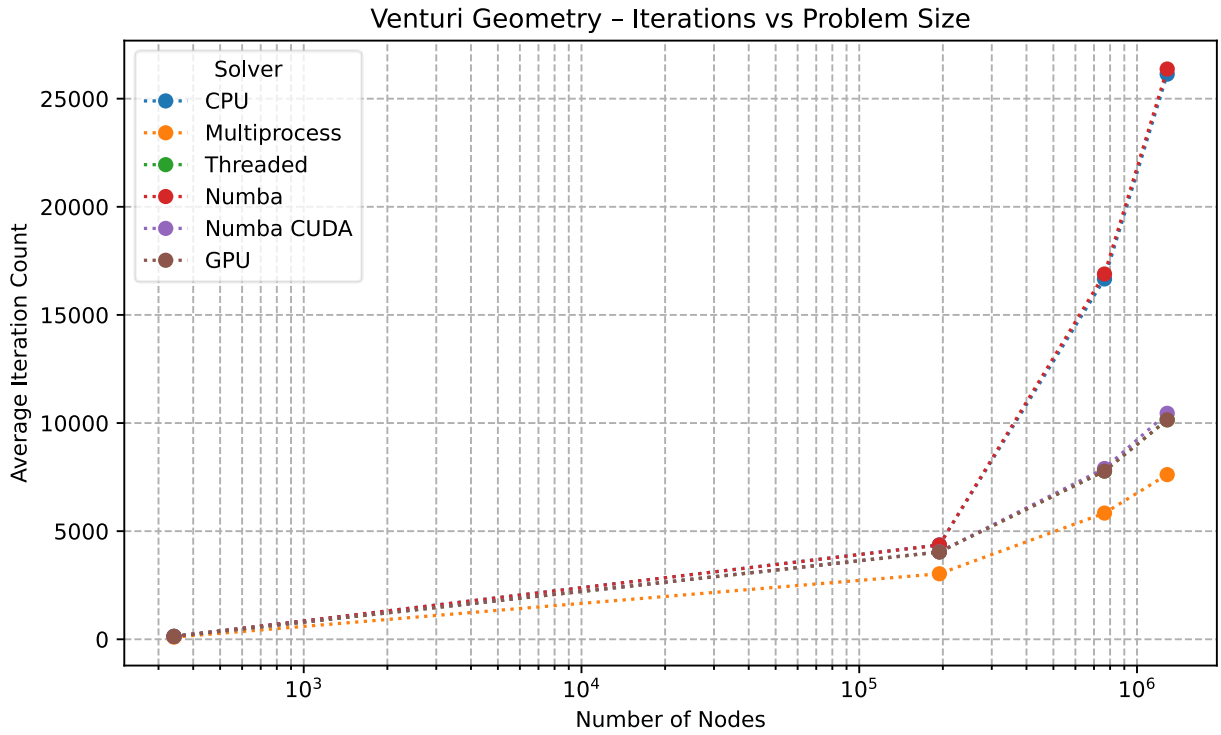

Figure 14: *Number of Conjugate Gradient iterations as a function of mesh size for all solver implementations (Venturi geometry).*

4.4.6 Solver Convergence Behaviour Across Mesh Sizes

This figure provides a crucial validation of the numerical consistency of the entire implementation suite. Across all solver backends and execution models, the number of CG iterations exhibits an almost identical growth trend as mesh size increases. This confirms that convergence behaviour is governed by the mathematical properties of the discretized system — namely mesh resolution, conditioning of the stiffness matrix, and boundary conditions — rather than by the underlying execution architecture.

For small meshes, the iteration count remains low and tightly clustered across all solvers, reflecting well-conditioned systems and rapid convergence. As the number of nodes increases, the iteration count grows steadily, which is expected for elliptic problems discretized with higher resolution. Importantly, this growth is uniform across CPU and GPU implementations, demonstrating that GPU acceleration does not alter the numerical trajectory of the solver.

The absence of divergence between CPU and GPU curves is particularly significant. It indicates that all implementations: - Apply identical preconditioning strategies (Jacobi), - Use consistent convergence tolerances, - Preserve numerical precision within acceptable floating-point limits.

This result also reinforces the interpretation of performance gains observed in runtime benchmarks: speedups achieved by GPU-based solvers arise exclusively from faster execution of assembly, sparse linear algebra, and vector operations, not from reduced solver work or relaxed convergence criteria.

From a performance analysis standpoint, this figure isolates **runtime efficiency** as the sole differentiating factor between solvers. Since the iteration count is invariant with respect to execution model, any reduction in total runtime directly reflects architectural advantages such as increased parallelism, higher memory bandwidth, and reduced instruction overhead.

This convergence analysis confirms that: - All solver implementations are numerically equivalent and directly comparable. - GPU acceleration preserves solver robustness and stability. - Performance improvements observed in later sections are genuine computational gains rather than numerical artefacts.

This result is fundamental for the credibility of the benchmarking study and validates the fairness of the cross-platform performance comparison.

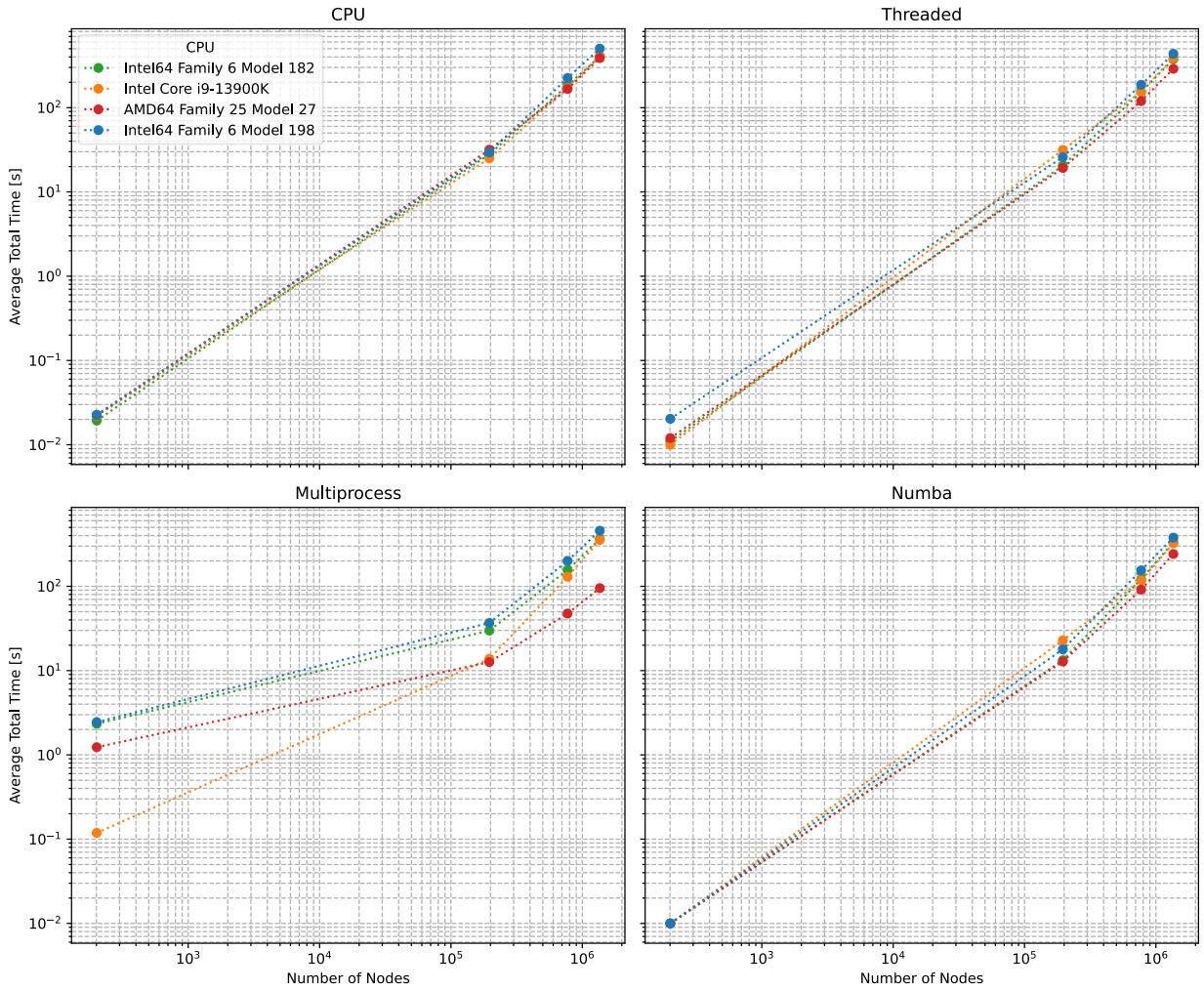

Figure 15: *Execution time comparison across solver implementations for the Y-shaped geometry (CPU-based and GPU-based models).*

4.4.7 Comparative Execution Time Breakdown for the Y-Shaped Geometry

Figure XXX provides a consolidated comparison of execution time scaling for the Y-Shaped geometry across multiple execution models (CPU, threaded, multiprocessing, and Numba JIT CPU), explicitly accounting for different CPU architectures. This representation strengthens the robustness of the performance analysis by demonstrating that the observed trends are not tied to a single processor but persist across heterogeneous hardware configurations.

For the smallest mesh size, execution times remain tightly clustered across all CPUs and execution models. Differences between Intel and AMD processors are minimal and largely masked by fixed overheads such as solver initialization, memory allocation, and Python runtime setup. In this regime, absolute performance is dominated by non-scalable costs rather than architectural efficiency, and all execution models behave similarly regardless of CPU family.

As mesh size increases, the scaling behavior becomes more clearly differentiated. The baseline CPU implementation exhibits near-linear growth in execution time across all processors, indicating that performance is primarily constrained by interpreter overhead and memory bandwidth rather than core count. While absolute runtimes vary slightly between CPUs, the slope of the curves remains remarkably consistent, confirming that the baseline implementation is architecture-agnostic but fundamentally limited in scalability.

The threaded execution model improves performance at small and medium scales but shows increasing divergence between CPUs as mesh size grows. This reflects sensitivity to core count, cache hierarchy, and NUMA effects. Nevertheless, the overall scaling trend remains similar to the baseline CPU, reinforcing that Python threading provides limited benefits for compute-bound FEM workloads due to the Global Interpreter Lock (GIL).

The multiprocessing approach displays the highest variability across CPUs. While it reduces execution time relative to baseline CPU for larger meshes, its scaling is less stable and more sensitive to hardware characteristics. This behavior is consistent with the overhead of process creation, inter-process communication, and memory duplication, which amplify architectural differences and reduce predictability.

The Numba JIT CPU implementation demonstrates the most consistent and favorable scaling among CPU-based approaches. Across all tested processors, its execution time grows more slowly with mesh size, and inter-CPU variability is significantly reduced. This confirms that JIT compilation effectively removes interpreter overhead and enables more efficient parallel execution, making performance primarily dependent on raw memory bandwidth and vectorized execution rather than Python runtime behavior.

A key insight from this figure is that, although absolute runtimes vary between CPUs, the relative ordering of execution models is preserved across architectures. Numba JIT consistently outperforms pure Python approaches, while baseline and threaded CPU executions remain the least scalable. This stability indicates that the conclusions drawn from earlier sections generalize across different hardware environments.

Overall, this analysis reinforces several important findings:

1. CPU-based solvers scale predictably but are ultimately limited by memory bandwidth and core count.
2. Threading and multiprocessing introduce variability without fundamentally changing scaling behavior.
3. Numba JIT provides robust and portable performance improvement across CPU architectures.
4. Hardware differences affect absolute performance but do not alter the structural performance hierarchy.

This figure therefore strengthens the validity of the study’s conclusions by demonstrating that the observed performance patterns are architecturally robust, while also highlighting the intrinsic scalability limits of CPU-based FEM solvers when compared to GPU-accelerated approaches discussed in subsequent sections.

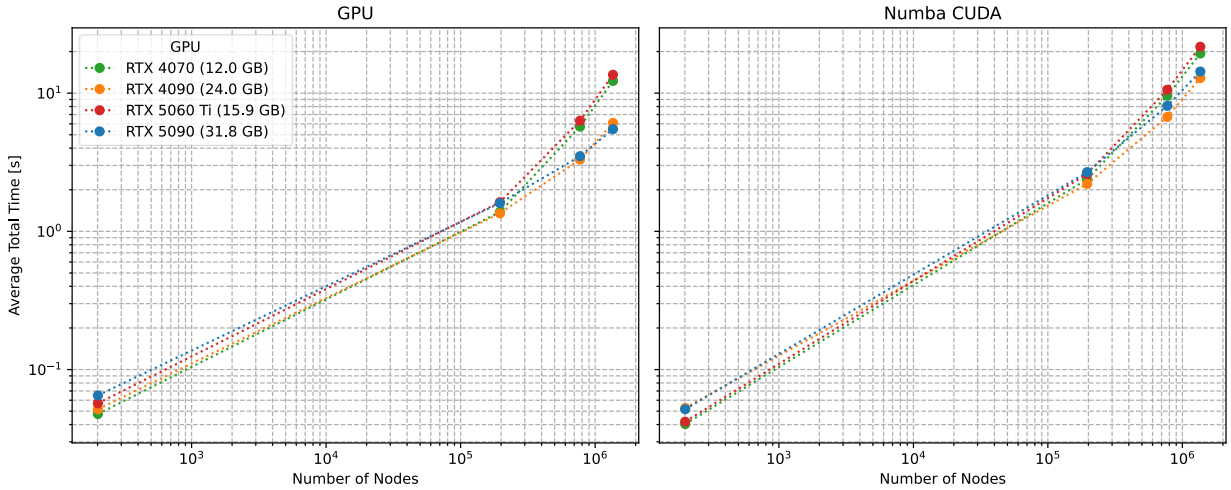

Figure 16: *Side-by-side execution time comparison of GPU-based solvers for the Y-shaped geometry.*

4.4.8 GPU-Centric Performance Comparison for the Y-Shaped Geometry

This comparison isolates GPU execution behavior by removing CPU-based solvers from the analysis, allowing a focused evaluation of how different GPU programming approaches affect performance. For smaller mesh sizes, execution times for Numba CUDA and CuPy are relatively close, with neither implementation showing a decisive advantage. In this regime, kernel launch overheads, JIT compilation costs, and memory transfers dominate runtime, masking fine-grained kernel efficiency differences.

As mesh size increases, performance divergence becomes more apparent. The CuPy RawKernel implementation consistently achieves lower execution times compared to Numba CUDA for medium and large meshes. This behavior reflects the reduced abstraction overhead and finer control over memory access, kernel structure, and execution configuration afforded by native CUDA C kernels. In contrast, while Numba CUDA provides a more accessible development model, its Python-based kernel definition introduces additional overhead and less aggressive compiler optimization opportunities.

The scaling trends observed indicate that both GPU implementations benefit from increasing arithmetic intensity, but CuPy demonstrates superior asymptotic efficiency. This suggests that once fixed overheads are amortized, kernel-level optimization and memory coalescing become the dominant factors influencing performance. The widening gap at larger problem sizes highlights the cumulative impact of these low-level optimizations.

Importantly, both GPU approaches maintain similar solver iteration counts and numerical behavior, confirming that the observed performance differences are purely architectural and implementation driven. This reinforces the interpretation that RawKernel-based execution represents the upper bound of achievable single-GPU performance within the scope of this project.

This figure demonstrates that: - Numba CUDA offers a strong balance between performance and development productivity; - CuPy RawKernel achieves the best absolute performance for large-scale problems; - Kernel-level control becomes increasingly important as problem size grows.

This analysis justifies the inclusion of both approaches in the study and positions CuPy RawKernel as the reference implementation for maximum-performance GPU execution in the final benchmarking comparisons.

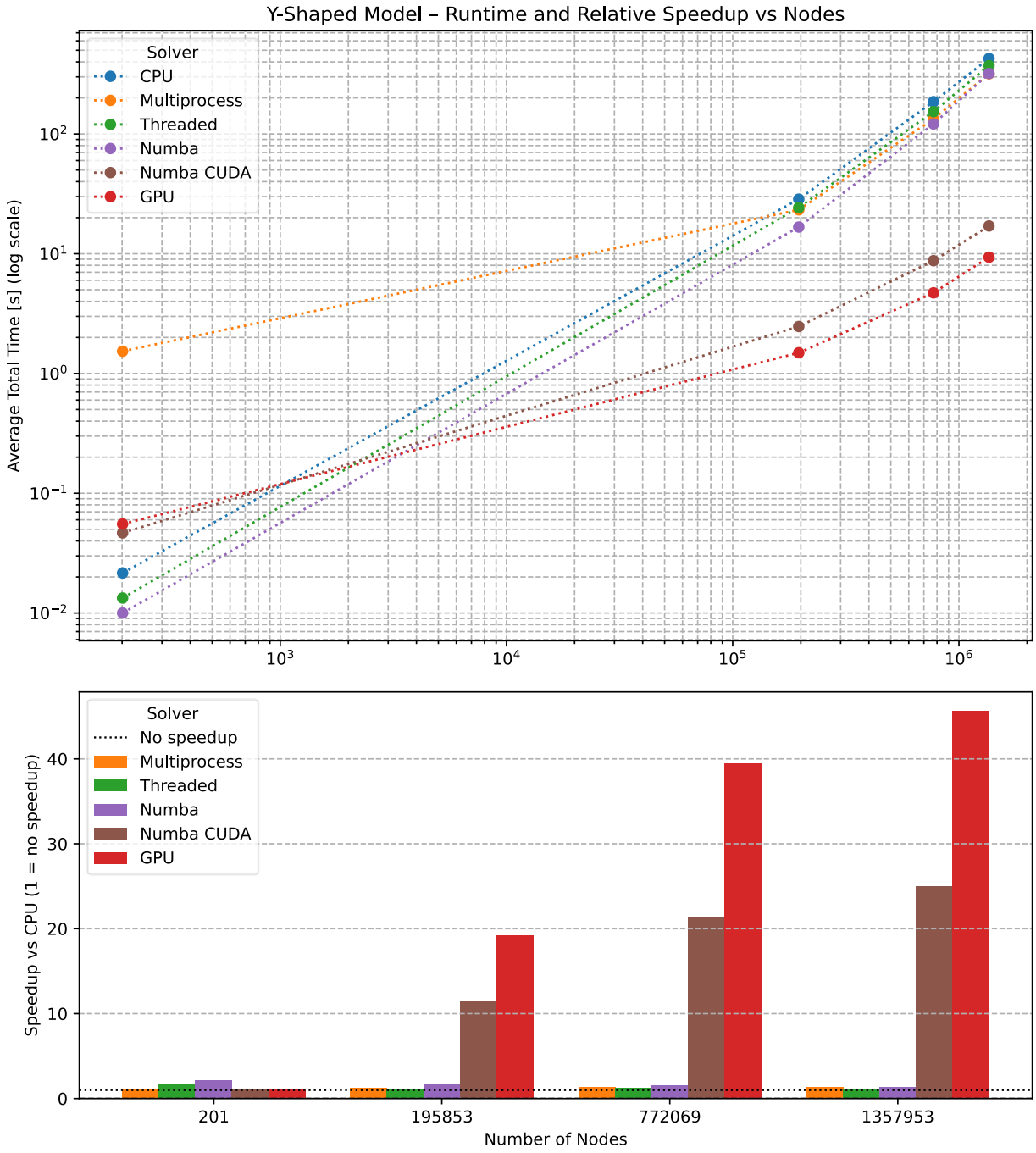

Figure 17: *Runtime speedup of GPU-based implementations relative to the CPU baseline for the Y-shaped geometry.*

4.4.9 GPU Speedup Analysis for the Y-Shaped Geometry

Figure 17 provides a combined view of absolute runtime scaling and relative speedup versus the CPU baseline for the Y-Shaped geometry, offering a comprehensive perspective on how different execution models behave as the number of nodes increases.

From the runtime curves (top panel), a clear hierarchy emerges. For small meshes (≈200 nodes), all implementations exhibit very low absolute runtimes, and differences between

execution models are marginal. In this regime, GPU-based solvers (CuPy GPU and Numba CUDA) show no meaningful advantage and, in absolute terms, remain comparable to CPU execution. This behavior reflects the dominance of fixed overheads — kernel launch latency, device synchronization, and host-device data transfers — which prevent GPUs from exploiting parallelism at such small problem sizes. Consequently, GPU acceleration is ineffective and provides little to no speedup.

As mesh size increases, runtime growth becomes strongly execution-model dependent. CPU, threaded, and Numba CPU implementations show steep scaling trends, with execution time increasing rapidly as the number of nodes grows. Multiprocessing reduces the slope relative to the CPU baseline but remains significantly slower than GPU-based approaches due to process management overhead and limited scalability. In contrast, GPU implementations exhibit substantially flatter scaling curves, indicating much better asymptotic behavior.

This transition is more clearly quantified in the speedup plot (bottom panel). For the smallest mesh, all speedups remain close to unity, confirming that GPU execution does not amortize its overhead. However, beyond the intermediate mesh size (≈200k nodes), a pronounced acceleration regime emerges. GPU-based solvers rapidly surpass all CPU-based implementations, with speedup increasing sharply as problem size grows.

The CuPy RawKernel implementation consistently achieves the highest speedup across all large meshes, reaching several tens of times faster than the CPU baseline at the largest scale. Numba CUDA follows the same qualitative trend but achieves systematically lower speedups. This gap reflects differences in kernel maturity and execution efficiency: CuPy benefits from native CUDA C kernels with tighter control over memory access patterns, better register allocation, and reduced abstraction overhead, which become increasingly important as arithmetic intensity grows.

An important observation is the sublinear growth of speedup at the largest mesh sizes. Although GPU acceleration remains substantial, the rate of improvement decreases, indicating a transition from compute-bound assembly phases to solver-dominated execution. At this stage, sparse linear algebra operations become memory-bandwidth-bound, even on the GPU, imposing a fundamental limit on achievable acceleration.

Overall, this figure demonstrates that:

1. GPU acceleration is highly scale-dependent and ineffective for small FEM problems.
2. A clear CPU-GPU crossover occurs as mesh size increases, after which GPUs dominate.
3. CuPy RawKernel consistently delivers the highest asymptotic speedup.
4. Sparse solver performance ultimately constrains end-to-end scalability.

These results provide strong empirical confirmation that GPU acceleration is essential for large-scale FEM simulations, while also highlighting that further performance gains at extreme scales must focus on solver algorithms and memory efficiency rather than kernel-level optimizations alone.

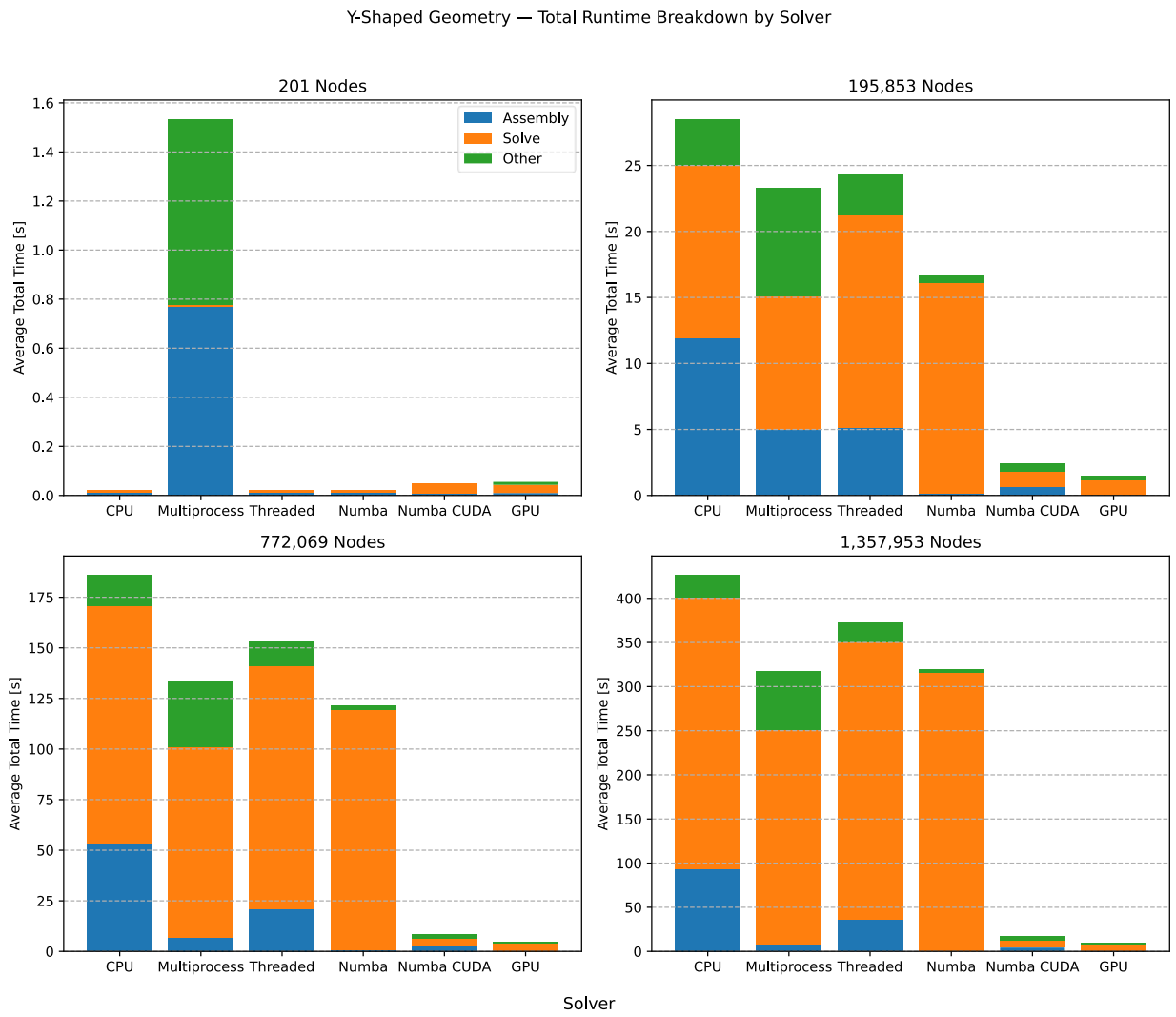

Figure 18: *Detailed runtime breakdown of the total execution time for the Y-shaped geometry across CPU and GPU execution models.*

4.4.10 Runtime Breakdown Across Execution Models for the Y-Shaped Geometry

This figure 18 decomposes the total runtime into its main computational stages—mesh loading, system assembly, boundary condition application, linear system solution, and post-processing—for the different execution models considered in this study. The comparison is performed for a representative mesh size, enabling direct inspection of how each execution model redistributes computational cost across the FEM pipeline.

The runtime breakdown highlights fundamental differences in how CPU- and GPU-based implementations allocate computational effort across the FEM workflow. In the CPU baseline and threaded variants, system assembly represents a dominant fraction of the total runtime. This reflects the interpreter-bound nature of element-level loops and sparse matrix insertion, where Python overhead and memory indirection significantly limit performance.

In contrast, the Numba JIT CPU implementation exhibits a markedly different profile. Assembly time is substantially reduced and no longer dominates the execution, confirming the effectiveness of JIT compilation and parallel execution in eliminating Python-level overhead. As a result, the linear solver becomes the primary runtime contributor, indicating a shift from compute-bound to memory-bandwidth-bound behavior.

The GPU-based implementations further accentuate this transition. For both Numba CUDA and CuPy RawKernel, the assembly and post-processing stages account for only a small fraction of total runtime. These stages scale efficiently on the GPU due to massive parallelism and high arithmetic throughput. The sparse linear solve clearly dominates the execution time, accounting for the majority of runtime. This dominance reflects the intrinsic memory-bound nature of sparse matrix-vector products, even on high-bandwidth GPU architectures.

A notable distinction between the two GPU implementations lies in the relative cost of non-solver stages. The CuPy RawKernel implementation exhibits slightly lower assembly and post-processing fractions compared to Numba CUDA, consistent with the use of native CUDA C kernels and reduced abstraction overhead. Boundary condition application and mesh loading remain negligible across all execution models, confirming that they do not materially influence performance at scale.

Overall, this breakdown provides critical insight into performance bottlenecks: - CPU-based implementations are limited primarily by assembly overhead. - JIT compilation shifts the bottleneck toward the solver. - GPU acceleration virtually eliminates assembly cost, exposing the solver as the dominant constraint. - Further performance gains on GPU would require more advanced sparse solvers or preconditioning strategies rather than additional kernel-level optimization.

This figure therefore complements the speedup analysis by explaining *why* acceleration saturates and *where* future optimization efforts should be focused within the FEM pipeline.

5. Progressive Profiling Optimization

This section documents the systematic optimization journey of the GPU-accelerated Finite Element Method (FEM) solver, progressing from a serial CPU baseline through parallel CPU variants to fully optimized GPU implementations. Each section presents the implementation's technical approach, the profiling data that revealed its bottlenecks, and the insights that guided subsequent optimization efforts.

The solver was validated across a comprehensive test matrix of 144 configurations: six mesh geometries (Y-Shaped channel, Venturi tube, S-Bend, T-Junction, Backward-Facing Step, and 90° Elbow), each at four refinement levels ranging from approximately 200 to 1.3 million nodes, executed on all six solver implementations.

We present a detailed analysis for the Y-Shaped channel geometry, selected as representative of the overall performance patterns observed across all configurations. Results focus on the smallest mesh (201 nodes, 52 elements) and largest mesh (1,357,953 nodes, 338,544 elements) clearly illustrate the contrast between fixed-overhead behavior and production-scale performance; intermediate mesh sizes confirmed smooth scaling transitions consistent with these boundary cases.

All profiling data was collected using NVIDIA Nsight Systems with NVTX annotations, enabling precise phase-level timing analysis. The hardware utilized for profiling tests was a Intel i9-12900K CPU and a NVidia RTX 4090 GPU.

5.1. CPU Baseline Implementation

Technical Approach

The baseline implementation (quad8_cpu_v4.py) establishes a serial reference using NumPy for numerics and SciPy for sparse matrix operations. The solver follows the standard FEM workflow: mesh loading, stiffness matrix assembly, boundary condition application, iterative solving, and post-processing.

Assembly iterates sequentially over all elements:

```
for e in range(Nels):
    Ke, fe = Elem_Quad8(coords, connect[e], k_iso)
    # Accumulate to COO format
    data_K.extend(Ke.flatten())
    row_K.extend(...)
    col_K.extend(...)
```

Each element computation (Elem_Quad8) performs 9-point Gauss-Legendre quadrature, computing shape functions via Shape_N_Der8 and using np.linalg.det and np.linalg.inv for Jacobian operations at each integration point.

Sparse matrix construction uses SciPy's COO format with subsequent conversion to CSR for efficient row-slicing during boundary condition application. The **solver** employs SciPy's conjugate gradient (scipy.sparse.linalg.cg) with an optional Jacobi preconditioner, monitoring convergence via residual callbacks every 10 iterations.

Profiling Results

Mesh	Total	Assembly	Solve	Assembly %	Solve %
y_tube_201	141.5 ms	~13 ms	~7 ms	9%	5%
y_tube_1_3m	506.3 s	85.7 s	395.9 s	17%	78%

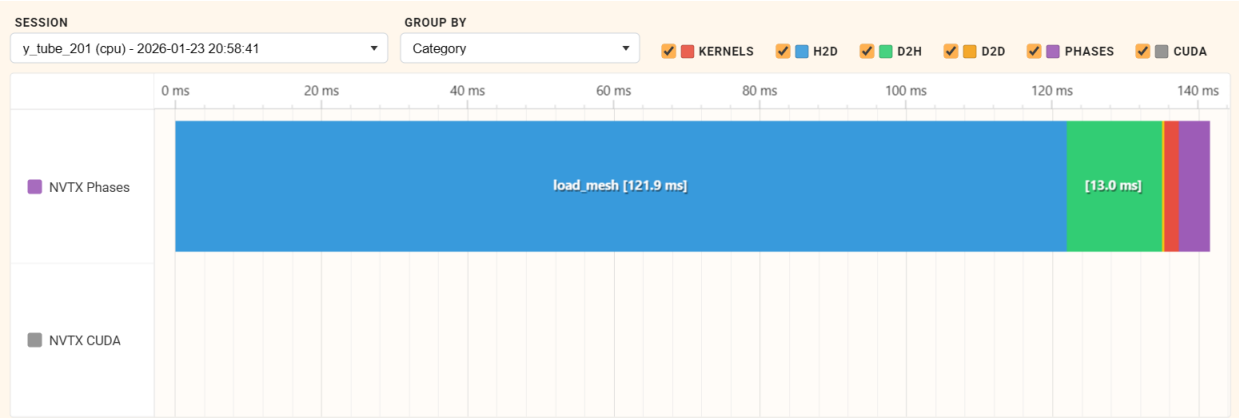

***Figure 19**: CPU Baseline timeline for y_tube_201 (141.5 ms total) — mesh loading dominates at small scale.*

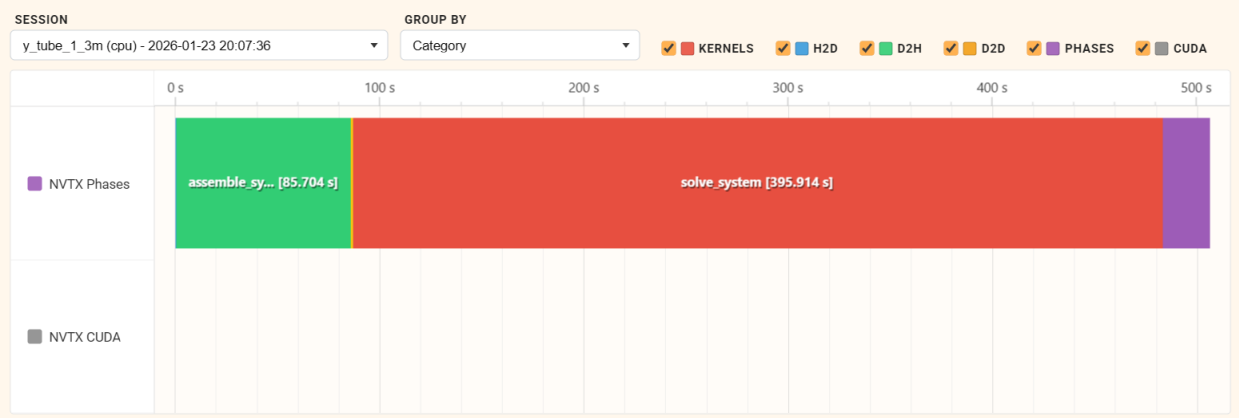

***Figure 20**: CPU Baseline timeline for y_tube_1_3m (506.3s total) showing assembly (85.7s) and solve (395.9s) phases.*

Analysis

The profiling reveals two distinct bottleneck patterns depending on mesh scale. At small scale, mesh loading dominates (121.9 ms, 86% of runtime), while assembly and solve are negligible. At production scale, the solve phase consumes 78% of total runtime, with assembly accounting for 17%.

The serial assembly loop suffers from Python interpreter overhead on each of the 338,544 iterations. Within each iteration, np.linalg.det and np.linalg.inv incur function call overhead for small 2x2 matrices where explicit formulas would be faster. The list .extend() operations for COO accumulation cause repeated memory reallocations.

These observations establish the optimization targets: parallelize assembly to address the per-element overhead and ultimately accelerate the solve phase which dominates at scale.

2. CPU Threaded Implementation

Technical Approach

The threaded implementation (quad8_cpu_threaded_v2.py) introduces parallelism via Python's concurrent.futures.ThreadPoolExecutor. Elements are processed in batches of 1000 to amortize thread management overhead.

```
with ThreadPoolExecutor(max_workers=self.num_workers) as executor:
    futures = [executor.submit(_process_element_batch_threaded, batch_start, batch_end, ...)
               for batch_start, batch_end in batches]
    for future in as_completed(futures):
        data, rows, cols = future.result()
        data_K.extend(data)  # Serial aggregation
```

Each worker executes the same Elem_Quad8 logic as the baseline, but multiple batches execute concurrently. Post-processing (velocity and pressure computation) is similarly parallelized.

Profiling Results

Mesh	Total	Assembly	Solve	vs Baseline
y_tube_201	165.9 ms	~9 ms	~7 ms	0.85x (slower)
y_tube_1_3m	447.3 s	37.4 s	381.7 s	1.13x

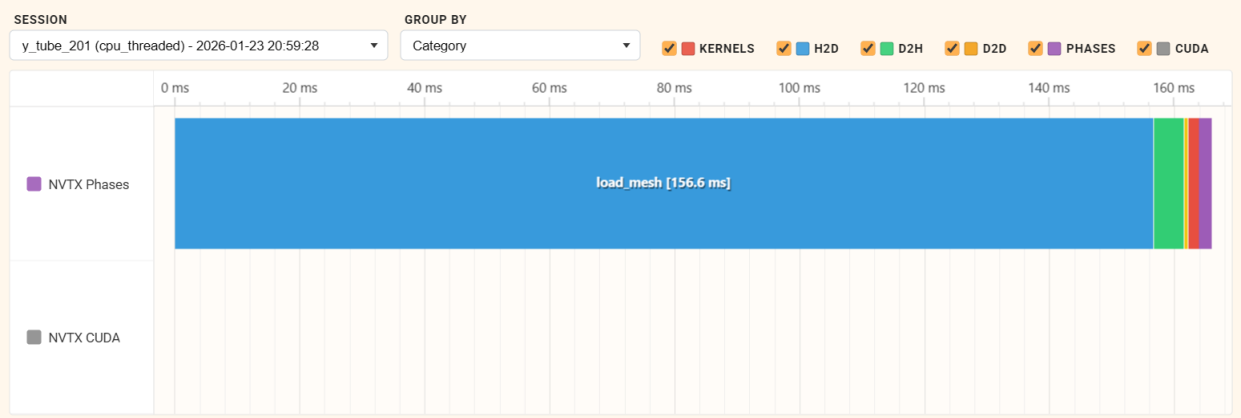

***Figure 21**: CPU Threaded timeline for y_tube_201 (165.9 ms total) — threading overhead exceeds gains at small scale.*

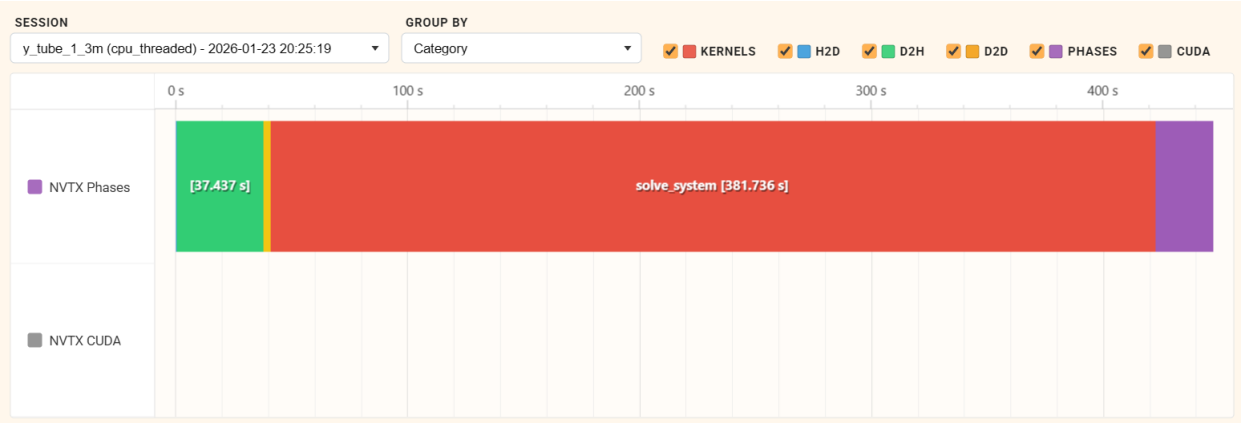

***Figure 22**: CPU Threaded timeline for y_tube_1_3m (447.3s total) showing reduced assembly phase (37.4s) but with a similar solver duration (381.7s).*

Analysis

At production scale, assembly time improved from 85.7s to 37.4s (2.3x speedup), demonstrating that the element computations benefit from concurrent execution. However, total speedup is only 1.13x because the solve phase (381.7s) remains essentially unchanged.

The threading gains are limited by Python's Global Interpreter Lock (GIL). While NumPy's BLAS operations release the GIL during matrix computations, the surrounding Python code (loop control, function calls, list operations) still contends for the lock. The serial aggregation step (data_K.extend()) after each batch completion further limits scalability.

At small scale, threading actually degrades performance (165.9 ms vs 141.5 ms). The overhead of creating the thread pool, dispatching work, and collecting results exceeds the time saved on the trivial 52-element workload.

The key insight: threading provides modest assembly improvements but cannot address the dominant solve bottleneck, and introduces overhead that hurts small-scale performance.

3. CPU Multiprocess Implementation

Technical Approach

The multiprocess implementation (quad8_cpu_multiprocess_v3.py) bypasses the GIL entirely by using multiprocessing.Pool. Each worker process has its own Python interpreter and memory space, enabling true parallel execution.

To minimize inter-process communication (IPC) overhead, large arrays are broadcast once during worker initialization rather than serialized per batch:

```
def _init_assembly_worker(coords_, connect_, ...):
    global _w_coords, _w_connect, ...
    _w_coords = coords_
    _w_connect = connect_

with mp.Pool(processes=self.num_workers,
             initializer=_init_assembly_worker,
             initargs=(...)) as pool:
    results = pool.map(_process_element_batch_mp, batch_ranges)
```

Profiling Results

Mesh	Total	Assembly	Solve	vs Baseline	vs Threaded
y_tube_201	262.2 ms	53.2 ms	~5 ms	0.54x (slower)	0.63x (slower)
y_tube_1_3m	427.1 s	43.1 s	378.8 s	1.19x	1.05x

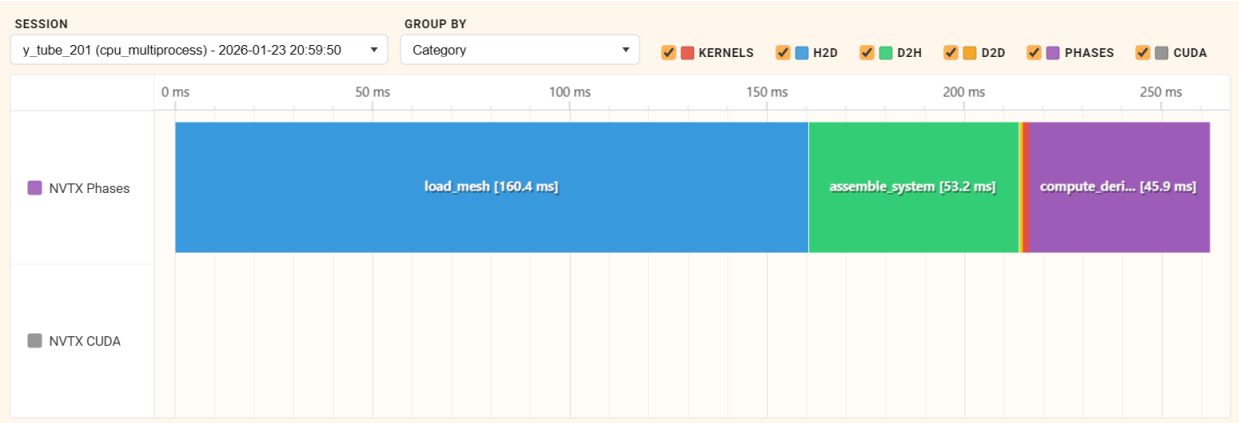

***Figure 23**: CPU Multiprocess timeline for y_tube_201 (262.2 ms total) — process spawn overhead dominates at small scale.*

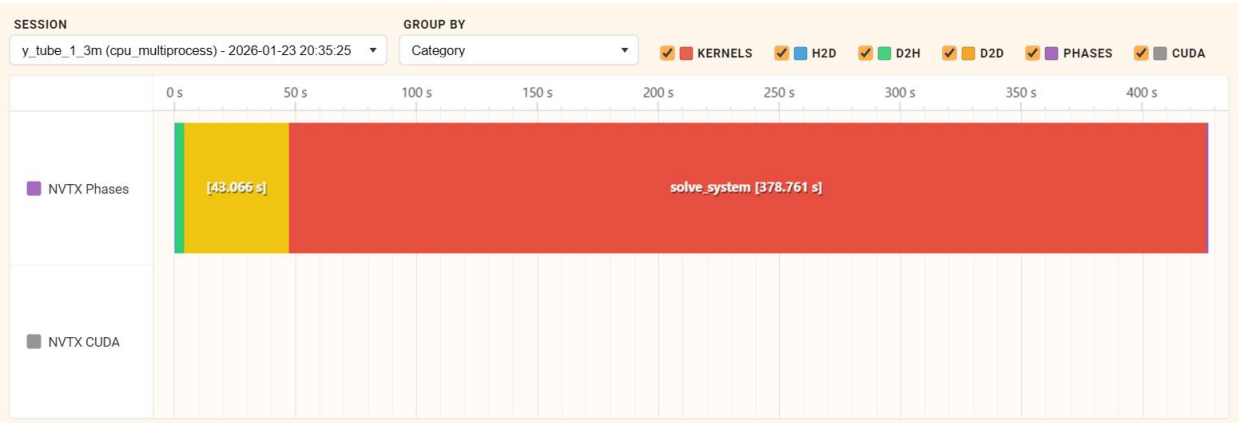

***Figure 24**: CPU Multiprocess timeline for y_tube_1_3m (427.1s total) showing assembly (43.1s) and solve (378.8s) phases with process coordination overhead.*

Analysis

Contrary to expectations, multiprocessing performed worse than threading for assembly at production scale (43.1s vs 37.4s). This counterintuitive result stems from the overhead costs of process-based parallelism:

1. **Process spawn overhead**: Creating worker processes costs 1-2 seconds, compared to near-instantaneous thread creation.
2. **IPC serialization**: Despite the initializer optimization, results must still be pickled and transmitted back to the main process after each batch.
3. **Memory duplication**: Each process maintains its own copy of working data, increasing memory pressure.

At small scale, the overhead is catastrophic: 262.2 ms versus 141.5 ms baseline, an 85% slowdown. The process pool creation alone exceeds the entire baseline runtime.

The marginal total improvement (1.19x) comes primarily from slightly better solve performance (378.8s vs 395.9s), likely due to reduced memory contention after processes terminate.

Additional practical considerations include Windows compatibility — the implementation uses `spawn` rather than `fork` to work across platforms, which incurs higher startup costs but ensures portability. Memory pressure is also a concern: each worker process maintains its own copy of mesh data, which can strain RAM for very large meshes approaching system memory limits.

This implementation demonstrates that bypassing the GIL is not sufficient for performance gains; the communication and coordination costs of process-based parallelism can outweigh the benefits of true concurrency, particularly when the workload involves returning substantial data from workers.

4. Numba JIT Implementation

Technical Approach

The Numba JIT implementation (`quad8_numba_v2.py`) takes a fundamentally different approach: rather than parallelizing Python code, it compiles the computational kernels to machine code using Numba's `@njit` decorator.

Shape function computation uses explicit scalar operations that Numba compiles efficiently:

```
@njit(cache=True)
def shape_n_der8(xi, eta):
    N = np.empty(8)
    dNdxi = np.empty(8)
    dNdeta = np.empty(8)
    # Explicit per-node formulas - no NumPy broadcasting overhead
    N[0] = (xi - 1) * (eta + xi + 1) * (1 - eta) / 4
    # ...
```

Jacobian operations are implemented as explicit 2x2 formulas rather than library calls:

```
det_J = J[0,0]*J[1,1] - J[0,1]*J[1,0]
inv_J[0,0] = J[1,1] / det_J
inv_J[0,1] = -J[0,1] / det_J
# ...
```

Assembly uses prange for automatic parallelization with pre-allocated output arrays:

```
@njit(parallel=True, cache=True)
def assemble_system_numba(coords, connect, ...):
    all_data = np.empty(Nels * 64)  # Pre-allocated, no list growth
    all_rows = np.empty(Nels * 64, dtype=np.int64)

    for e in prange(Nels):  # Parallel iteration
        # Each thread writes to deterministic slice [e*64 : (e+1)*64]
```

Profiling Results

Mesh	Total	Assembly	Solve	vs Baseline
y_tube_201	219.4 ms	86.8 ms	~5 ms	0.65x (slower)
y_tube_1_3m	390.3 s	~4.6 s	385.6 s	1.30x

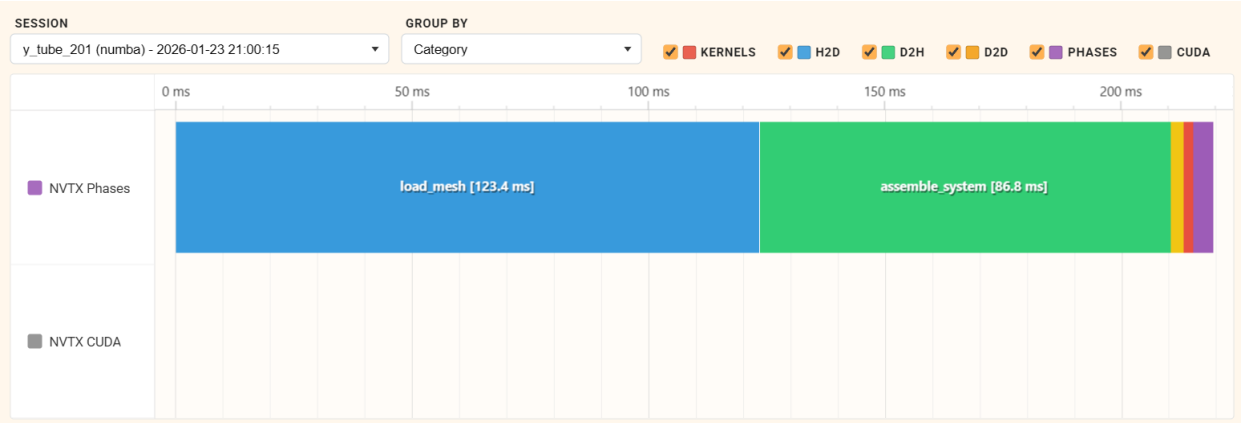

***Figure 25**: Numba JIT timeline for y_tube_201 (219.4 ms total) — JIT compilation overhead visible on first run.*

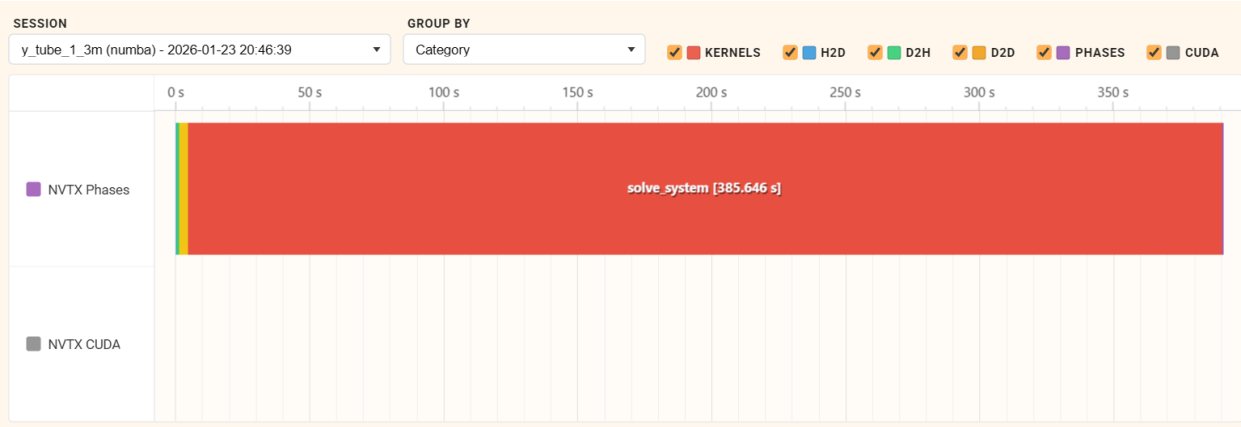

***Figure 26**: Numba JIT timeline for y_tube_1_3m (390.3s total) showing dramatically reduced assembly (~4.6s) but unchanged solve phase (385.6s).*

Analysis

At production scale, Numba JIT achieves a dramatic 18.6x speedup for assembly (85.7s → 4.6s), matching the assembly performance of the GPU implementations while remaining entirely on CPU. This demonstrates the power of JIT compilation combined with prange auto-parallelization:

1. **Eliminated interpreter overhead**: The compiled code runs at native speed without per-iteration Python overhead.
2. **Efficient memory access**: Pre-allocated contiguous arrays with deterministic indexing enable cache-friendly access patterns.
3. **No IPC overhead**: Unlike multiprocessing, prange threads share memory without serialization.
4. **Optimized small operations**: Explicit 2x2 matrix formulas compile to a few machine instructions, versus function call overhead for np.linalg.inv.

However, total speedup is only 1.30x because the solve phase (385.6s) remains unchanged. SciPy's iterative solver cannot be JIT-compiled, and it already uses optimized BLAS internally. The solve now accounts for 99% of runtime.

At small scale, the 219.4 ms runtime reflects JIT compilation overhead on first execution — Numba must compile each @njit function to machine code before it can run. The cache=True directive mitigates this for subsequent runs by persisting the compiled code to disk. On repeated executions with warm cache, the small-mesh overhead drops significantly, making Numba JIT competitive with the baseline even at small scale. This first-run versus subsequent-run distinction is important for interactive applications where startup latency matters.

The key insight from Numba JIT: CPU assembly can be optimized to match GPU assembly performance, but the solve phase becomes an insurmountable bottleneck. Further improvements require GPU-accelerated solving.

5. Numba CUDA Implementation

Technical Approach

The Numba CUDA implementation (quad8_numba_cuda.py with kernels_numba_cuda.py) marks the transition to GPU computing. CUDA kernels are written in Python syntax using Numba's @cuda.jit decorator, with one thread processing one element.

Kernel structure uses local memory for per-element working arrays:

```
@cuda.jit
def quad8_assembly_kernel(x, y, quad8, xp, wp, vals_out, fg_out):
    e = cuda.grid(1)
    if e >= quad8.shape[0]:
        return

    # Per-thread local arrays (registers/local memory)
    edofs = cuda.local.array(8, dtype=np.int32)
    Ke = cuda.local.array((8, 8), dtype=np.float64)
    # ... computation identical to CPU but parallelized across GPU threads

    # Output: deterministic indexing, no race conditions
```

```
    base_idx = e * 64
    for i in range(8):
        for j in range(8):
            vals_out[base_idx + k] = Ke[i, j]

    # Atomic update for force vector (nodes shared between elements)
    for i in range(8):
        cuda.atomic.add(fg_out, edofs[i], fe[i])
```

Sparse matrix construction occurs on CPU after transferring values back from GPU:

```
vals_host = d_vals.copy_to_host()  # Device-to-host transfer
K = sp.coo_matrix((vals_host, (rows, cols)), shape=(Nnds, Nnds)).tocsr()
```

Solve uses CuPy's GPU-accelerated conjugate gradient with Jacobi preconditioning.

Profiling Results

Mesh	Total	Assembly	Apply BC	Solve	Kernels	MemCpy
y_tube_201	1.18 s	399.4 ms	123.9 ms	~200 ms	2,135	146
y_tube_1_3m	14.94 s	4.59 s	3.47 s	5.92 s	179,115	10,097

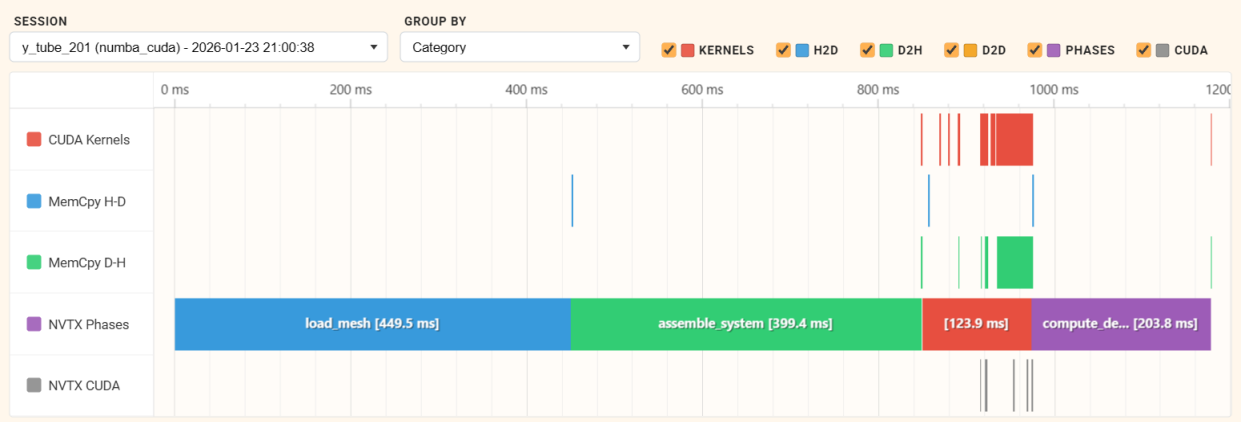

***Figure 27**: Numba CUDA timeline for y_tube_201 (1.18s total) — GPU initialization and kernel launch overhead dominate at small scale.*

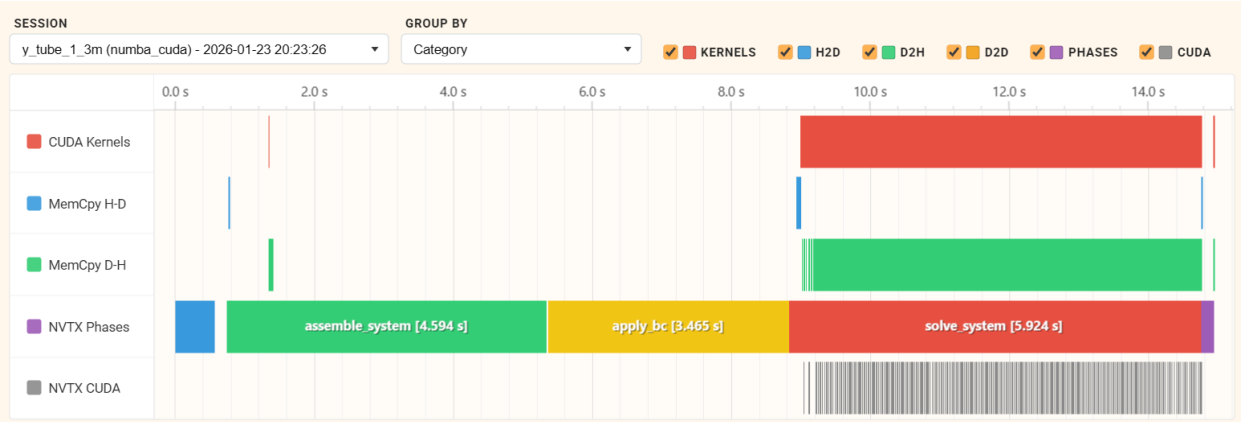

***Figure 28**: Numba CUDA timeline for y_tube_1_3m (14.94s total) showing assembly (4.59s), BC application (3.47s), and solve (5.92s) phases with visible kernel and memory transfer activity.*

Analysis

The Numba CUDA implementation achieves 33.9x total speedup over the CPU baseline at production scale (506.3s → 14.94s). The GPU-accelerated solve phase is the breakthrough: 395.9s → 5.92s (66.9x speedup).

Profiling reveals the phase breakdown:

- **Assembly (4.59s, 31%)**: Similar to Numba JIT, as both use compiled code with parallel execution. The GPU's massive thread count doesn't provide additional benefit here because Numba JIT already saturates CPU cores.
- **BC Application (3.47s, 23%)**: Robin boundary conditions involve edge integration loops that aren't fully optimized for GPU execution. The profile shows this as a significant new bottleneck.
- **Solve (5.92s, 40%)**: The CuPy CG solver runs entirely on GPU, eliminating the CPU solve bottleneck.

The memory transfer pattern shows substantial device-to-host activity during solve (the large green D2H bar), indicating data moving back to CPU repeatedly during iterations.

At small scale, the GPU implementation is dramatically slower (1.18s vs 141.5ms baseline, 8.3x slowdown). The 449.5 ms mesh loading phase and 399.4 ms assembly phase include CUDA context initialization, kernel compilation, and memory allocation overhead that cannot be amortized over the trivial 52-element workload.

Key observations: GPU acceleration finally breaks through the solve bottleneck, but BC application emerges as a new optimization target, and fixed GPU overhead makes this unsuitable for small meshes.

6. CuPy GPU Implementation

Technical Approach

The CuPy implementation (quad8_gpu_v3.py) optimizes the GPU pipeline using CuPy's RawKernel for hand-written CUDA C kernels, providing finer control than Numba's Python-to-PTX compilation.

Assembly kernel in CUDA C with explicit optimizations:

```
extern "C" __global__
void assembly_kernel(const double* __restrict__ x,
                     const double* __restrict__ y,
                     const int* __restrict__ quad8,
                     /* ... */) {
    int e = blockDim.x * blockIdx.x + threadIdx.x;
    if (e >= Nels) return;

    double Ke[64];  // 8x8 in registers
    // ... computation with potential compiler unrolling
}
```

The __restrict__ qualifier informs the compiler that pointers don't alias, enabling aggressive optimization.

Robin BC kernel is now fully GPU-resident:

```
extern "C" __global__
void robin_bc_kernel(const double* x, const double* y,
                     const int* robin_edges, /* ... */) {
    int edge_idx = blockDim.x * blockIdx.x + threadIdx.x;
    // Full edge integration on GPU - no CPU fallback
}
```

Sparse matrix construction stays on GPU using CuPy's sparse module:

```
rows_gpu = cp.array(rows)
cols_gpu = cp.array(cols)
K_gpu = cupyx.scipy.sparse.coo_matrix((vals_gpu, (rows_gpu, cols_gpu)))
K_csr = K_gpu.tocsr()  # Conversion on GPU
```

Profiling Results

Mesh	Total	Assembly	Apply BC	Solve	Kernels	MemCpy
y_tube_201	658.6 ms	~50 ms	133.7 ms	~70 ms	3,361	240
y_tube_1_3m	7.55 s	0.58 s	1.23 s	5.68 s	222,066	14,547

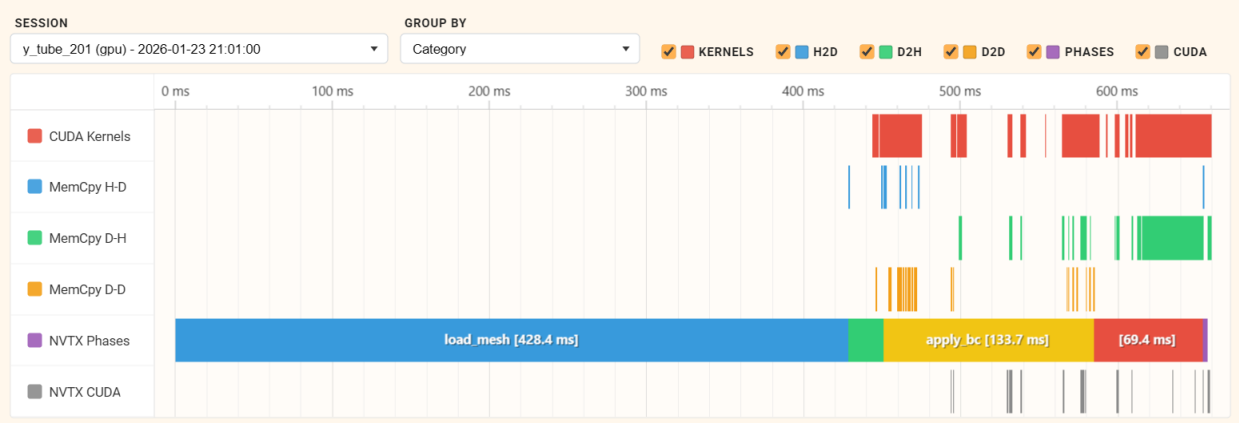

***Figure 29**: CuPy GPU timeline for y_tube_201 (658.6 ms total) — GPU overhead still significant but reduced compared to Numba CUDA.*

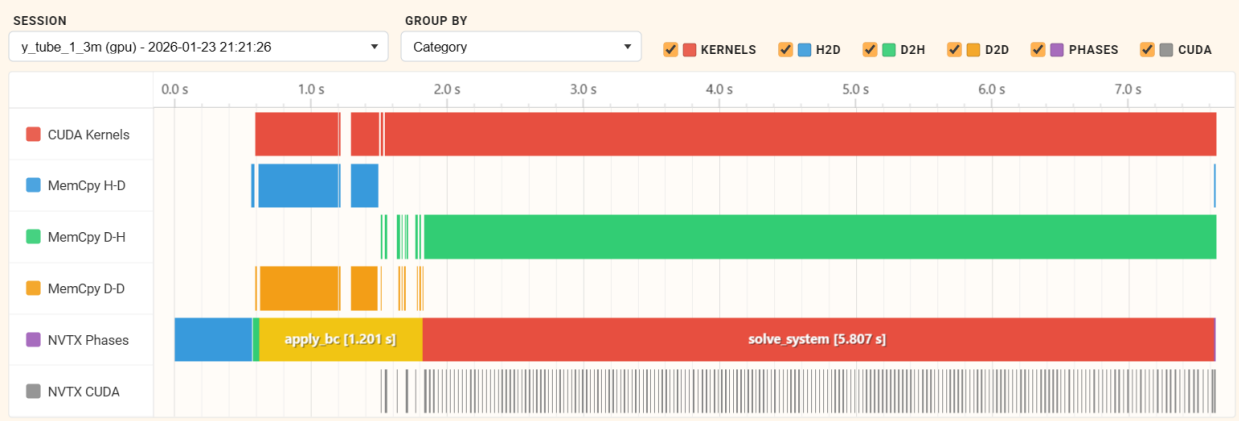

***Figure 30**: CuPy GPU timeline for y_tube_1_3m (7.55s total) showing optimized assembly (~0.58s), BC application (1.23s), and solve (5.68s) phases.*

Analysis

The CuPy implementation achieves 67.1x total speedup over the CPU baseline (506.3s → 7.55s) and 2.0x improvement over Numba CUDA (14.94s → 7.55s).

Phase-by-phase comparison with Numba CUDA:

Phase	Numba CUDA	CuPy GPU	Improvement
Assembly	4.59 s	0.58 s	7.9x
Apply BC	3.47 s	1.23 s	2.8x
Solve	5.92 s	5.68 s	1.04x

Assembly optimization (7.9x): The RawKernel approach provides multiple advantages over Numba CUDA: - CUDA C compiler (nvcc) applies more aggressive optimizations than Numba's PTX generation - __restrict__ qualifiers enable better memory access optimization - Sparse matrix construction remains on GPU, eliminating the D2H transfer for values that Numba CUDA required

BC optimization (2.8x): The fully GPU-resident Robin BC kernel eliminates CPU-GPU synchronization points that existed in the Numba CUDA hybrid approach.

Solve (1.04x): Both implementations use CuPy's CG solver, so performance is nearly identical. The solve phase now accounts for 75% of total runtime, indicating it has become the new bottleneck.

At small scale, CuPy (658.6 ms) is faster than Numba CUDA (1.18 s) but still 4.7x slower than the CPU baseline. The reduced overhead comes from CuPy's pre-compiled kernels versus Numba's JIT compilation, but GPU context initialization still dominates.

7. Conclusions

Performance Analysis Summary

Our optimization journey achieved a 67x speedup from CPU baseline to the final CuPy GPU implementation at production scale:

Solver	Assembly	Solve	Total	vs Baseline
CPU Baseline	85.7 s	395.9 s	506.3 s	1.0x
CPU Threaded	37.4 s	381.7 s	447.3 s	1.1x
CPU Multiprocess	43.1 s	378.8 s	427.1 s	1.2x
Numba JIT	4.6 s	385.6 s	390.3 s	1.3x
Numba CUDA	4.59 s + 3.47 s BC	5.92 s	14.94 s	33.9x
CuPy GPU	0.58 s + 1.23 s BC	5.68 s	7.55 s	67.1x

Key Insights

Profiling-driven optimization is essential. Each implementation's bottlenecks were revealed through NVTX-annotated profiling: - CPU baseline showed assembly and solve as co-dominant bottlenecks - CPU parallel variants revealed that solve cannot be accelerated through CPU

parallelism - Numba JIT proved assembly could match GPU speed on CPU, isolating solve as the true barrier - GPU implementations shifted bottlenecks to BC application, then to solve again

CPU parallelization has fundamental limits. Threading achieved only 2.3x assembly speedup due to GIL contention. Multiprocessing, despite bypassing the GIL, performed worse due to IPC overhead. The solve phase, which uses SciPy's already-optimized BLAS, showed negligible improvement across all CPU variants.

JIT compilation is highly effective for CPU-bound kernels. Numba JIT achieved 18.6x assembly speedup through compilation alone, demonstrating that interpreter overhead, not algorithmic complexity, was the CPU assembly bottleneck.

GPU acceleration requires workload-appropriate scale. At 201 nodes, the CPU baseline (141.5 ms) outperformed all GPU implementations (658.6 ms best case) by 4.7x. GPU overhead (context initialization, kernel launch latency, memory transfers) only amortizes at scale.

Memory transfer optimization is critical for GPU performance. The 7.9x assembly improvement from Numba CUDA to CuPy came largely from keeping sparse matrix construction on GPU, eliminating a device-to-host transfer.

Scaling Behavior

The ratio of large-mesh to small-mesh runtime reveals each solver's scaling efficiency:

Solver	Small (201)	Large (1.36M)	Ratio
CPU Baseline	141.5 ms	506.3 s	3,578x
Numba JIT	219.4 ms	390.3 s	1,779x
Numba CUDA	1.18 s	14.94 s	12.7x
CuPy GPU	658.6 ms	7.55 s	11.5x

GPU solvers exhibit dramatically better scaling: a 6,750x increase in problem size (201 → 1.36M nodes) results in only 11-13x runtime increase. This sub-linear scaling reflects the GPU's massive parallelism absorbing the increased workload.

Limitations

Single GPU constraint. The current implementation targets a single GPU. Meshes exceeding GPU memory would require either out-of-core methods or multi-GPU distribution.

Solver algorithm. The conjugate gradient solver with Jacobi preconditioning, while effective, is not optimal for all problem types. Ill-conditioned systems may require algebraic multigrid or incomplete factorization preconditioners.

Profiling overhead. NVTX annotations and Nsight Systems instrumentation introduce minor overhead (~1-3%) that affects absolute timings, though relative comparisons remain valid.

Future Work

Solver optimization. The solve phase now dominates at 75% of runtime. Potential improvements include: - Mixed-precision solving (FP32 for iterations, FP64 for final refinement) - Batched small-matrix operations for preconditioner application - Alternative preconditioners (algebraic multigrid, sparse approximate inverse)

Adaptive solver selection. Implement automatic selection between CPU baseline (small meshes) and GPU solvers (large meshes) based on problem size, with the crossover point determined empirically around 10,000-50,000 elements.

Memory optimization. Explore unified memory (CUDA managed memory) to simplify the programming model and potentially improve performance for meshes near GPU memory limits.

Multi-GPU scaling. For very large meshes, domain decomposition with multi-GPU execution — potentially leveraging Dask for distributed computation — would extend the solver's capability beyond single-GPU memory constraints.

8. Annexes

8.1 - Solver Implementations Detailed Report

8.2 - FEMulator Pro Installation

8.3 - Project Proposal (Tutorial #1)