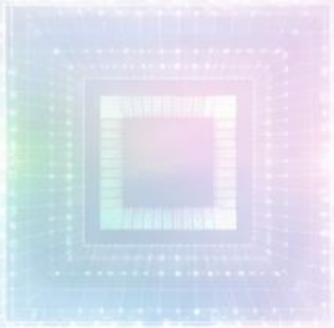
INSTITUTO UNIVERSITÁRIO DE LISBOA

CGAD



High-Performance Graphics Computing

TUTORIAL #2

HIGH-PERFORMANCE GPU-ACCELERATED **FINITE ELEMENT ANALYSIS**

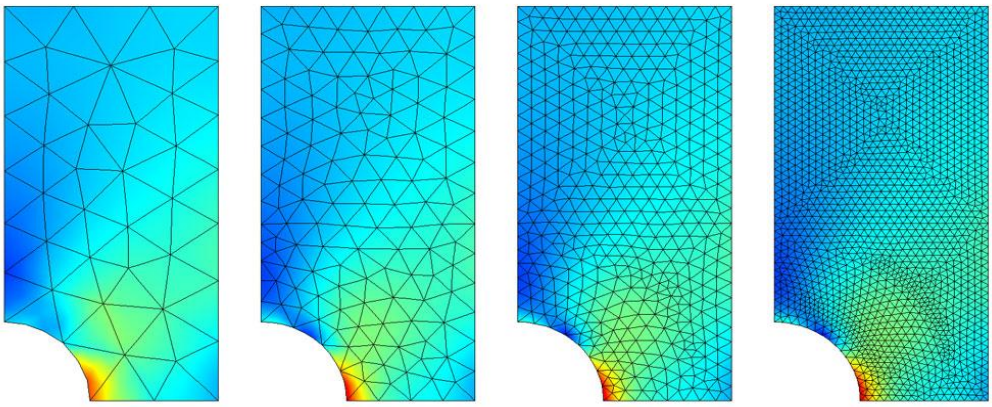
António Cruz (140129), **Bruno Santos** (140586), **Pedro Miranda** (129268), **Ricardo Kayseller** (95813)

Table of Contents

1. General Overview - 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.



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.

Elliptic 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 elliptic 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:

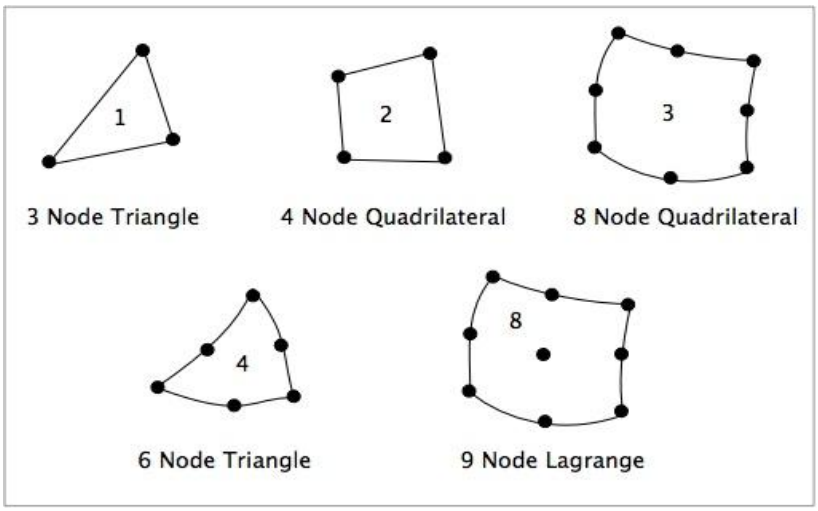
1. **Mathematical well-posedness**: Unique solution guaranteed under appropriate boundary conditions
2. **Symmetric positive-definite system**: Enables use of efficient iterative solvers (CG)
3. **Predictable convergence**: Facilitates consistent timing measurements
4. **Representative workload**: Assembly 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.



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 counter-clockwise 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:

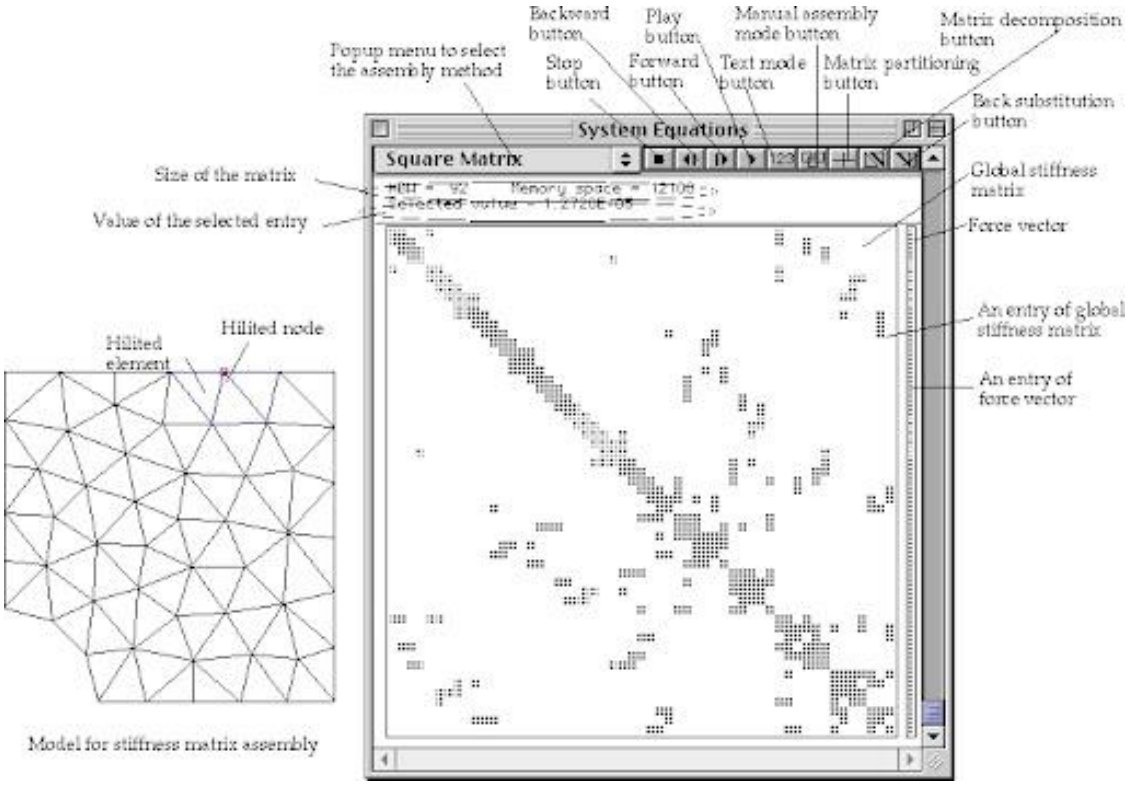
$$Ku = f$$

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.



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 supports, loads, or environmental interactions acting on the domain. The most commonly 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/m^3 (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 += (∇N^T · 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 access 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_0
r_0 = f - K u_0
p_0 = r_0

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 by a member of the group, 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. 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

3.2.2.1. NumPy and SciPy Ecosystem

The baseline implementation is built on Python's scientific computing 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.

3.2.2.2. 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.

3.2.2.3. Relevance for FEM

The sequential CPU baseline fulfills several essential roles in the FEM workflow:

- Provides a clear and traceable mapping between the mathematical formulation and the 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

3.2.3.1. Mesh Loading

Mesh data is loaded primarily from binary HDF5 files. This choice minimizes parsing overhead and ensures that input/output costs remain negligible relative to computation, even for large meshes.

3.2.3.2. System Assembly

The global stiffness matrix and load vector are assembled using a classical element-by-element FEM approach:

1. The global sparse matrix is initialized in a format optimized for incremental insertion.
2. Elements are processed sequentially.
3. For each element, an 8×8 local stiffness matrix and corresponding load contributions are computed using numerical quadrature.
4. Local contributions are scattered into the global sparse matrix.

After assembly, the global matrix is converted to a compressed sparse format optimized for sparse matrix–vector products during the solution phase. This two-phase strategy balances insertion efficiency during assembly with arithmetic efficiency during iterative solution.

3.2.3.3. Boundary Condition Application

Boundary conditions are applied after assembly using standard FEM techniques:

- **Robin boundary conditions (inlet)** are enforced through numerical integration of boundary contributions.
- **Dirichlet boundary conditions (outlet)** are imposed using the penalty method for implementation simplicity.

The computational cost of boundary condition application is small relative to assembly and solution phases.

3.2.3.4. Linear System Solution

The resulting linear system is solved using the Conjugate Gradient (CG) method provided by SciPy. To ensure robust and consistent convergence:

- The system is diagonally equilibrated to improve numerical conditioning.
- A Jacobi (diagonal) preconditioner is applied.

The same solver configuration and convergence criteria are used across all implementations, ensuring identical iteration counts and comparable numerical behavior.

3.2.3.5. Post-Processing

Post-processing computes derived quantities such as velocity fields and pressure from the solved potential field. These operations involve additional element-level loops and are executed sequentially.

While not dominant, post-processing introduces a measurable overhead for large meshes.

3.2.4. Optimization Techniques Applied

3.2.4.1. Sparse Matrix Format Selection

Different sparse matrix formats are employed at different stages of the computation:

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

This separation minimizes assembly overhead while ensuring efficient memory access during iterative solution.

3.2.4.2. Diagonal Equilibration

Prior to solving, the linear system is diagonally equilibrated to improve conditioning. This scaling reduces sensitivity to variations in element size and improves convergence behavior, particularly for large or heterogeneous meshes.

3.2.4.3. Preconditioning Strategy

A Jacobi (diagonal) preconditioner is employed within the Conjugate Gradient solver. Despite its simplicity, this preconditioner provides a favorable trade-off between implementation complexity and convergence robustness, ensuring stable and reproducible iteration counts.

3.2.4.4. Vectorized Inner Operations

Within each element computation, dense linear algebra operations are expressed using NumPy array operations. These operations are executed in optimized compiled libraries, partially mitigating Python interpreter overhead at the inner-kernel level.

3.2.5. Challenges and Limitations

3.2.5.1 Sequential Element Loop

The assembly phase relies on an explicit Python loop over all elements. For large meshes, this results in linear scaling dominated by interpreter overhead rather than arithmetic intensity.

3.2.5.2 Global Interpreter Lock (GIL) Constraints

Although numerical kernels release the GIL, Python-level control flow and sparse matrix indexing remain serialized. As a result, multi-threaded execution provides limited benefit for this implementation.

3.2.5.3 Sparse Matrix Insertion Overhead

Incremental updates to the global sparse matrix incur significant overhead due to dynamic memory allocation, object management, and indirect indexing. These costs dominate assembly time for large problem sizes.

3.2.5.4 Memory Access Patterns

Element assembly involves scattered reads of nodal data and scattered writes to the global sparse matrix. This access pattern exhibits poor spatial locality, leading to cache inefficiencies and increased memory traffic.

3.2.5.5 Observed Execution Behavior

3.2.5.5.1 Element-Level Execution and Interpreter Overhead

Assembly follows a strictly element-by-element execution model aligned with the FEM formulation. Performance is dominated by Python loop execution and sparse matrix indexing rather than floating-point computation, resulting in interpreter-bound behavior.

3.2.5.5.2 Sparse Matrix Format Trade-offs

Sparse matrix assembly is performed using a format optimized for incremental insertion, followed by conversion to a compressed format optimized for sparse matrix–vector operations.

This conversion introduces additional overhead but is required for efficient solver execution. In the baseline implementation, conversion overhead is amortized over multiple solver iterations.

3.2.5.5.3 Impact of Preconditioning on Convergence

Solver convergence is highly sensitive to preconditioning. In the absence of preconditioning, the Conjugate Gradient method exhibits significantly increased iteration counts, sensitivity to problem scaling, and potential convergence failure.

The Jacobi preconditioner improves numerical conditioning and stabilizes convergence with negligible computational overhead, ensuring consistent iteration counts across problem sizes.

3.2.5.5.4 Residual Evaluation and Solver Diagnostics

Convergence monitoring is based on explicit evaluation of the true residual norm rather than solver-internal estimates. This provides a consistent convergence criterion across implementations and enables early detection of numerical anomalies.

The additional cost of residual evaluation is limited to a sparse matrix–vector product per monitoring step and is negligible relative to overall solver runtime.

3.2.6. Performance Characteristics and Baseline Role

3.2.6.1 Expected Scalings

From an algorithmic perspective, the CPU baseline exhibits the following computational complexity:

Stage	Complexity	Dominant Factor
Mesh loading	O(N_nodes)	I/O bandwidth
Assembly	O(N_elements × 64 × 9)	Python loop overhead
Boundary condition application	O(N_boundary)	Minor relative cost
Linear system solution	O(iterations × nnz)	SpMV memory bandwidth
Post-processing	O(N_elements × 8 × 4)	Python loop overhead

The constant factors reflect the fixed size of element stiffness matrices and the numerical quadrature scheme employed.

3.2.6.2 Profiling Observations

For large meshes, the expected distribution of execution time is:

- **Assembly:** approximately 50–70% of total runtime
- **Solve:** approximately 20–40%, governed by sparse matrix–vector products and iteration count
- **Post-processing:** approximately 5–15%
- **Mesh I/O and boundary conditions:** typically below 5%

3.2.6.3 Baseline Role

The CPU baseline establishes the following reference points:

- **Correctness reference:** All alternative implementations must produce numerically equivalent results.
- **Performance floor:** Any parallel CPU or GPU-based approach must improve upon this execution time.
- **Solver behavior reference:** Convergence behavior and iteration counts are expected to remain consistent across implementations.

This implementation therefore defines the reference execution profile for all reported speedups, scalability analyses, and efficiency metrics.

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. 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

3.3.2.1 Python Threading and the Global Interpreter Lock

Python's Global Interpreter Lock (GIL) enforces serialized execution of Python bytecode, preventing true parallel execution of CPU-bound workloads across threads. This simplifies memory management but significantly constrains scalability for numerical applications implemented at the Python level.

However, many NumPy operations release the GIL during execution, including:

- Vectorized array arithmetic
- Dense linear algebra routines (BLAS/LAPACK)
- Element-wise mathematical kernels

This behavior enables limited concurrency when the computation is structured to maximize time spent inside GIL-released NumPy kernels, while minimizing Python-level control flow.

3.3.2.2 ThreadPoolExecutor Execution Model

The ThreadPoolExecutor abstraction provides a pool of reusable worker threads and a future-based execution model.

Key characteristics include:

- Persistent worker threads, reducing creation overhead
- Asynchronous task submission via Future objects
- Automatic synchronization and cleanup through context management
- Dynamic scheduling that enables basic load balancing

This abstraction simplifies parallel orchestration while preserving shared-memory access to NumPy arrays.

3.3.2.3 Implications for FEM Workloads

Relative to the CPU baseline, the expected impact of threading on FEM operations is mixed:

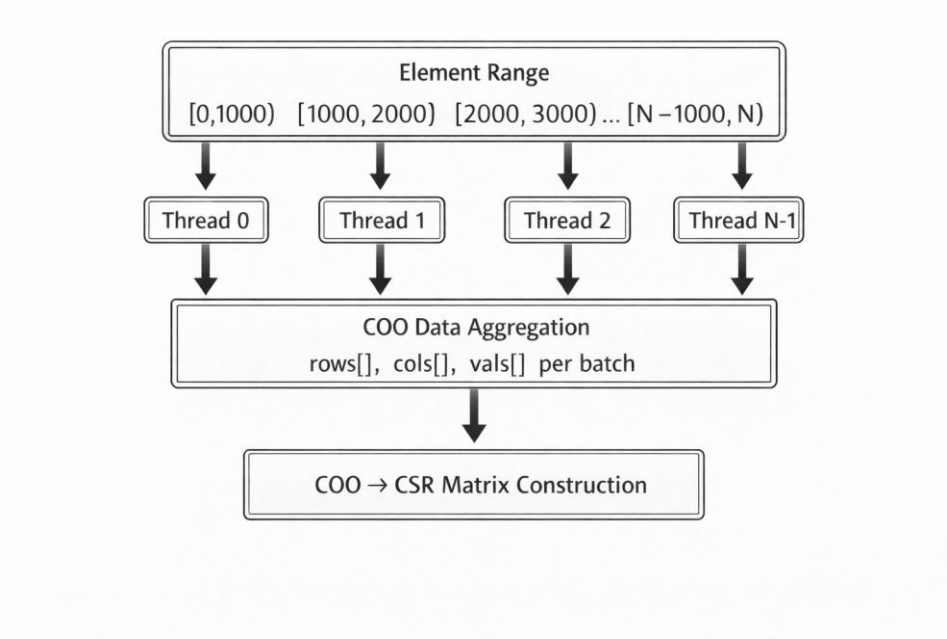
Operation	GIL Released	Expected Benefit
Python loop iteration	No	None
Sparse matrix indexing	No	None
NumPy dense kernels	Yes	Moderate
Element-wise NumPy ops	Yes	Moderate

The overall benefit therefore depends on increasing the ratio of GIL-free numerical computation relative to GIL-held 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:



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 writes 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 complete, 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

3.3.4.1 Batch Size Selection

Batch size is a critical tuning parameter controlling the balance between coordination overhead and load balance. Empirical testing indicates that batch sizes between 500 and 2000 elements provide the best trade-off for typical problem sizes.

3.3.4.2 Pre-allocation of Thread-Local Buffers

Each batch allocates fixed-size arrays once per thread invocation, avoiding repeated dynamic memory allocation within inner loops. This reduces overhead and improves cache locality.

3.3.4.3 Inlined Element Computation

Element stiffness computation is implemented directly within the batch function to minimize function call overhead and maximize time spent in GIL-released NumPy kernels.

3.3.4.4 Shared Read-Only Data

Mesh coordinates, connectivity, and quadrature data are shared across threads as read-only NumPy arrays. This avoids memory duplication while maintaining thread safety.

3.3.5. Challenges and Limitations

3.3.5.1 GIL Contention

Despite NumPy releasing the GIL during numerical kernels, a substantial fraction of execution time remains GIL-bound due to Python loops, indexing, and sparse data manipulation. This fundamentally limits scalability.

3.3.5.2 Memory Bandwidth Saturation

All threads share the same memory subsystem, leading to contention and diminishing returns beyond a modest number of threads.

3.3.5.3 Thread Management Overhead

Task submission, scheduling, and result aggregation introduce non-negligible overhead, which dominates execution time for small problem sizes.

3.3.5.4 Limited Solver Parallelism

The solver phase remains effectively sequential at the Python level. While underlying BLAS libraries may use threads, overall solver performance is memory-bound and does not benefit significantly from additional Python threading.

3.3.6. Performance Characteristics and Role

3.3.6.1 Expected Scaling Behavior

Thread-level parallelism yields sub-linear speedup governed by Amdahl's Law. Only portions of the assembly and post-processing phases benefit from concurrent execution.

3.3.6.2 Practical Speedup Regime

Empirical behavior typically shows:

- Modest gains with 2–4 threads
- Diminishing returns beyond 4–8 threads
- Potential slowdowns when contention outweighs parallel benefits

3.3.6.3 Role in the Implementation Suite

This implementation serves as an intermediate reference between the sequential CPU baseline and more aggressive parallelization strategies. It highlights the structural limitations imposed by the GIL and motivates approaches that bypass it entirely.

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. 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.

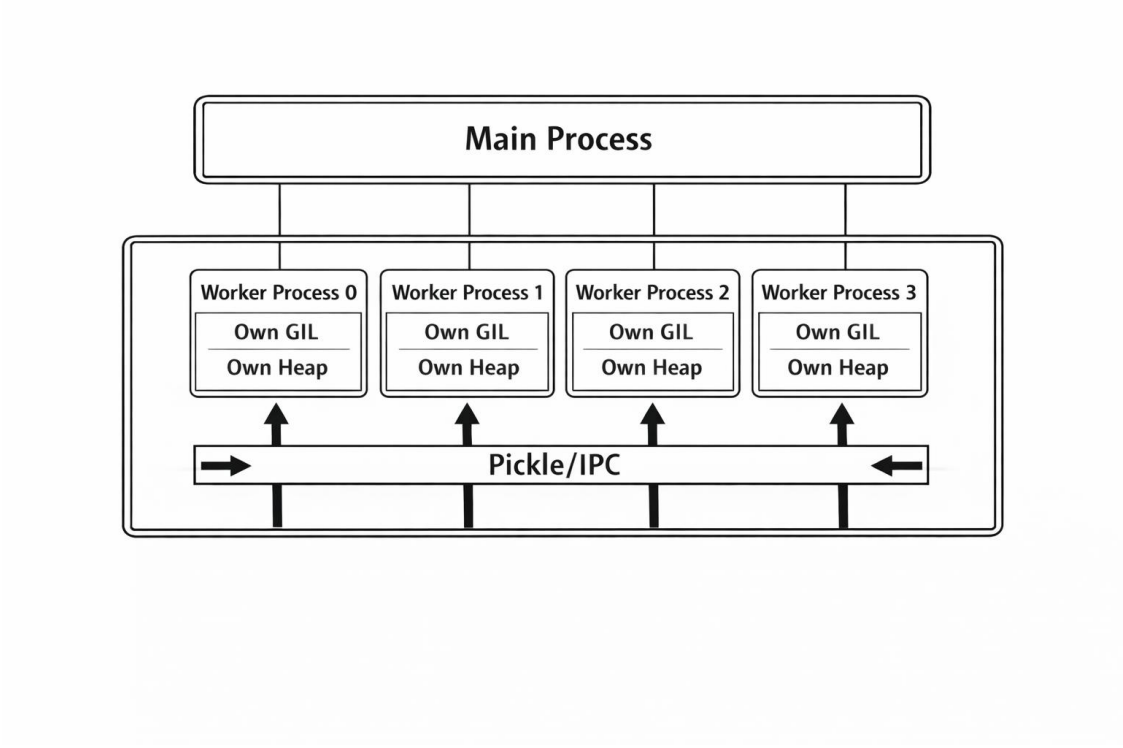
Attribute	Description
Technology	multiprocessing.Pool (Python stdlib)
Execution Model	Multi-process, separate memory spaces

Role	True CPU parallelism and GIL bypass demonstration
Dependencies	NumPy, SciPy, multiprocessing (stdlib)

3.4.2. Technology Background

3.4.2.1 Python Multiprocessing

The multiprocessing execution model spawns multiple independent worker processes. Each worker runs its own Python interpreter with an isolated memory space and its own Global Interpreter Lock.



Key characteristics:

- **Separate memory**: Each process has isolated address space
- **Independent GIL**: No GIL contention between processes
- **IPC required**: Data must be serialized (pickled) for transfer
- **Higher overhead**: Process creation and coordination are more expensive than threads

3.4.2.2 multiprocessing.Pool

The Pool abstraction manages a fixed number of worker processes and distributes work among them 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

Compared to the threaded implementation, which can collect results asynchronously, map() returns results in submission order, simplifying aggregation.

3.4.2.3 Pickle Serialization

Inter-process communication relies on pickle serialization:

- All input arguments are serialized and sent to workers
- Return values are serialized and sent back to the main process
- Worker functions must be defined at module level
- Large arrays incur significant serialization overhead

3.4.2.4 Relevance for FEM

Relative to threading, multiprocessing offers true parallelism but introduces additional overheads:

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 FEM assembly with element-independent computation, multiprocessing can approach near-linear speedup if IPC overhead is amortized.

3.4.3. Implementation Strategy

3.4.3.1 Module-Level Function Requirement

A critical constraint of multiprocessing is that worker logic must be defined at module level to be serializable. This imposes structural constraints compared to class-centric designs.

All computational kernels and batch-processing logic must therefore reside at top-level scope.

3.4.3.2 Batch Processing Architecture

The global element set is partitioned into contiguous batches. Each batch is processed independently by a worker process.

Each batch contains:

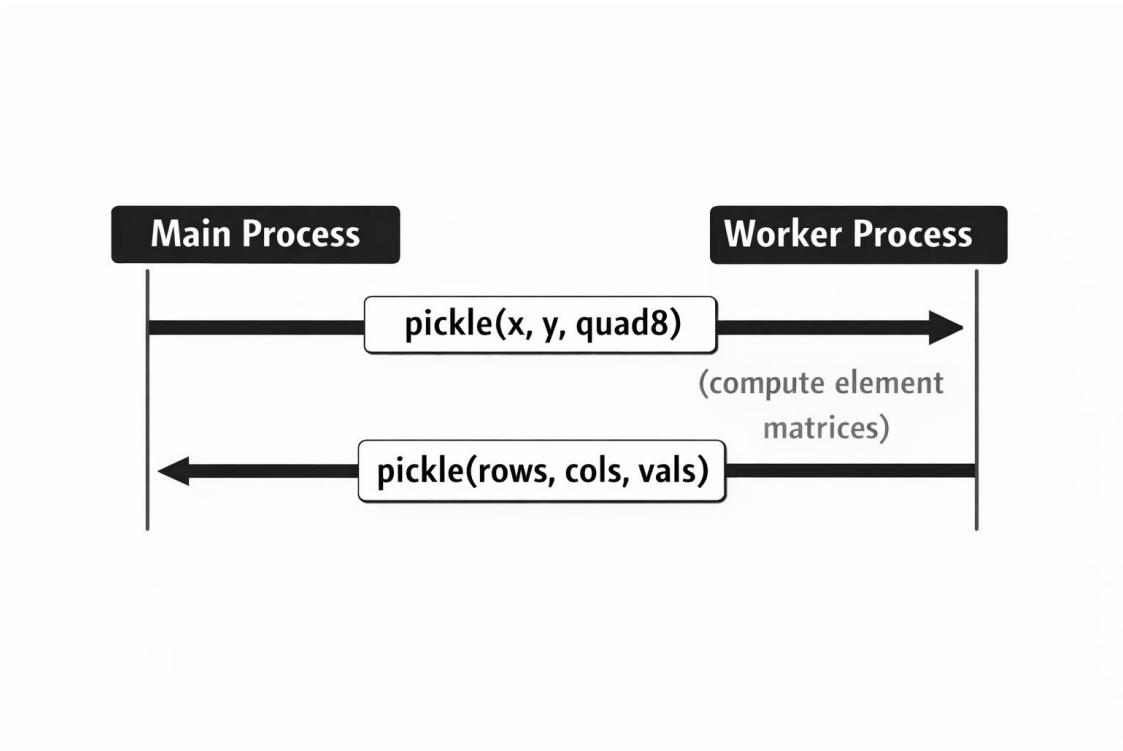
- Element index range
- Coordinate data
- Connectivity information

- Quadrature data

Batching amortizes IPC overhead and reduces scheduling frequency.

3.4.3.3 Data Serialization Implications

Unlike threading, multiprocessing requires explicit data transfer per batch:



For large meshes, serialization frequency and volume become dominant performance constraints.

3.4.3.4 COO Assembly Strategy

As in the threaded implementation, assembly uses a coordinate-based sparse representation:

- Workers generate independent COO contributions
- The main process concatenates all partial results
- COO → CSR conversion merges duplicates automatically

This avoids concurrent updates to shared sparse structures.

3.4.3.5 Post-Processing Parallelization

Derived field computation follows the same batching strategy. The solution field must also be serialized and transmitted to workers, increasing IPC overhead during post-processing.

3.4.3.6 Linear System Solution

The linear solver is executed in the main process using the same configuration as other implementations, ensuring consistent convergence behavior and numerical equivalence.

3.4.4. Optimization Techniques Applied

3.4.4.1 Batch Size for IPC Amortization

Larger batches reduce IPC frequency but limit 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

3.4.4.2 Tuple-Based Argument Packing

All data required for batch processing is grouped and transmitted together. This simplifies orchestration but increases serialization cost per task.

3.4.4.3 COO Assembly for Parallel Safety

Independent per-batch output generation avoids shared-state mutation. Duplicate summation is deferred to the final sparse matrix conversion.

3.4.4.4 Worker Count Configuration

Worker count typically matches available CPU cores. While this maximizes parallelism, it also increases memory duplication and IPC traffic.

3.4.5. Challenges and Limitations

3.4.5.1 Serialization Overhead

Serialization dominates overhead:

- Input data is serialized for each batch
- Output data is serialized back to the main process
- Small batch sizes exacerbate overhead

3.4.5.2 Memory Duplication

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)

3.4.5.3 Process Startup Cost

Process creation introduces fixed overhead:

Component	Typical Time

Fork/spawn	10–50 ms per process
Interpreter initialization	50–100 ms per process
Module imports	Variable
Pool creation (4 workers)	200–500 ms

3.4.5.4 Limited Shared State

Workers cannot directly modify shared data. All results must be merged in the main process, introducing a sequential aggregation phase.

3.4.5.5 Pickle Constraints

Serialization requirements restrict code structure and increase implementation complexity.

3.4.6. Performance Characteristics

3.4.6.1 Scaling Model

Multiprocessing performance can be approximated as:

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

where:

- (T_{serial}): Sequential computation time
- (N): Number of worker processes
- (T_{overhead}): IPC and process management overhead

3.4.6.2 Break-Even Analysis

Multiprocessing becomes beneficial when computation dominates overhead:

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

3.4.6.3 Memory Bandwidth Considerations

All processes share the same memory subsystem. Bandwidth saturation and NUMA effects can limit scaling on multi-socket systems.

3.4.6.4 Comparison with Threading

Relative 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. 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

3.5.2.1 Just-In-Time Compilation

Numba is a Just-In-Time compiler that translates Python functions into optimized machine code using the LLVM compiler infrastructure. The compilation pipeline transforms Python source code through intermediate representations into native CPU instructions.

Key advantages relative to interpreted execution include:

- Elimination of Python interpreter overhead
- Native execution speed comparable to C/Fortran
- Automatic loop unrolling and inlining
- SIMD vectorization via LLVM
- Execution without Global Interpreter Lock (GIL) constraints

3.5.2.2 The @njit Compilation Model

The implementation uses Numba's @njit decorator to enforce *nopython* mode. In this mode:

- All Python bytecode is bypassed
- Type inference is resolved at compile time
- Unsupported Python and NumPy features are disallowed

Compilation caching is enabled to persist generated machine code across executions, amortizing compilation cost.

3.5.2.3 Parallel Execution with prange

Parallelism is achieved using Numba's prange, which distributes loop iterations across CPU threads. Unlike Python threading:

- Execution occurs without GIL constraints
- Threads operate in shared memory
- Work distribution follows an OpenMP-style model
- Near-linear speedup is achievable for independent iterations

3.5.2.4 Relevance for FEM

For FEM workloads, JIT compilation directly targets the dominant computational bottlenecks:

- Element stiffness matrix computation
- Element-level assembly loops

- Derived field computation

Sparse matrix construction and iterative solvers remain in SciPy, preserving numerical equivalence with previous implementations.

3.5.3. Implementation Strategy

3.5.3.1 Function-Level JIT Compilation

All computational kernels are implemented as Numba-compiled functions. Element stiffness computation, boundary condition evaluation, and post-processing kernels are explicitly written using loop-based formulations compatible with Numba's *nopython* mode.

This ensures that the entire element-level computation executes in compiled code without interpreter intervention.

3.5.3.2 Parallel Element Assembly

Global assembly is implemented through a parallel loop over elements. Each iteration:

1. Gathers nodal coordinates for a single element
2. Computes the local stiffness matrix and load vector
3. Writes local contributions into pre-allocated COO arrays

Parallelism is applied at the element level using prange, ensuring that each element is processed independently and concurrently.

3.5.3.3 Explicit Loop-Based Kernels

Unlike vectorized NumPy formulations, all numerical operations are expressed as explicit loops. This allows LLVM to:

- Fully unroll small fixed-size loops
- Inline function calls
- Eliminate temporary array allocations
- Apply SIMD vectorization to inner loops

This explicit structure is critical for achieving high performance in JIT-compiled FEM kernels.

3.5.3.4 Parallel Post-Processing

Derived field computation (velocity and magnitude) follows the same compiled parallel pattern. Each element gathers local solution values, evaluates gradients at integration points, and stores results in element-wise output arrays.

3.5.3.5 Solver Integration

Sparse matrix construction and the Conjugate Gradient solver are executed outside Numba using SciPy. The JIT boundary is placed at the array level: Numba generates COO-format data, and SciPy handles sparse matrix assembly and solution.

3.5.4. Optimization Techniques Applied

3.5.4.1 Interpreter Elimination

The dominant optimization is the complete removal of Python interpreter overhead from element-level computation. All inner loops execute as native machine code.

3.5.4.2 Loop Unrolling and Inlining

Small, fixed-size loops are fully unrolled by LLVM. Nested function calls within JIT-compiled code are typically inlined, eliminating function call overhead.

3.5.4.3 SIMD Vectorization

LLVM applies SIMD vectorization to inner arithmetic loops where data layout permits, enabling multiple operations per CPU cycle.

3.5.4.4 Memory Access Optimization

COO data is written sequentially in element-major order, improving cache locality and reducing memory write overhead.

3.5.4.5 Shared-Memory Parallelism

Parallel execution uses shared memory without data duplication, preserving memory efficiency relative to multiprocessing-based approaches.

3.5.5. Challenges and Limitations

3.5.5.1 Compilation Overhead

Initial execution incurs JIT compilation cost on the order of hundreds of milliseconds. This overhead is amortized for large problems and eliminated on subsequent runs via caching.

3.5.5.2 Limited NumPy and SciPy Support

Only a subset of NumPy functionality is supported in *nopython* mode. Unsupported operations must be rewritten using explicit loops, increasing implementation complexity.

3.5.5.3 Debugging Complexity

Debugging JIT-compiled code is more difficult than debugging pure Python code. Stack traces may be less informative, and interactive debuggers cannot step into compiled regions.

3.5.5.4 Memory Allocation in Parallel Regions

Allocating arrays inside parallel loops increases overhead. Performance is improved by minimizing allocations within parallel regions.

3.5.5.5 Solver Dominance at Scale

As assembly and post-processing accelerate, the sparse solver increasingly dominates total runtime and limits further speedup.

3.5.6. Performance Characteristics and Role

3.5.6.1 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

3.5.6.2 Profiling Observations

For large meshes:

- Assembly and post-processing times are reduced by one to two orders of magnitude
- Solver time becomes the dominant runtime component
- Parallel efficiency remains high until memory bandwidth saturation

3.5.6.3 Role in the Implementation Suite

The Numba JIT CPU implementation represents the highest-performing CPU-based solution in this study. It establishes the upper bound for shared-memory CPU execution and serves as the reference point for evaluating GPU-based 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. 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

3.6.2.1 Numba CUDA Programming Model

Numba extends its JIT compilation framework to NVIDIA GPUs through the @cuda.jit decorator. Python functions annotated as CUDA kernels are compiled to PTX (Parallel Thread Execution) code and executed on the GPU.

GPU execution follows the CUDA SIMT model, where thousands of lightweight threads execute the same kernel concurrently.

3.6.2.2 CUDA Execution Hierarchy

GPU kernels are launched using a hierarchical structure:

- **Grid**: All threads launched by a kernel invocation
- **Block**: A group of threads that can cooperate via shared memory
- **Thread**: The smallest execution unit
- **Warp**: A group of 32 threads executing in lockstep

Threads are indexed using cuda.grid(1), enabling a direct mapping between thread index and FEM element index.

3.6.2.3 GPU Memory Hierarchy

GPU memory is organized in multiple tiers:

- **Registers**: Fastest, thread-private storage
- **Local memory**: Thread-private, may spill to device memory
- **Shared memory**: Fast, block-shared memory
- **Global memory**: Large but high-latency device memory

The implementation primarily uses registers and local memory for element-level arrays and global memory for mesh data and assembled results.

3.6.2.4 Relevance for FEM

GPU execution is particularly well suited for FEM workloads with large numbers of independent elements. Element-level stiffness matrix computation exhibits high arithmetic intensity and minimal inter-thread dependency, making it ideal for SIMT execution.

3.6.3. Implementation Strategy

3.6.3.1 Kernel-Based Element Assembly

Element assembly is implemented as a GPU kernel where each thread processes a single element. For each element, the 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

All computations are performed using explicit loops compatible with Numba CUDA.

3.6.3.2 Thread-to-Element Mapping

A one-thread-per-element strategy is used:

- Each GPU thread computes one element
- Threads are launched in 1D grids
- Excess threads exit early if the element index exceeds the mesh size

This mapping ensures uniform work distribution and avoids inter-thread synchronization during element computation.

3.6.3.3 Local Memory Usage

Per-thread temporary arrays are allocated using `cuda.local.array`, including:

- Element DOF indices
- Nodal coordinates
- Local stiffness matrix and load vector
- Shape functions and derivatives

These arrays are private to each thread, eliminating race conditions and synchronization overhead.

3.6.3.4 Force Vector Assembly with Atomics

Because multiple elements share nodes, assembly of the global force vector requires atomic operations. Thread-safe accumulation is performed using `cuda.atomic.add` to ensure correctness.

3.6.3.5 Post-Processing on GPU

Derived field computation (velocity and magnitude) is implemented as a separate GPU kernel. Each thread processes one element, evaluates gradients at integration points, and stores averaged results.

3.6.3.6 Solver Integration

The linear system is solved on the GPU using CuPy's sparse Conjugate Gradient solver. Sparse matrices are converted to CuPy formats, allowing the entire solution phase to execute on the GPU before transferring results back to CPU memory.

3.6.4. Optimization Techniques Applied

3.6.4.1 Massive Parallelism

The GPU executes tens of thousands of threads concurrently, enabling element-level parallelism far beyond CPU core counts.

3.6.4.2 Block Size Tuning

Kernel launch configuration is tuned to balance occupancy and register pressure. A block size of 128 threads provides good performance for the register-heavy FEM kernels.

3.6.4.3 Memory Coalescing

Global memory accesses are structured so that consecutive threads write to contiguous memory regions, improving memory coalescing and bandwidth utilization.

3.6.4.4 Register and Local Memory Management

Small per-thread arrays are kept in registers where possible. Larger arrays may spill to local memory, but remain private and efficiently cached.

3.6.4.5 Warp Divergence Minimization

Kernel control flow is designed to minimize conditional branches. Aside from bounds checking at kernel entry, all threads follow identical execution paths.

3.6.5. Challenges and Limitations

3.6.5.1 Debugging Complexity

Debugging GPU kernels is significantly more difficult than CPU code. Python debuggers cannot be used inside kernels, and runtime errors may lead to silent failures or kernel crashes.

3.6.5.2 Limited NumPy Support

Only a restricted subset of NumPy functionality is available in CUDA kernels. Unsupported operations must be reimplemented using explicit arithmetic and loops.

3.6.5.3 Atomic Operation Overhead

Atomic updates to the global force vector introduce serialization and reduce scalability. While acceptable for this problem, atomics may become a bottleneck for higher connectivity or more complex FEM formulations.

3.6.5.4 Memory Transfer Overhead

Data must be explicitly transferred between host and device memory. For small problem sizes, PCIe transfer overhead can dominate execution time.

3.6.5.5 Partial CPU-GPU Workflow

Some tasks, such as COO index generation and boundary condition application, remain on the CPU due to complexity or limited performance benefit on GPU.

3.6.6. Performance Characteristics and Role

3.6.6.1 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

3.6.6.2 Profiling Observations

For large meshes:

- GPU occupancy reaches 50–75%
- Assembly speedup is substantial compared to CPU JIT
- Sparse solver performance becomes memory-bound
- End-to-end speedup improves with increasing problem size

3.6.6.3 Role in the Implementation Suite

The Numba CUDA implementation is the first fully GPU-based approach in the study. It demonstrates the feasibility of GPU acceleration using Python-native kernels and establishes a reference point for evaluating raw CUDA kernel implementations.

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. Implementation 6: GPU CuPy (RawKernel)

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

3.7.2.1 CuPy Overview

CuPy is a NumPy-compatible array library for GPU computing. It provides:

- **Drop-in NumPy replacement**: import cupy as cp mirrors NumPy API
- **GPU arrays**: Data stored in GPU memory (VRAM)
- **Sparse matrices**: CSR/CSC/COO formats on GPU
- **Iterative solvers**: CG, GMRES, etc. running entirely on GPU
- **RawKernel**: Direct CUDA C/C++ kernel execution

3.7.2.2 RawKernel Interface

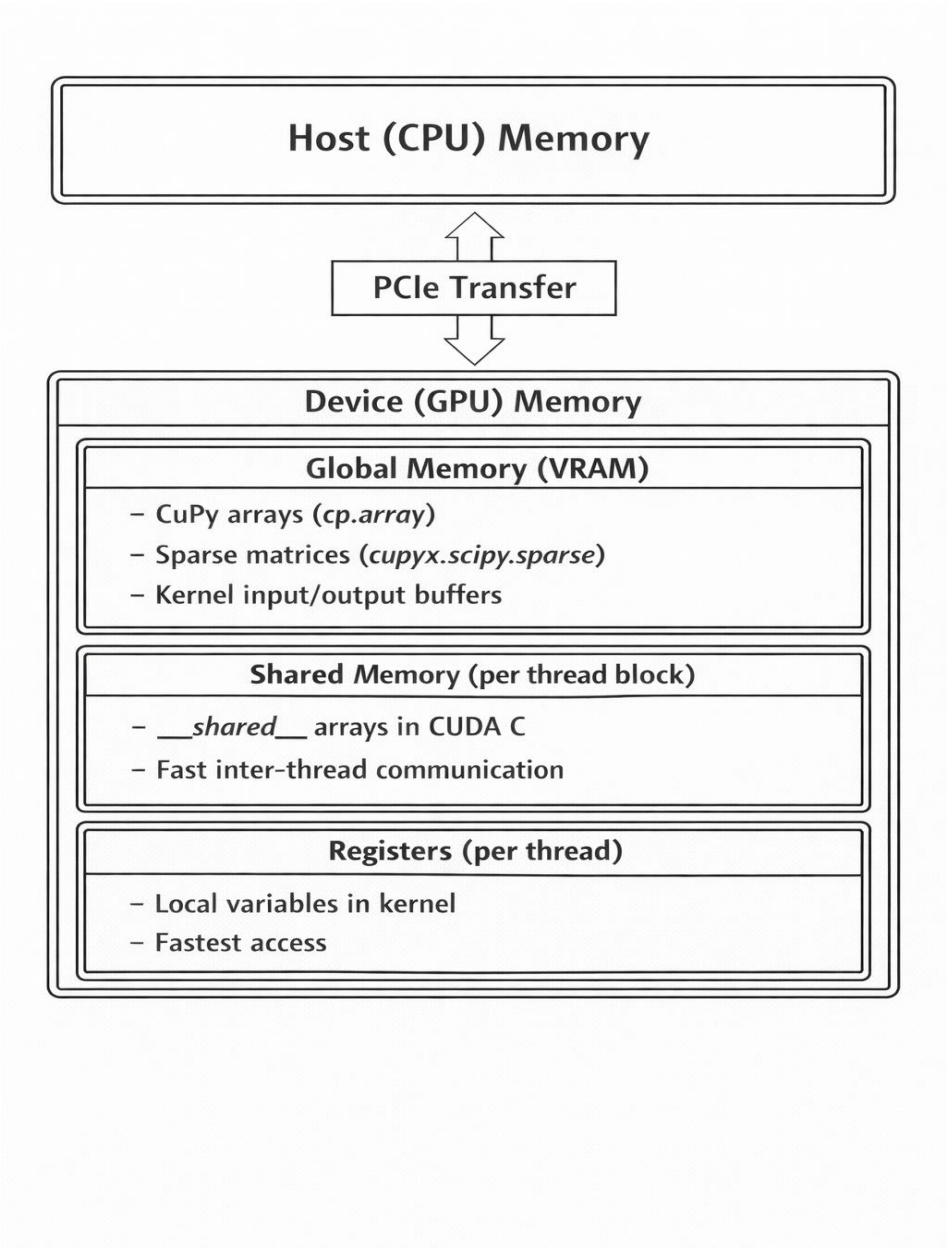
CuPy's RawKernel allows embedding CUDA C code directly in Python. This provides:

- Full CUDA C feature set
- Maximum performance (no Python overhead in kernel)
- Direct control over memory, synchronization, shared memory
- Compiled once, cached for reuse

3.7.2.3 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
Debugging	Python-like	GPU debugger
Compilation	JIT per function	JIT per kernel string

3.7.2.4 GPU Memory Model



3.7.3. Implementation Strategy

3.7.3.1 CUDA C Kernel Architecture

The implementation defines two primary CUDA C kernels as string literals embedded in Python:

Assembly Kernel (quad8_assembly_kernel): - One thread per element - Computes 8×8 element stiffness matrix - Writes 64 values to global COO value array - Atomic update to global force vector

Post-Processing Kernel (quad8_postprocess_kernel): - One thread per element - Computes velocity gradient at 4 Gauss points - Averages to element centroid velocity - Writes velocity components and magnitude

3.7.3.2 Kernel Source Structure

The assembly kernel follows a CUDA C structure with:

- Thread index derived from blockIdx, blockDim, and threadIdx
- Thread-local arrays for element data and local matrices
- Fixed integration loops matching the CPU formulation
- Scatter step writing flattened element stiffness matrix values
- Atomic force vector accumulation

3.7.3.3 GPU-Accelerated COO Index Generation

Unlike the Numba CUDA version which generates COO indices on CPU, this implementation uses vectorized CuPy operations on GPU:

- Generates all (N_{el}) row indices in parallel
- Generates all (N_{el}) column indices in parallel
- Avoids CPU-GPU synchronization for index generation
- Uses CuPy's optimized array operations for index construction

The kernel computes only the COO values; index arrays are generated on GPU.

3.7.3.4 Sparse Matrix Construction

After kernel execution, COO data is converted to CSR:

- COO sparse matrix is created on GPU using CuPy sparse
- COO → CSR conversion merges duplicates automatically
- Entire sparse matrix remains GPU-resident

3.7.3.5 GPU Sparse Solver

The linear system is solved entirely on GPU using CuPy's sparse solvers:

- Diagonal equilibration is performed on GPU
- Jacobi preconditioning is implemented on GPU using a linear operator abstraction
- Conjugate Gradient (CG) runs fully on GPU
- Solution is de-equilibrated on GPU

All solver operations (SpMV, vector updates, dot products) run without CPU round-trips.

3.7.3.6 GPU Post-Processing

Velocity computation is performed using a dedicated RawKernel:

- One thread per element
- Gradient evaluated at 4 Gauss points
- Element-averaged velocity and magnitude stored in GPU arrays
- Pressure computed from Bernoulli using vectorized GPU operations

3.7.4. Optimization Techniques Applied

3.7.4.1 CUDA C Shape Function Derivatives

Shape function derivatives are computed inline inside the kernel using explicit CUDA C expressions, avoiding function call overhead and enabling compiler optimization.

3.7.4.2 Jacobian and Inverse in CUDA C

The Jacobian matrix, determinant, and inverse are computed explicitly inside the kernel:

- Explicit loops over the 8 nodes
- Fixed-size operations suitable for unrolling
- Determinant and inverse computed directly from 2×2 Jacobian

3.7.4.3 Atomic Force Vector Update

Force vector accumulation is performed using CUDA atomics (atomicAdd) to ensure correctness when multiple elements share nodes.

3.7.4.4 Solver Fallback Strategy

The implementation includes a solver fallback mechanism:

- Attempts CG solve first
- Falls back to GMRES if CG fails
- Improves robustness under cases of numerical difficulty

3.7.5. Challenges and Limitations

3.7.5.1 CUDA C Complexity

RawKernel development requires CUDA expertise, including:

- Thread hierarchy reasoning
- Memory hierarchy and access optimization
- Occupancy, register pressure, and divergence control

This increases development cost relative to Python-kernel approaches.

3.7.5.2 Debugging Challenges

Debugging RawKernel code is harder than Python or Numba CUDA:

- Python debugger is not applicable inside kernels
- printf debugging is limited and expensive
- Memory errors can crash kernels without clear error traces
- Race conditions are difficult to diagnose

3.7.5.3 Kernel Compilation Overhead

RawKernel code is JIT-compiled on first use:

Kernel	Compilation Time

Assembly	~200-500 ms
Post-processing	~100-200 ms

CuPy caches compiled kernels so subsequent executions have near-zero compilation cost.

3.7.5.4 GPU Memory Constraints

Large problems can be limited by GPU VRAM:

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

For very large meshes, multi-GPU or out-of-core strategies may be required.

3.7.5.5 CuPy Sparse Solver Limitations

CuPy sparse solvers have limitations relative to SciPy:

- Fewer preconditioner options
- Some solver features less mature
- Occasional numerical differences

GMRES fallback addresses robustness concerns.

3.7.6. Performance Characteristics and Role

3.7.6.1 GPU Utilization Analysis

Metric	Typical Value	Optimization Target
Occupancy	50-75%	Register pressure
Memory throughput	70-85% peak	Coalescing
Compute utilization	60-80%	Algorithm efficiency

3.7.6.2 Performance Breakdown

Expected time distribution for 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

3.7.6.3 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

3.7.6.4 Comparison with CPU Implementations

Aspect	CPU Baseline	Numba CPU	GPU CuPy
Assembly	O(N) sequential	O(N/P) parallel	O(N/T) massively parallel
Threads/cores	1	4-32	1000s
Memory 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

At this stage of the project, the experimental evaluation is conducted using a selected set of servers (**ids: #1/#4/#5**) and problem sizes. In a subsequent phase, the study will be extended to include a larger pool of computational servers as well as significantly larger models, with increased numbers of nodes and elements, in order to further highlight performance, scalability, and hardware sensitivity. The current setup therefore represents an initial and controlled benchmarking baseline.

Contributing Servers

Benchmark data aggregated from **5 servers**:

#	Hostname	CPU	Cores	RAM	GPU	VRAM	Records
1	DESKTOP-3MCDHQ7	AMD64 Family 25 Model 97 St...	12	-	NVIDIA GeForce RT...	15.9 GB	237
2	DESKTOP-B968RT3	AMD64 Family 25 Model 97 St...	12	-	NVIDIA GeForce RT...	15.9 GB	33
3	KRATOS	Intel64 Family 6 Model 183 ...	28	-	NVIDIA GeForce RT...	12.0 GB	81
4	MERCURY	13th Gen Intel(R) Core(TM) ...	20	94.3 GB	NVIDIA GeForce RT...	24.0 GB	275
5	RICKYROG700	Intel64 Family 6 Model 198 ...	24	-	NVIDIA GeForce RT...	31.8 GB	238

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	11,973,636
Backward-Facing Step	XL	1,283,215	426,686	20,066,869
Elbow 90°	XS	411	111	5,063
Elbow 90°	M	161,984	53,344	2,503,138
Elbow 90°	L	623,153	206,435	9,712,725
Elbow 90°	XL	1,044,857	346,621	16,304,541
S-Bend	XS	387	222	4,109
S-Bend	M	196,078	64,787	3,048,794
S-Bend	L	765,441	254,034	11,952,725
S-Bend	XL	1,286,039	427,244	20,097,467
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	20,186,313

Venturi	XS	341	86	4,061
Venturi	M	194,325	64,334	3,023,503
Venturi	L	763,707	253,704	11,934,351
Venturi	XL	1,284,412	427,017	20,083,132
Y-Shaped	XS	201	52	2,571
Y-Shaped	M	195,853	48,607	2,336,363
Y-Shaped	L	772,069	192,308	9,242,129
Y-Shaped	XL	1,357,953	338,544	16,265,217

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

In line with this scope, the benchmarks presented below are conducted on three representative systems and on meshes with a limited number of nodes and elements, ensuring consistent and reproducible measurements:

System	GPU Model	VRAM	Benchmark Relevance
RICKYROG700	RTX 5090	31.8 GB	Upper performance ceiling
MERCURY	RTX 4090	24 GB	High-end reference GPU
DESKTOP-B968RT3	RTX 5060 Ti	15.9 GB	Mid-range GPU

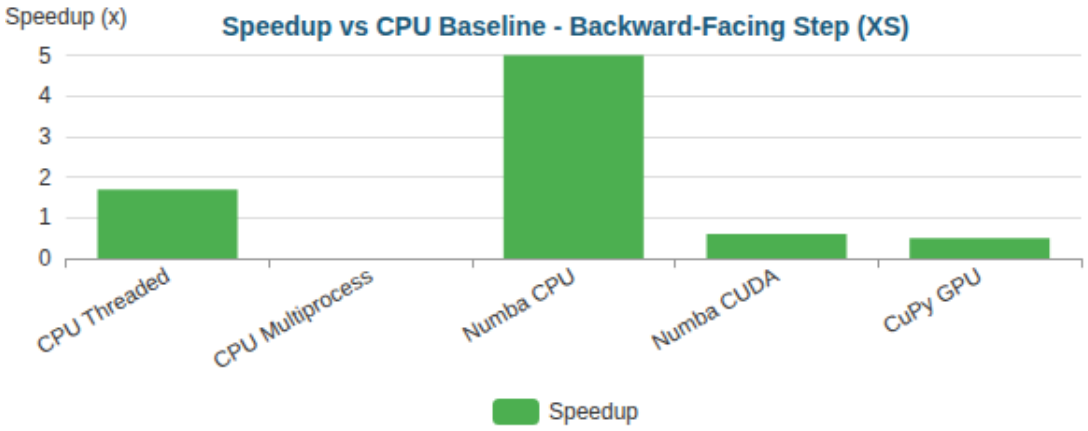
4.5 RTX 5090 Performance

Key results from performance benchmarks comparing FEM solver implementations.

Backward-Facing Step (XS) (287 nodes)

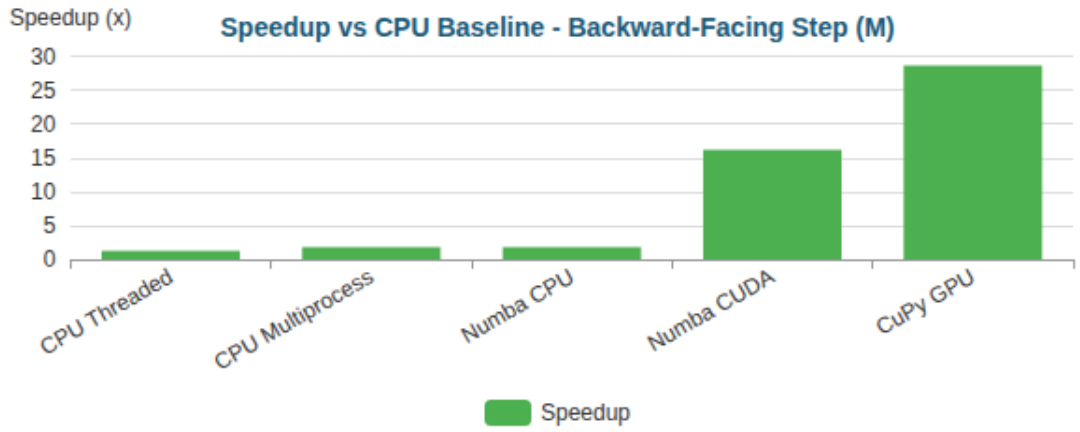
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	30ms ± 2ms	1.0x	3

CPU Threaded	18ms ± 4ms	1.7x	3
CPU Multiprocess	996ms ± 48ms	0.0x	3
Numba CPU	<0.01s ± 0ms	5.0x	3
Numba CUDA	49ms ± 6ms	0.6x	3
CuPy GPU	58ms ± 0ms	0.5x	3



Backward-Facing Step (M) (195,362 nodes)

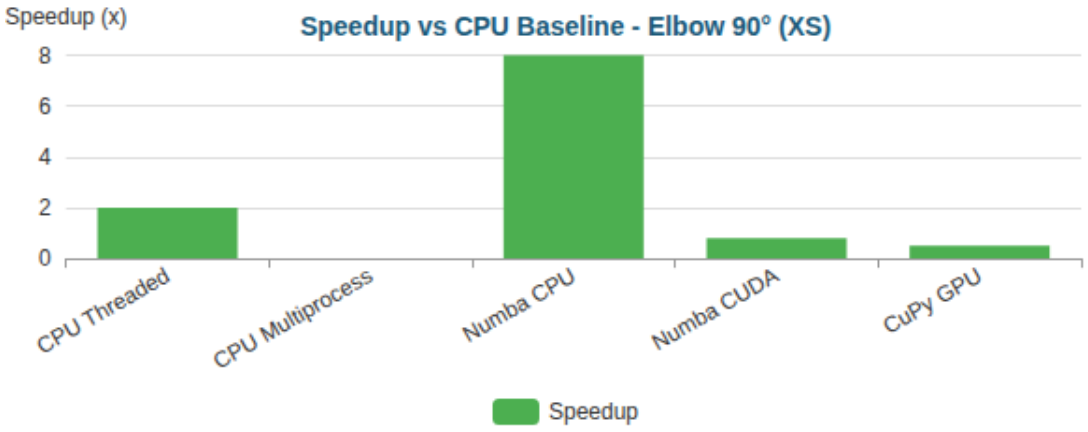
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	41.59s ± 0.12s	1.0x	3
CPU Threaded	31.23s ± 0.18s	1.3x	3
CPU Multiprocess	23.66s ± 0.22s	1.8x	3
Numba CPU	23.41s ± 0.17s	1.8x	3
Numba CUDA	2.56s ± 0.10s	16.2x	3
CuPy GPU	1.45s ± 0.01s	28.6x	3



Elbow 90° (XS) (411 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	42ms ± 5ms	1.0x	3
CPU Threaded	21ms ± 4ms	2.0x	3

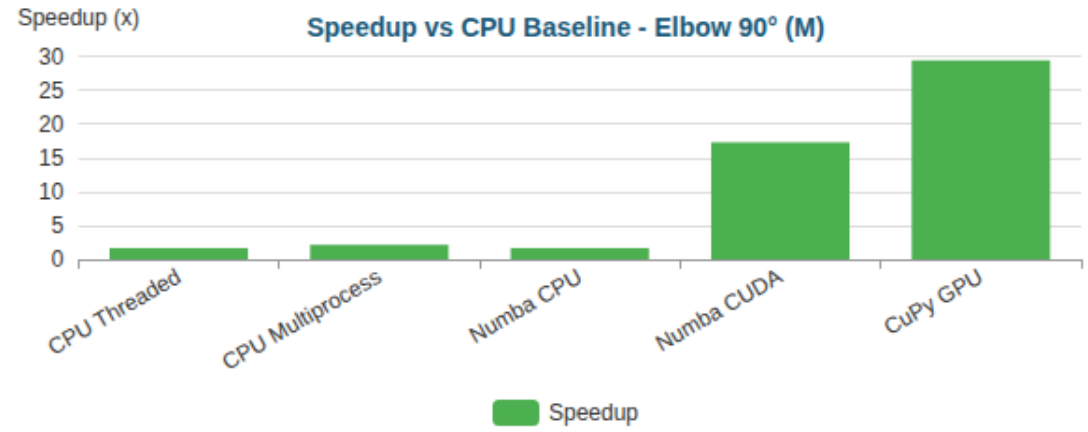
CPU Multiprocess	911ms ± 28ms	0.0x	3
Numba CPU	<0.01s ± 1ms	8.0x	3
Numba CUDA	53ms ± 7ms	0.8x	3
CuPy GPU	82ms ± 4ms	0.5x	3



Bar

Elbow 90° (M) (161,984 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	36.82s ± 0.16s	1.0x	3
CPU Threaded	21.98s ± 0.16s	1.7x	3
CPU Multiprocess	16.65s ± 0.48s	2.2x	3
Numba CPU	21.35s ± 0.18s	1.7x	3
Numba CUDA	2.12s ± 0.07s	17.3x	3
CuPy GPU	1.25s ± 0.02s	29.4x	3

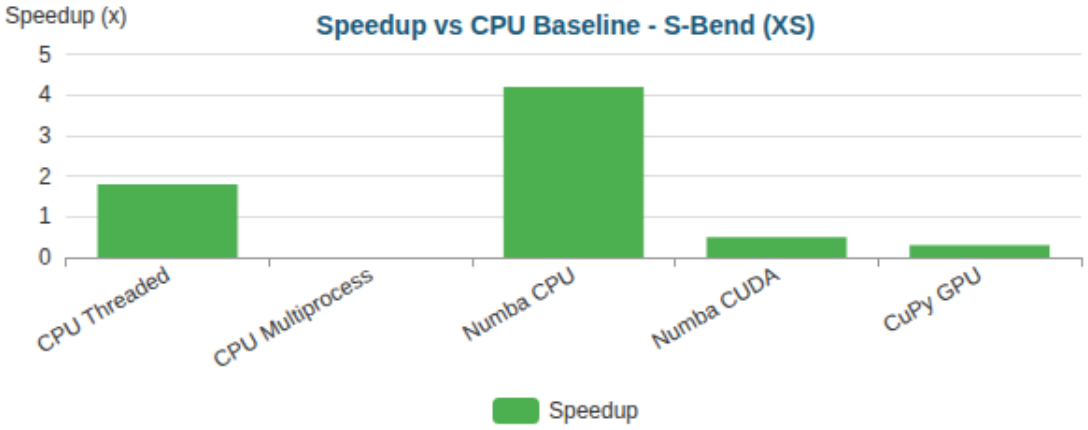


Bar

S-Bend (XS) (387 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	27ms ± 1ms	1.0x	3

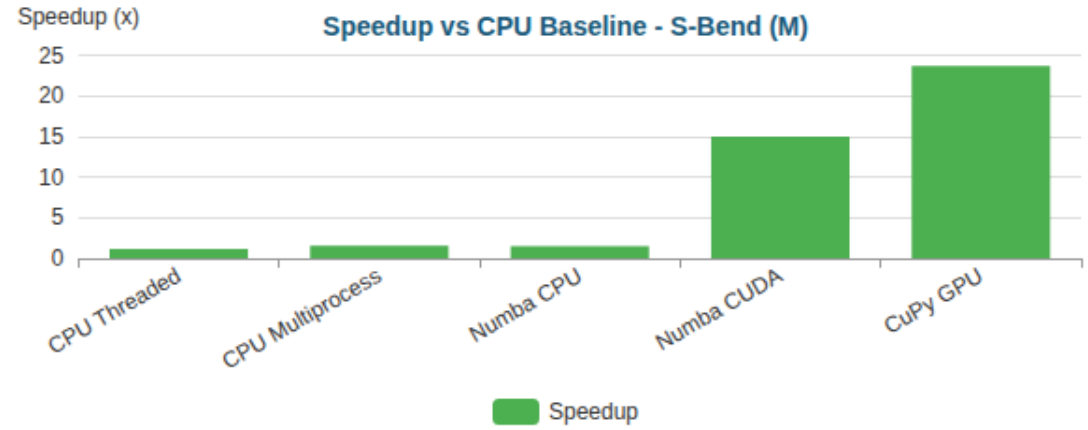
Implementation	Total Time	Speedup vs Baseline	N
CPU Threaded	15ms ± 3ms	1.8x	3
CPU Multiprocess	860ms ± 42ms	0.0x	3
Numba CPU	<0.01s ± 1ms	4.2x	3
Numba CUDA	54ms ± 5ms	0.5x	3
CuPy GPU	82ms ± 3ms	0.3x	3



Bar

S-Bend (M) (196,078 nodes)

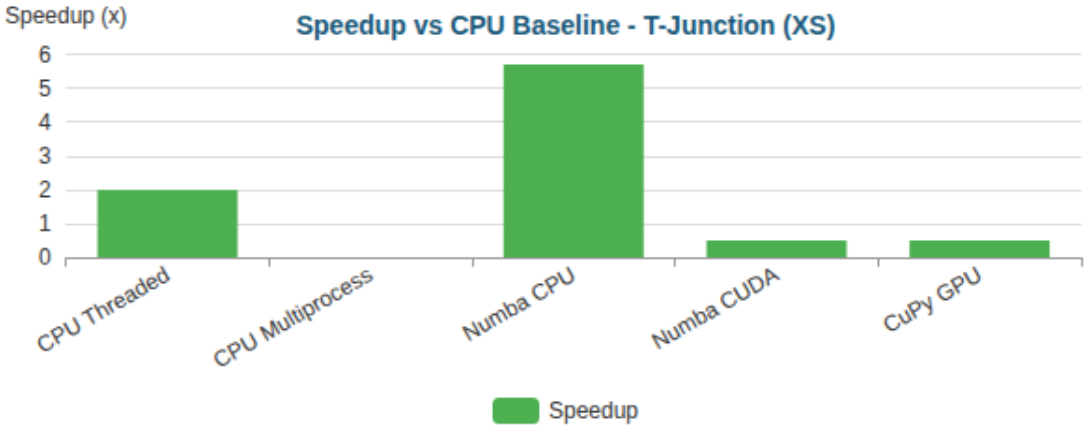
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	42.00s ± 0.14s	1.0x	3
CPU Threaded	34.94s ± 0.62s	1.2x	3
CPU Multiprocess	26.73s ± 0.47s	1.6x	3
Numba CPU	27.79s ± 0.19s	1.5x	3
Numba CUDA	2.80s ± 0.14s	15.0x	3
CuPy GPU	1.77s ± 0.12s	23.7x	3



Bar

T-Junction (XS) (393 nodes)

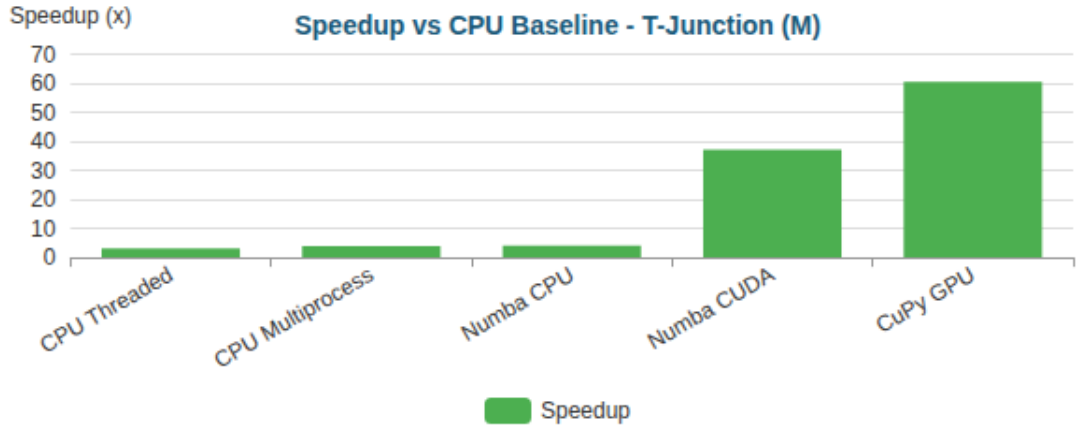
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	34ms ± 1ms	1.0x	3
CPU Threaded	17ms ± 1ms	2.0x	3
CPU Multiprocess	890ms ± 23ms	0.0x	3
Numba CPU	<0.01s ± 0ms	5.7x	3
Numba CUDA	70ms ± 4ms	0.5x	3
CuPy GPU	71ms ± 5ms	0.5x	3



Bar

T-Junction (M) (196,420 nodes)

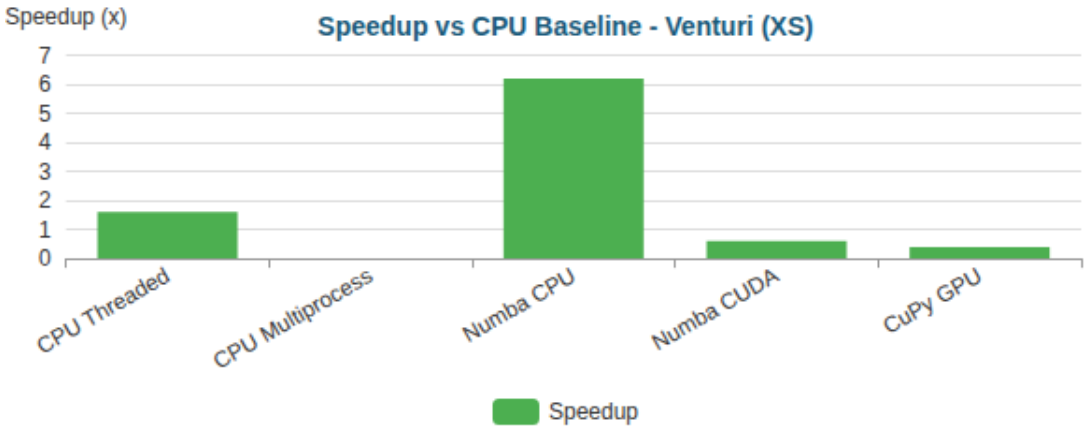
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	1m 39.2s ± 105.6s	1.0x	3
CPU Threaded	31.54s ± 0.61s	3.1x	3
CPU Multiprocess	25.29s ± 0.04s	3.9x	3
Numba CPU	24.14s ± 0.35s	4.1x	3
Numba CUDA	2.67s ± 0.10s	37.2x	3
CuPy GPU	1.64s ± 0.05s	60.5x	3



Bar

Venturi (XS) (341 nodes)

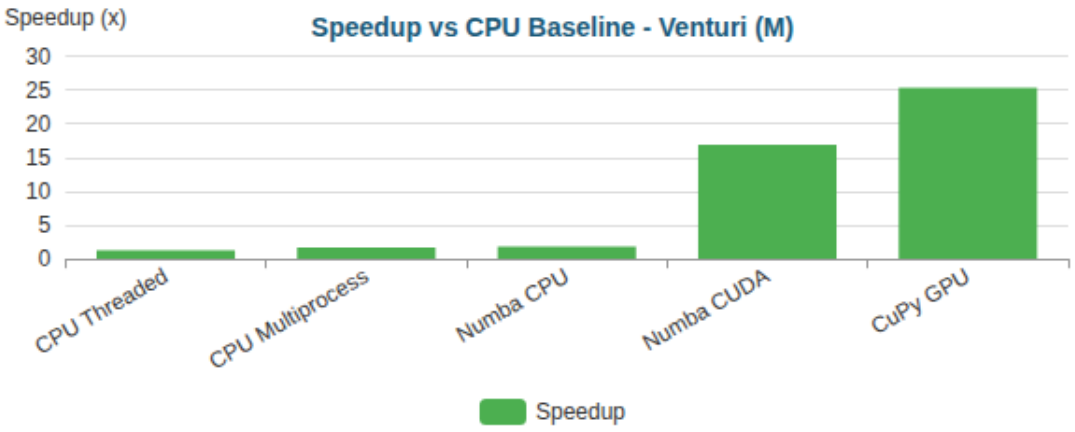
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	30ms ± 2ms	1.0x	3
CPU Threaded	18ms ± 4ms	1.6x	3
CPU Multiprocess	1.05s ± 0.02s	0.0x	3
Numba CPU	<0.01s ± 1ms	6.2x	3
Numba CUDA	52ms ± 5ms	0.6x	3
CuPy GPU	76ms ± 4ms	0.4x	3



Bar

Venturi (M) (194,325 nodes)

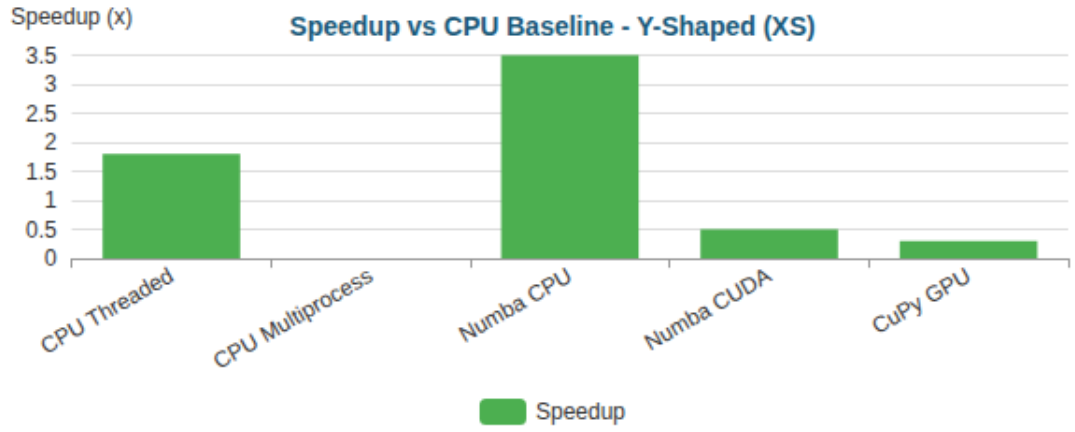
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	41.96s ± 0.16s	1.0x	3
CPU Threaded	31.34s ± 0.42s	1.3x	3
CPU Multiprocess	24.63s ± 1.60s	1.7x	3
Numba CPU	23.02s ± 0.36s	1.8x	3
Numba CUDA	2.48s ± 0.03s	16.9x	3
CuPy GPU	1.66s ± 0.03s	25.3x	3



Bar

Y-Shaped (XS) (201 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	19ms ± 2ms	1.0x	3
CPU Threaded	11ms ± 1ms	1.8x	3
CPU Multiprocess	1.00s ± 0.01s	0.0x	3
Numba CPU	<0.01s ± 0ms	3.5x	3
Numba CUDA	40ms ± 2ms	0.5x	3
CuPy GPU	59ms ± 4ms	0.3x	3

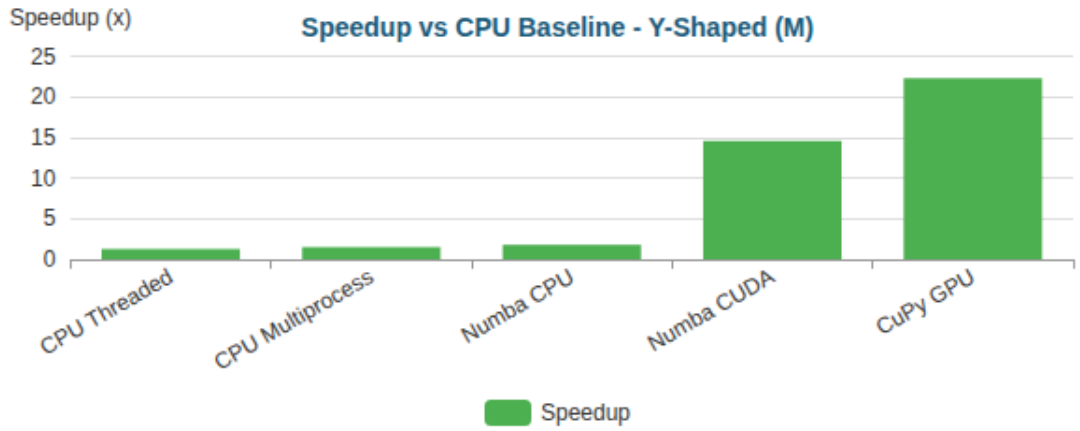


Bar

Y-Shaped (M) (195,853 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	34.38s ± 1.80s	1.0x	3
CPU Threaded	26.06s ± 0.34s	1.3x	3
CPU Multiprocess	22.53s ± 0.22s	1.5x	3
Numba CPU	19.15s ± 0.03s	1.8x	3
Numba CUDA	2.36s ± 0.05s	14.6x	3

Implementation	Total Time	Speedup vs Baseline	N
CuPy GPU	1.54s ± 0.01s	22.3x	3



Bar

Critical Analysis

Critical Analysis

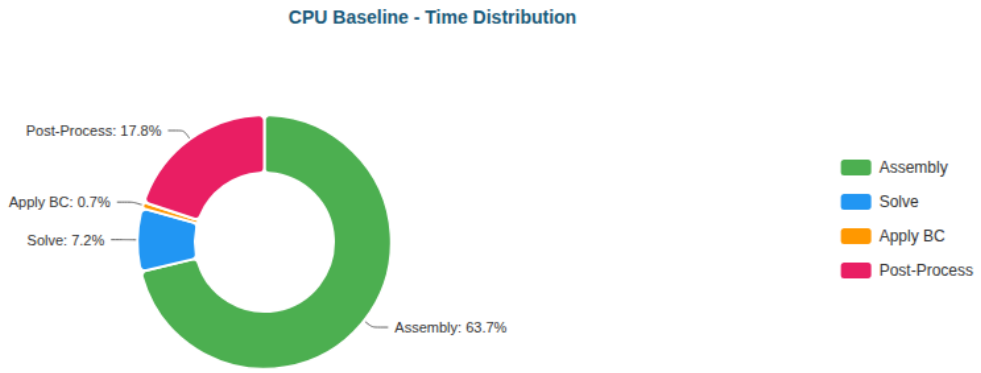
Bottleneck Evolution

As optimizations progress, the computational bottleneck shifts:

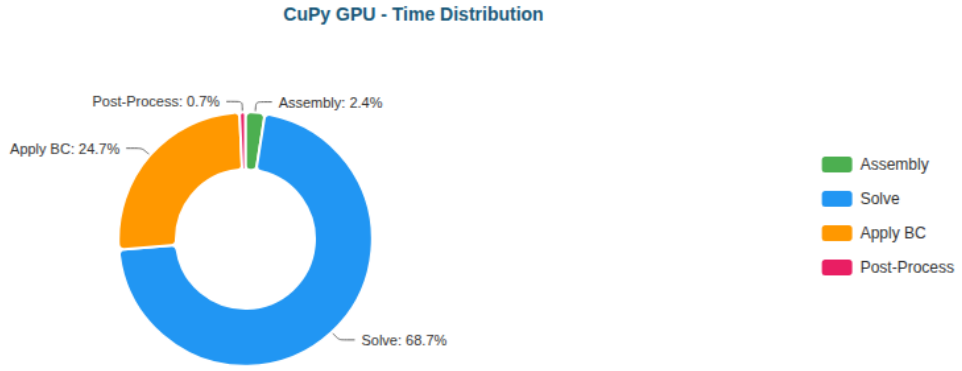
Backward-Facing Step (XS) - 287 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (64%)	Post-Proc (18%)
CPU Threaded	Assembly (46%)	Post-Proc (22%)
CPU Multiprocess	Assembly (50%)	Post-Proc (50%)
Numba CPU	Solve (46%)	BC (13%)
Numba CUDA	Solve (87%)	Assembly (6%)
CuPy GPU	Solve (69%)	BC (25%)

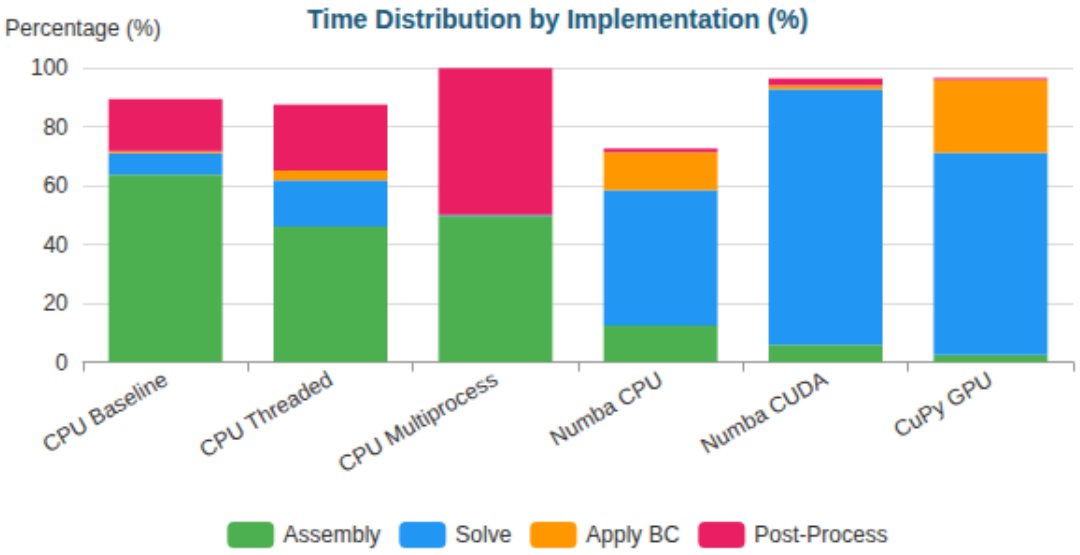
Time Distribution:



Pie



Pie



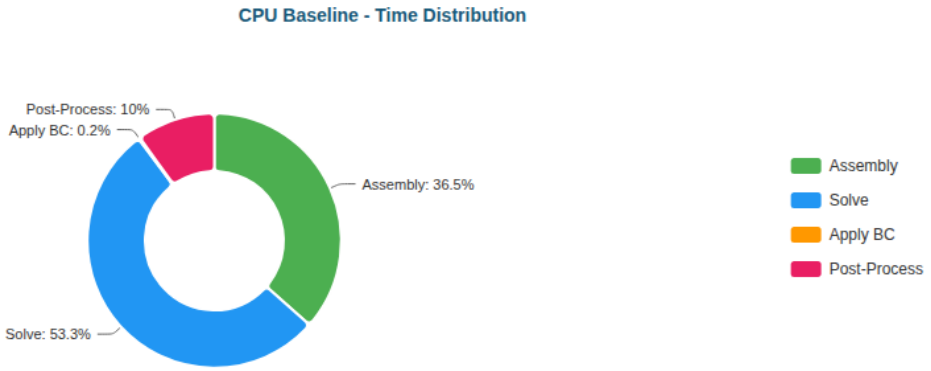
Stacked Bar

Backward-Facing Step (M) - 195,362 nodes

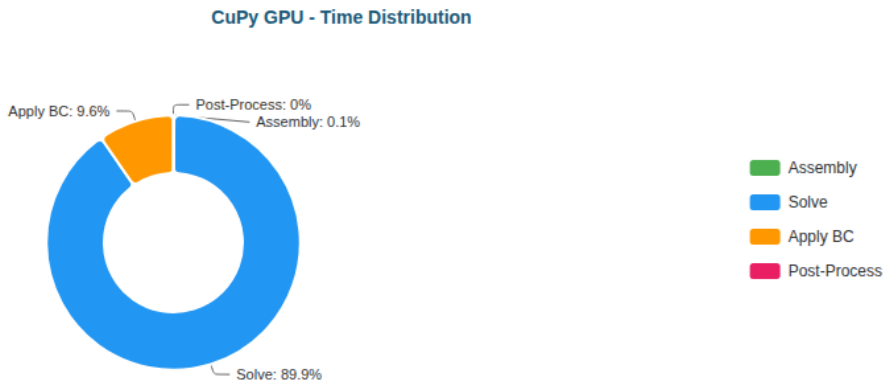
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (53%)	Assembly (36%)
CPU Threaded	Solve (72%)	Assembly (17%)
CPU Multiprocess	Solve (88%)	Assembly (5%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (51%)	Assembly (25%)
CuPy GPU	Solve (90%)	BC (10%)

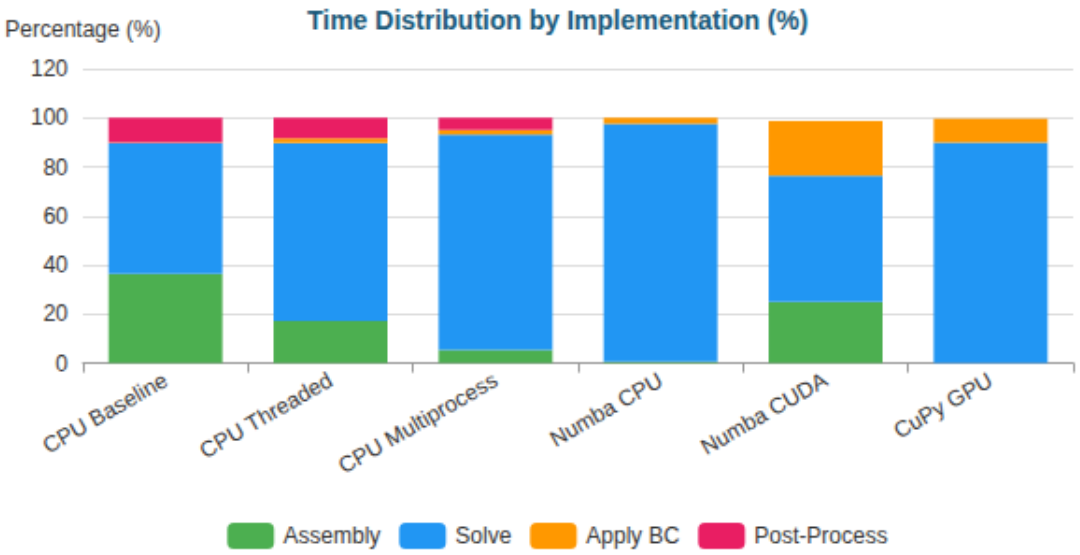
Time Distribution:



Pie



Pie

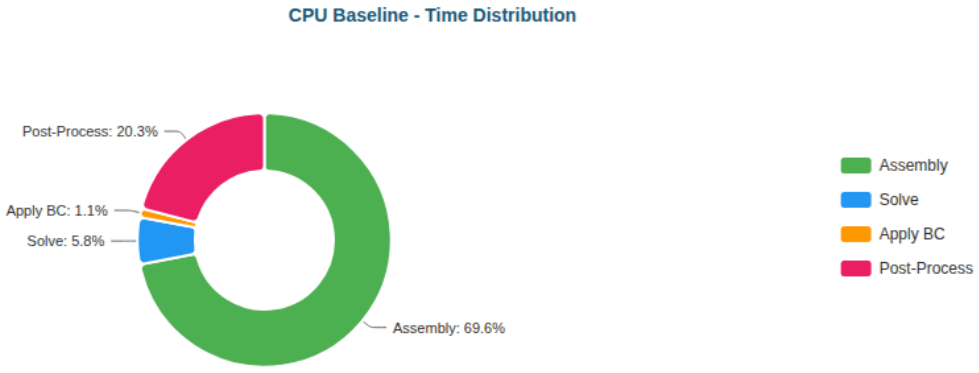


Stacked Bar

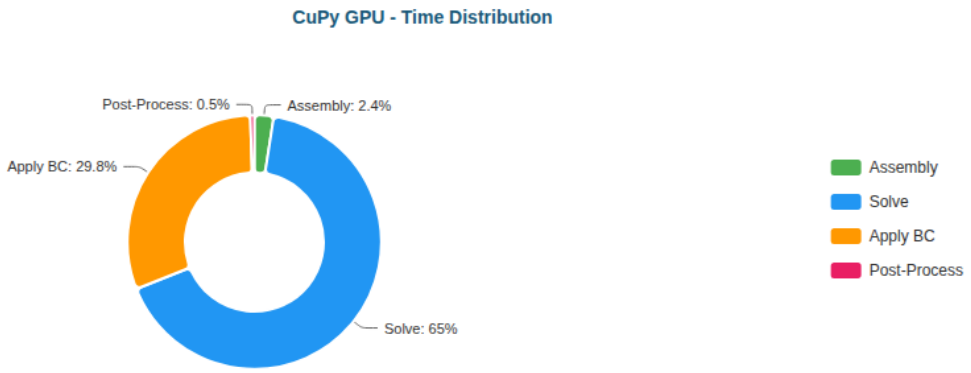
Elbow 90° (XS) - 411 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (70%)	Post-Proc (20%)
CPU Threaded	Assembly (50%)	Post-Proc (24%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (45%)	BC (17%)
Numba CUDA	Solve (87%)	Assembly (6%)
CuPy GPU	Solve (65%)	BC (30%)

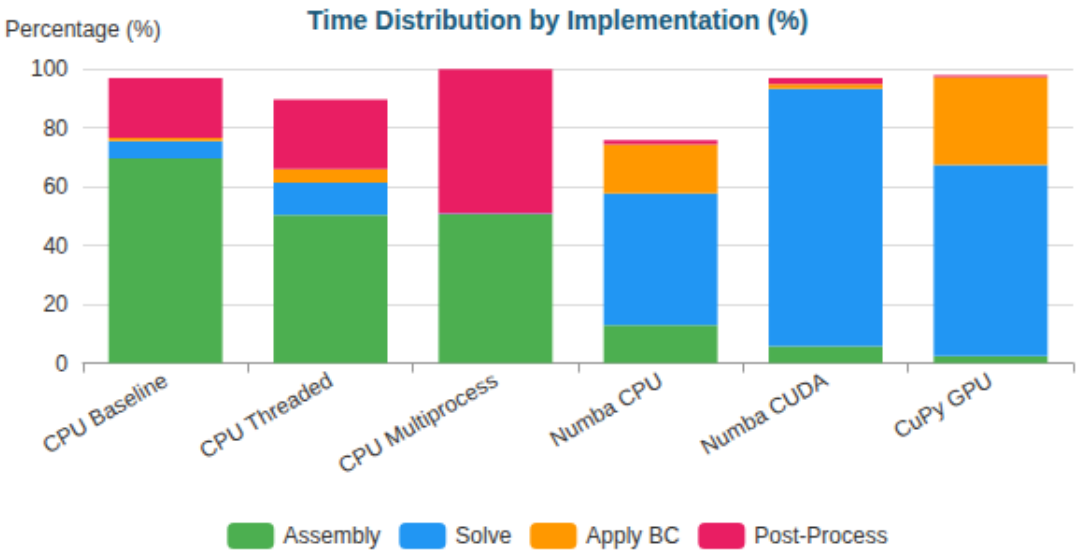
Time Distribution:



Pie



Pie

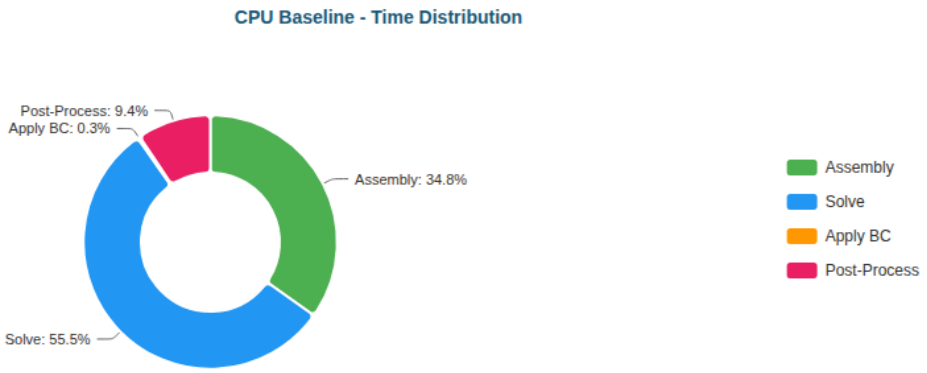


Stacked Bar

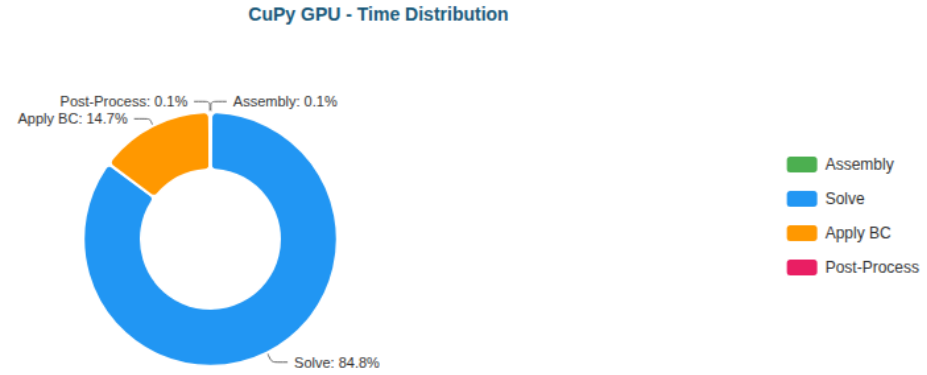
Elbow 90° (M) - 161,984 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (56%)	Assembly (35%)
CPU Threaded	Solve (68%)	Assembly (20%)
CPU Multiprocess	Solve (84%)	Assembly (7%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (50%)	Assembly (25%)
CuPy GPU	Solve (85%)	BC (15%)

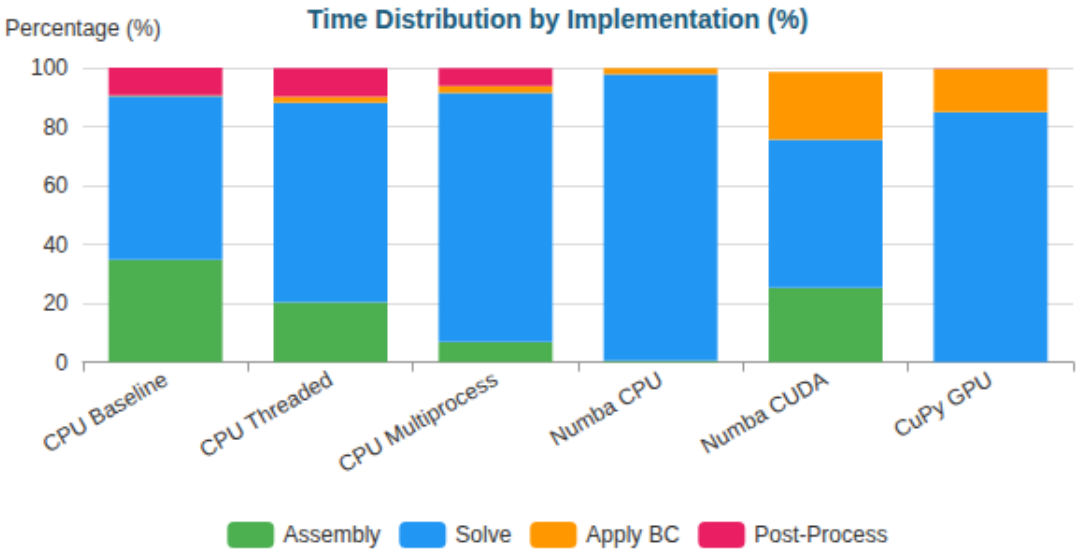
Time Distribution:



Pie



Pie



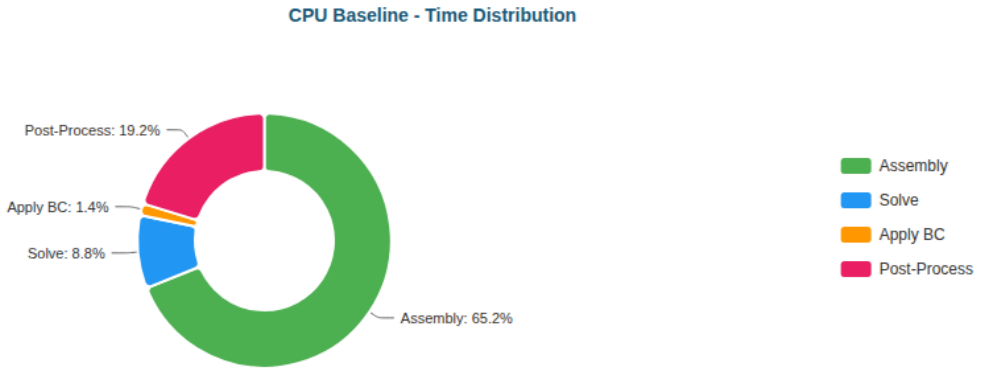
Stacked Bar

S-Bend (XS) - 387 nodes

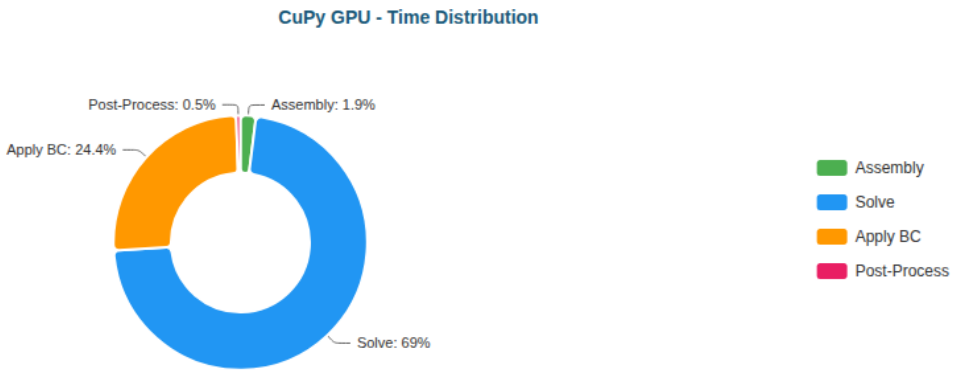
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (65%)	Post-Proc (19%)
CPU Threaded	Assembly (46%)	Post-Proc (26%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (44%)	BC (17%)
Numba CUDA	Solve (84%)	Assembly (8%)
CuPy GPU	Solve (69%)	BC (24%)

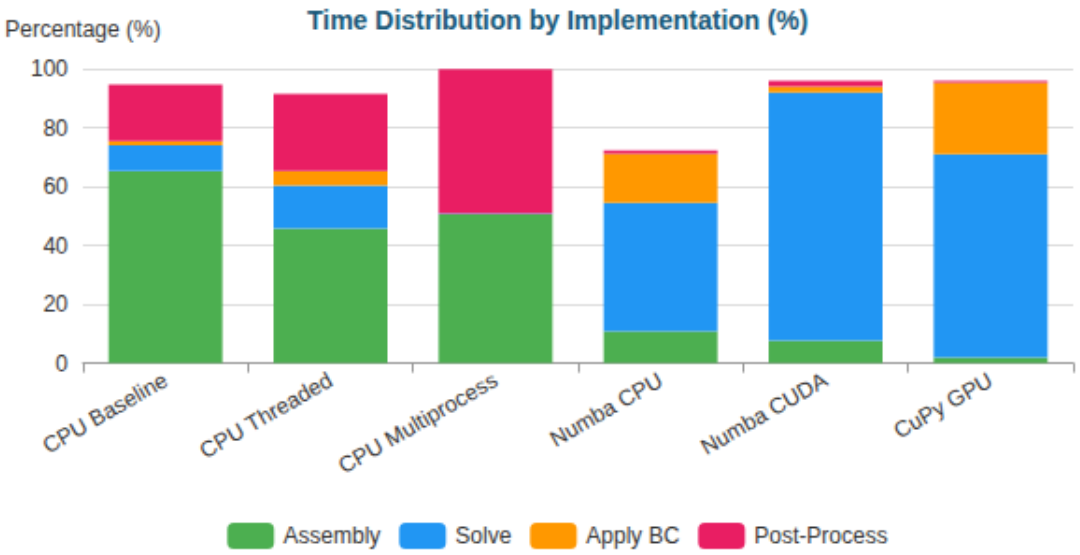
Time Distribution:



Pie



Pie

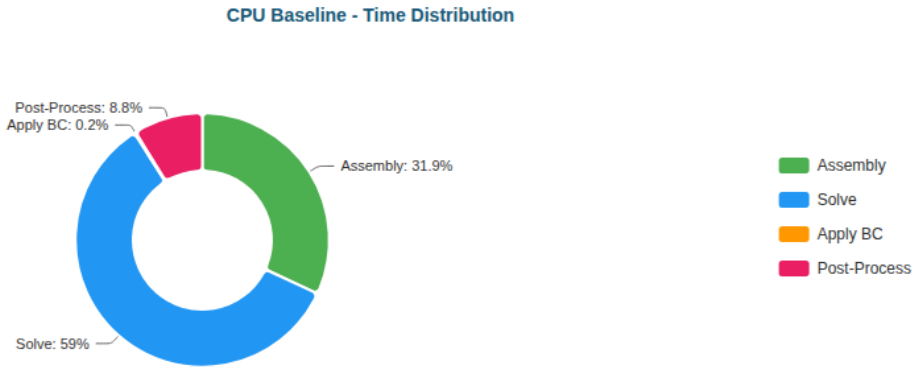


Stacked Bar

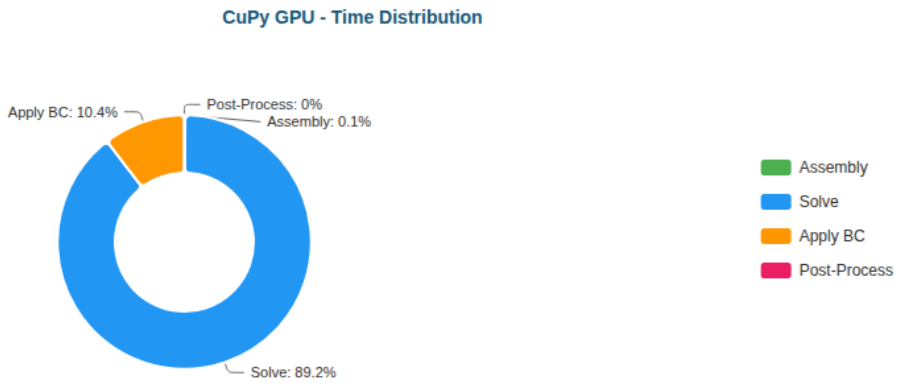
S-Bend (M) - 196,078 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (59%)	Assembly (32%)
CPU Threaded	Solve (74%)	Assembly (17%)
CPU Multiprocess	Solve (89%)	Assembly (5%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (55%)	Assembly (24%)
CuPy GPU	Solve (89%)	BC (10%)

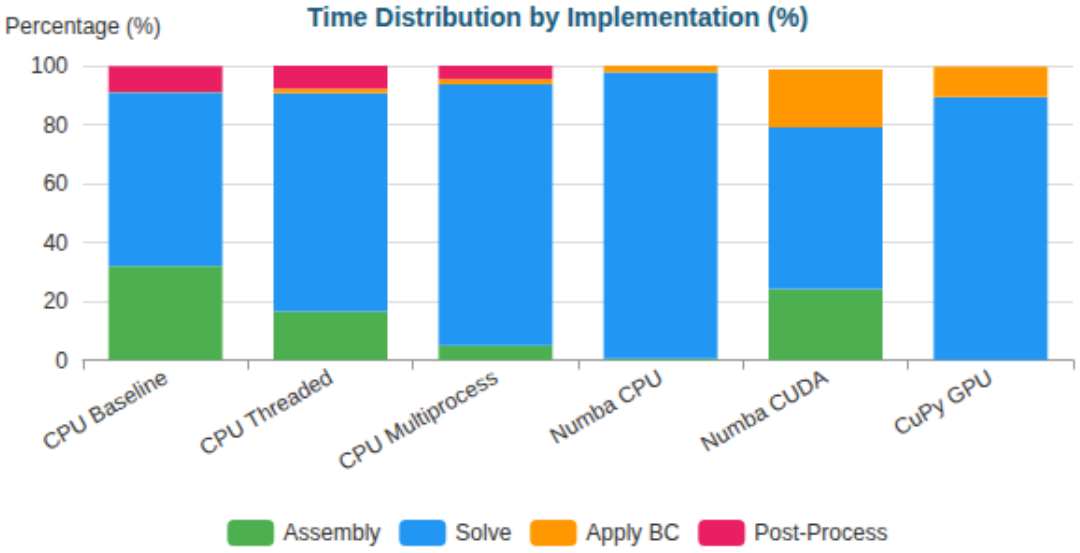
Time Distribution:



Pie



Pie

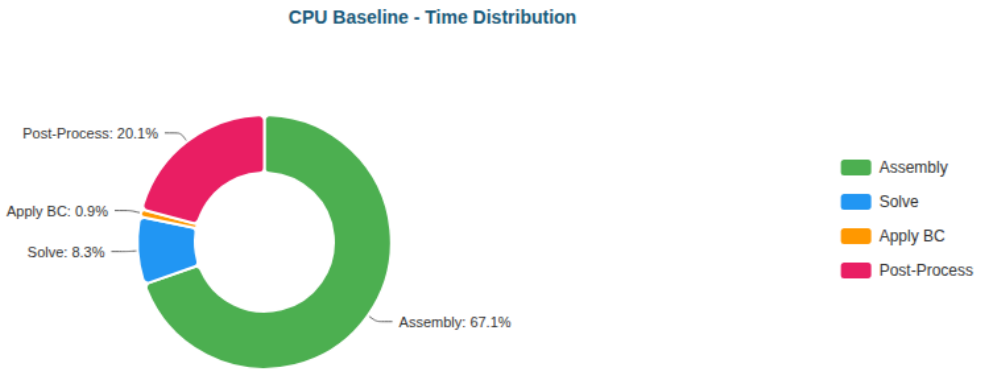


Stacked Bar

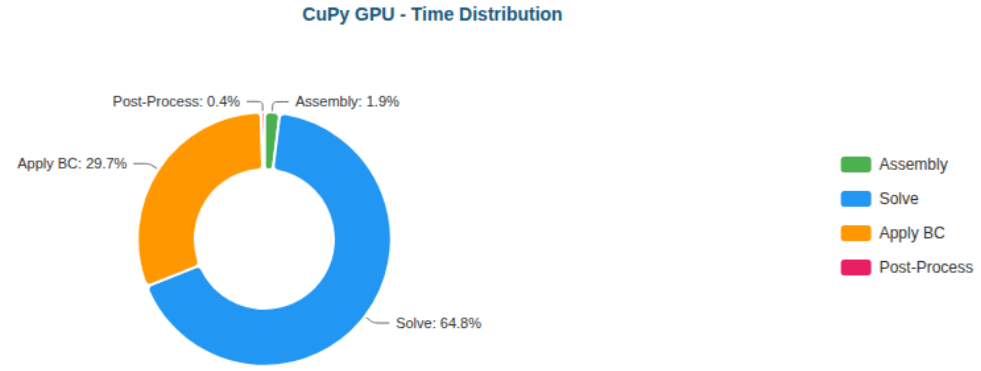
T-Junction (XS) - 393 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (67%)	Post-Proc (20%)
CPU Threaded	Assembly (52%)	Post-Proc (22%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (47%)	BC (16%)
Numba CUDA	Solve (89%)	Assembly (5%)
CuPy GPU	Solve (65%)	BC (30%)

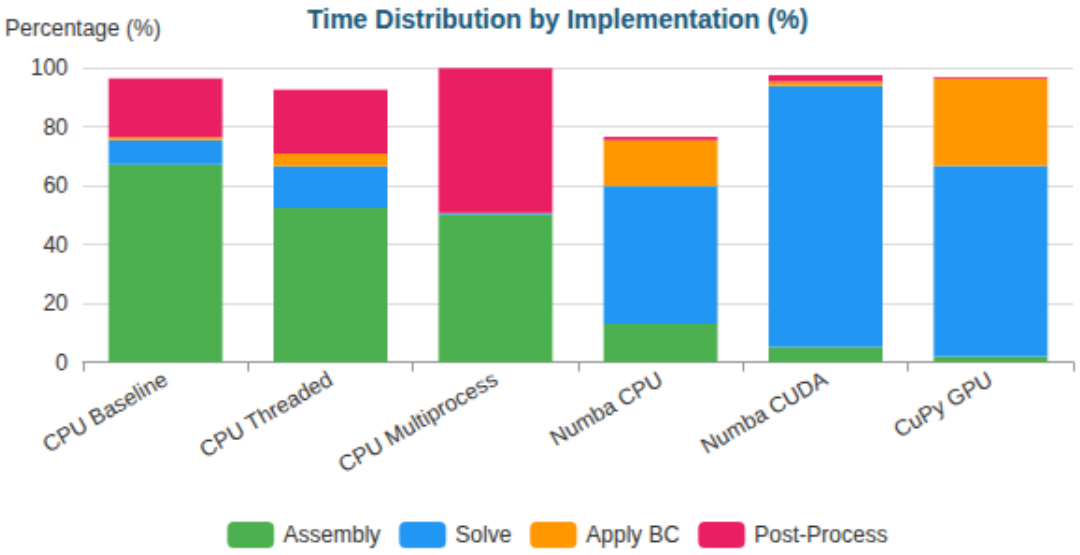
Time Distribution:



Pie



Pie



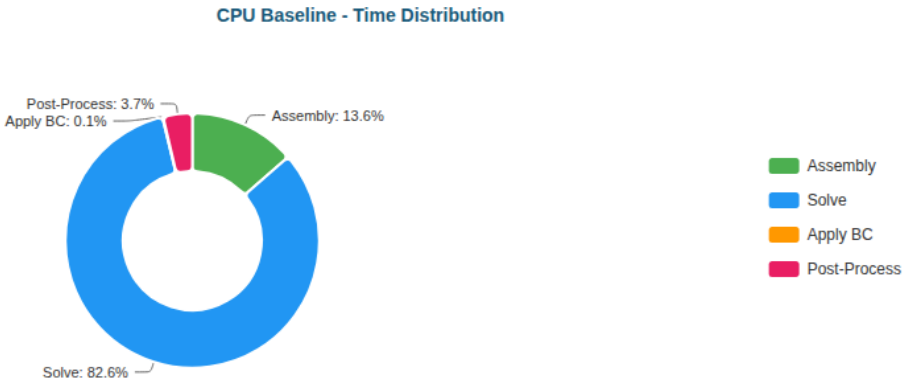
Stacked Bar

T-Junction (M) - 196,420 nodes

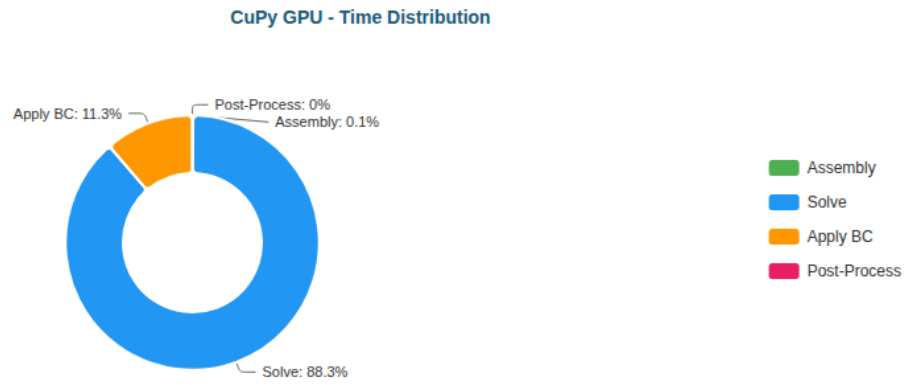
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (83%)	Assembly (14%)
CPU Threaded	Solve (75%)	Assembly (16%)
CPU Multiprocess	Solve (89%)	Assembly (5%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (55%)	Assembly (24%)
CuPy GPU	Solve (88%)	BC (11%)

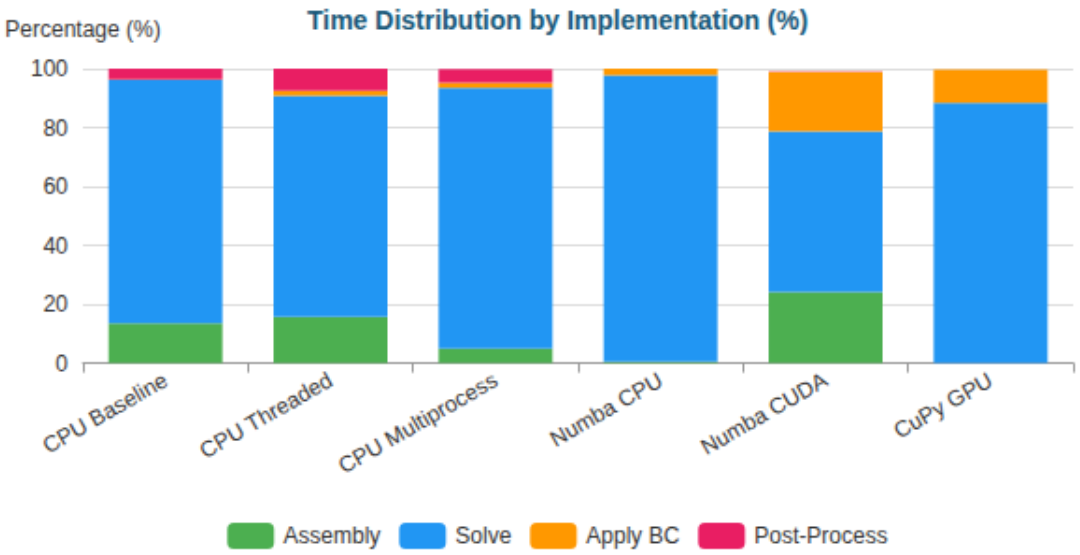
Time Distribution:



Pie



Pie

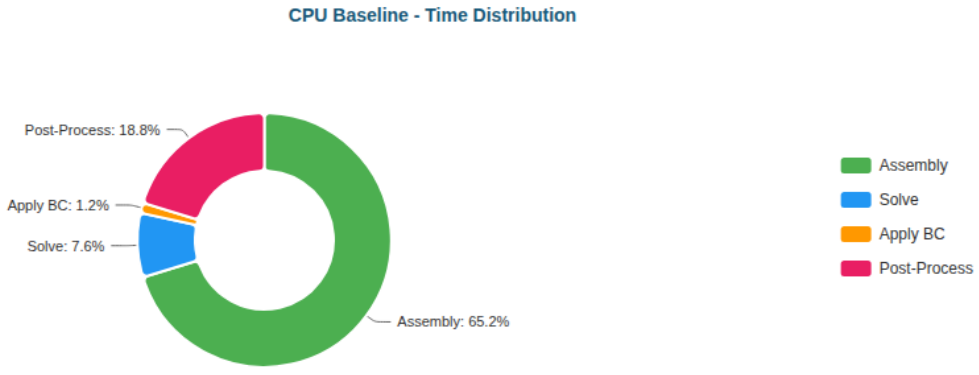


Stacked Bar

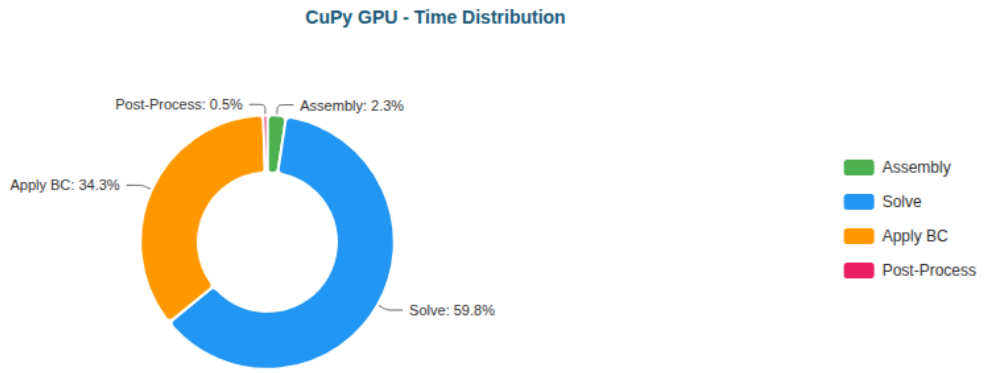
Venturi (XS) - 341 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (65%)	Post-Proc (19%)
CPU Threaded	Assembly (47%)	Post-Proc (22%)
CPU Multiprocess	Post-Proc (50%)	Assembly (50%)
Numba CPU	Solve (45%)	BC (16%)
Numba CUDA	Solve (85%)	Assembly (7%)
CuPy GPU	Solve (60%)	BC (34%)

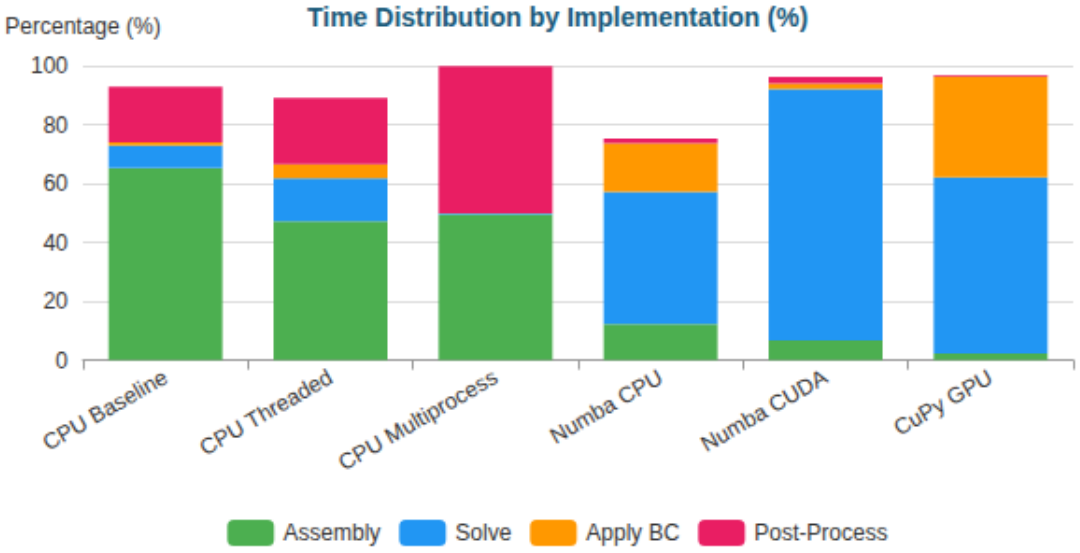
Time Distribution:



Pie



Pie

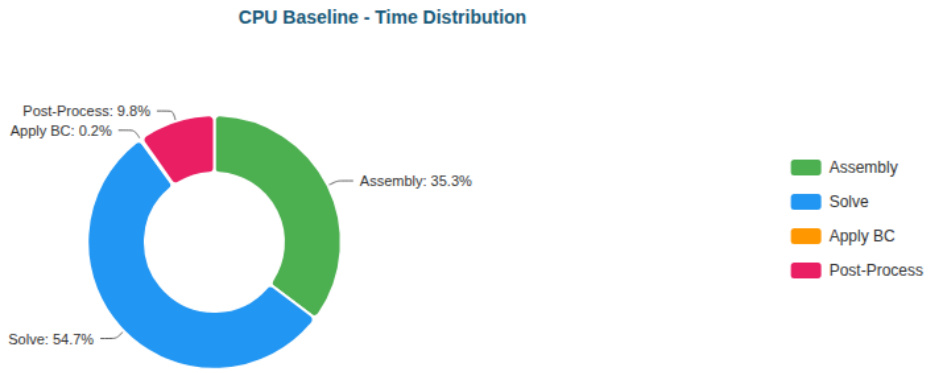


Stacked Bar

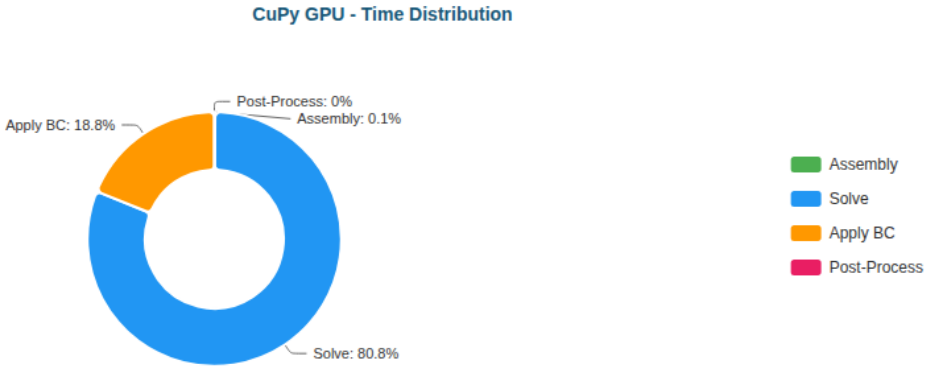
Venturi (M) - 194,325 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (55%)	Assembly (35%)
CPU Threaded	Solve (72%)	Assembly (17%)
CPU Multiprocess	Solve (87%)	Assembly (6%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (52%)	Assembly (25%)
CuPy GPU	Solve (81%)	BC (19%)

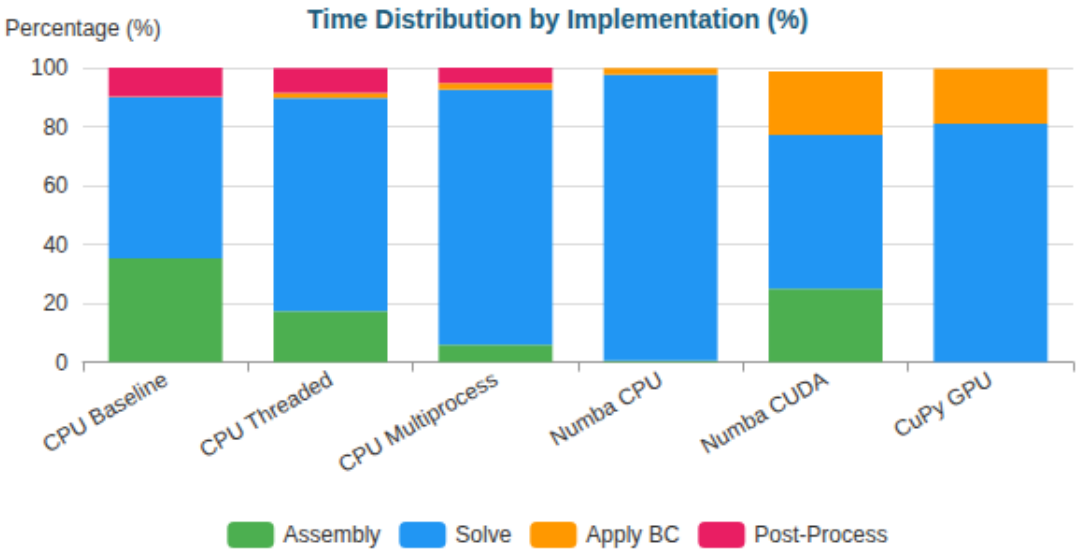
Time Distribution:



Pie



Pie



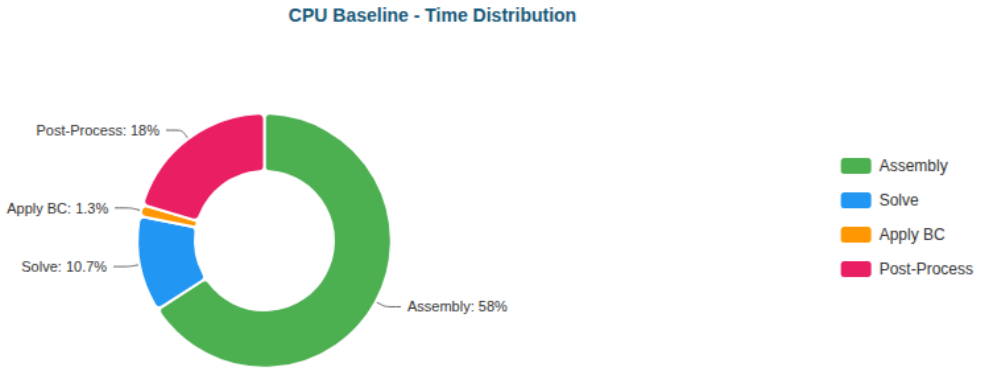
Stacked Bar

Y-Shaped (XS) - 201 nodes

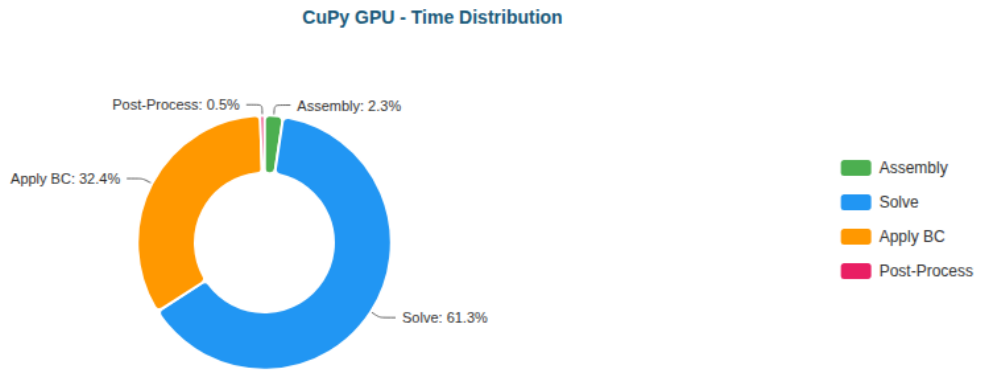
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (58%)	Post-Proc (18%)
CPU Threaded	Assembly (45%)	Solve (18%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (39%)	Assembly (13%)
Numba CUDA	Solve (80%)	Assembly (8%)
CuPy GPU	Solve (61%)	BC (32%)

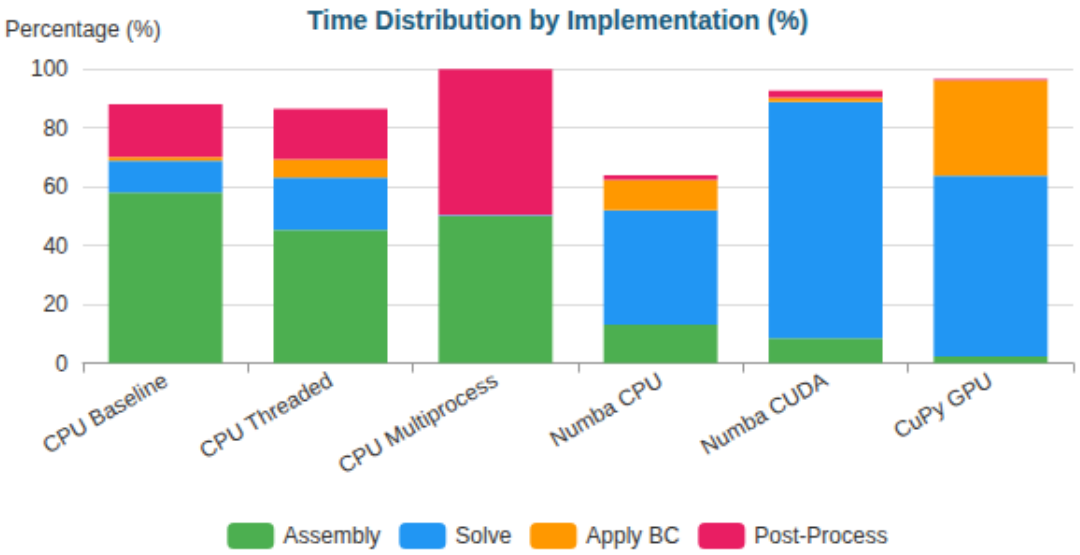
Time Distribution:



Pie



Pie

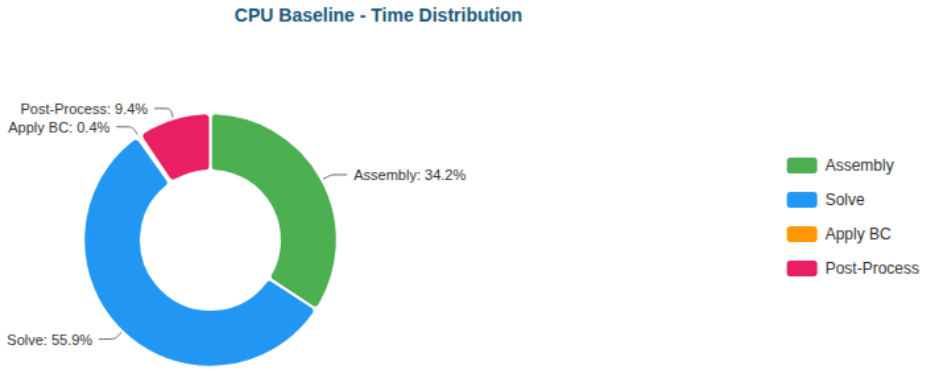


Stacked Bar

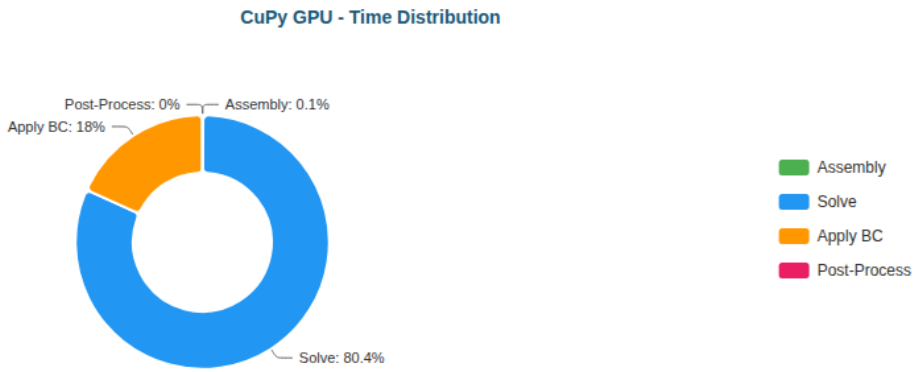
Y-Shaped (M) - 195,853 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (56%)	Assembly (34%)
CPU Threaded	Solve (74%)	Assembly (16%)
CPU Multiprocess	Solve (87%)	Assembly (6%)
Numba CPU	Solve (96%)	BC (3%)
Numba CUDA	Solve (51%)	BC (26%)
CuPy GPU	Solve (80%)	BC (18%)

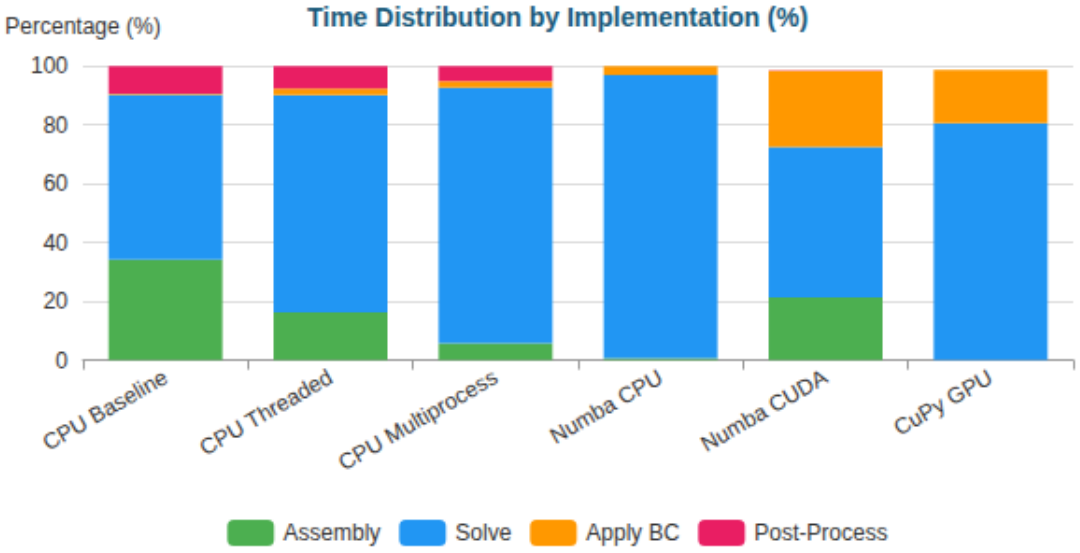
Time Distribution:



Pie



Pie



Stacked Bar

Why Each Optimization Helps

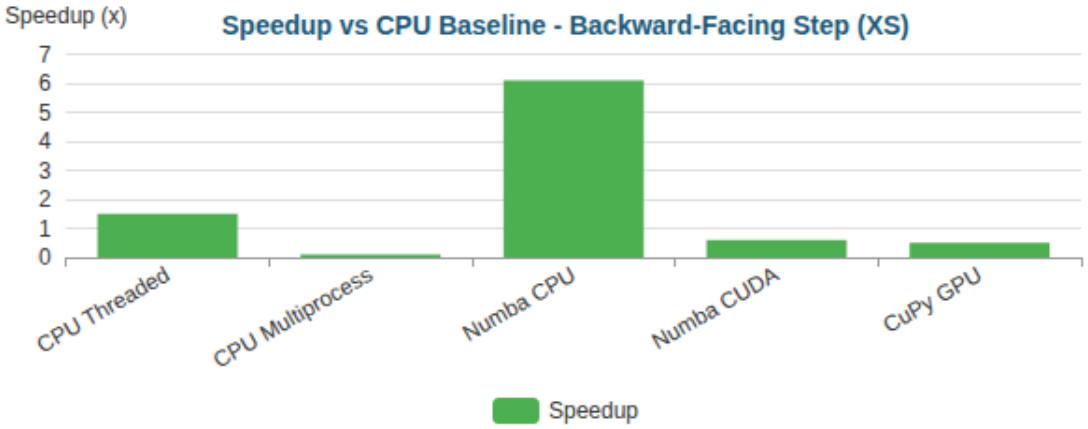
Transition	Reason
Baseline → Threaded	Limited by Python GIL; threads only help for I/O
Threaded → Multiprocess	Bypasses GIL via separate processes; IPC overhead limits gains
Multiprocess → Numba CPU	JIT compilation eliminates interpreter overhead; true parallel loops
Numba CPU → Numba CUDA	GPU parallelism: thousands of threads vs dozens of CPU cores
Numba CUDA → CuPy GPU	CUDA C kernels more optimized than Numba-generated PTX

4.6 RTX 4090 Performance

Key results from performance benchmarks comparing FEM solver implementations.

Backward-Facing Step (XS) (287 nodes)

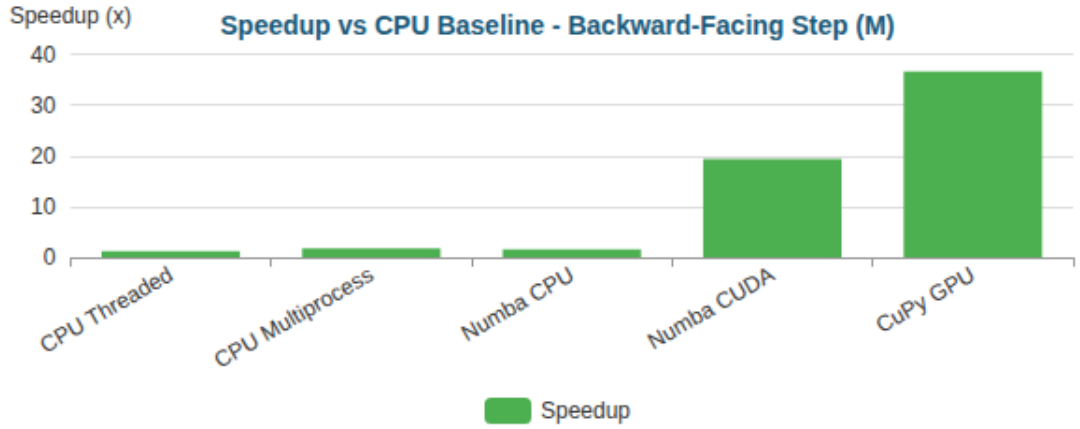
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	25ms ± 1ms	1.0x	3
CPU Threaded	17ms ± 2ms	1.5x	3
CPU Multiprocess	233ms ± 10ms	0.1x	3
Numba CPU	<0.01s ± 0ms	6.1x	3
Numba CUDA	45ms ± 10ms	0.6x	3
CuPy GPU	51ms ± 4ms	0.5x	3



Bar

Backward-Facing Step (M) (195,362 nodes)

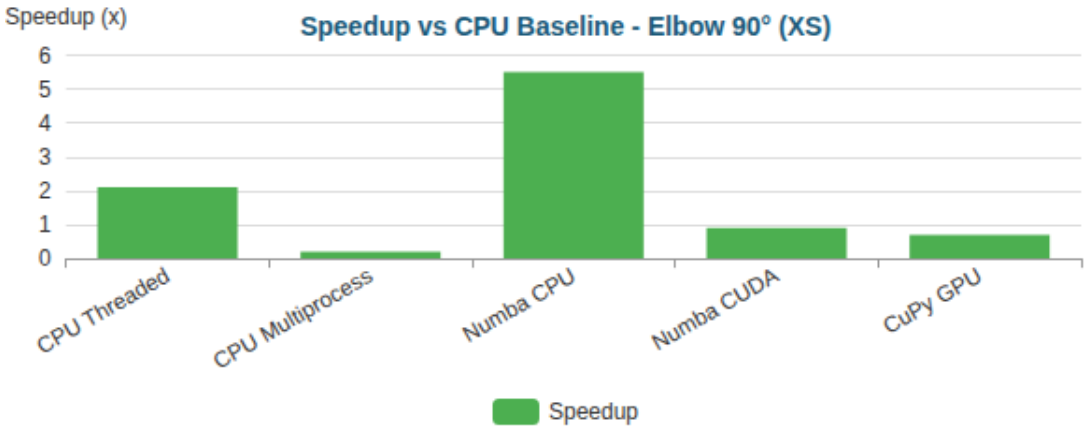
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	46.68s ± 1.66s	1.0x	3
CPU Threaded	37.41s ± 0.38s	1.2x	3
CPU Multiprocess	26.60s ± 0.18s	1.8x	3
Numba CPU	29.17s ± 2.04s	1.6x	3
Numba CUDA	2.40s ± 0.03s	19.4x	3
CuPy GPU	1.27s ± 0.02s	36.6x	3



Bar

Elbow 90° (XS) (411 nodes)

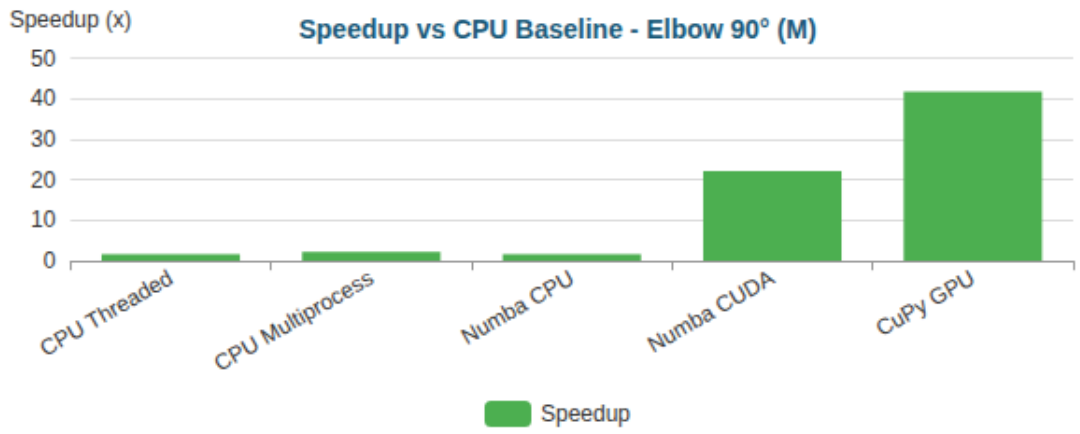
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	38ms ± 1ms	1.0x	3
CPU Threaded	18ms ± 1ms	2.1x	3
CPU Multiprocess	230ms ± 5ms	0.2x	3
Numba CPU	<0.01s ± 2ms	5.5x	3
Numba CUDA	44ms ± 1ms	0.9x	3
CuPy GPU	53ms ± 2ms	0.7x	3



Bar

Elbow 90° (M) (161,984 nodes)

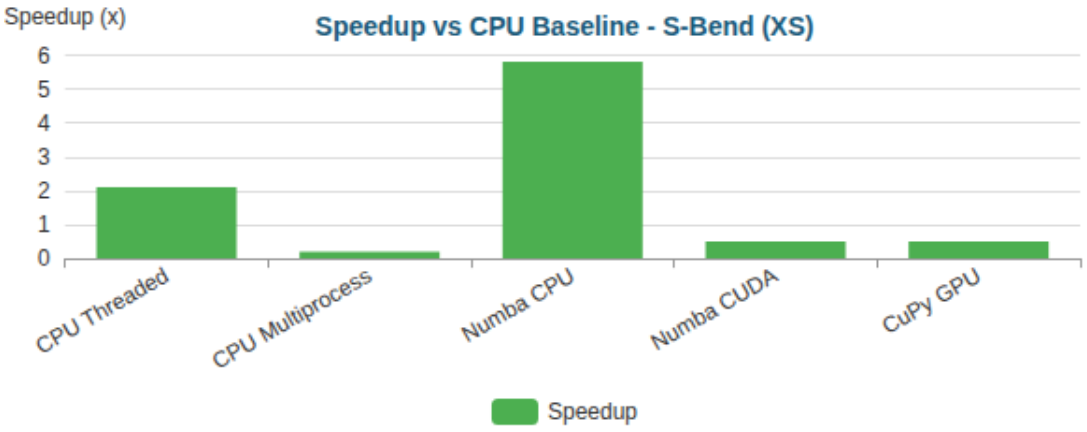
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	43.70s ± 0.62s	1.0x	3
CPU Threaded	27.97s ± 0.44s	1.6x	3
CPU Multiprocess	19.95s ± 1.64s	2.2x	3
Numba CPU	27.40s ± 1.48s	1.6x	3
Numba CUDA	1.97s ± 0.02s	22.1x	3
CuPy GPU	1.05s ± 0.05s	41.7x	3



Bar

S-Bend (XS) (387 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	34ms ± 1ms	1.0x	3
CPU Threaded	17ms ± 3ms	2.1x	3
CPU Multiprocess	219ms ± 6ms	0.2x	3
Numba CPU	<0.01s ± 2ms	5.8x	3
Numba CUDA	63ms ± 6ms	0.5x	3
CuPy GPU	62ms ± 8ms	0.5x	3

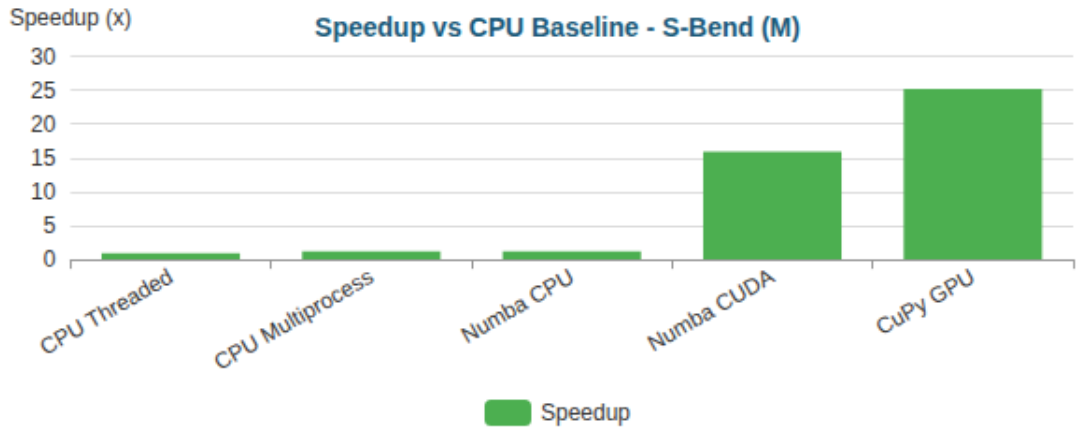


Bar

S-Bend (M) (196,078 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	38.82s ± 0.42s	1.0x	3
CPU Threaded	42.24s ± 1.49s	0.9x	3
CPU Multiprocess	31.68s ± 1.12s	1.2x	3
Numba CPU	33.08s ± 0.33s	1.2x	3
Numba CUDA	2.44s ± 0.05s	15.9x	3

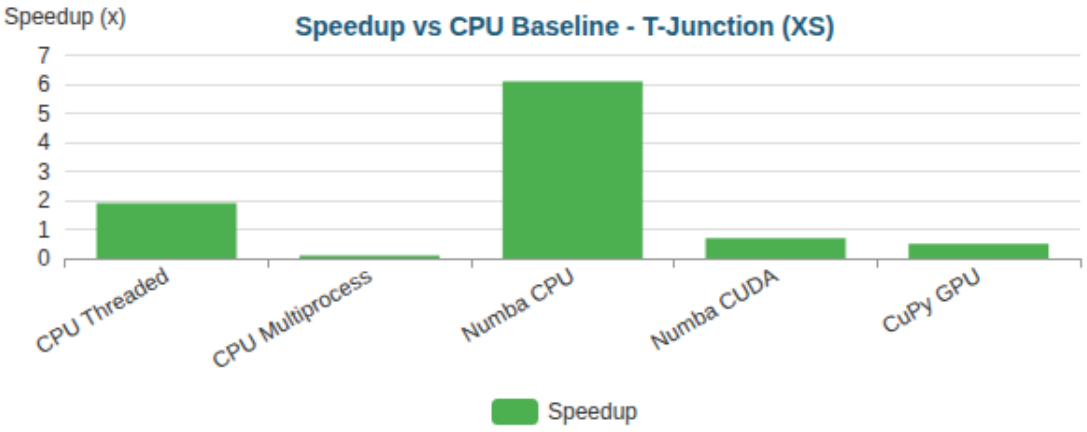
Implementation	Total Time	Speedup vs Baseline	N
CuPy GPU	1.54s ± 0.05s	25.2x	3



Bar

T-Junction (XS) (393 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	35ms ± 1ms	1.0x	3
CPU Threaded	18ms ± 2ms	1.9x	3
CPU Multiprocess	232ms ± 8ms	0.1x	3
Numba CPU	<0.01s ± 1ms	6.1x	3
Numba CUDA	48ms ± 4ms	0.7x	3
CuPy GPU	64ms ± 8ms	0.5x	3

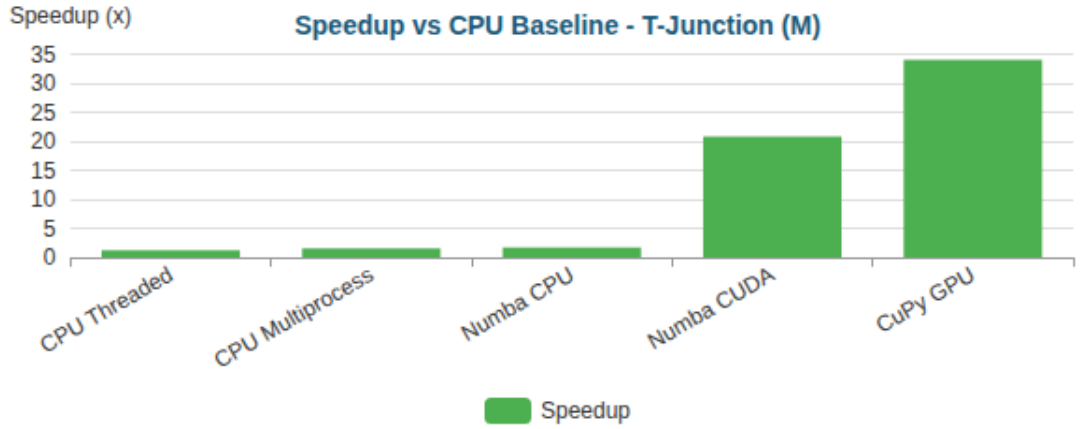


Bar

T-Junction (M) (196,420 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	48.55s ± 0.08s	1.0x	3
CPU Threaded	41.12s ± 0.61s	1.2x	3
CPU Multiprocess	29.56s ± 1.14s	1.6x	3

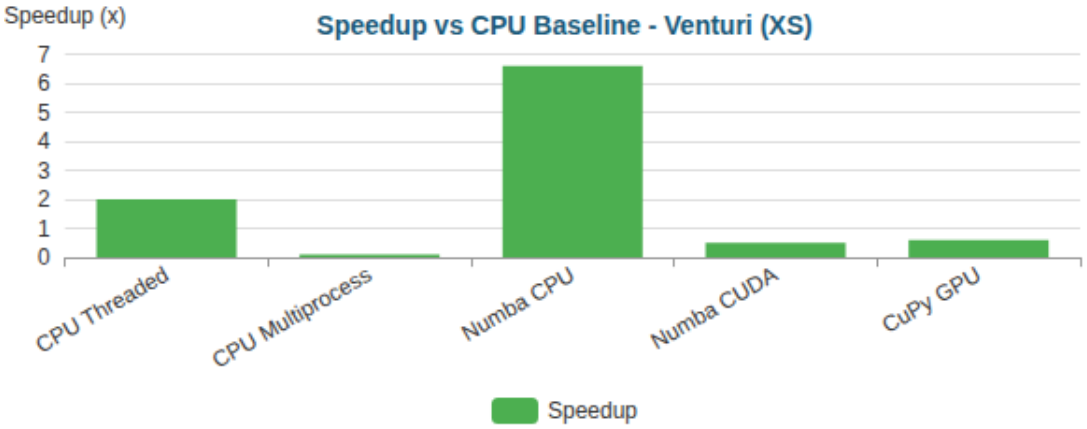
Implementation	Total Time	Speedup vs Baseline	N
Numba CPU	28.00s ± 1.65s	1.7x	3
Numba CUDA	2.33s ± 0.03s	20.8x	3
CuPy GPU	1.43s ± 0.01s	34.0x	3



Bar

Venturi (XS) (341 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	30ms ± 2ms	1.0x	3
CPU Threaded	15ms ± 3ms	2.0x	3
CPU Multiprocess	203ms ± 11ms	0.1x	3
Numba CPU	<0.01s ± 0ms	6.6x	3
Numba CUDA	60ms ± 9ms	0.5x	3
CuPy GPU	53ms ± 2ms	0.6x	3

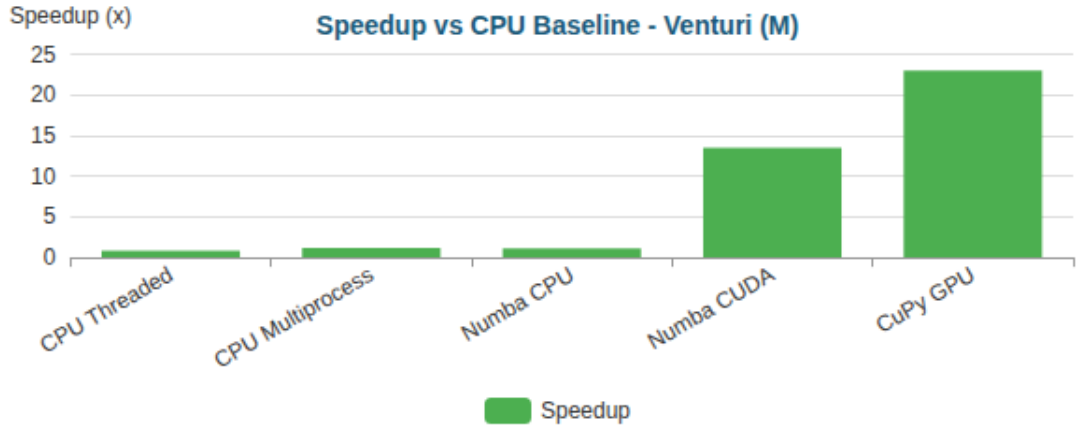


Bar

Venturi (M) (194,325 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	32.22s ± 0.16s	1.0x	3

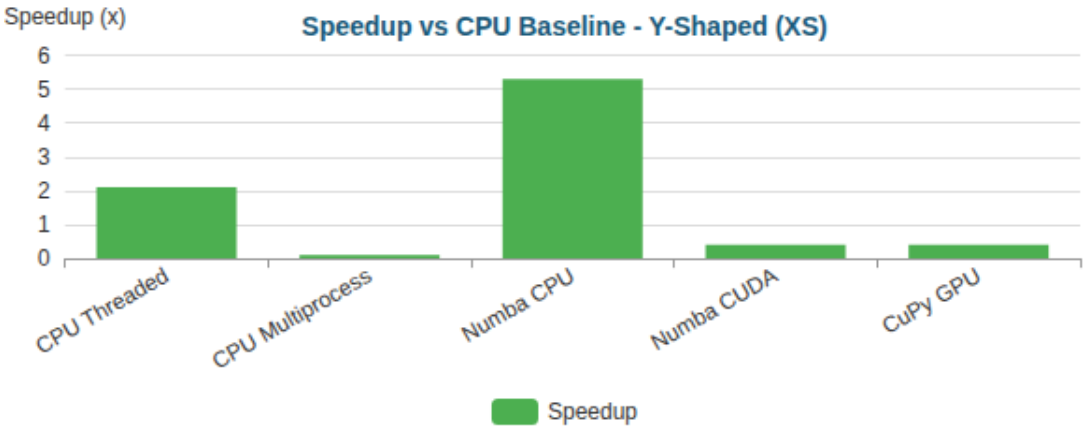
Implementation	Total Time	Speedup vs Baseline	N
CPU Threaded	38.35s ± 0.25s	0.8x	3
CPU Multiprocess	27.15s ± 1.38s	1.2x	3
Numba CPU	29.96s ± 0.27s	1.1x	3
Numba CUDA	2.38s ± 0.08s	13.5x	3
CuPy GPU	1.40s ± 0.05s	23.0x	3



Bar

Y-Shaped (XS) (201 nodes)

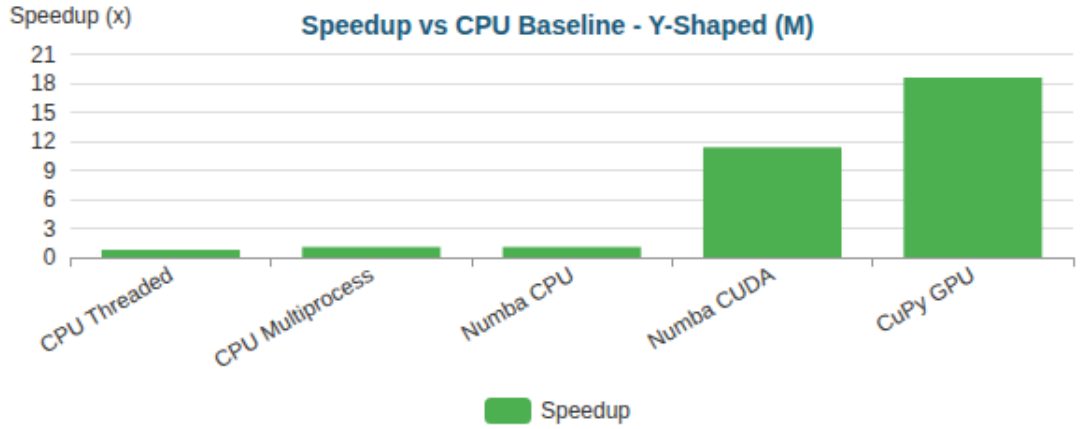
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	20ms ± 3ms	1.0x	6
CPU Threaded	<0.01s ± 1ms	2.1x	3
CPU Multiprocess	169ms ± 5ms	0.1x	3
Numba CPU	<0.01s ± 0ms	5.3x	3
Numba CUDA	53ms ± 13ms	0.4x	3
CuPy GPU	52ms ± 6ms	0.4x	3



Bar

Y-Shaped (M) (195,853 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	25.05s ± 0.49s	1.0x	3
CPU Threaded	31.45s ± 0.54s	0.8x	3
CPU Multiprocess	23.40s ± 0.34s	1.1x	3
Numba CPU	22.86s ± 0.32s	1.1x	3
Numba CUDA	2.20s ± 0.08s	11.4x	3
CuPy GPU	1.35s ± 0.01s	18.6x	3



Bar

Critical Analysis

Critical Analysis

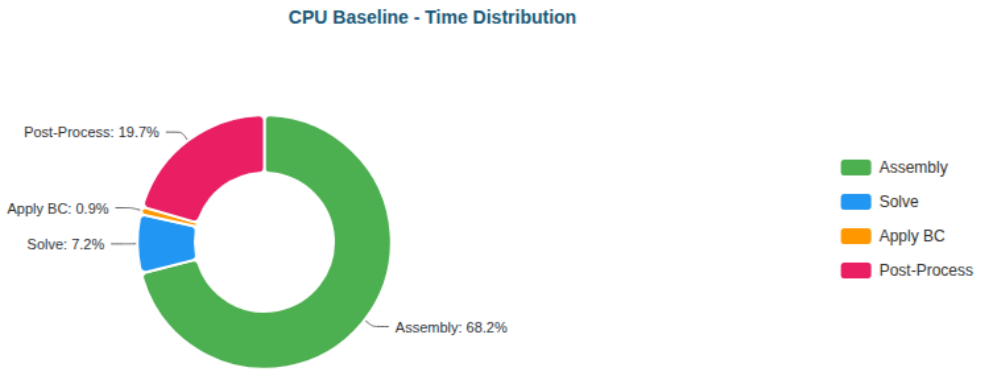
Bottleneck Evolution

As optimizations progress, the computational bottleneck shifts:

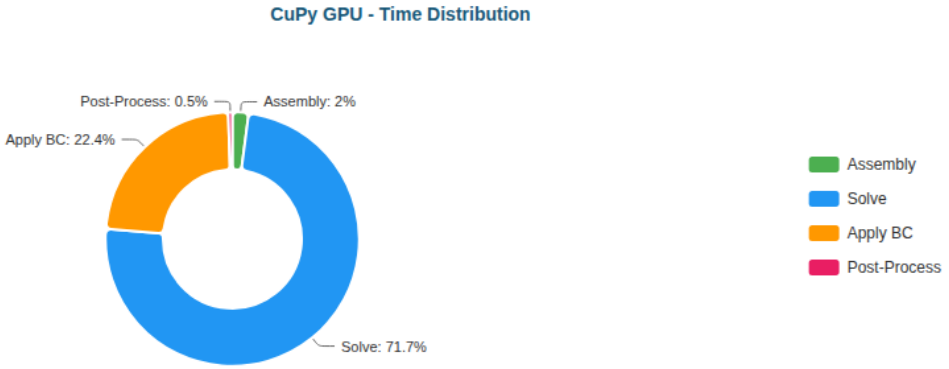
Backward-Facing Step (XS) - 287 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (68%)	Post-Proc (20%)
CPU Threaded	Assembly (52%)	Post-Proc (21%)
CPU Multiprocess	Assembly (50%)	Post-Proc (48%)
Numba CPU	Solve (45%)	Assembly (14%)
Numba CUDA	Solve (87%)	Assembly (6%)
CuPy GPU	Solve (72%)	BC (22%)

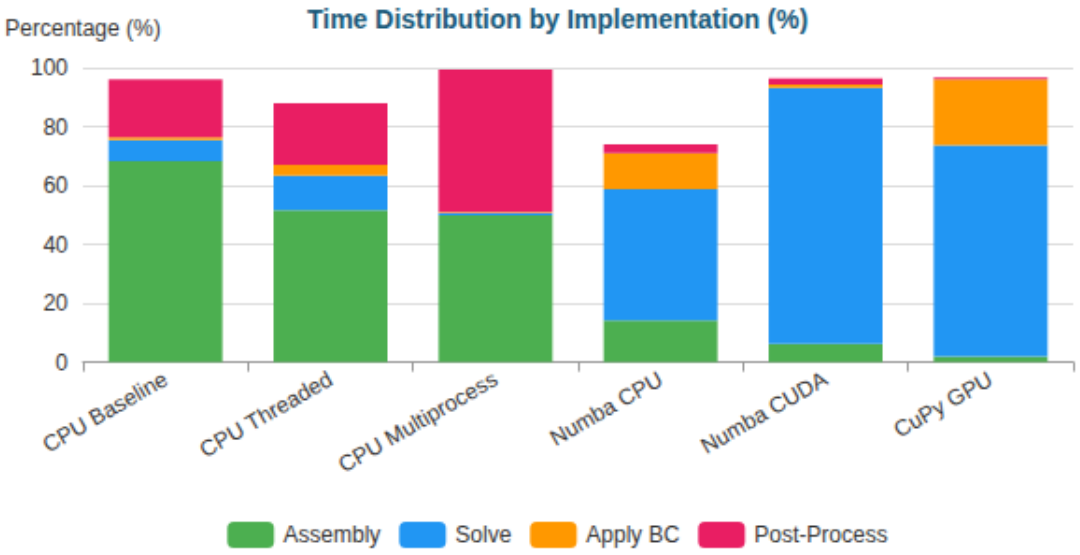
Time Distribution:



Pie



Pie



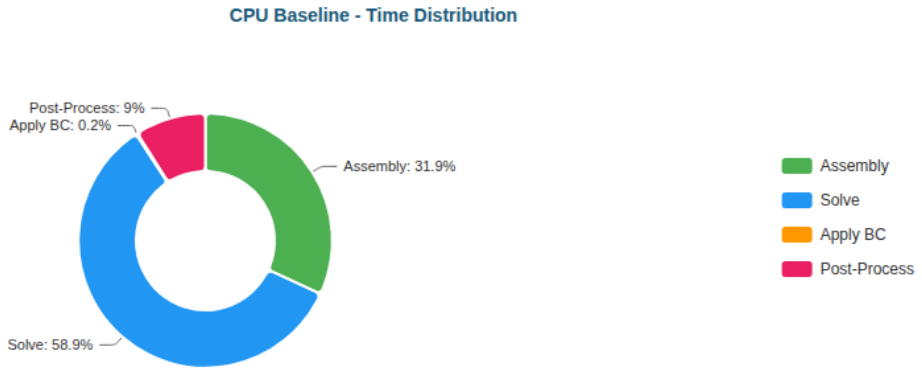
Stacked Bar

Backward-Facing Step (M) - 195,362 nodes

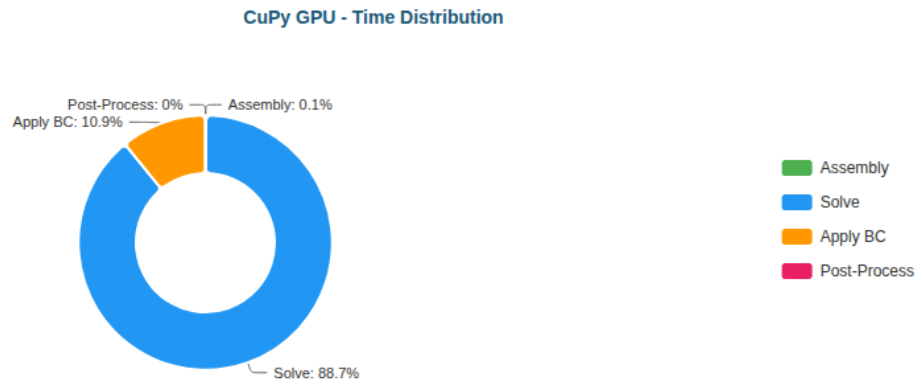
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (59%)	Assembly (32%)
CPU Threaded	Solve (67%)	Assembly (19%)
CPU Multiprocess	Solve (92%)	Assembly (3%)
Numba CPU	Solve (98%)	BC (1%)
Numba CUDA	Solve (49%)	Assembly (30%)
CuPy GPU	Solve (89%)	BC (11%)

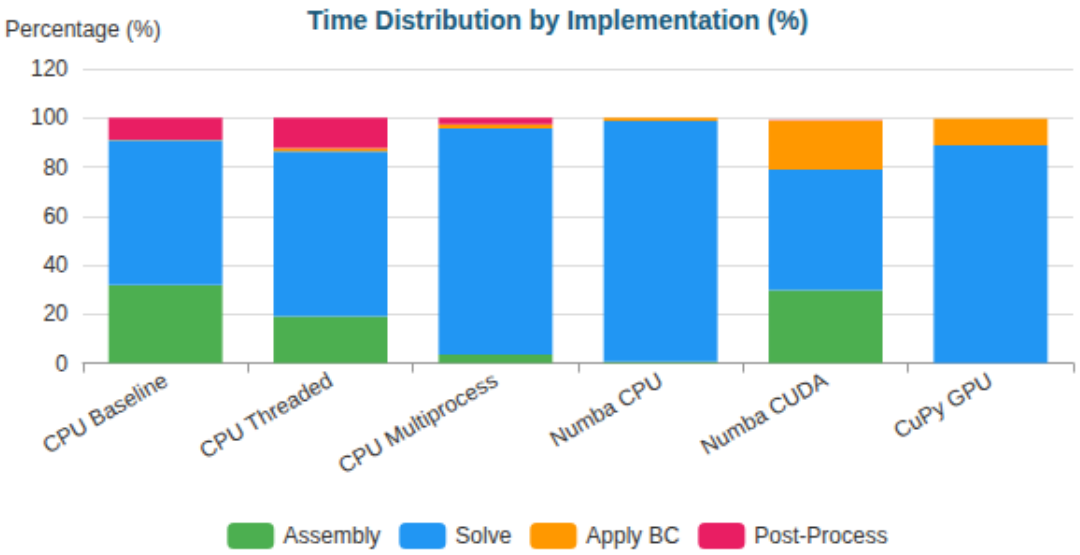
Time Distribution:



Pie



Pie

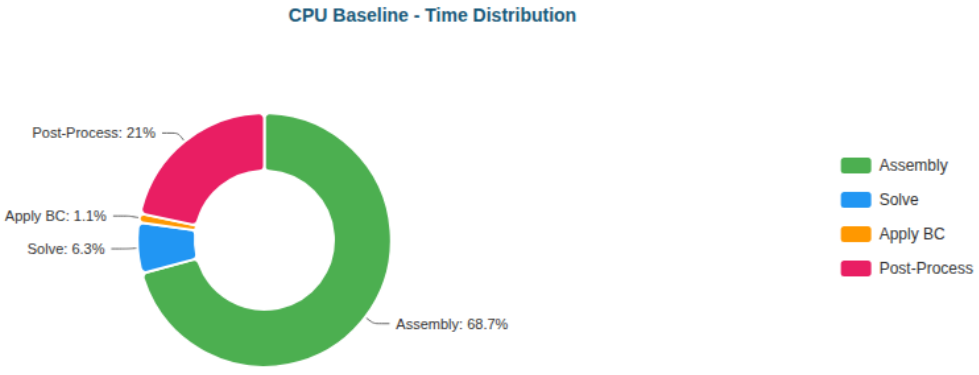


Stacked Bar

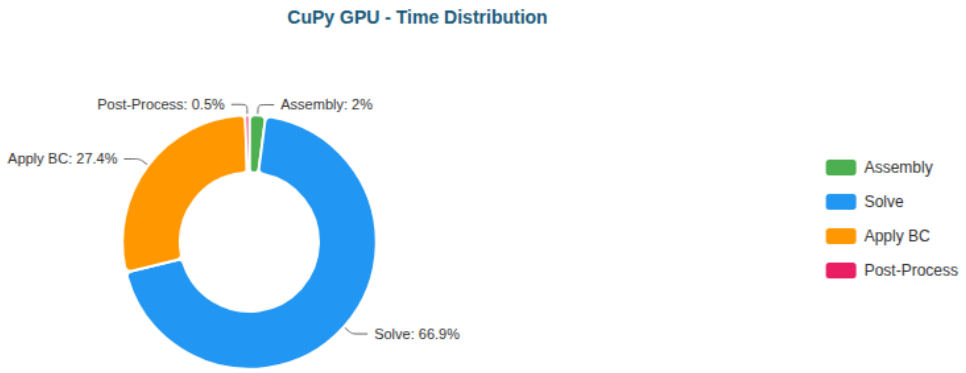
Elbow 90° (XS) - 411 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (69%)	Post-Proc (21%)
CPU Threaded	Assembly (53%)	Post-Proc (24%)
CPU Multiprocess	Assembly (51%)	Post-Proc (47%)
Numba CPU	Solve (39%)	BC (16%)
Numba CUDA	Solve (86%)	Assembly (7%)
CuPy GPU	Solve (67%)	BC (27%)

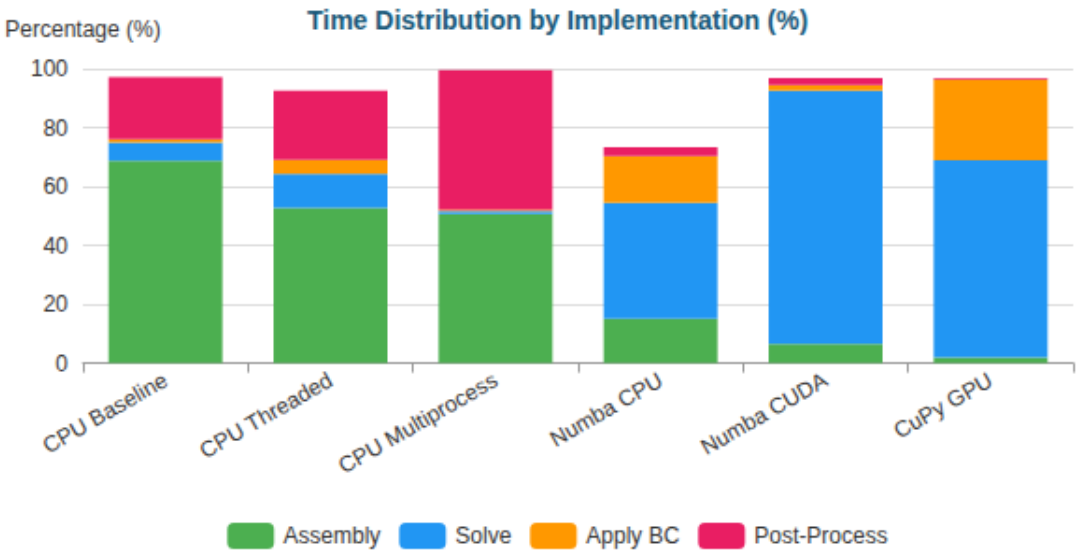
Time Distribution:



Pie



Pie

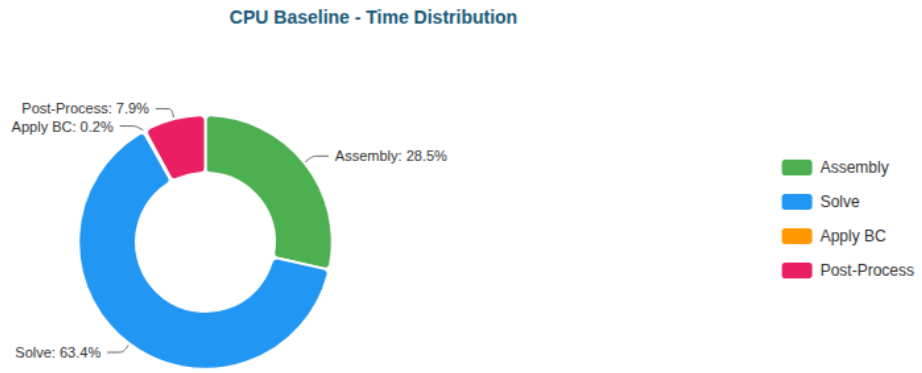


Stacked Bar

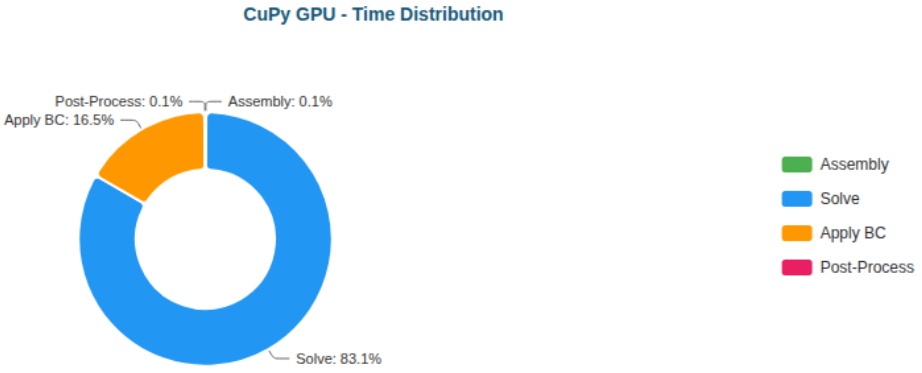
Elbow 90° (M) - 161,984 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (63%)	Assembly (28%)
CPU Threaded	Solve (64%)	Assembly (21%)
CPU Multiprocess	Solve (92%)	Assembly (4%)
Numba CPU	Solve (98%)	BC (1%)
Numba CUDA	Solve (49%)	Assembly (31%)
CuPy GPU	Solve (83%)	BC (16%)

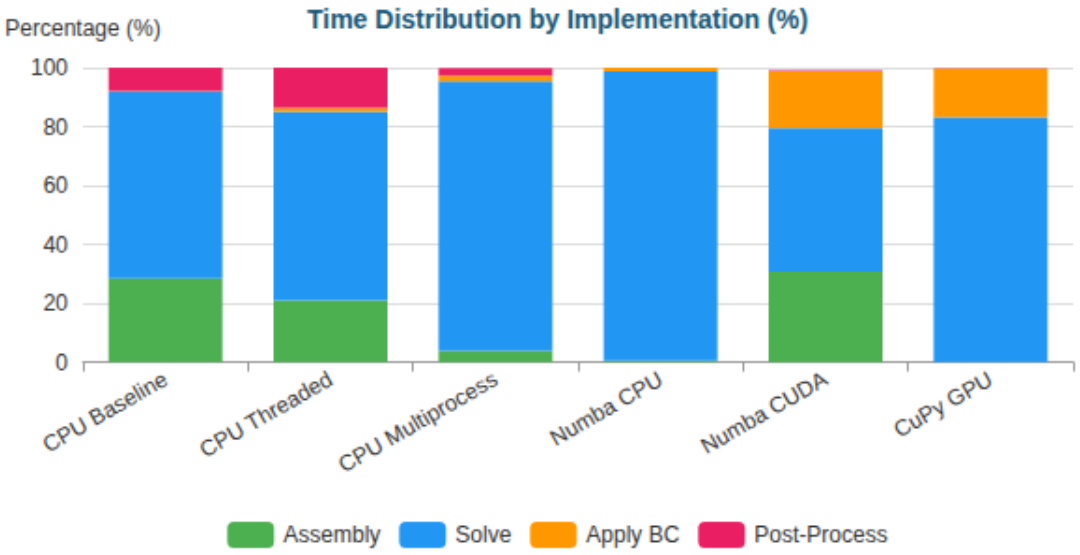
Time Distribution:



Pie



Pie



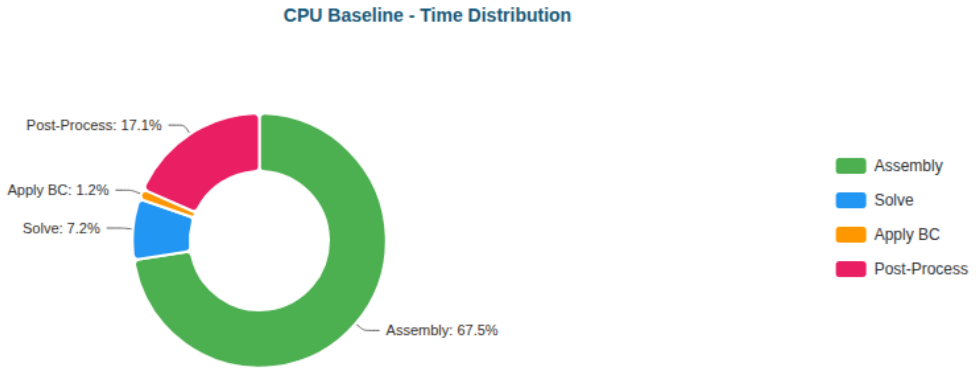
Stacked Bar

S-Bend (XS) - 387 nodes

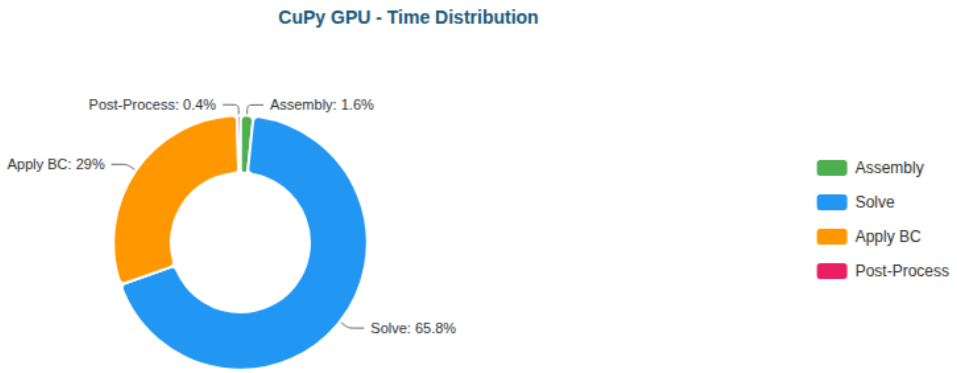
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (68%)	Post-Proc (17%)
CPU Threaded	Assembly (55%)	Post-Proc (17%)
CPU Multiprocess	Assembly (51%)	Post-Proc (46%)
Numba CPU	Solve (39%)	BC (16%)
Numba CUDA	Solve (91%)	Assembly (4%)
CuPy GPU	Solve (66%)	BC (29%)

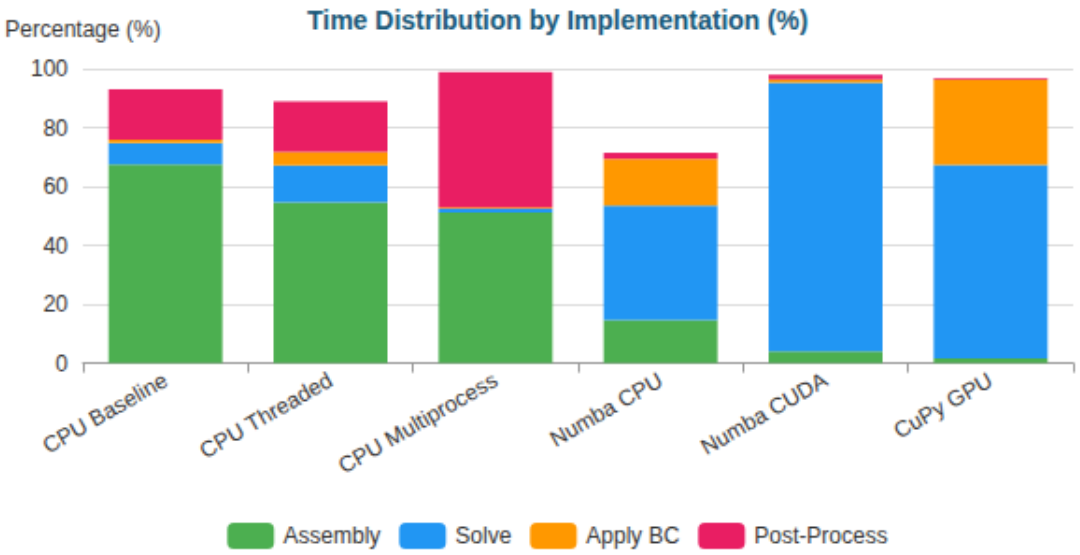
Time Distribution:



Pie



Pie

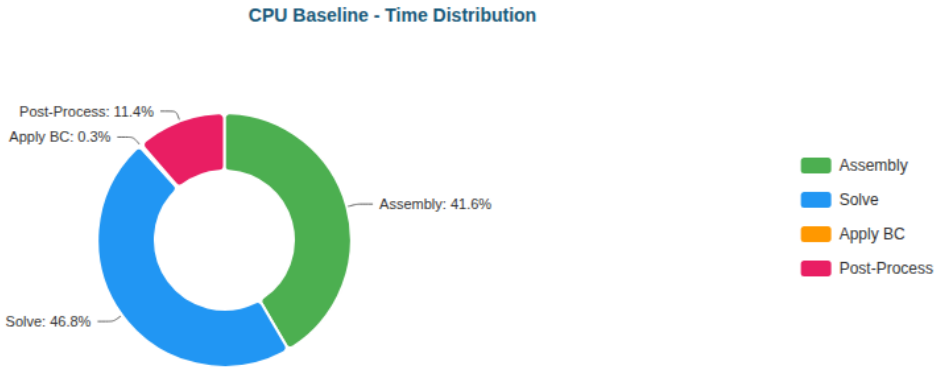


Stacked Bar

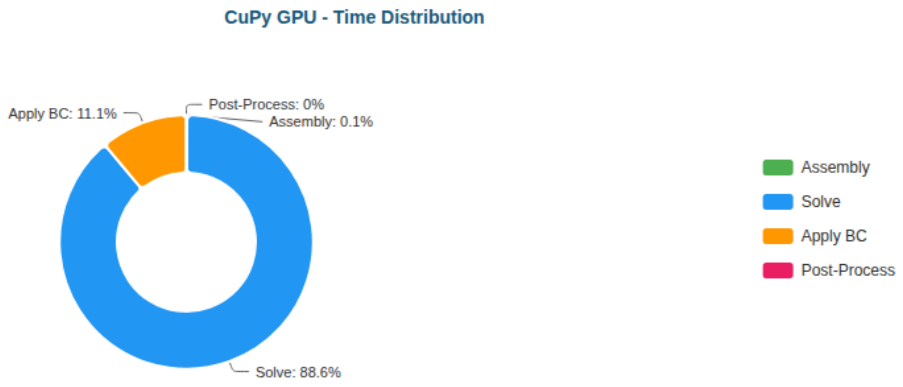
S-Bend (M) - 196,078 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (47%)	Assembly (42%)
CPU Threaded	Solve (71%)	Assembly (17%)
CPU Multiprocess	Solve (94%)	Assembly (3%)
Numba CPU	Solve (98%)	BC (1%)
Numba CUDA	Solve (53%)	Assembly (28%)
CuPy GPU	Solve (89%)	BC (11%)

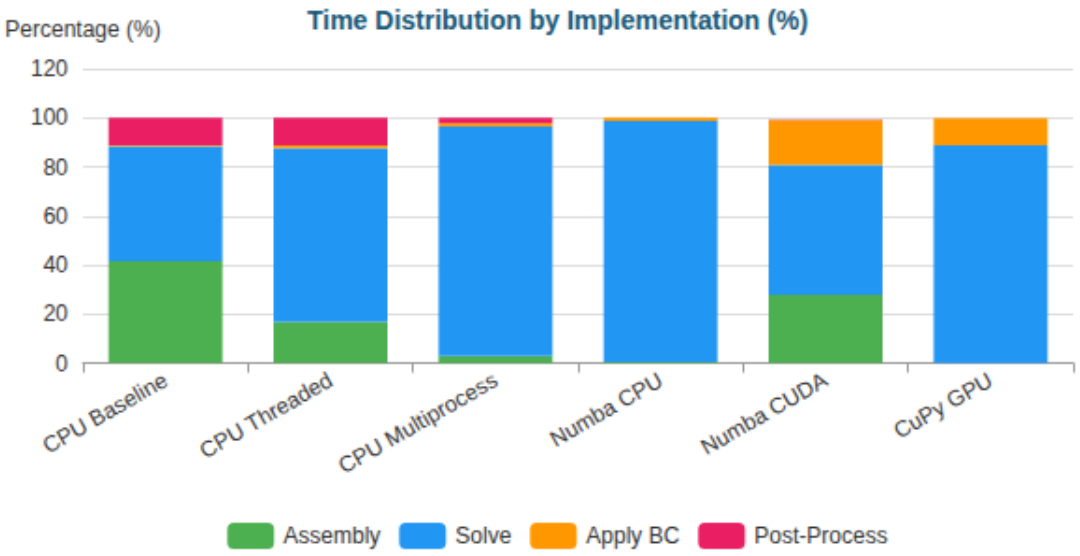
Time Distribution:



Pie



Pie

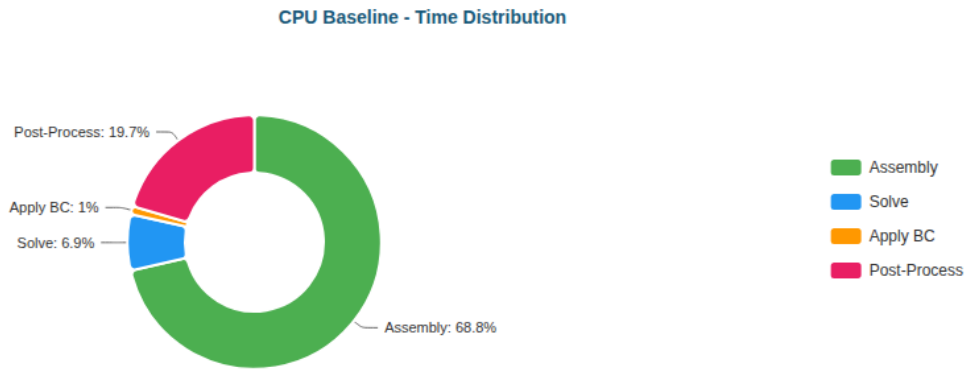


Stacked Bar

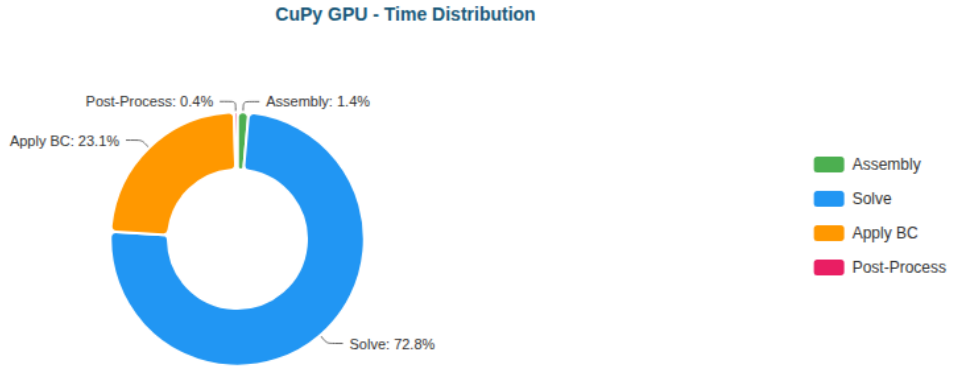
T-Junction (XS) - 393 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (69%)	Post-Proc (20%)
CPU Threaded	Assembly (52%)	Post-Proc (23%)
CPU Multiprocess	Assembly (52%)	Post-Proc (46%)
Numba CPU	Solve (41%)	Assembly (15%)
Numba CUDA	Solve (85%)	Assembly (8%)
CuPy GPU	Solve (73%)	BC (23%)

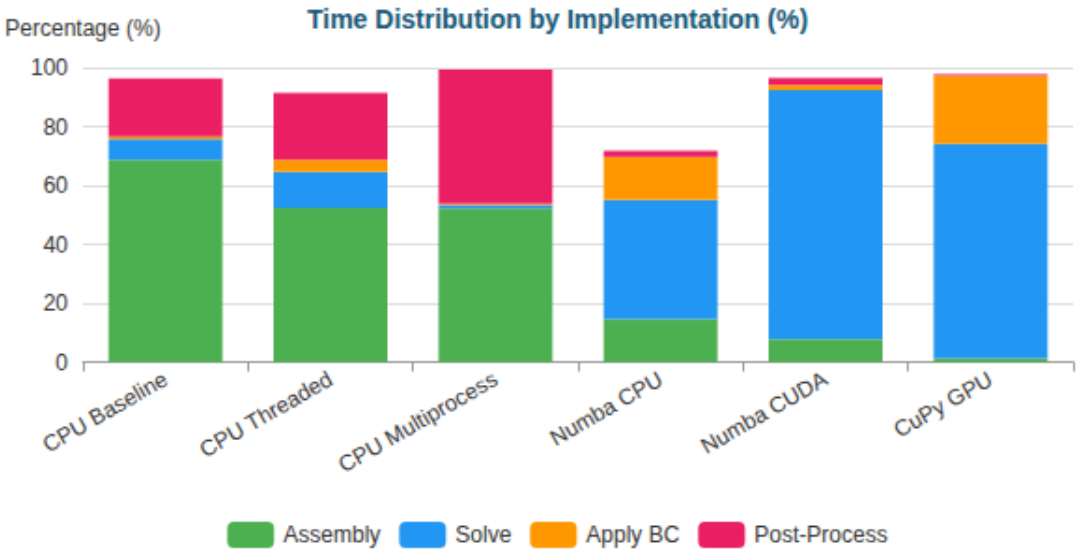
Time Distribution:



Pie



Pie



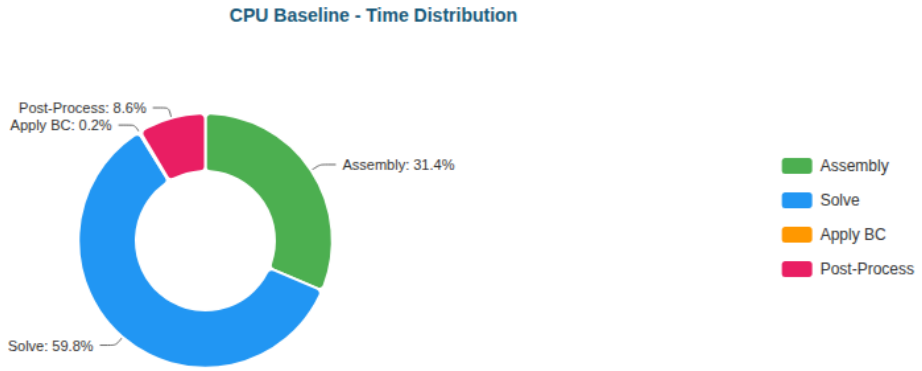
Stacked Bar

T-Junction (M) - 196,420 nodes

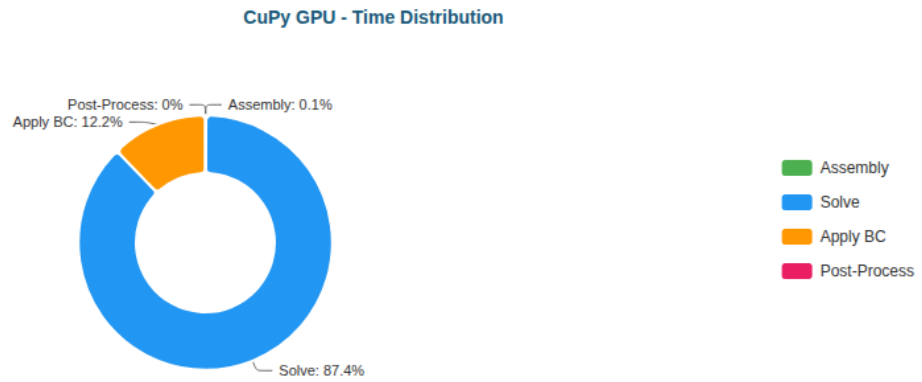
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (60%)	Assembly (31%)
CPU Threaded	Solve (70%)	Assembly (18%)
CPU Multiprocess	Solve (93%)	Assembly (3%)
Numba CPU	Solve (98%)	BC (1%)
Numba CUDA	Solve (51%)	Assembly (29%)
CuPy GPU	Solve (87%)	BC (12%)

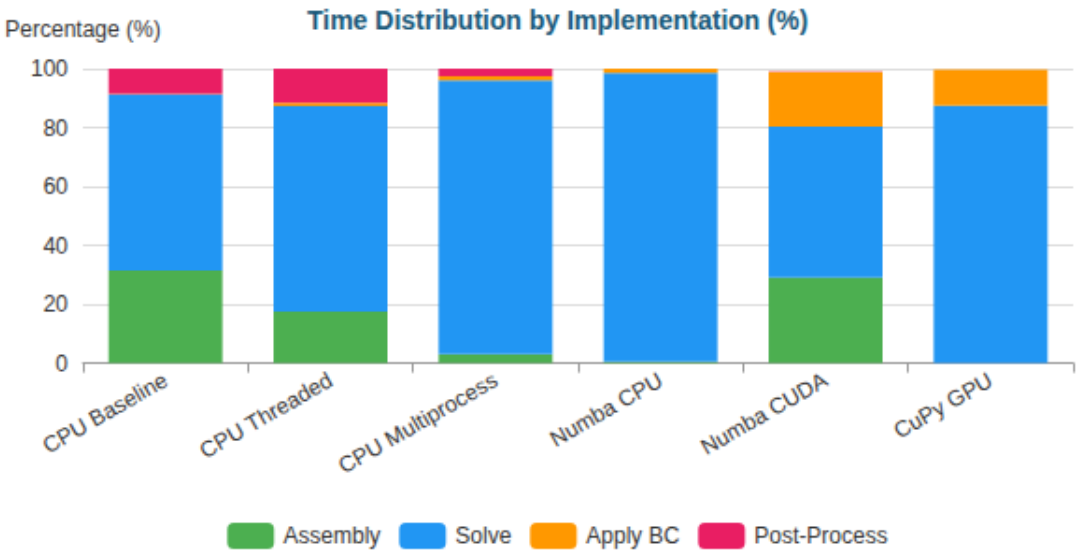
Time Distribution:



Pie



Pie

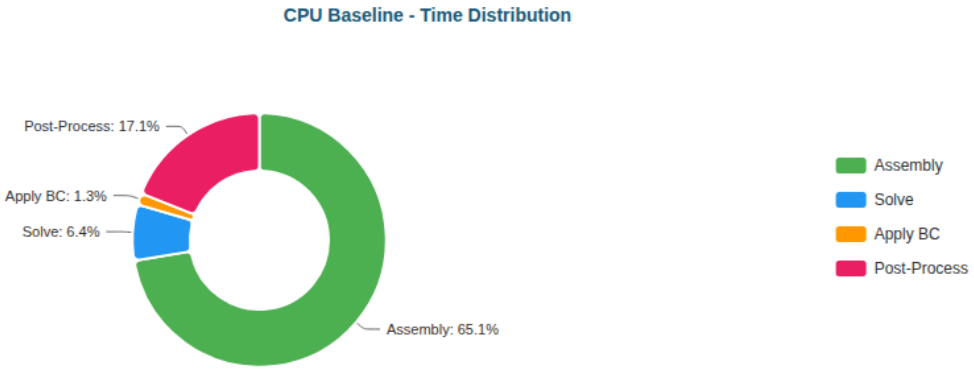


Stacked Bar

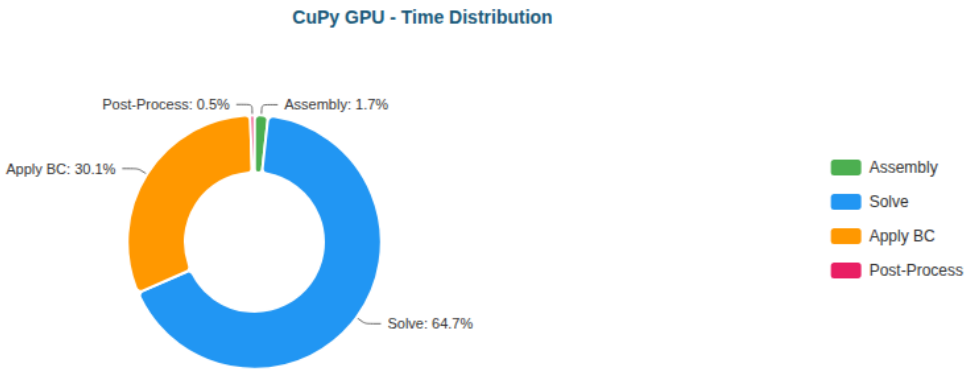
Venturi (XS) - 341 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (65%)	Post-Proc (17%)
CPU Threaded	Assembly (50%)	Post-Proc (23%)
CPU Multiprocess	Assembly (52%)	Post-Proc (46%)
Numba CPU	Solve (43%)	BC (16%)
Numba CUDA	Solve (90%)	Assembly (5%)
CuPy GPU	Solve (65%)	BC (30%)

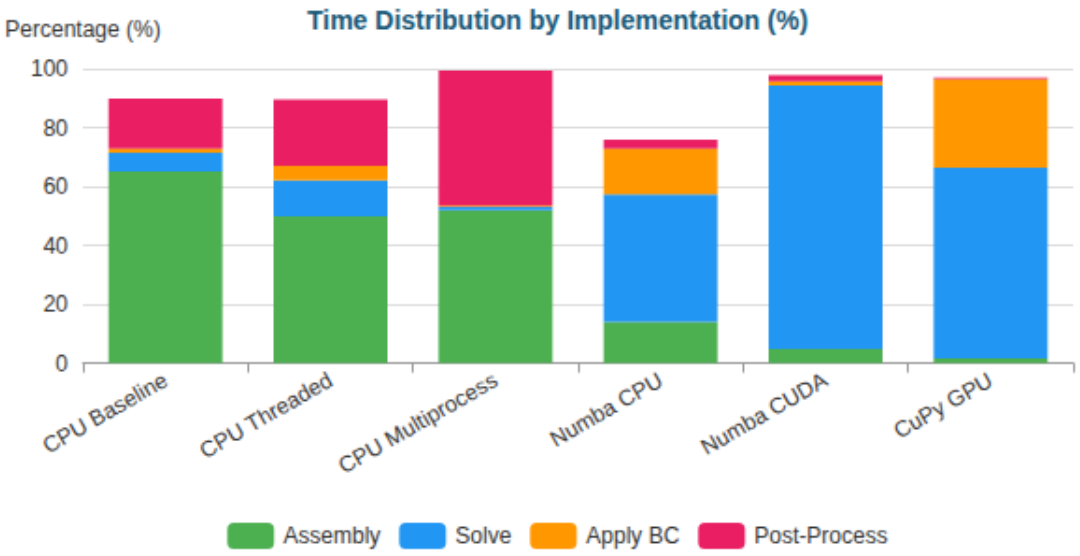
Time Distribution:



Pie



Pie

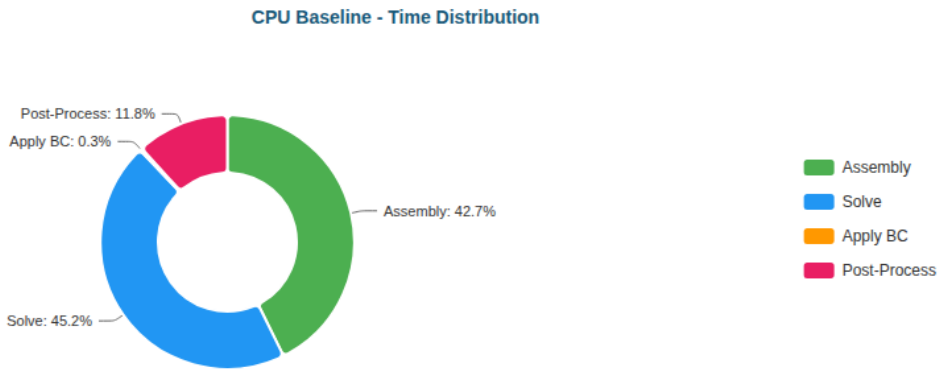


Stacked Bar

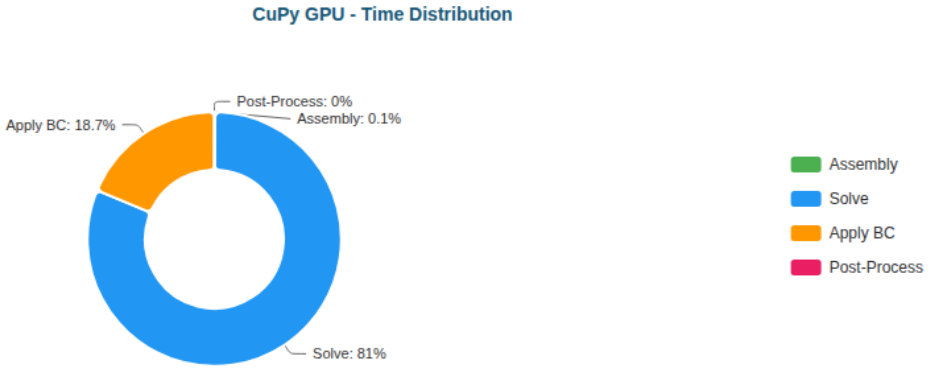
Venturi (M) - 194,325 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Solve (45%)	Assembly (43%)
CPU Threaded	Solve (68%)	Assembly (19%)
CPU Multiprocess	Solve (92%)	Assembly (3%)
Numba CPU	Solve (98%)	BC (1%)
Numba CUDA	Solve (48%)	Assembly (29%)
CuPy GPU	Solve (81%)	BC (19%)

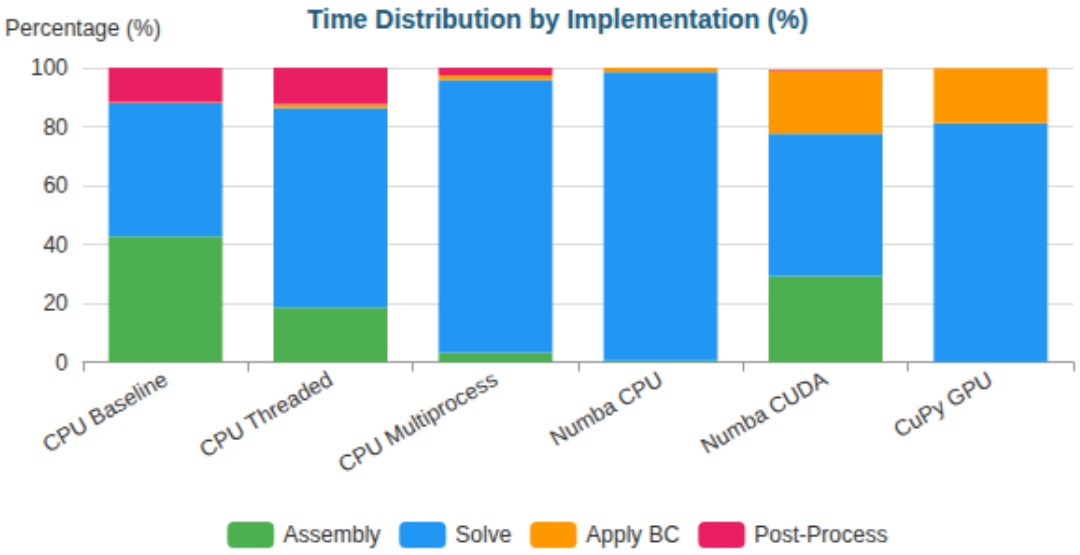
Time Distribution:



Pie



Pie



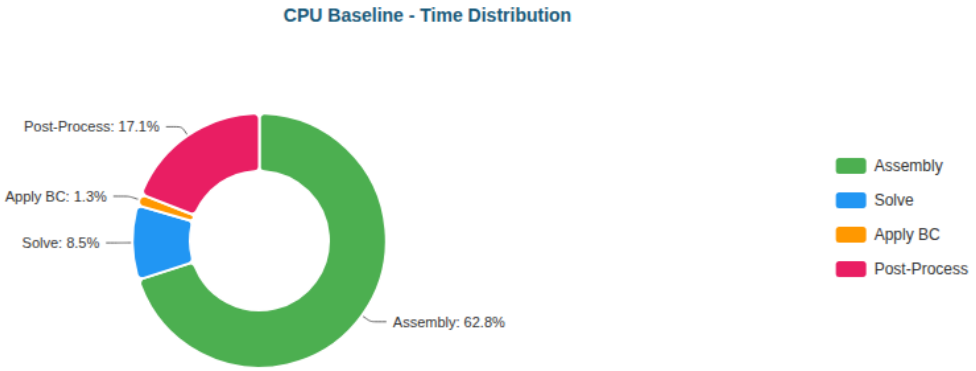
Stacked Bar

Y-Shaped (XS) - 201 nodes

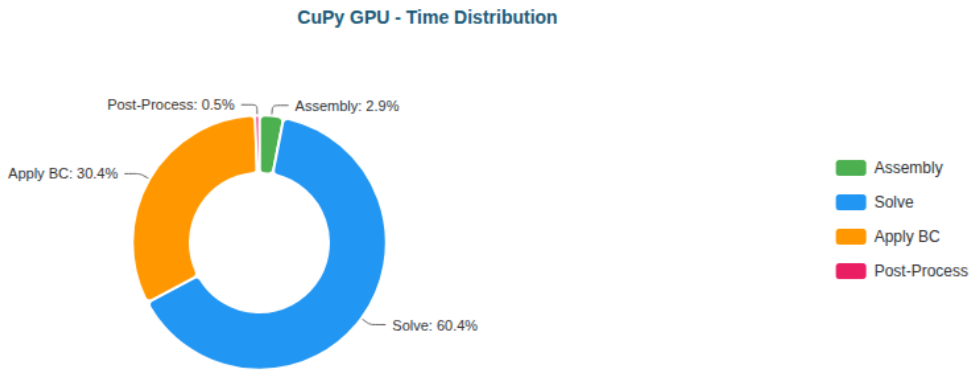
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (63%)	Post-Proc (17%)
CPU Threaded	Assembly (44%)	Post-Proc (22%)
CPU Multiprocess	Assembly (50%)	Post-Proc (48%)
Numba CPU	Solve (39%)	Assembly (14%)
Numba CUDA	Solve (89%)	Assembly (5%)
CuPy GPU	Solve (60%)	BC (30%)

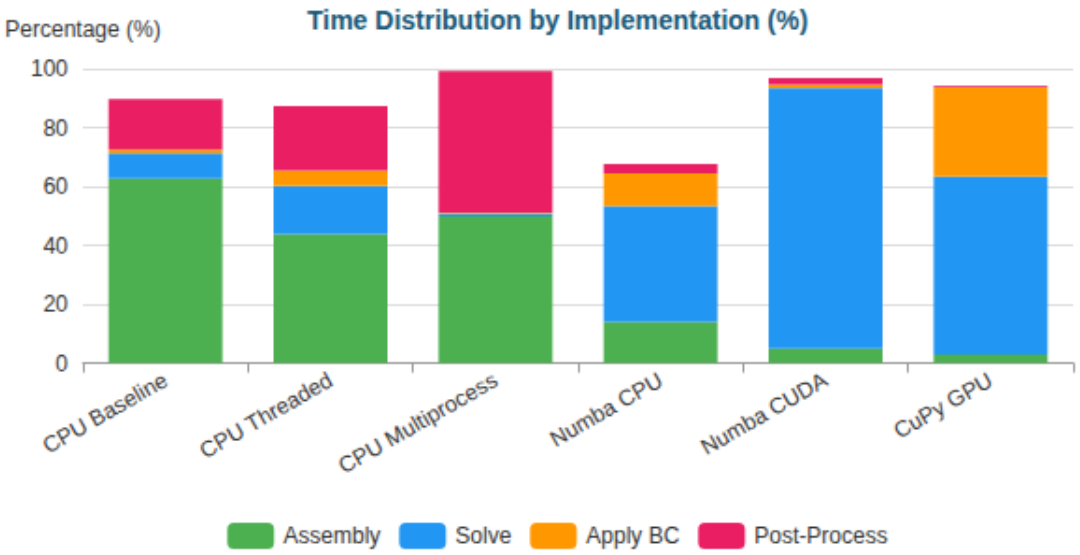
Time Distribution:



Pie



Pie

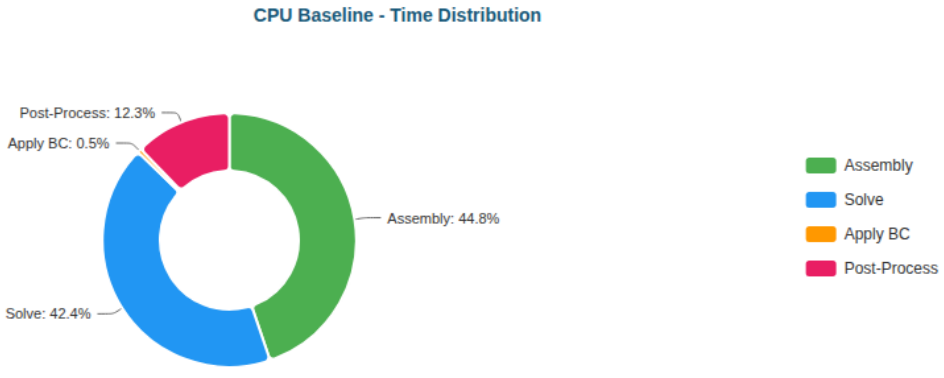


Stacked Bar

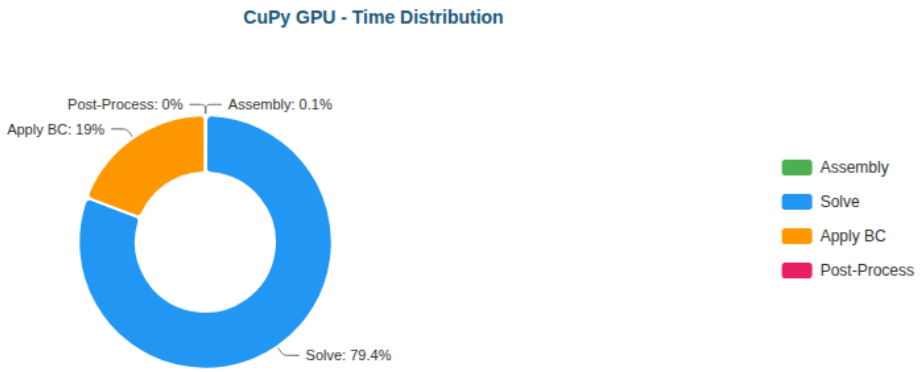
Y-Shaped (M) - 195,853 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (45%)	Solve (42%)
CPU Threaded	Solve (71%)	Assembly (17%)
CPU Multiprocess	Solve (92%)	Assembly (3%)
Numba CPU	Solve (97%)	BC (2%)
Numba CUDA	Solve (50%)	Assembly (26%)
CuPy GPU	Solve (79%)	BC (19%)

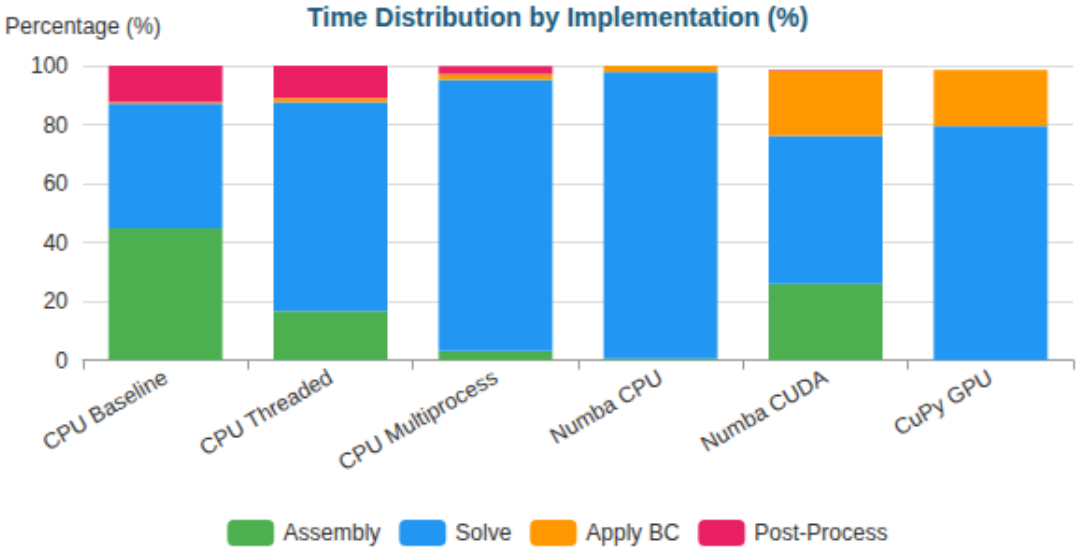
Time Distribution:



Pie



Pie



Stacked Bar

Why Each Optimization Helps

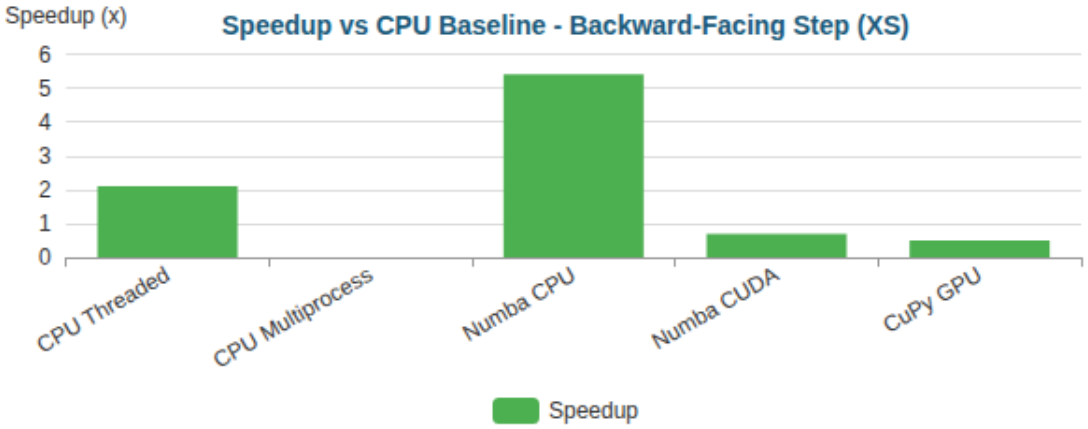
Transition	Reason
Baseline → Threaded	Limited by Python GIL; threads only help for I/O
Threaded → Multiprocess	Bypasses GIL via separate processes; IPC overhead limits gains
Multiprocess → Numba CPU	JIT compilation eliminates interpreter overhead; true parallel loops
Numba CPU → Numba CUDA	GPU parallelism: thousands of threads vs dozens of CPU cores
Numba CUDA → CuPy GPU	CUDA C kernels more optimized than Numba-generated PTX

4.7 RTX 5060 Ti Performance

Key results from performance benchmarks comparing FEM solver implementations.

Backward-Facing Step (XS) (287 nodes)

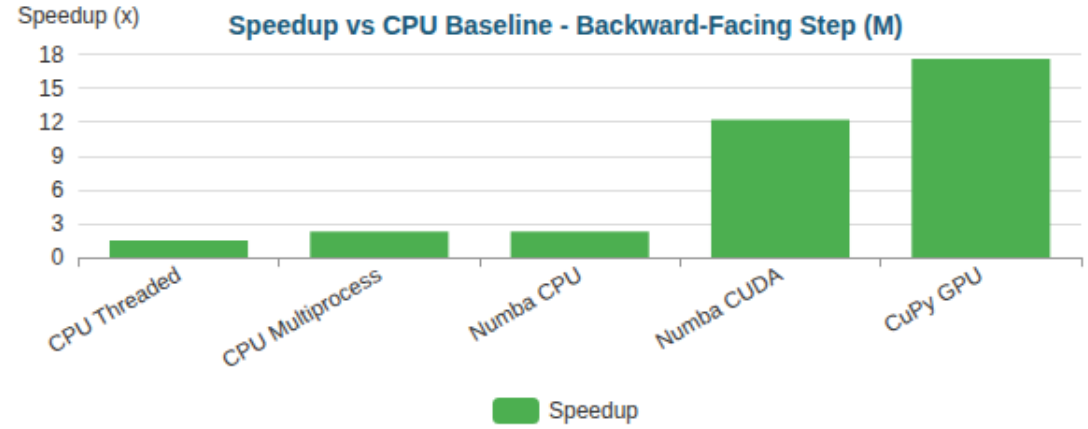
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	35ms ± 1ms	1.0x	3
CPU Threaded	17ms ± 0ms	2.1x	3
CPU Multiprocess	796ms ± 24ms	0.0x	3
Numba CPU	<0.01s ± 0ms	5.4x	3
Numba CUDA	53ms ± 0ms	0.7x	3
CuPy GPU	70ms ± 9ms	0.5x	3



Bar

Backward-Facing Step (M) (195,362 nodes)

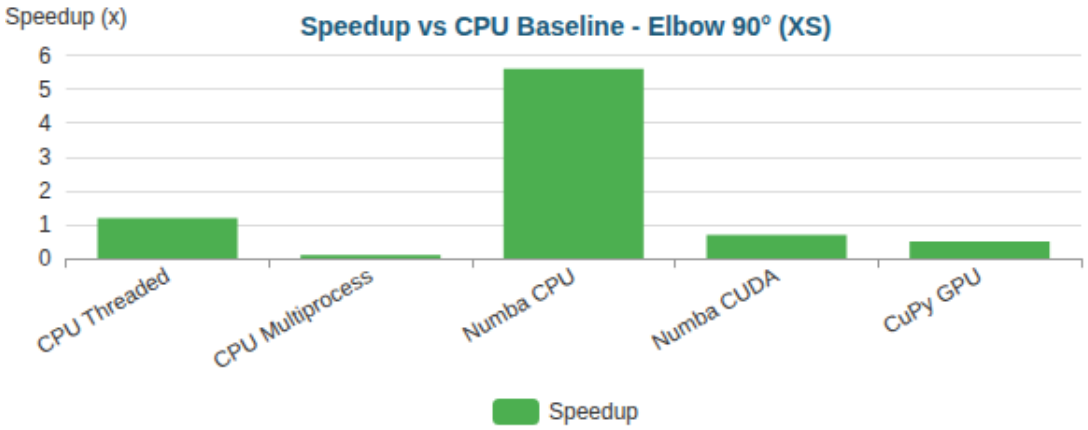
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	39.07s ± 0.20s	1.0x	3
CPU Threaded	25.93s ± 0.65s	1.5x	3
CPU Multiprocess	17.22s ± 0.02s	2.3x	3
Numba CPU	16.88s ± 2.95s	2.3x	3
Numba CUDA	3.19s ± 0.10s	12.2x	3
CuPy GPU	2.22s ± 0.06s	17.6x	3



Bar

Elbow 90° (XS) (411 nodes)

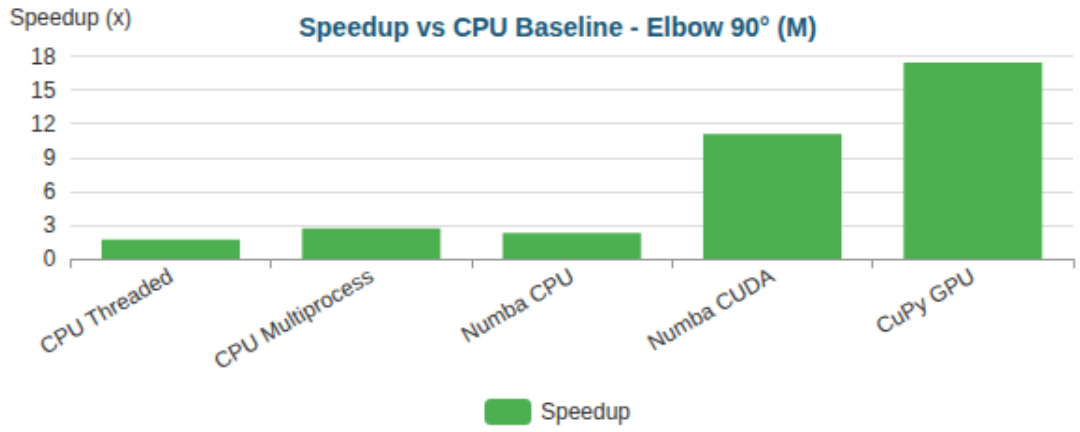
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	44ms ± 0ms	1.0x	3
CPU Threaded	36ms ± 2ms	1.2x	3
CPU Multiprocess	800ms ± 20ms	0.1x	3
Numba CPU	<0.01s ± 0ms	5.6x	3
Numba CUDA	59ms ± 3ms	0.7x	3
CuPy GPU	94ms ± 8ms	0.5x	3



Bar

Elbow 90° (M) (161,984 nodes)

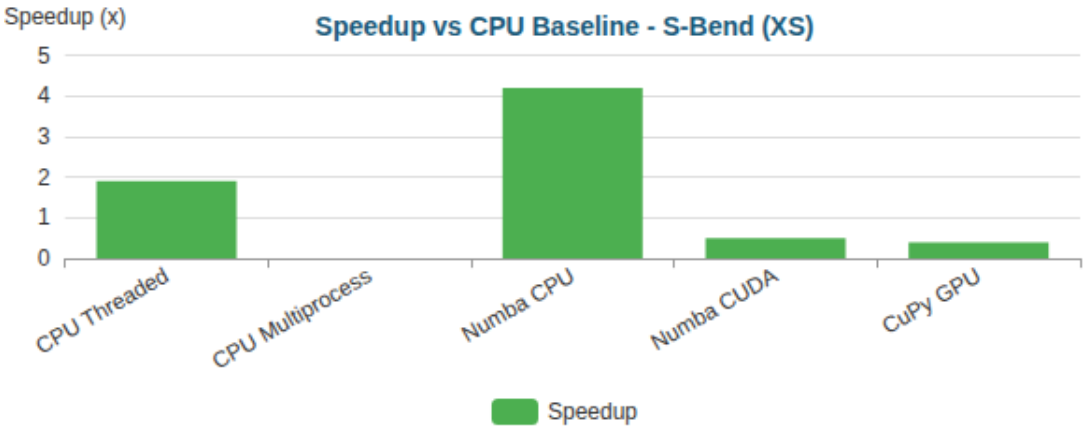
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	33.09s ± 0.59s	1.0x	3
CPU Threaded	19.19s ± 0.07s	1.7x	3
CPU Multiprocess	12.42s ± 0.04s	2.7x	3
Numba CPU	14.30s ± 0.05s	2.3x	3
Numba CUDA	2.98s ± 0.61s	11.1x	3
CuPy GPU	1.90s ± 0.06s	17.4x	3



Bar

S-Bend (XS) (387 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	34ms ± 0ms	1.0x	3
CPU Threaded	18ms ± 0ms	1.9x	3
CPU Multiprocess	789ms ± 23ms	0.0x	3
Numba CPU	<0.01s ± 0ms	4.2x	3
Numba CUDA	63ms ± 4ms	0.5x	3
CuPy GPU	91ms ± 7ms	0.4x	3

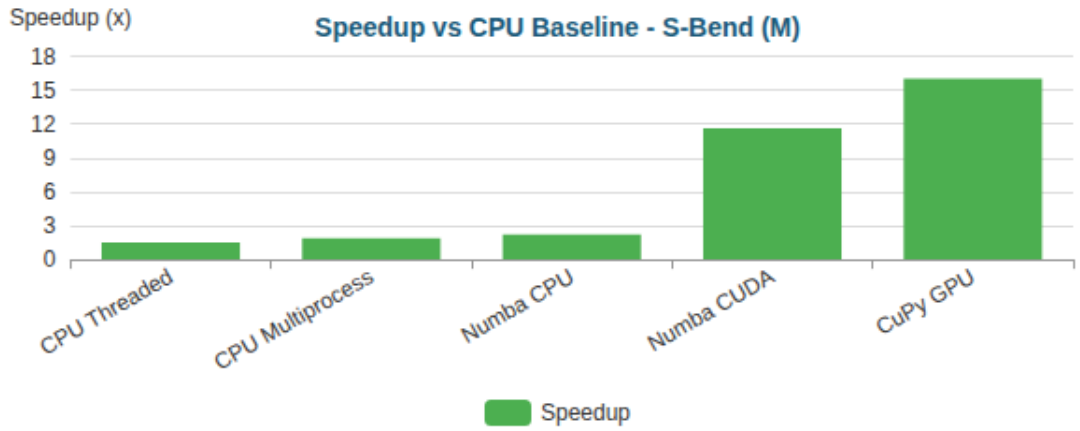


Bar

S-Bend (M) (196,078 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	40.28s ± 0.10s	1.0x	3
CPU Threaded	26.05s ± 0.07s	1.5x	3
CPU Multiprocess	21.00s ± 0.71s	1.9x	3
Numba CPU	18.06s ± 0.23s	2.2x	3
Numba CUDA	3.47s ± 0.09s	11.6x	3

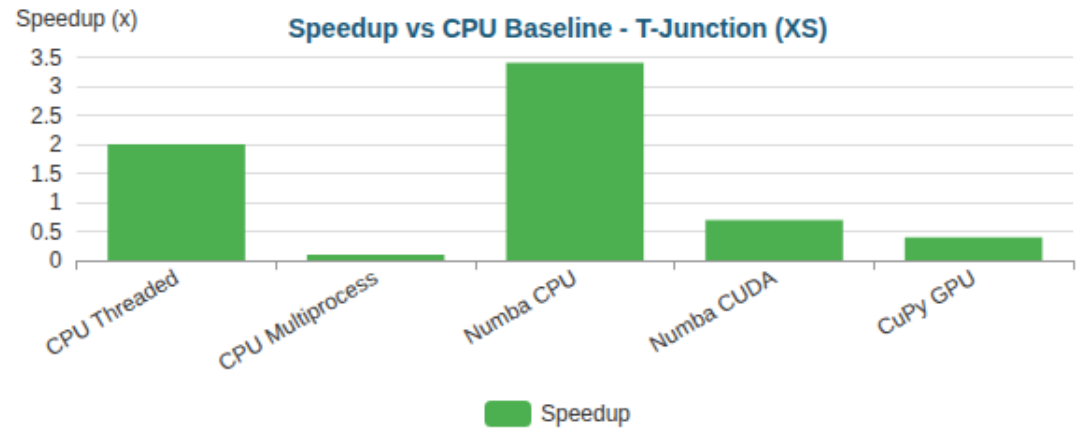
Implementation	Total Time	Speedup vs Baseline	N
CuPy GPU	2.52s ± 0.10s	16.0x	3



Bar

T-Junction (XS) (393 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	44ms ± 1ms	1.0x	3
CPU Threaded	22ms ± 0ms	2.0x	3
CPU Multiprocess	828ms ± 66ms	0.1x	3
Numba CPU	13ms ± 4ms	3.4x	3
Numba CUDA	65ms ± 1ms	0.7x	3
CuPy GPU	101ms ± 17ms	0.4x	3

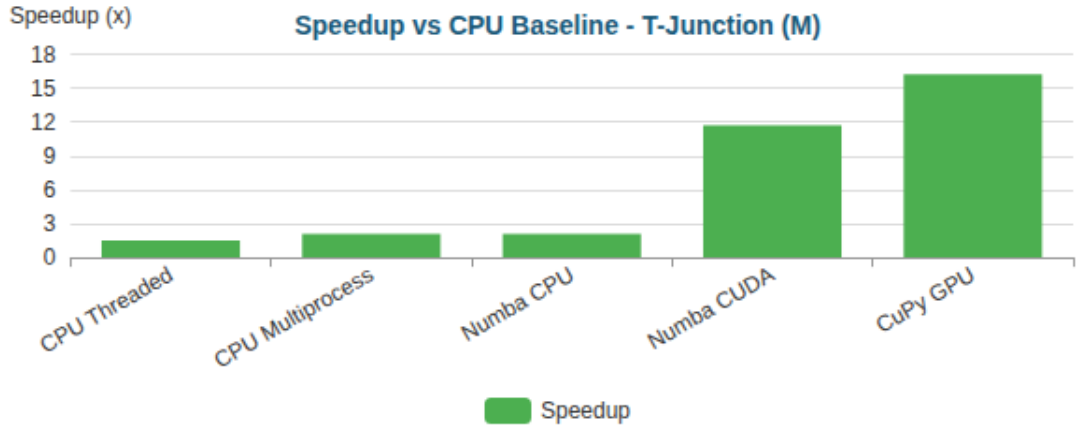


Bar

T-Junction (M) (196,420 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	39.19s ± 0.47s	1.0x	3
CPU Threaded	25.37s ± 0.43s	1.5x	3
CPU Multiprocess	18.31s ± 0.15s	2.1x	3

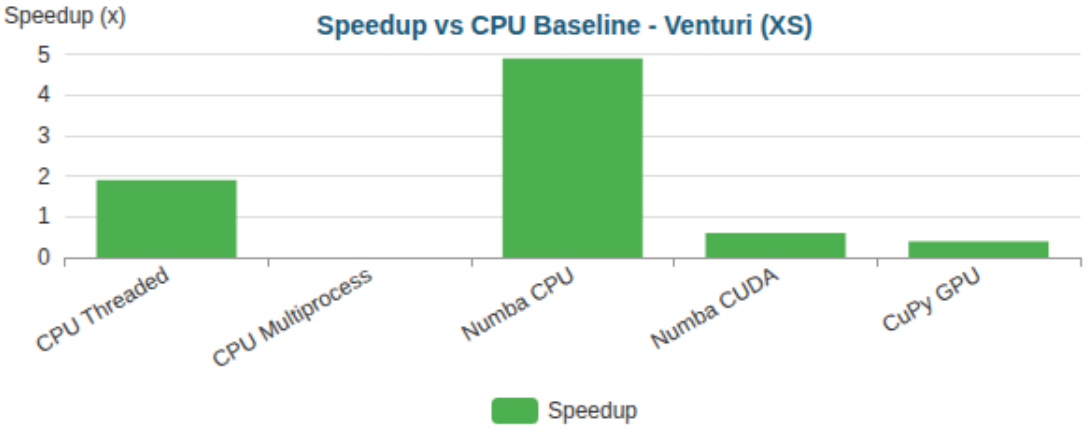
Implementation	Total Time	Speedup vs Baseline	N
Numba CPU	18.77s ± 5.17s	2.1x	3
Numba CUDA	3.35s ± 0.03s	11.7x	3
CuPy GPU	2.43s ± 0.24s	16.2x	3



Bar

Venturi (XS) (341 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	34ms ± 0ms	1.0x	3
CPU Threaded	18ms ± 0ms	1.9x	3
CPU Multiprocess	889ms ± 17ms	0.0x	3
Numba CPU	<0.01s ± 0ms	4.9x	3
Numba CUDA	53ms ± 1ms	0.6x	3
CuPy GPU	92ms ± 16ms	0.4x	3

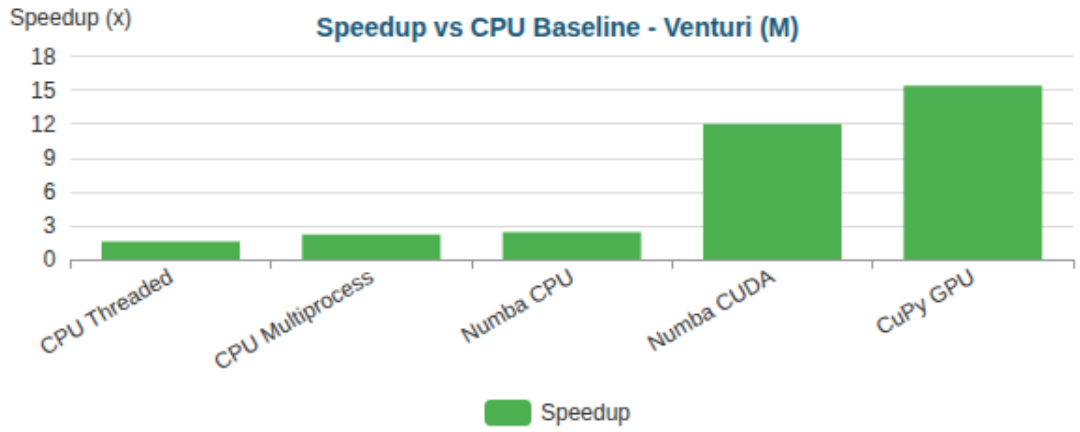


Bar

Venturi (M) (194,325 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	38.42s ± 0.09s	1.0x	3

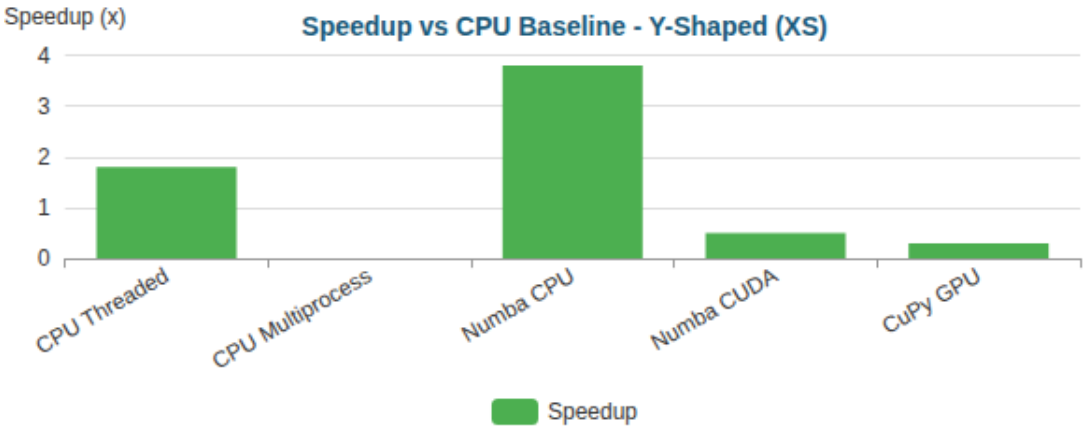
Implementation	Total Time	Speedup vs Baseline	N
CPU Threaded	23.74s ± 0.41s	1.6x	3
CPU Multiprocess	17.85s ± 0.55s	2.2x	3
Numba CPU	15.80s ± 0.42s	2.4x	3
Numba CUDA	3.20s ± 0.05s	12.0x	3
CuPy GPU	2.50s ± 0.11s	15.4x	3



Bar

Y-Shaped (XS) (201 nodes)

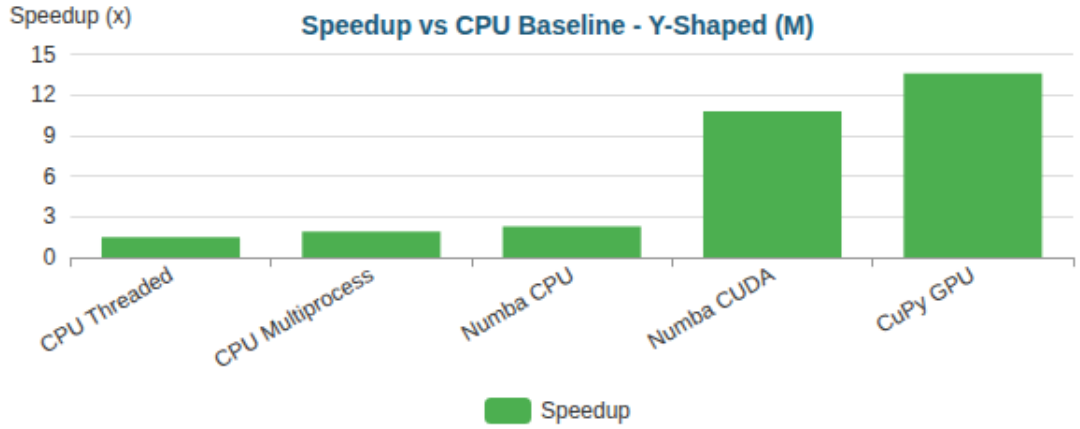
Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	23ms ± 1ms	1.0x	3
CPU Threaded	13ms ± 1ms	1.8x	3
CPU Multiprocess	889ms ± 9ms	0.0x	3
Numba CPU	<0.01s ± 0ms	3.8x	3
Numba CUDA	44ms ± 1ms	0.5x	3
CuPy GPU	65ms ± 7ms	0.3x	3



Bar

Y-Shaped (M) (195,853 nodes)

Implementation	Total Time	Speedup vs Baseline	N
CPU Baseline	31.21s ± 0.74s	1.0x	3
CPU Threaded	20.27s ± 0.04s	1.5x	3
CPU Multiprocess	16.05s ± 0.79s	1.9x	3
Numba CPU	13.41s ± 0.09s	2.3x	3
Numba CUDA	2.89s ± 0.04s	10.8x	3
CuPy GPU	2.30s ± 0.05s	13.6x	3



Bar

Critical Analysis

Critical Analysis

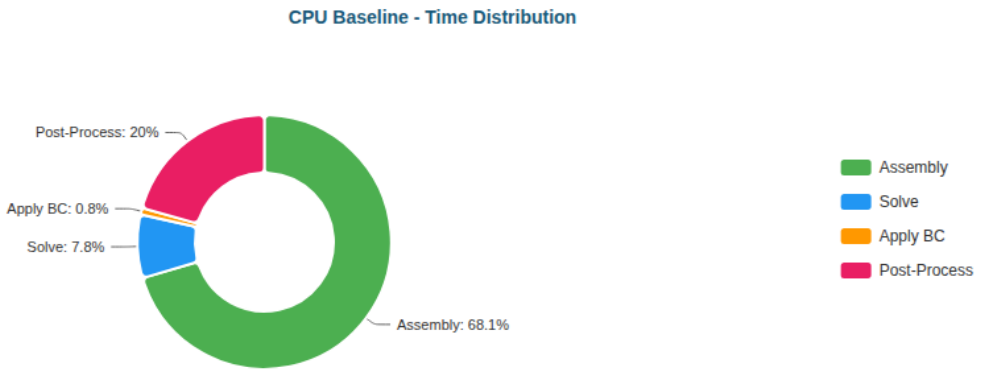
Bottleneck Evolution

As optimizations progress, the computational bottleneck shifts:

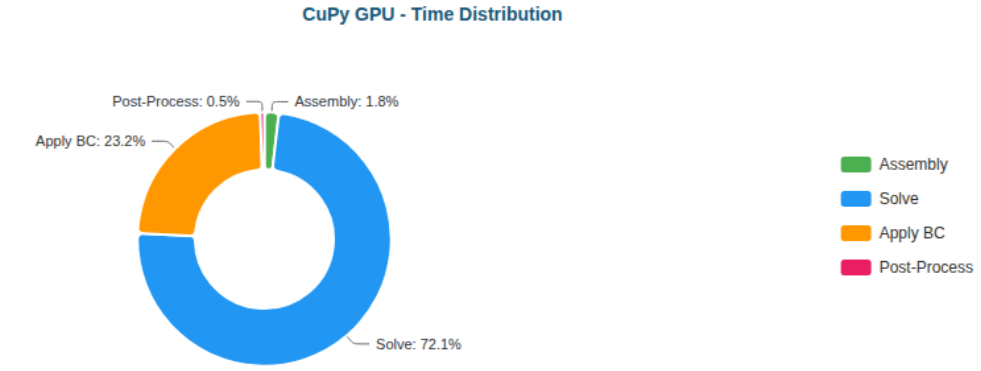
Backward-Facing Step (XS) - 287 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (68%)	Post-Proc (20%)
CPU Threaded	Assembly (51%)	Post-Proc (25%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (55%)	BC (16%)
Numba CUDA	Solve (88%)	Assembly (6%)
CuPy GPU	Solve (72%)	BC (23%)

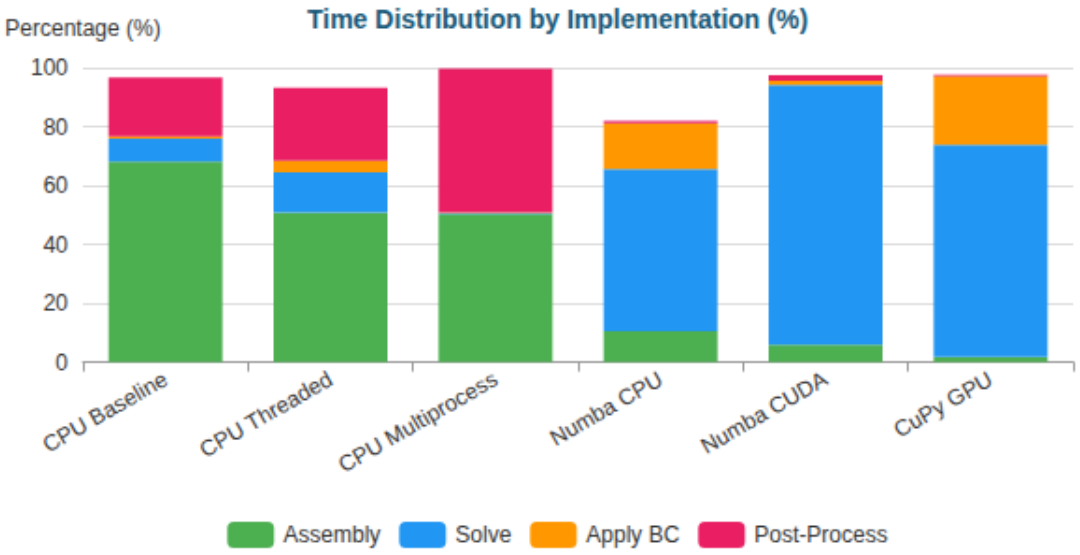
Time Distribution:



Pie



Pie



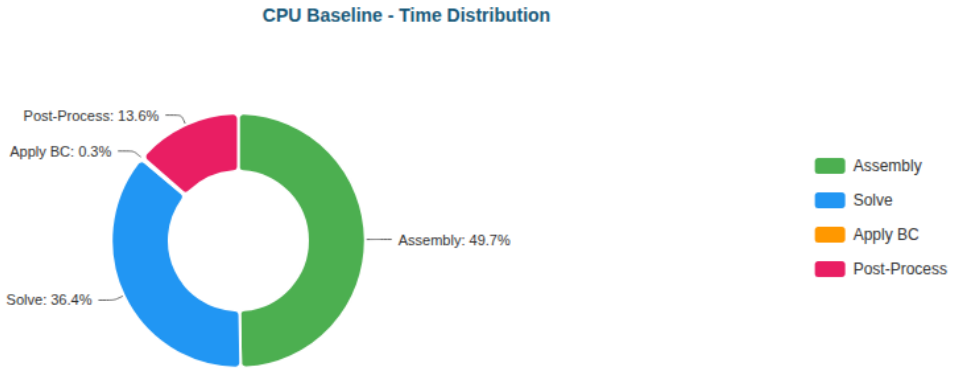
Stacked Bar

Backward-Facing Step (M) - 195,362 nodes

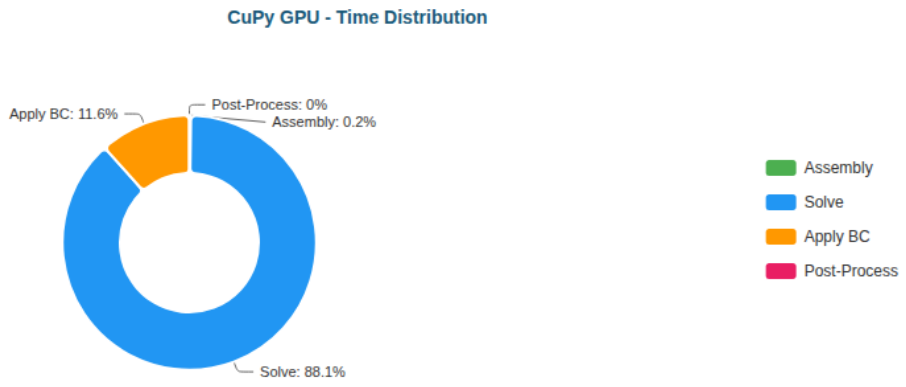
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (50%)	Solve (36%)
CPU Threaded	Solve (56%)	Assembly (28%)
CPU Multiprocess	Solve (78%)	Assembly (12%)
Numba CPU	Solve (94%)	BC (4%)
Numba CUDA	Solve (50%)	Assembly (27%)
CuPy GPU	Solve (88%)	BC (12%)

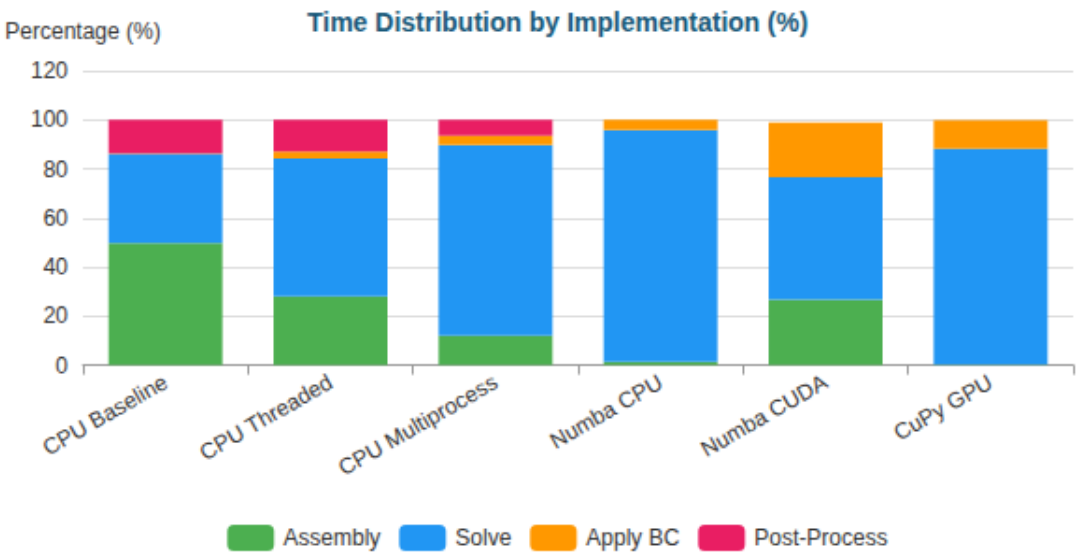
Time Distribution:



Pie



Pie

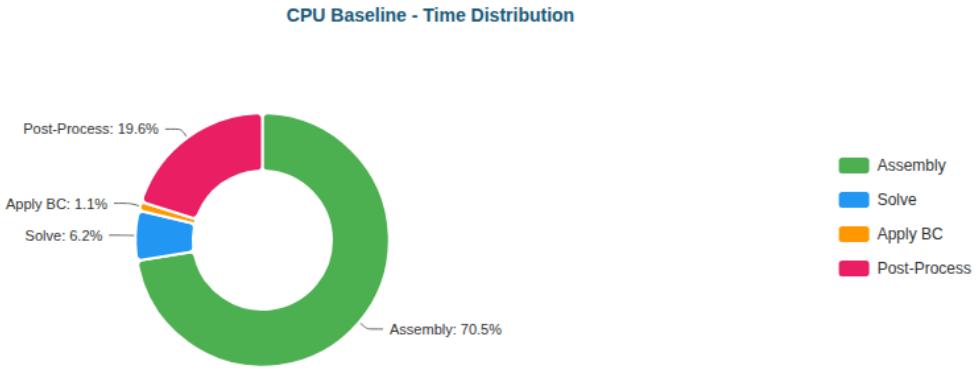


Stacked Bar

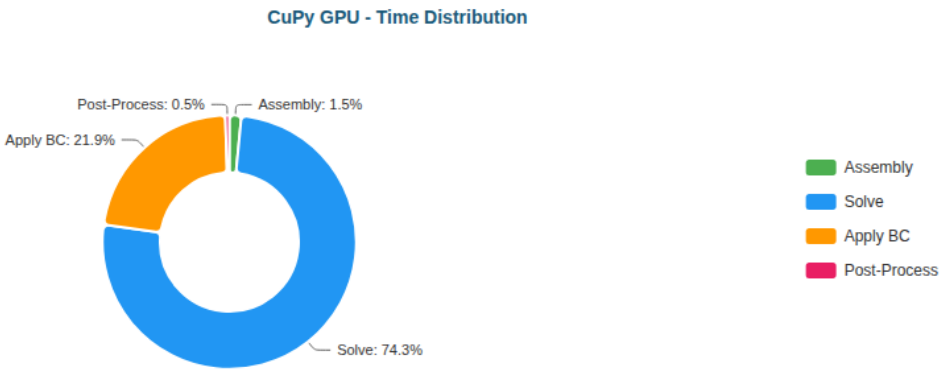
Elbow 90° (XS) - 411 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (71%)	Post-Proc (20%)
CPU Threaded	Assembly (59%)	Post-Proc (24%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (53%)	BC (22%)
Numba CUDA	Solve (87%)	Assembly (6%)
CuPy GPU	Solve (74%)	BC (22%)

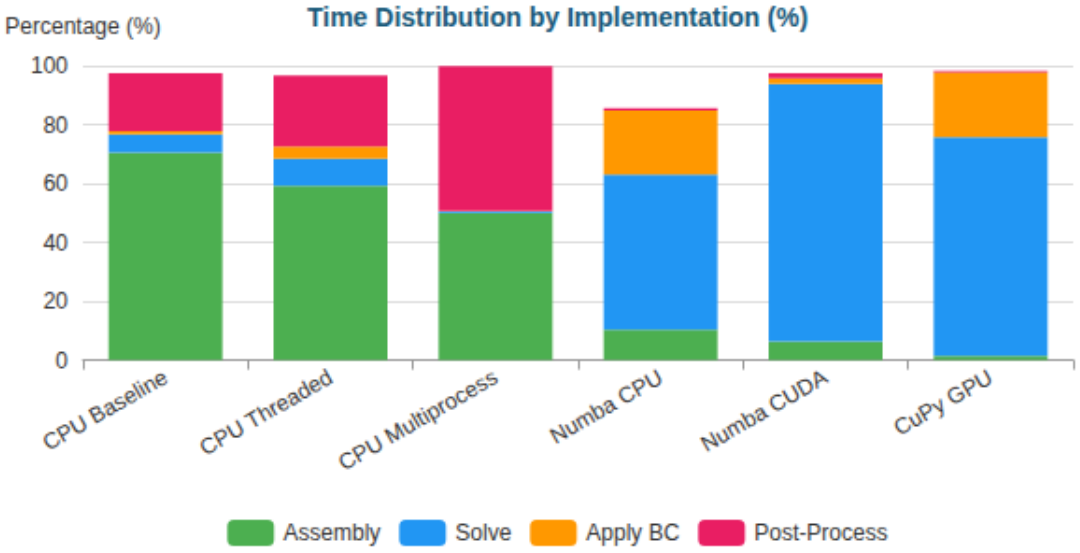
Time Distribution:



Pie



Pie

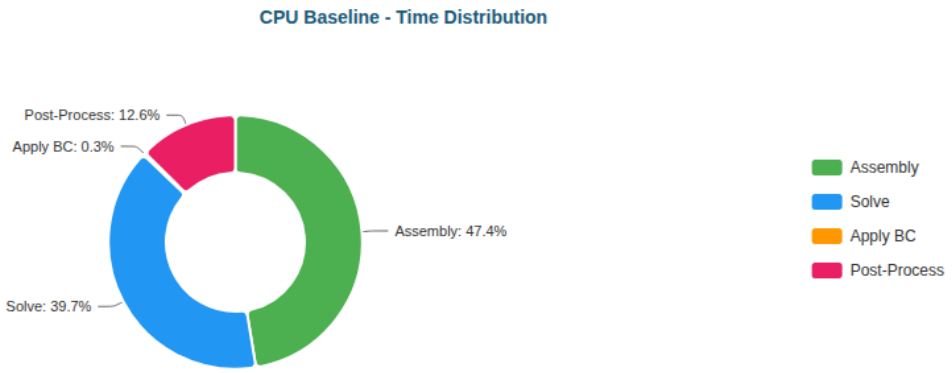


Stacked Bar

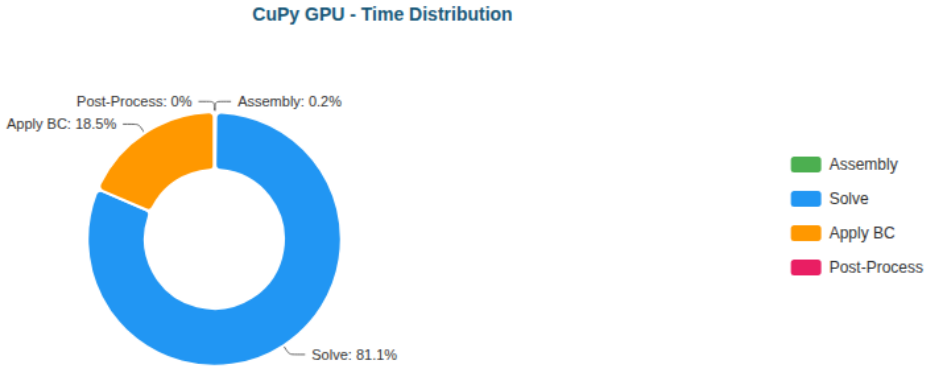
Elbow 90° (M) - 161,984 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (47%)	Solve (40%)
CPU Threaded	Solve (51%)	Assembly (31%)
CPU Multiprocess	Solve (73%)	Assembly (15%)
Numba CPU	Solve (94%)	BC (4%)
Numba CUDA	Solve (48%)	Assembly (28%)
CuPy GPU	Solve (81%)	BC (19%)

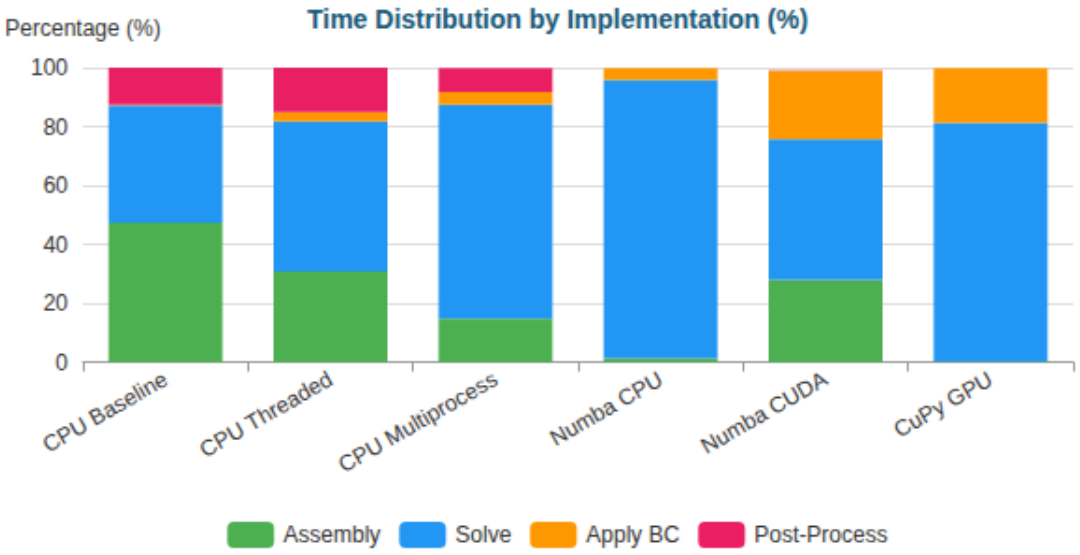
Time Distribution:



Pie



Pie



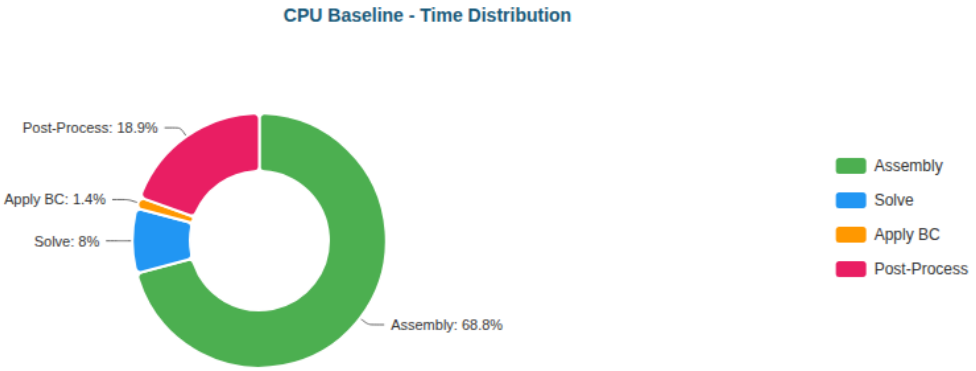
Stacked Bar

S-Bend (XS) - 387 nodes

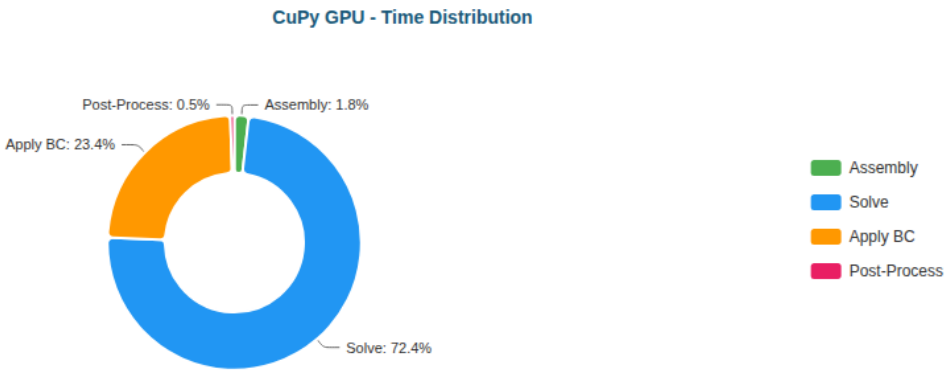
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (69%)	Post-Proc (19%)
CPU Threaded	Assembly (50%)	Post-Proc (20%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (52%)	BC (21%)
Numba CUDA	Solve (89%)	Assembly (5%)
CuPy GPU	Solve (72%)	BC (23%)

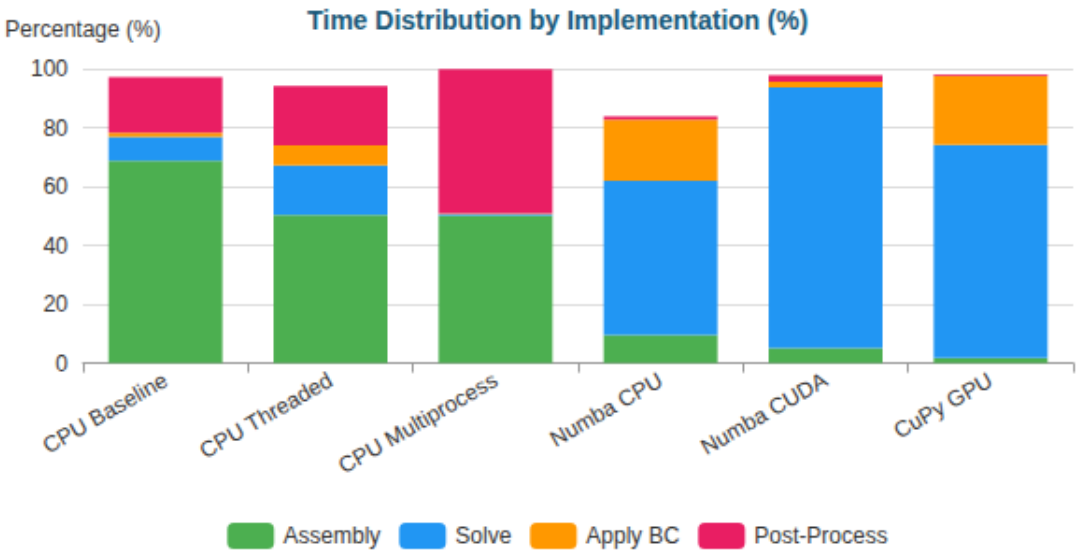
Time Distribution:



Pie



Pie

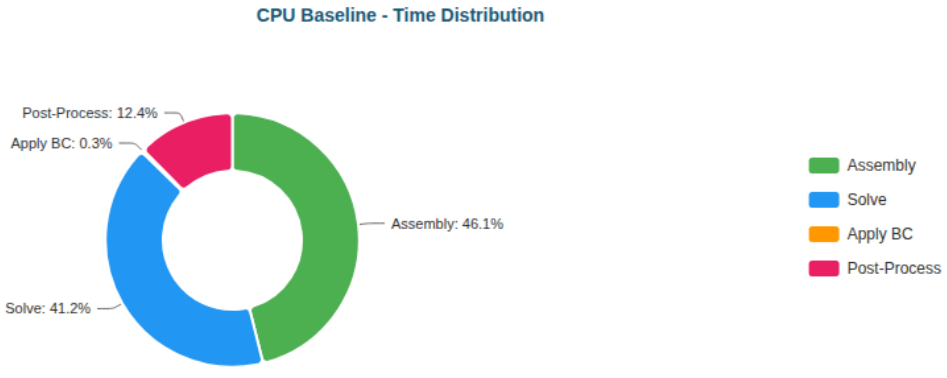


Stacked Bar

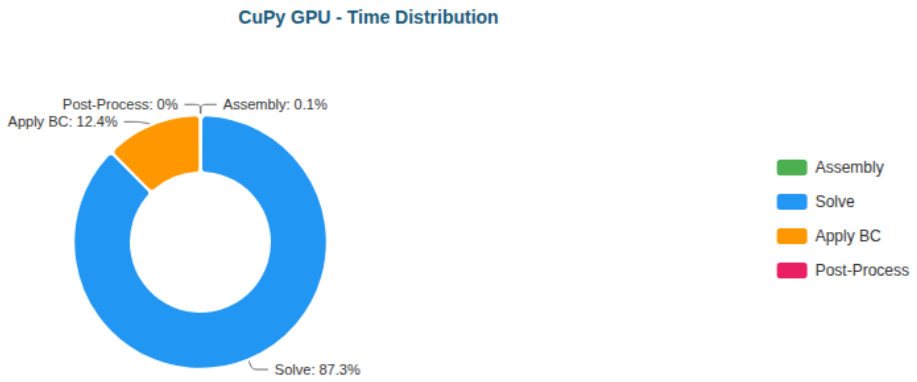
S-Bend (M) - 196,078 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (46%)	Solve (41%)
CPU Threaded	Solve (58%)	Assembly (26%)
CPU Multiprocess	Solve (80%)	Assembly (11%)
Numba CPU	Solve (95%)	BC (4%)
Numba CUDA	Solve (52%)	Assembly (26%)
CuPy GPU	Solve (87%)	BC (12%)

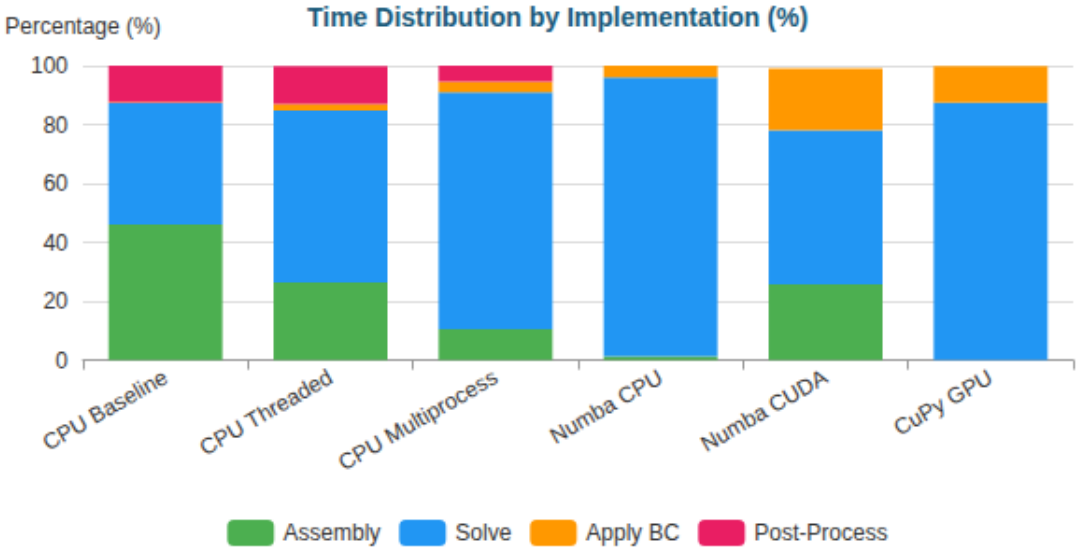
Time Distribution:



Pie



Pie

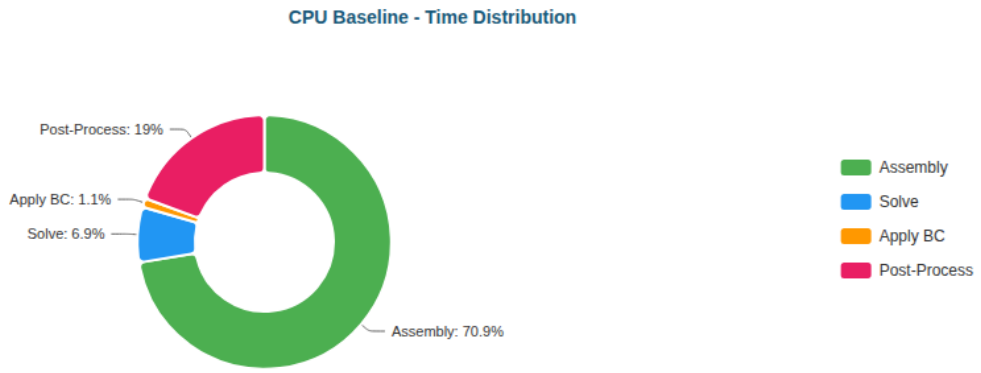


Stacked Bar

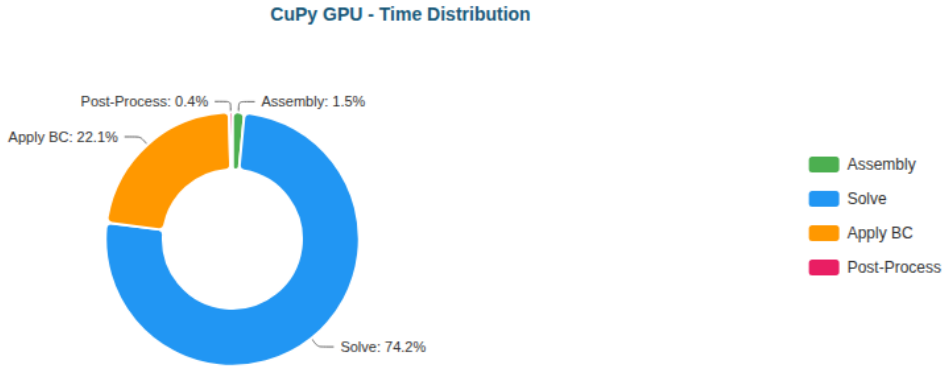
T-Junction (XS) - 393 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (71%)	Post-Proc (19%)
CPU Threaded	Assembly (53%)	Post-Proc (24%)
CPU Multiprocess	Assembly (51%)	Post-Proc (48%)
Numba CPU	Solve (59%)	BC (17%)
Numba CUDA	Solve (88%)	Assembly (6%)
CuPy GPU	Solve (74%)	BC (22%)

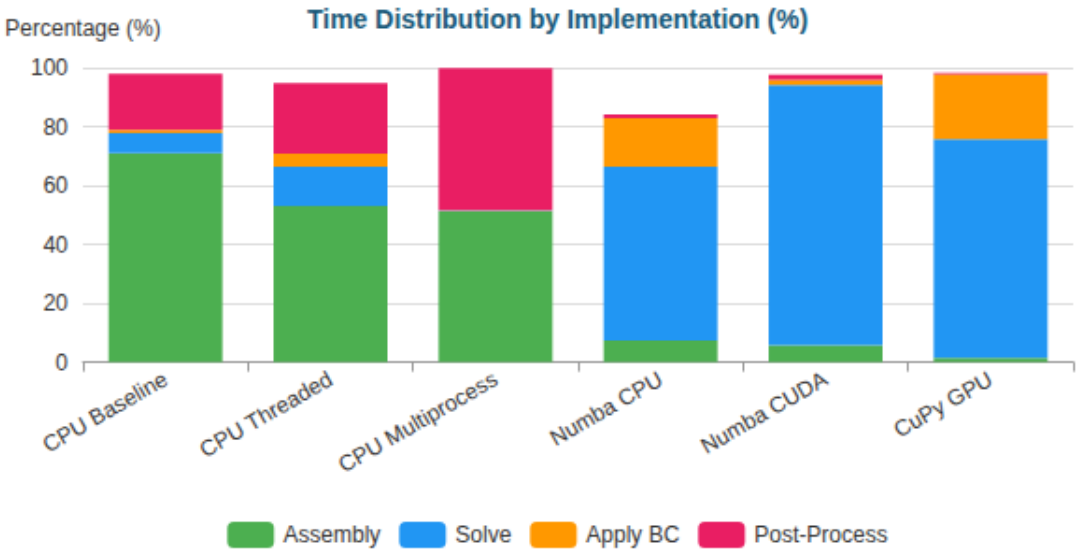
Time Distribution:



Pie



Pie



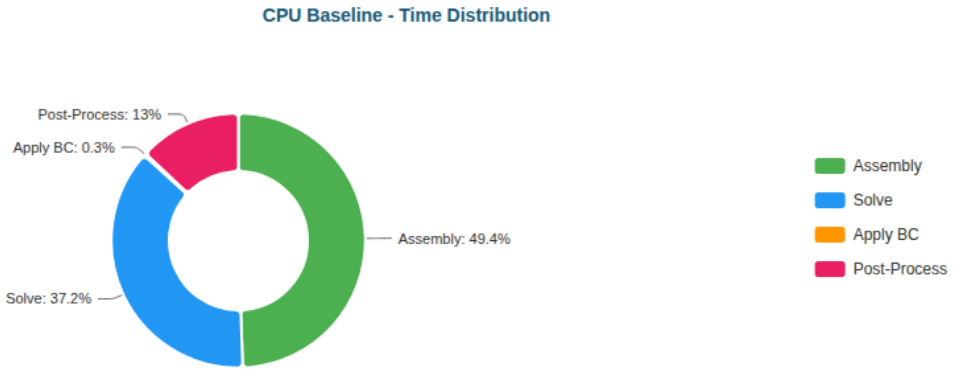
Stacked Bar

T-Junction (M) - 196,420 nodes

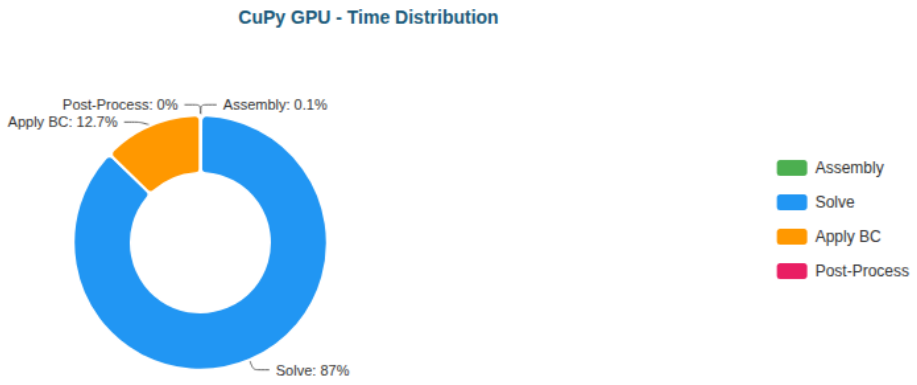
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (49%)	Solve (37%)
CPU Threaded	Solve (57%)	Assembly (28%)
CPU Multiprocess	Solve (79%)	Assembly (11%)
Numba CPU	Solve (95%)	BC (4%)
Numba CUDA	Solve (51%)	Assembly (27%)
CuPy GPU	Solve (87%)	BC (13%)

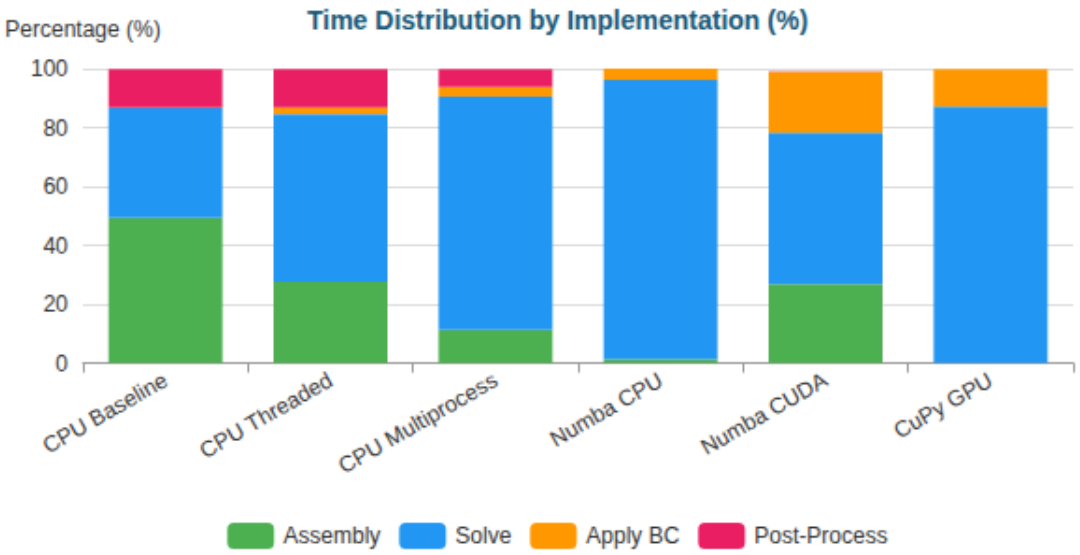
Time Distribution:



Pie



Pie

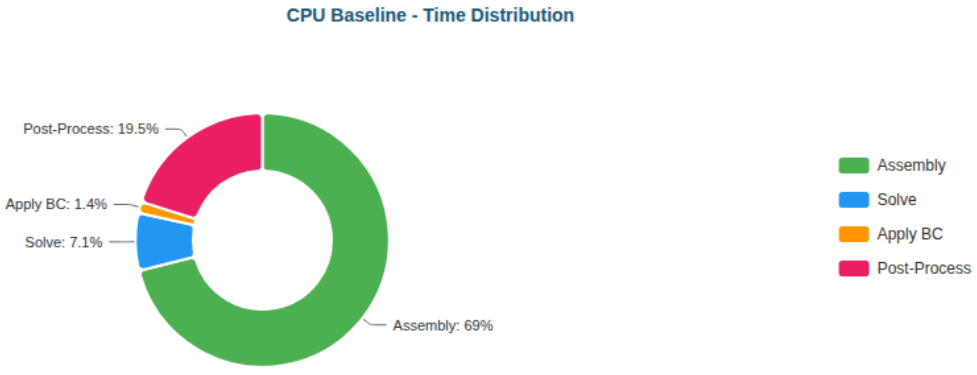


Stacked Bar

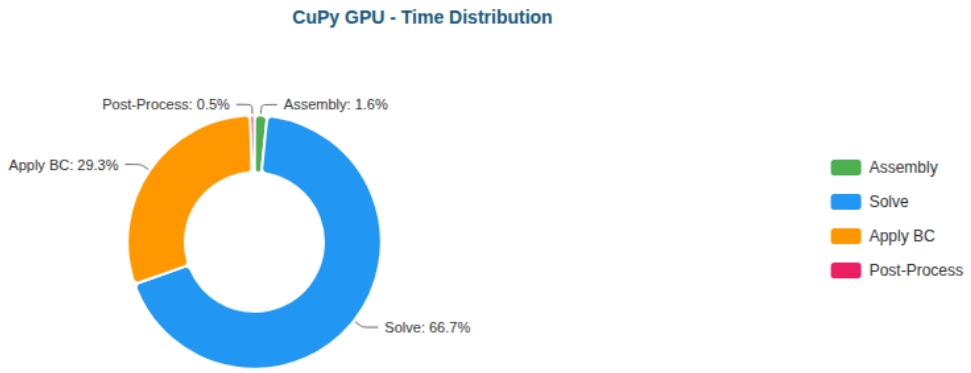
Venturi (XS) - 341 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (69%)	Post-Proc (19%)
CPU Threaded	Assembly (51%)	Post-Proc (24%)
CPU Multiprocess	Assembly (50%)	Post-Proc (49%)
Numba CPU	Solve (49%)	BC (21%)
Numba CUDA	Solve (87%)	Assembly (6%)
CuPy GPU	Solve (67%)	BC (29%)

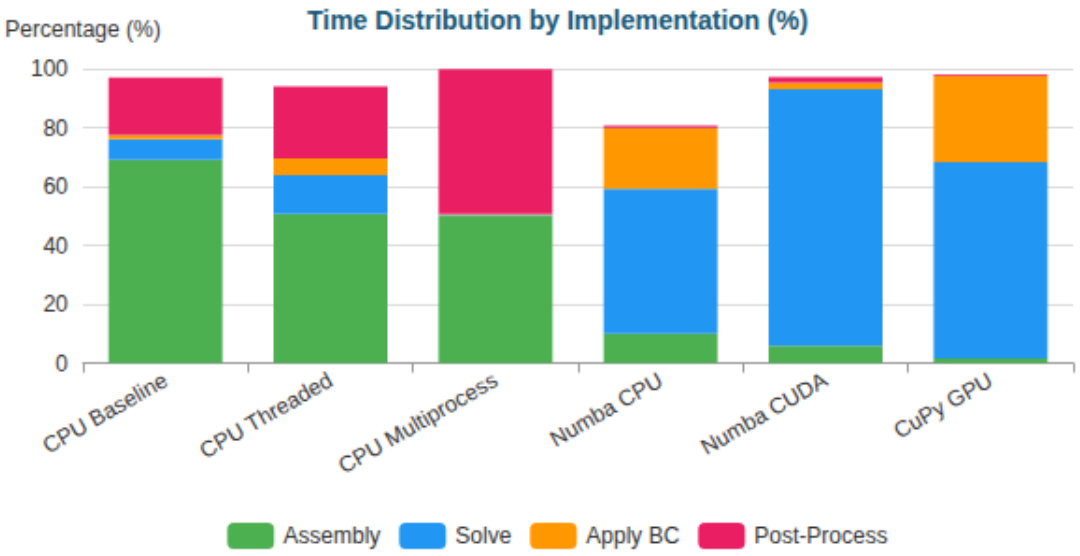
Time Distribution:



Pie



Pie

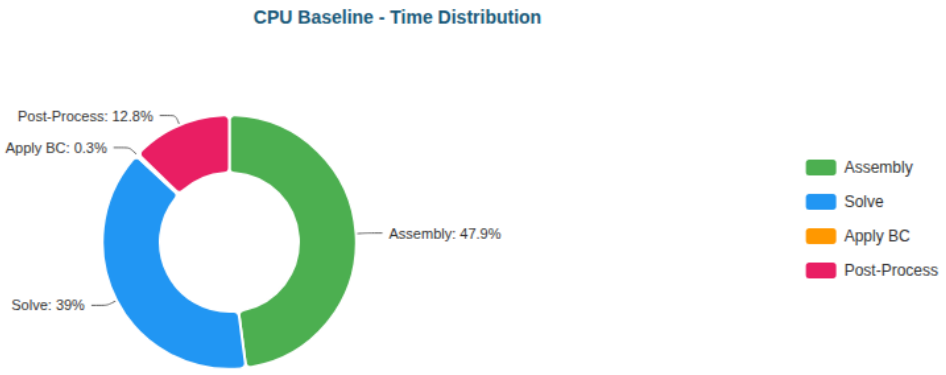


Stacked Bar

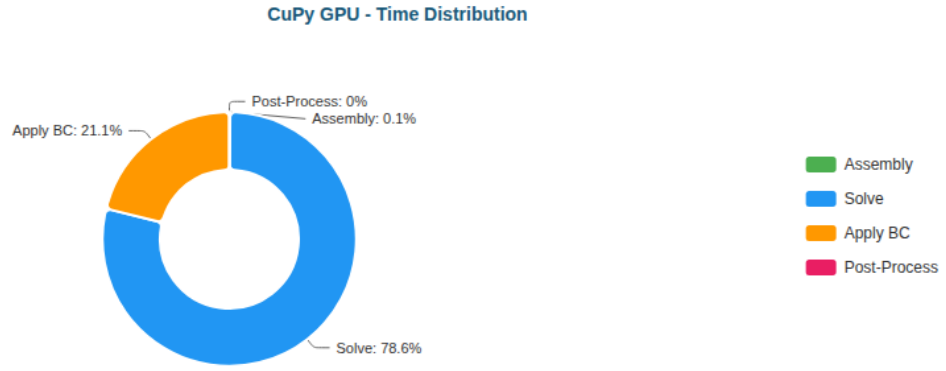
Venturi (M) - 194,325 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (48%)	Solve (39%)
CPU Threaded	Solve (55%)	Assembly (29%)
CPU Multiprocess	Solve (78%)	Assembly (12%)
Numba CPU	Solve (94%)	BC (5%)
Numba CUDA	Solve (49%)	Assembly (28%)
CuPy GPU	Solve (79%)	BC (21%)

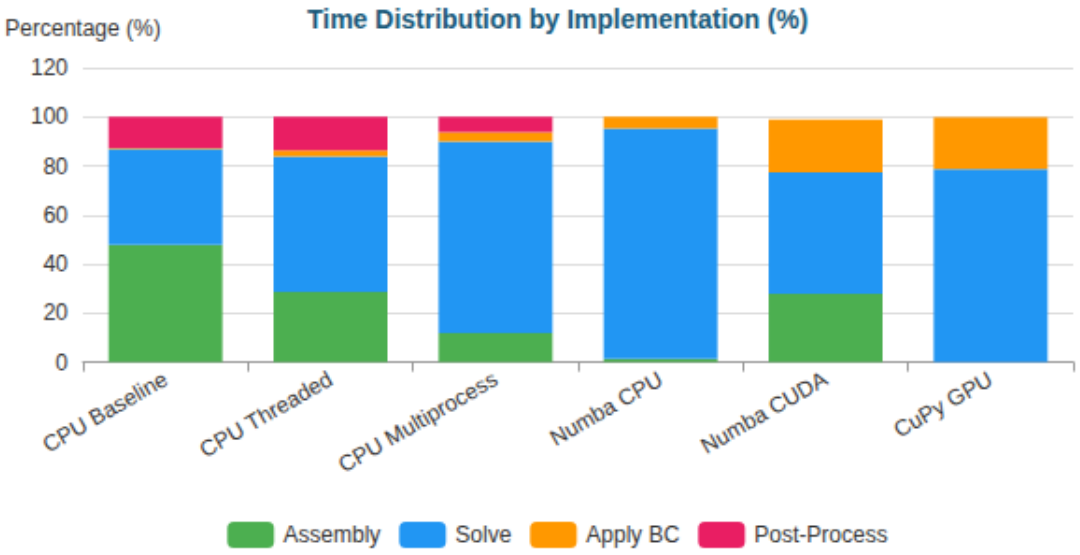
Time Distribution:



Pie



Pie



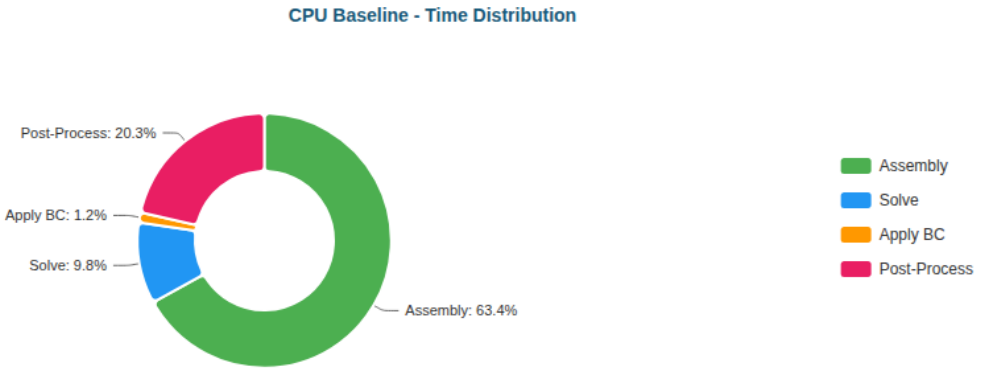
Stacked Bar

Y-Shaped (XS) - 201 nodes

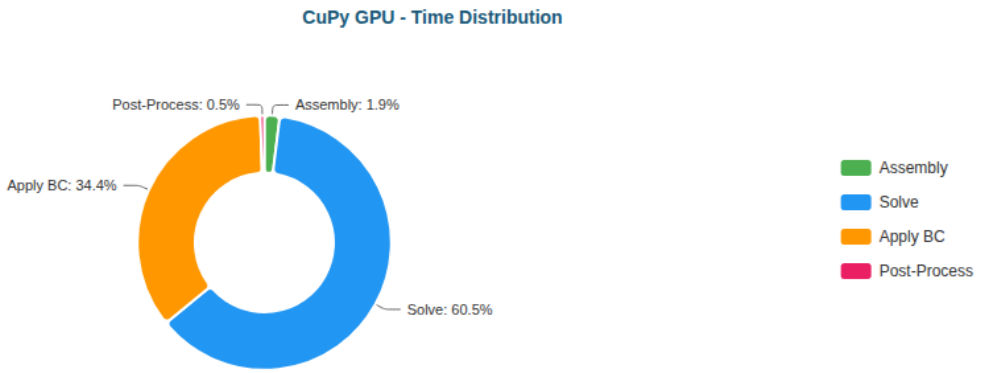
Implementation	Primary Bottleneck	Secondary Bottleneck

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (63%)	Post-Proc (20%)
CPU Threaded	Assembly (44%)	Post-Proc (26%)
CPU Multiprocess	Assembly (51%)	Post-Proc (49%)
Numba CPU	Solve (52%)	BC (15%)
Numba CUDA	Solve (86%)	Assembly (6%)
CuPy GPU	Solve (61%)	BC (34%)

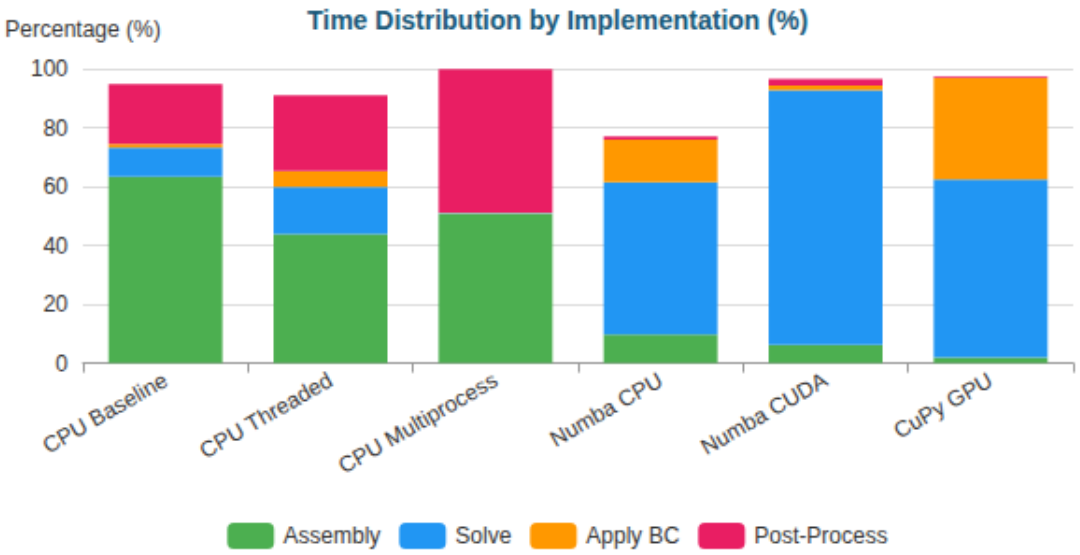
Time Distribution:



Pie



Pie

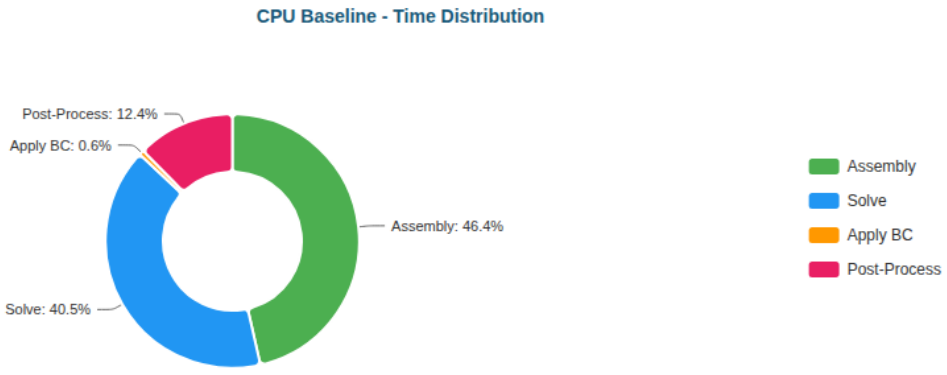


Stacked Bar

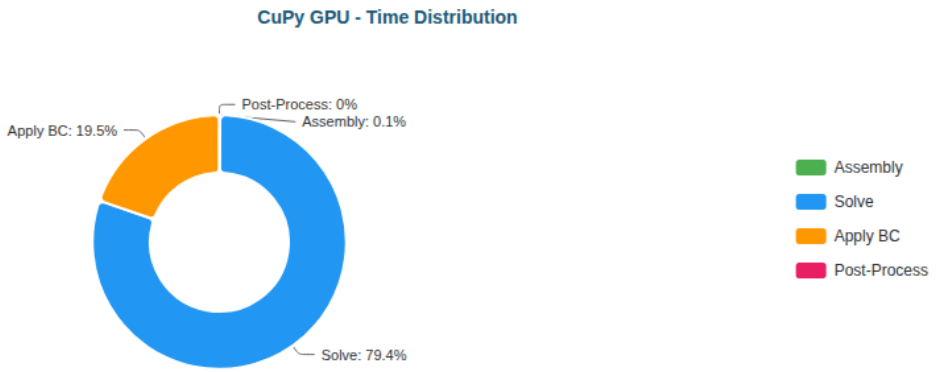
Y-Shaped (M) - 195,853 nodes

Implementation	Primary Bottleneck	Secondary Bottleneck
CPU Baseline	Assembly (46%)	Solve (41%)
CPU Threaded	Solve (59%)	Assembly (26%)
CPU Multiprocess	Solve (79%)	Assembly (10%)
Numba CPU	Solve (93%)	BC (6%)
Numba CUDA	Solve (48%)	BC (26%)
CuPy GPU	Solve (79%)	BC (20%)

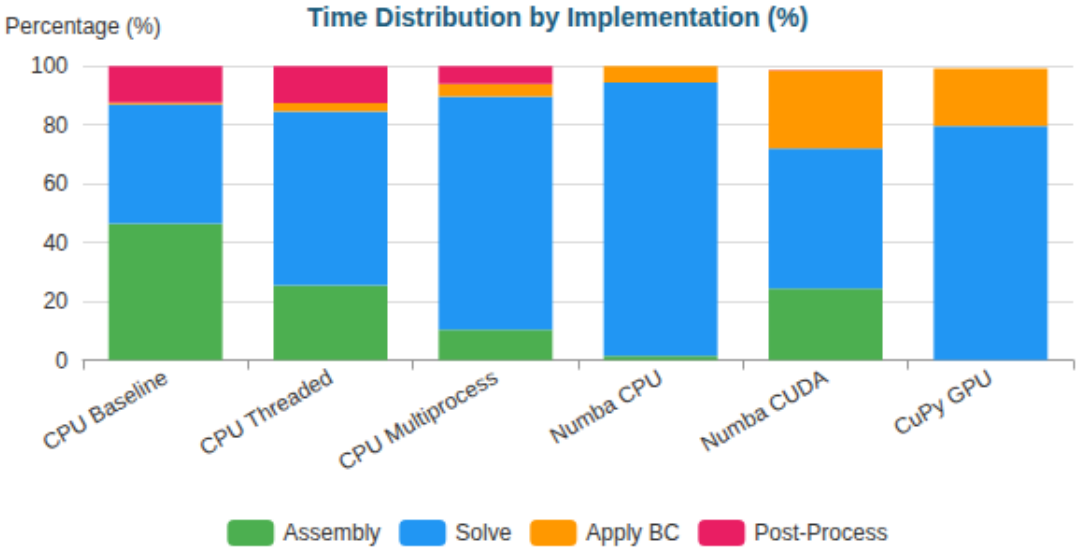
Time Distribution:



Pie



Pie



Why Each Optimization Helps

Transition	Reason
Baseline → Threaded	Limited by Python GIL; threads only help for I/O
Threaded → Multiprocess	Bypasses GIL via separate processes; IPC overhead limits gains
Multiprocess → Numba CPU	JIT compilation eliminates interpreter overhead; true parallel loops
Numba CPU → Numba CUDA	GPU parallelism: thousands of threads vs dozens of CPU cores
Numba CUDA → CuPy GPU	CUDA C kernels more optimized than Numba-generated PTX

4.8 Cross-Platform Comparative Analysis

This section consolidates the benchmark results presented in Sections 4.5–4.7 into a unified comparative analysis.

Rather than reiterating individual measurements, the focus here is on **interpreting performance trends**, **explaining architectural effects**, and **extracting general conclusions** regarding execution models and GPU classes.

4.8.1 CPU vs GPU: Where the Paradigm Shifts

Across all geometries and medium-to-large meshes, a clear and consistent transition point emerges:

- **Small meshes (XS)**
  GPU execution is systematically slower than optimized CPU variants due to:
  - kernel launch overhead,
  - PCIe latency,
  - underutilization of GPU parallelism.
- **Medium meshes (M)**
  GPU acceleration becomes dominant, with speedups ranging from:
  - **~11× (RTX 5060 Ti)**
  - **~20–40× (RTX 4090)**
  - **~30–60× (RTX 5090)**

This confirms that GPU acceleration is not universally beneficial, but **highly problem-size dependent**.

4.8.2 CPU Scaling Limits

The benchmark reveals well-defined limits for CPU-based optimization strategies.

CPU Strategy	Observed Benefit	Limiting Factor
Threading	1.2× – 2.1×	Python GIL
Multiprocessing	1.5× – 2.7×	IPC overhead
Numba JIT	2× – 6×	Memory bandwidth

Even with aggressive JIT compilation, **CPU performance saturates early**. For medium meshes, the solver becomes:

- **memory-bound**, and
- dominated by **sparse matrix–vector products**.

This explains why Numba CPU converges to similar performance as multiprocessing for large problems.

4.8.3 GPU Acceleration: Numba CUDA vs CuPy RawKernel

A consistent hierarchy is observed across all GPUs:

GPU Execution Model	Characteristics	Performance
Numba CUDA	Python-defined kernels, easier development	High
CuPy RawKernel	Native CUDA C, full control	Highest

Key observations:

- **CuPy GPU consistently outperforms Numba CUDA** for medium meshes.
- Gains range from **1.3× to 1.8×** over Numba CUDA.

- The advantage increases with:
  - mesh size,
  - solver dominance,
  - memory bandwidth pressure.

This confirms that **kernel maturity and low-level control matter** once GPU execution becomes solver-bound.

4.8.4 Cross-GPU Performance Scaling

A core objective of this benchmark was to separate **software scaling** from **hardware scaling**.

Aggregate Speedup (Medium Meshes)

GPU	Typical Speedup vs CPU Baseline
RTX 5060 Ti	11× – 18×
RTX 4090	20× – 42×
RTX 5090	25× – 60×

However, performance does **not** scale linearly with theoretical FLOPs.

Interpretation

- The FEM solver is **memory-bandwidth dominated**, not compute-bound.
- Higher-end GPUs benefit from:
  - larger L2 cache,
  - higher memory throughput,
  - better latency hiding.
- The RTX 5090 advantage is strongest for:
  - CG-heavy cases,
  - large sparse matrices,
  - solver-dominated workloads.

This confirms that **architectural balance**, not raw FLOPs, drives FEM performance.

4.8.5 Bottleneck Evolution Across Platforms

A central insight from the benchmark is the **systematic migration of bottlenecks**:

Execution Stage	CPU Baseline	Numba CPU	GPU (CuPy)
Assembly	Dominant	Minor	Negligible
Solve	Secondary	Dominant	Overwhelming
Apply BC	Minor	Minor	Non-negligible
Post-processing	Visible	Minimal	Negligible

Key implications:

- GPU acceleration **eliminates assembly as a bottleneck**.
- The **linear solver dominates runtime** in all optimized variants.
- On GPU, boundary condition application becomes visible due to:

  - atomic operations,
  - irregular memory access,
  - limited arithmetic intensity.

This validates the design decision to prioritize GPU-resident solvers.

4.8.6 Efficiency vs Absolute Performance

While the RTX 5090 delivers the highest absolute performance, efficiency considerations are relevant:

GPU	Relative Performance	Cost / Power Consideration
RTX 5060 Ti	Moderate	High efficiency per cost
RTX 4090	Very high	Balanced performance
RTX 5090	Extreme	Diminishing returns

For production environments, this suggests:

- **Mid-range GPUs** are sufficient for moderate FEM workloads.
- **High-end GPUs** are justified for:
  - very large meshes,
  - repeated simulations,
  - solver-dominated pipelines.

4.8.7 Robustness and Numerical Consistency

Crucially, acceleration does **not** alter numerical behavior:

- Identical CG iteration counts across platforms.
- Consistent residual norms at convergence.
- No divergence or fallback behavior observed.

This confirms that performance gains are achieved **without sacrificing numerical correctness**.

4.8.8 Consolidated Summary

Aspect	Key Conclusion
CPU optimization	Quickly saturates
GPU benefit	Strongly size-dependent
Best execution model	CuPy RawKernel
Dominant bottleneck	Sparse solver
Best scaling factor	Memory bandwidth
Best overall GPU	RTX 5090
Best cost-efficiency	RTX 5060 Ti

4.8.9 Final Insight

The benchmark demonstrates that **GPU acceleration fundamentally changes the performance landscape of FEM solvers**, but only when:

- the problem size is sufficiently large,
- data remains resident on the GPU,
- solver execution dominates the pipeline.

Beyond this point, performance becomes a function of **memory architecture rather than algorithmic complexity**, placing modern GPUs at a decisive advantage over CPUs for large-scale finite element simulations.

Conclusions

Key Findings

Conclusions based on mean times across 3 servers.

Backward-Facing Step (XS) - 287 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.5x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.8x speedup.
3. **JIT Compilation:** Numba CPU delivers 5.4x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 71% of total time.

Backward-Facing Step (M) - 195,362 nodes

1. **Maximum Speedup:** CuPy GPU achieves 25.1x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.3x speedup.
3. **JIT Compilation:** Numba CPU delivers 1.8x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 89% of total time.

Elbow 90° (XS) - 411 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.5x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.6x speedup.
3. **JIT Compilation:** Numba CPU delivers 6.1x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 69% of total time.

Elbow 90° (M) - 161,984 nodes

1. **Maximum Speedup:** CuPy GPU achieves 26.2x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.6x speedup.
3. **JIT Compilation:** Numba CPU delivers 1.7x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 83% of total time.

S-Bend (XS) - 387 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.4x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.9x speedup.
3. **JIT Compilation:** Numba CPU delivers 4.7x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 69% of total time.

S-Bend (M) - 196,078 nodes

1. **Maximum Speedup:** CuPy GPU achieves 20.7x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.2x speedup.
3. **JIT Compilation:** Numba CPU delivers 1.5x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 88% of total time.

T-Junction (XS) - 393 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.5x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 2.0x speedup.
3. **JIT Compilation:** Numba CPU delivers 4.6x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 71% of total time.

T-Junction (M) - 196,420 nodes

1. **Maximum Speedup:** CuPy GPU achieves 30.4x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.7x speedup.
3. **JIT Compilation:** Numba CPU delivers 2.4x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 87% of total time.

Venturi (XS) - 341 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.4x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.9x speedup.
3. **JIT Compilation:** Numba CPU delivers 6.0x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 64% of total time.

Venturi (M) - 194,325 nodes

1. **Maximum Speedup:** CuPy GPU achieves 20.6x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.2x speedup.

3. **JIT Compilation:** Numba CPU delivers 1.7x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 80% of total time.

Y-Shaped (XS) - 201 nodes

1. **Maximum Speedup:** CuPy GPU achieves 0.4x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.9x speedup.
3. **JIT Compilation:** Numba CPU delivers 4.1x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 61% of total time.

Y-Shaped (M) - 195,853 nodes

1. **Maximum Speedup:** CuPy GPU achieves 17.3x speedup over CPU Baseline.
2. **Threading Effect:** CPU Threaded shows 1.2x speedup.
3. **JIT Compilation:** Numba CPU delivers 1.6x speedup by eliminating interpreter overhead.
4. **GPU Bottleneck:** On GPU, the iterative solver consumes 80% of total time.

Recommendations

Use Case	Recommended Implementation
Development/debugging	CPU Baseline or Numba CPU
Production (no GPU)	Numba CPU
Production (with GPU)	CuPy GPU
Small meshes (<10K nodes)	Numba CPU (GPU overhead not worthwhile)
Large meshes (>100K nodes)	CuPy GPU