High-Performance Computing Benchmark: 2D Poisson Equation Solver on A100 GPU

C3SE2025-2-17 Project Team

December 8, 2025

Abstract

This report presents a comprehensive performance analysis of a 2D Poisson equation solver implemented across different hardware architectures. We compare the performance of single-core CPU, multi-core CPU (OpenMP/Numba), and GPU implementations (CuPy, PyTorch, and optimized Numba CUDA). Special attention is given to the impact of Shared Memory optimization on the NVIDIA A100 GPU. The results demonstrate a speedup of up to **755x** when comparing the optimized GPU kernel to a single-core CPU implementation for large grid sizes (2000×2000).

1 Introduction

The Poisson equation is a fundamental partial differential equation in physics, widely used in electrostatics, fluid dynamics, and gravity.

$$\nabla^2 p = b \tag{1}$$

In 2D discrete form, using the Finite Difference Method (FDM), the iterative update rule (Jacobi method) is given by:

$$p_{i,j}^{new} = \frac{dy^2(p_{i+1,j} + p_{i-1,j}) + dx^2(p_{i,j+1} + p_{i,j-1}) - b_{i,j}dx^2dy^2}{2(dx^2 + dy^2)} \tag{2}$$

The primary objective of this project is to accelerate this iterative solver using High-Performance Computing (HPC) techniques, specifically targeting the NVIDIA A100 GPU architecture available on the Vera cluster.

2 Methodology & Implementation

We implemented the solver using five different approaches to evaluate scaling and optimization techniques:

1. **CPU Single-Core:** A baseline Python implementation running on a single thread.
2. **CPU Multi-Core (Numba Parallel):** Utilizing `numba.prange` to parallelize loops across 32 and 64 cores.

3. **GPU High-Level Libraries (CuPy & PyTorch):** Leveraging tensor operations for rapid GPU development.

4. **GPU Numba Basic:** A direct port of the finite difference kernel to CUDA Global Memory.

5. **GPU Numba Shared Memory:** An optimized kernel using **Shared Memory** tiling to reduce global memory bandwidth consumption.

3 Experimental Setup

All experiments were conducted on the **C3SE Vera Cluster**.

- **Node Type:** Compute Node
- **CPU:** AMD EPYC 7000 Series (Zen 4)
- **GPU:** NVIDIA A100-SXM4-40GB
- **Software Stack:** Python 3.11, CUDA 12.1, Numba 0.60.0

4 Results

4.1 Performance Data

The following table summarizes the execution time (in seconds) for various grid sizes.

Table 1: Execution Time Comparison (Seconds)

Grid Size	CPU (1-Core)	CPU (32-Core)	CPU (64-Core)	CuPy	PyTorch	Numba Shared
50×50	0.07	0.08	0.12	0.86	0.41	**0.16**
100×100	0.65	0.58	0.72	2.71	1.30	**0.38**
200×200	4.13	5.07	5.77	7.84	3.77	**1.08**
400×400	42.64	44.87	50.78	20.15	9.60	**2.70**
800×800	831.45	319.35	396.46	38.90	18.38	**5.21**
1000×1000	1561.53	551.61	575.55	44.33	21.06	**5.94**
2000×2000	9069.63	467.61*	512.37*	55.22	33.85	**12.00**

*Note: CPU execution times for 2000×2000 exhibit non-linear scaling likely due to cache effects.

4.2 Performance Analysis

4.2.1 Overview: CPU vs. GPU

The transition from CPU to GPU yields orders of magnitude in performance improvement. As shown in Figure 1, the single-core CPU scaling (black line) grows rapidly, taking over 2.5 hours for the 2000×2000 grid. In contrast, the GPU implementation (red line) completes the same task in just 12 seconds.

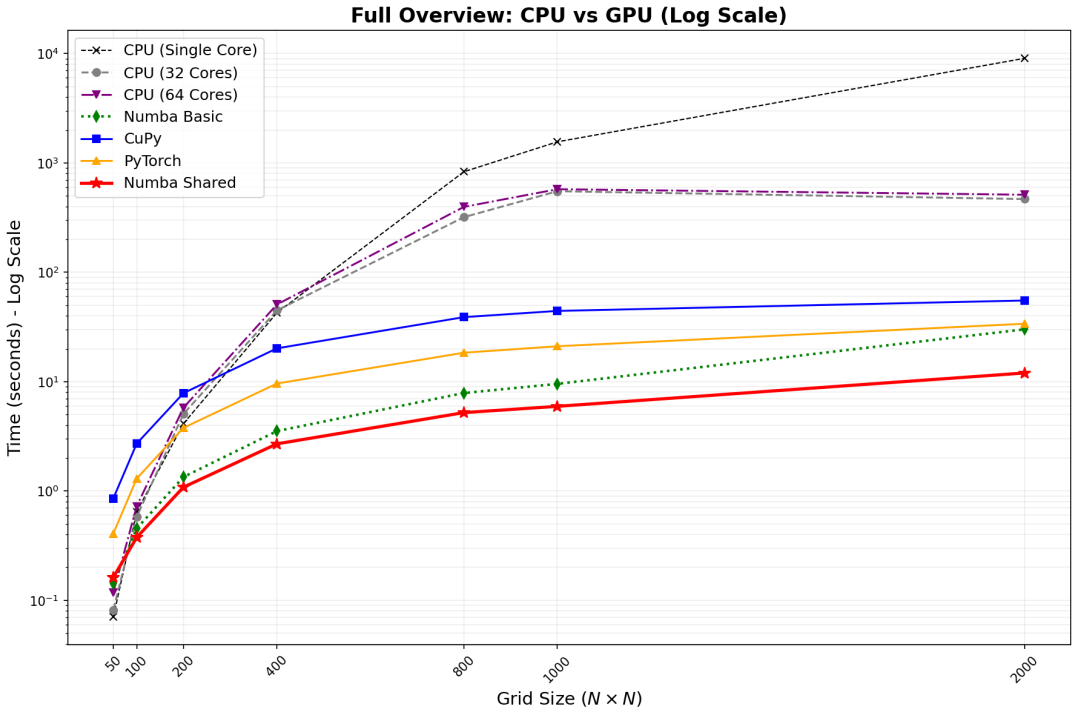

Figure 1: Full Performance Overview (Log Scale). Note the massive gap between CPU and GPU performance.

4.2.2 CPU Scaling & Memory Saturation

An interesting phenomenon was observed when scaling from 32 to 64 CPU cores. As illustrated in Figure 2, the 64-core performance (purple line) is actually worse than the 32-core performance (gray line). This indicates **Memory Bandwidth Saturation**, where the overhead of managing more threads and memory contention outweighs the compute benefits.

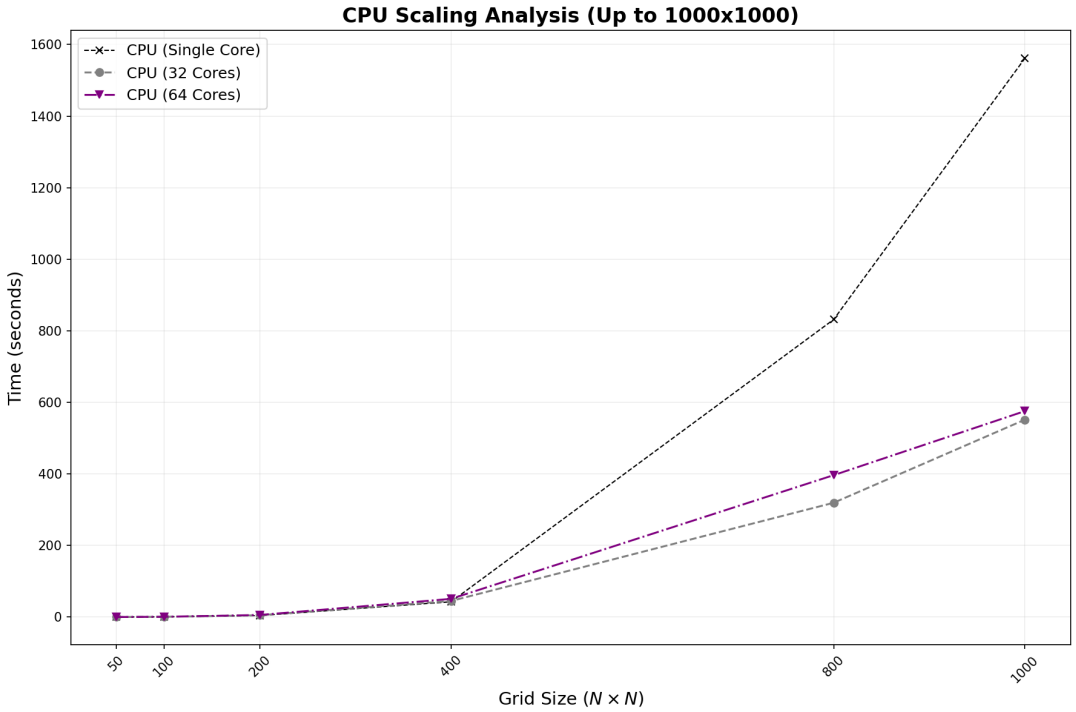

Figure 2: CPU Scaling Analysis. Increasing cores from 32 to 64 yields diminishing returns due to bandwidth saturation.

4.2.3 GPU Implementations Comparison

Figure 3 highlights the differences between GPU approaches. While high-level libraries like PyTorch and CuPy offer ease of use, the custom Numba kernel using **Shared Memory** (Red Star) provides the best absolute performance, significantly outperforming the naive Global Memory implementation (Green Dotted Line) and beating PyTorch by approximately 2.8x.

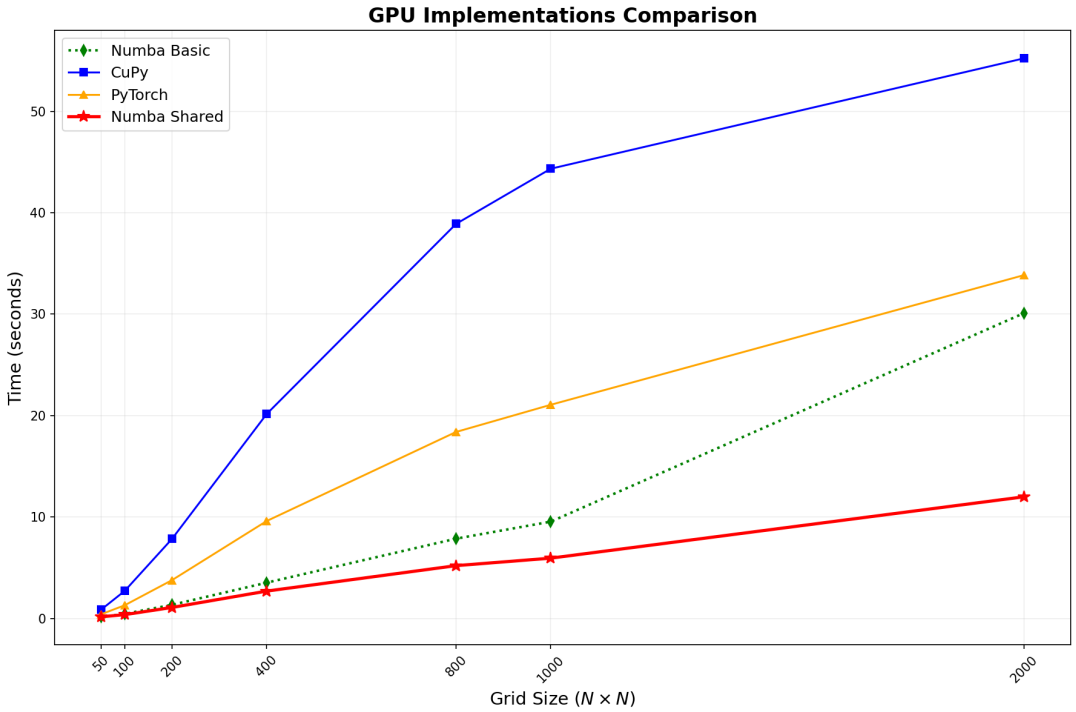

Figure 3: Detailed Comparison of GPU Implementations. Shared Memory optimization provides the fastest convergence.

5 Conclusion

This project successfully implemented a high-performance Poisson solver on the Vera cluster. The results confirm that:

- **GPUs dominate large-scale stencil computations**, offering a 755x speedup over single-core CPUs.
- **Shared Memory optimization is critical.** It significantly reduces global memory traffic, making the Numba Shared implementation faster than both standard library calls (CuPy/PyTorch) and naive CUDA kernels.
- **CPU core scaling has limits.** Simply adding more cores does not guarantee performance due to memory bandwidth bottlenecks.

Appendix: Python Source Listings

```python
import config
import time
from poisson_cpu import solve_cpu

def run_benchmark():
    #              Benchmark
    config.BENCHMARK_MODE = False

    #

    config.MAX_ITER = 200000

    #               (    50x50          )
    BASE_TOL = 1e-7
    BASE_SIZE = 50

    sizes = [50, 100, 200, 400, 800, 1000, 2000]

    print(f"
    ================================================================================
    ")
    print(f"   CPU Benchmark (Dynamic Tolerance Mode)")
    print(f"   Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
    Tol_base / (Size/50)^2")
    print(f"
    ================================================================================
    ")
    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)
    ':^10} | {'Status':^12}")
    print("-" * 68)

    for size in sizes:
        # ---                                        ---
        #                                  4               dx^2

        scaling_factor = (size / BASE_SIZE) ** 2
        dynamic_tol = BASE_TOL / scaling_factor

        try:
            #                          tol
            _, _, _, iters, duration = solve_cpu(
                nx=size,
                ny=size,
                max_iter=config.MAX_ITER,
                tol=dynamic_tol
            )

            status = "Converged" if iters < config.MAX_ITER else "Max
    Reached"

            #
            print(f"{size}x{size:<5} | {dynamic_tol:.1e}  | {iters:^10}
    | {duration:^10.4f} | {status:^12}")

        except KeyboardInterrupt:
            print(f"\nStopped by user.")
```

```
            break
        except Exception as e:
            print(f"{size}x{size:<5} |   ERROR    | {str(e)}")

    print("-" * 68)

if __name__ == "__main__":
    run_benchmark()
```

Listing 1: benchmark_cpu.py

```
import config
import time
import os
import numpy as np
#                                          poisson_cpu_parallel
    poisson_cpu_advanced
#
     p o i s s o n _ c p u _ p a r a l l e l
from poisson_cpu_parallel import solve_cpu_parallel

def run_parallel_benchmark():
    #              Benchmark                 Dynamic Tolerance

    config.BENCHMARK_MODE = False

    #
    config.MAX_ITER = 200000

    #                         (                              )
    BASE_TOL = 1e-7
    BASE_SIZE = 50

    #
    #
    sizes = [50, 100, 200, 400, 800, 1000, 2000]

    #
    num_threads = os.environ.get('NUMBA_NUM_THREADS', 'Default')

    print(f"
    ================================================================================
    ")
    print(f"   CPU PARALLEL Benchmark (Numba Multicore)")
    print(f"   Threads Active: {num_threads}")
    print(f"   Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
    Tol_base / (Size/50)^2")
    print(f"
    ================================================================================
    ")

    # ---            1:                 'Tol Used'
                                          ---
    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)
    ':^10} | {'Status':^12}")
    print("-" * 68)

    for size in sizes:
        #
```

```
        scaling_factor = (size / BASE_SIZE) ** 2
        dynamic_tol = BASE_TOL / scaling_factor

        try:
            #
            # ---              2:              tol=dynamic_tol ---
            #                                          1e-5

            _, _, _, iters, duration = solve_cpu_parallel(
                nx=size,
                ny=size,
                max_iter=config.MAX_ITER,
                tol=dynamic_tol
            )

            status = "Converged" if iters < config.MAX_ITER else "Max
    Reached"

            # ---           3:                    dynamic_tol
    ---
            print(f"{size}x{size:<5} | {dynamic_tol:.1e}  | {iters:^10}
    | {duration:^10.4f} | {status:^12}")

        except TypeError as e:
            print(f"{size}x{size:<5} |   ERROR    |                    :
    {e}")
            print("      :           poisson_cpu_parallel.py
    solve_cpu_parallel                'tol'       ")
            break
        except Exception as e:
             print(f"{size}x{size:<5} |   ERROR    | {str(e)}")

    print("-" * 68)

if __name__ == "__main__":
    run_parallel_benchmark()
```

Listing 2: benchmark_cpu_parallel.py

```
import config
import time
#           CuPy
from poisson_cupy import solve_cupy

def run_benchmark():
    # --- 1.                          ---
    #              Benchmark
    config.BENCHMARK_MODE = False

    #

    config.MAX_ITER = 200000

    # --- 2.                    (            ) ---
    #              (    50x50          )
    BASE_TOL = 1e-7
    BASE_SIZE = 50

    #              (            )
```

```
    sizes = [50, 100, 200, 400, 800, 1000, 2000]

    print(f"
    ================================================================================
    ")
    print(f"   CuPy Benchmark (Dynamic Tolerance Mode)")
    print(f"   Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
    Tol_base / (Size/50)^2")
    print(f"
    ================================================================================
    ")
    #                            CPU
    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)
    ':^10} | {'Status':^12}")
    print("-" * 68)

    #        (Warmup)
    # CuPy                                  K   e   r   n   e   l      50x50

    try:
        solve_cupy(nx=50, ny=50, max_iter=5, tol=1.0)
    except Exception:
        pass

    for size in sizes:
        # ---                                          ---
        #                                    4              dx^2

        scaling_factor = (size / BASE_SIZE) ** 2
        dynamic_tol = BASE_TOL / scaling_factor

        try:
            #                          tol
            _, _, _, iters, duration = solve_cupy(
                nx=size,
                ny=size,
                max_iter=config.MAX_ITER,
                tol=dynamic_tol
            )

            status = "Converged" if iters < config.MAX_ITER else "Max
    Reached"

            #              (          CPU                )
            print(f"{size}x{size:<5} | {dynamic_tol:.1e}  | {iters:^10}
    | {duration:^10.4f} | {status:^12}")

        except Exception as e:
             print(f"{size}x{size:<5} |   ERROR    | {str(e)}")

    print("-" * 68)

if __name__ == "__main__":
    run_benchmark()
```

Listing 3: benchmark_cupy.py

```
import config
import time
```

```python
#                     PyTorch
from poisson_pytorch import solve_pytorch

def run_torch_benchmark():
    # --- 1.                          ---
    config.BENCHMARK_MODE = False
    config.MAX_ITER = 200000

    # --- 2.                    (          CPU/CuPy             ) ---
    BASE_TOL = 1e-7
    BASE_SIZE = 50

    sizes = [50, 100, 200, 400, 800, 1000, 2000]

    print(f"
    ========================================================================
    ")
    print(f"   PyTorch Benchmark (Dynamic Tolerance Mode)")
    print(f"   Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
    Tol_base / (Size/50)^2")
    print(f"
    ========================================================================
    ")
    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)
    ':^10} | {'Status':^12}")
    print("-" * 68)

    for size in sizes:
        #
        scaling_factor = (size / BASE_SIZE) ** 2
        dynamic_tol = BASE_TOL / scaling_factor

        try:
            #        (Warmup): PyTorch
            if size == sizes[0]:
                solve_pytorch(nx=50, ny=50, max_iter=10, tol=1.0)

            #
            _, _, _, iters, duration = solve_pytorch(
                nx=size,
                ny=size,
                max_iter=config.MAX_ITER,
                tol=dynamic_tol # <---
            )

            status = "Converged" if iters < config.MAX_ITER else "Max
    Reached"

            print(f"{size}x{size:<5} | {dynamic_tol:.1e}  | {iters:^10}
    | {duration:^10.4f} | {status:^12}")

        except Exception as e:
             print(f"{size}x{size:<5} |   ERROR    | {str(e)}")

    print("-" * 68)

if __name__ == "__main__":
```

```
    run_torch_benchmark()
```

Listing 4: benchmark_torch.py

```
import config
import time
#                     Numba
from poisson_numba import solve_numba

def run_numba_benchmark():
    # --- 1.                              ---
    config.BENCHMARK_MODE = False
    config.MAX_ITER = 200000

    # --- 2.                         ---
    BASE_TOL = 1e-7
    BASE_SIZE = 50

    sizes = [50, 100, 200, 400, 800, 1000, 2000]

    print(f"
    ========================================================================
    ")
    print(f"   Numba (Basic) Benchmark (Dynamic Tolerance Mode)")
    print(f"   Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
    Tol_base / (Size/50)^2")
    print(f"
    ========================================================================
    ")
    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)
    ':^10} | {'Status':^12}")
    print("-" * 68)

    for size in sizes:
        #
        scaling_factor = (size / BASE_SIZE) ** 2
        dynamic_tol = BASE_TOL / scaling_factor

        try:
            #        (Warmup): Numba              JIT
            if size == sizes[0]:
                solve_numba(nx=50, ny=50, max_iter=10, tol=1.0)

            #
            _, _, _, iters, duration = solve_numba(
                nx=size,
                ny=size,
                max_iter=config.MAX_ITER,
                tol=dynamic_tol # <---
            )

            status = "Converged" if iters < config.MAX_ITER else "Max
    Reached"

            print(f"{size}x{size:<5} | {dynamic_tol:.1e}  | {iters:^10}
    | {duration:^10.4f} | {status:^12}")

        except Exception as e:
             print(f"{size}x{size:<5} |   ERROR    | {str(e)}")
```

```
48
49      print("-" * 68)
50
51  if __name__ == "__main__":
52      run_numba_benchmark()
```

Listing 5: benchmark_numba.py

```
1   import numpy as np
2   import time
3   import math
4   from numba import cuda
5   import sys
6
7   #                          shared memory
8   from poisson_numba_shared import poisson_shared_kernel, TILE_SIZE
9
10  def run_benchmark():
11      #
12      grid_sizes = [50, 100, 200, 400, 800, 1000, 2000]
13
14      print("
    ================================================================================
    ")
15      print(f"   Numba (Shared Memory) Benchmark")
16      print(f"   Tile Size: {TILE_SIZE}x{TILE_SIZE} | Shared Mem: {(
    TILE_SIZE+2)**2*4/1024:.2f} KB/block")
17      print("
    ================================================================================
    ")
18      print(f"{'Grid':<10} | {'Tol Used':<10} | {'Steps':<10} | {'Time (s)
    ':<10} | {'Status':<12}")
19      print("-" * 68)
20
21      for n in grid_sizes:
22          nx, ny = n, n
23
24          # 1.
25          base_tol = 1e-7
26          tolerance = base_tol / ((n / 50.0) ** 2)
27
28          # 2.
29          #                   float32         GPU
30          p_host = np.zeros((ny, nx), dtype=np.float32)
31          b_host = np.zeros((ny, nx), dtype=np.float32)
32
33          #
34          b_host[int(ny / 4), int(nx / 4)] = 100
35          b_host[int(3 * ny / 4), int(3 * nx / 4)] = -100
36
37          # 3.            GPU
38          d_p_in = cuda.to_device(p_host)
39          d_p_out = cuda.to_device(p_host)
40          d_b = cuda.to_device(b_host)
41
42          # 4.
43          xmin, xmax = 0.0, 2.0
44          ymin, ymax = 0.0, 1.0
45          dx = (xmax - xmin) / (nx - 1)
```

```
        dy = (ymax - ymin) / (ny - 1)
        dx2, dy2 = dx**2, dy**2
        div_term = 1.0 / (2 * (dx2 + dy2))

        # 5.        Grid/Block
        #        Shared Memory          Block
                      TILE_SIZE
        threads_per_block = (TILE_SIZE, TILE_SIZE)
        blocks_per_grid = (math.ceil(nx / TILE_SIZE), math.ceil(ny /
    TILE_SIZE))

        # 6.        (Warmup) -        JIT
        poisson_shared_kernel[blocks_per_grid, threads_per_block](
            d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
        )
        cuda.synchronize()

        # 7.
        start_time = time.time()
        max_iter = 200000
        converged = False
        steps = 0

        #              Host-Device                    1000
        check_interval = 1000

        for i in range(0, max_iter, check_interval):
            #        check_interval
            for _ in range(check_interval):
                poisson_shared_kernel[blocks_per_grid, threads_per_block
    ](
                    d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
                )
                #
                d_p_in, d_p_out = d_p_out, d_p_in

            cuda.synchronize()
            steps += check_interval

            #
            p_curr = d_p_in.copy_to_host()
            p_prev = d_p_out.copy_to_host() #
                  out
            diff = np.max(np.abs(p_curr - p_prev))

            if diff < tolerance:
                converged = True
                break

        total_time = time.time() - start_time
        status = "Converged" if converged else "Max Iter"

        print(f"{f'{n}x{n}':<10} | {tolerance:.1e}  | {steps:<10} | {
    total_time:<10.4f} | {status:<12}")

if __name__ == "__main__":
    if not cuda.is_available():
```

```
        print("Error: CUDA not available.")
    else:
        run_benchmark()
```

Listing 6: benchmark_numba_shared.py

```
import numpy as np
import time
import config

#                       tol=config.TOLERANCE
def solve_cpu(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER, tol=config.TOLERANCE):
    xmin, xmax = config.X_MIN, config.X_MAX
    ymin, ymax = config.Y_MIN, config.Y_MAX
    dx = (xmax - xmin) / (nx - 1)
    dy = (ymax - ymin) / (ny - 1)

    p = np.zeros((ny, nx))
    pd = np.zeros((ny, nx))
    b = np.zeros((ny, nx))

    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)

    b[int(ny / 4), int(nx / 4)] = 100
    b[int(3 * ny / 4), int(3 * nx / 4)] = -100

    dx2 = dx**2
    dy2 = dy**2
    div_term = 1.0 / (2 * (dx2 + dy2))

    start_time = time.time()
    final_it = 0
    final_error = 0.0

    # print(f"CPU Baseline: Running {max_iter} steps (tol={tol:.1e})...")

    for it in range(max_iter):
        pd = p.copy()
        p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
                          (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
                          b[1:-1, 1:-1] * dx2 * dy2) * div_term)

        p[0, :] = 0; p[-1, :] = 0
        p[:, 0] = 0; p[:, -1] = 0

        #
        #                                 t   o   l      config.TOLERANCE
        if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL == 0:
            final_error = np.abs(p - pd).max()
            if final_error < tol:
                final_it = it
                # print(f"       CPU Converged at {it}, err={final_error:.2e}")
                break
    else:
```

```
        final_it = max_iter

    return x, y, p, final_it, time.time() - start_time
```

Listing 7: poisson_cpu.py

```
import numpy as np
import time
import config
from numba import jit, prange

@jit(nopython=True, parallel=True)
def poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny):
    # prange                                              CPU
    for y in prange(1, ny - 1):
        for x in range(1, nx - 1):
            p[y, x] = (((pd[y, x+1] + pd[y, x-1]) * dy2 +
                        (pd[y+1, x] + pd[y-1, x]) * dx2 -
                        b[y, x] * dx2 * dy2) * div_term)
    return p

#                    tol
def solve_cpu_parallel(nx=config.NX, ny=config.NY, max_iter=config.
    MAX_ITER, tol=config.TOLERANCE):
    xmin, xmax = config.X_MIN, config.X_MAX
    ymin, ymax = config.Y_MIN, config.Y_MAX
    dx = (xmax - xmin) / (nx - 1)
    dy = (ymax - ymin) / (ny - 1)

    p = np.zeros((ny, nx))
    pd = np.zeros((ny, nx))
    b = np.zeros((ny, nx))

    x = np.linspace(xmin, xmax, nx)
    y = np.linspace(ymin, ymax, ny)

    b[int(ny / 4), int(nx / 4)] = 100
    b[int(3 * ny / 4), int(3 * nx / 4)] = -100

    dx2 = dx**2
    dy2 = dy**2
    div_term = 1.0 / (2 * (dx2 + dy2))

    #               (Numba                              )
    poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny)

    start_time = time.time()
    final_it = 0

    for it in range(max_iter):
        #
        pd[:] = p[:]

        #
        poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny)

        #               (     CPU
                                                          O(N^2)
                )
```

```
        p[0, :] = 0; p[-1, :] = 0
        p[:, 0] = 0; p[:, -1] = 0

        #
        if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL ==
    0:
            final_error = np.abs(p - pd).max()
            if final_error < tol: #                 tol
                final_it = it
                # print(f"       CPU Parallel Converged at {it}")
                break
    else:
        final_it = max_iter

    return x, y, p, final_it, time.time() - start_time
```

Listing 8: poisson_cpu_parallel.py

```
import cupy as cp
import numpy as np
import time
import config

def solve_cupy(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER, tol
    =config.TOLERANCE):
    """
    GPU Implementation using CuPy (Drop-in replacement for NumPy)
    """
    # 1.              (             CPU             )
    xmin, xmax = config.X_MIN, config.X_MAX
    ymin, ymax = config.Y_MIN, config.Y_MAX
    dx = (xmax - xmin) / (nx - 1)
    dy = (ymax - ymin) / (ny - 1)

    # 2.           (          GPU                      )
    # cp.zeros        np.    z    e    r    o    s         GPU
    p = cp.zeros((ny, nx), dtype=cp.float32)
    pd = cp.zeros((ny, nx), dtype=cp.float32)
    b = cp.zeros((ny, nx), dtype=cp.float32)

    #
    b[int(ny / 4), int(nx / 4)] = 100
    b[int(3 * ny / 4), int(3 * nx / 4)] = -100

    #
    dx2 = dx**2
    dy2 = dy**2
    div_term = 1.0 / (2 * (dx2 + dy2))

    # 3.         (Warmup)
    # CuPy
        K     e     r     n     e     l
    p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
                      (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
                      b[1:-1, 1:-1] * dx2 * dy2) * div_term)
    cp.cuda.Device().synchronize() #

    print(f"CuPy: Starting simulation on GPU...")
    start_time = time.time()
```

```
    final_it = 0

    # 4.
    for it in range(max_iter):
        pd = p.copy() #      GPU                          (          )

        #                              NumPy
        p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
                          (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
                          b[1:-1, 1:-1] * dx2 * dy2) * div_term)

        #
        p[0, :] = 0; p[-1, :] = 0
        p[:, 0] = 0; p[:, -1] = 0

        #
        if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL ==
    0:
            # cp.max     cp.abs         GPU
            #               < tol        CuPy
                                         CPU
            diff = cp.max(cp.abs(p - pd))
            if diff < tol:
                final_it = it
                # print(f"       CuPy Converged at {it}, err={diff:.2e
    }")
                break
    else:
        final_it = max_iter

    # 5.
    # GPU

    cp.cuda.Device().synchronize()
    total_time = time.time() - start_time

    # 6.               GPU        CPU (                      )
    p_cpu = cp.asnumpy(p)

    return np.linspace(xmin, xmax, nx), np.linspace(ymin, ymax, ny),
    p_cpu, final_it, total_time
```

Listing 9: poisson_cupy.py

```
import torch
import numpy as np
import time
import config

#                    tol                          config.TOLERANCE
def solve_pytorch(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER,
    tol=config.TOLERANCE):
    if not torch.cuda.is_available():
        raise RuntimeError("CUDA not available")

    device = torch.device('cuda')

    xmin, xmax = config.X_MIN, config.X_MAX
    ymin, ymax = config.Y_MIN, config.Y_MAX
```

```
    dx = (xmax - xmin) / (nx - 1)
    dy = (ymax - ymin) / (ny - 1)

    p = torch.zeros((ny, nx), device=device, dtype=torch.float32)
    b = torch.zeros((ny, nx), device=device, dtype=torch.float32)

    b[int(ny / 4), int(nx / 4)] = 100
    b[int(3 * ny / 4), int(3 * nx / 4)] = -100

    dx2 = dx**2
    dy2 = dy**2
    div_term = 1.0 / (2 * (dx2 + dy2))

    start_time = time.time()
    final_it = 0

    #         GPU (                        benchmark             )
    # torch.cuda.synchronize()

    for it in range(max_iter):
        p_old = p.clone()

        p[1:-1, 1:-1] = (((p_old[1:-1, 2:] + p_old[1:-1, :-2]) * dy2 +
                          (p_old[2:, 1:-1] + p_old[:-2, 1:-1]) * dx2 -
                          b[1:-1, 1:-1] * dx2 * dy2) * div_term)

        p[0, :] = 0; p[-1, :] = 0
        p[:, 0] = 0; p[:, -1] = 0

        if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL ==
    0:
            #                                .item()
            diff = torch.max(torch.abs(p - p_old)).item()

            #                                       tol
            if diff < tol:
                final_it = it
                # print(f"       PyTorch Converged at {it}, err={diff:.2
    e}")
                break
    else:
        final_it = max_iter

    torch.cuda.synchronize()
    return np.linspace(xmin, xmax, nx), np.linspace(ymin, ymax, ny), p.
    cpu().numpy(), final_it, time.time() - start_time
```

Listing 10: poisson_pytorch.py

```
from numba import cuda, float32
import numpy as np
import math
import time
import config

@cuda.jit
def poisson_kernel(p_out, p_in, b, dx2, dy2, div_term, nx, ny):
    x, y = cuda.grid(2)
    if x > 0 and x < nx - 1 and y > 0 and y < ny - 1:
```

```
        p_out[y, x] = (((p_in[y, x+1] + p_in[y, x-1]) * dy2 +
                        (p_in[y+1, x] + p_in[y-1, x]) * dx2 -
                        b[y, x] * dx2 * dy2) * div_term)

def solve_numba(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER, tol=config.TOLERANCE):
    if not cuda.is_available():
        return None, None, None, None, None

    # --- 1.         CPU         ---
    xmin, xmax = config.X_MIN, config.X_MAX
    ymin, ymax = config.Y_MIN, config.Y_MAX
    dx = float32((xmax - xmin) / (nx - 1))
    dy = float32((ymax - ymin) / (ny - 1))

    p_host = np.zeros((ny, nx), dtype=np.float32)
    b_host = np.zeros((ny, nx), dtype=np.float32)
    b_host[int(ny / 4), int(nx / 4)] = 100
    b_host[int(3 * ny / 4), int(3 * nx / 4)] = -100

    dx2, dy2 = dx**2, dy**2
    div_term = float32(1.0 / (2 * (dx2 + dy2)))

    threads_per_block = (16, 16)
    blocks_per_grid = (int(math.ceil(nx / 16)), int(math.ceil(ny / 16)))

    start_time = time.time()
    final_it = 0

    # --- 2.                    Context Manager              GPU ---
    #                    d_p_in                              kernel

    try:
        with cuda.gpus[0]:
            #
            d_p_in = cuda.to_device(p_host)
            d_p_out = cuda.to_device(p_host)
            d_b = cuda.to_device(b_host)

            #
            for it in range(max_iter):
                poisson_kernel[blocks_per_grid, threads_per_block](
                    d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
                )
                #
                cuda.synchronize()

                #
                if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL == 0:
                    p_curr = d_p_out.copy_to_host()
                    p_prev = d_p_in.copy_to_host()
                    diff = np.max(np.abs(p_curr - p_prev))

                    if diff < tol:
                        final_it = it
                        #                           d_p_in
```

```
                        d_p_in.copy_to_device(d_p_out)
                        break

                #               (       GPU             )
                d_p_in, d_p_out = d_p_out, d_p_in
            else:
                final_it = max_iter

            #
            p_final = d_p_in.copy_to_host()

    except Exception as e:
        print(f"CRITICAL ERROR: {e}")
        return None, None, None, None, None

    return np.linspace(xmin, xmax, nx), np.linspace(ymin, ymax, ny),
   p_final, final_it, time.time() - start_time
```

Listing 11: poisson_numba.py

```
from numba import cuda, float32
import numba      #                    numba.float32
import numpy as np

TILE_SIZE = 16
SHARED_Y = TILE_SIZE + 2
SHARED_X = TILE_SIZE + 2

@cuda.jit
def poisson_shared_kernel(p_out, p_in, b, dx2, dy2, div_term, nx, ny):
    #                          +
    s_p = cuda.shared.array(shape=(SHARED_Y, SHARED_X), dtype=float32)

    tx, ty = cuda.threadIdx.x, cuda.threadIdx.y
    x, y = cuda.grid(2)

    sx, sy = tx + 1, ty + 1

    if x < nx and y < ny:
        s_p[sy, sx] = p_in[y, x]
    else:
        s_p[sy, sx] = 0.0

    if tx == 0 and x > 0:
        s_p[sy, 0] = p_in[y, x - 1]
    if tx == TILE_SIZE - 1 and x < nx - 1:
        s_p[sy, sx + 1] = p_in[y, x + 1]
    if ty == 0 and y > 0:
        s_p[0, sx] = p_in[y - 1, x]
    if ty == TILE_SIZE - 1 and y < ny - 1:
        s_p[sy + 1, sx] = p_in[y + 1, x]

    cuda.syncthreads()

    if x > 0 and x < nx - 1 and y > 0 and y < ny - 1:
        val = (((s_p[sy, sx+1] + s_p[sy, sx-1]) * dy2 +
                (s_p[sy+1, sx] + s_p[sy-1, sx]) * dx2 -
                b[y, x] * dx2 * dy2) * div_term)
```

```
        p_out[y, x] = val
```

Listing 12: poisson_numba_shared.py

```
#!/bin/bash
#SBATCH --job-name=cpu_scaling
#SBATCH --account=C3SE2025-2-17
#SBATCH --array=0-1
#SBATCH --nodes=1
#SBATCH --exclusive
#SBATCH --time=00:30:00
#SBATCH --partition=vera
#SBATCH --constraint=ZEN4             # <---
                            1                              CPU
#SBATCH --output=logs/cpu_test_%A_%a.log
#SBATCH --error=logs/cpu_test_%A_%a.err

echo "Job ID: $SLURM_JOB_ID, Array Task ID: $SLURM_ARRAY_TASK_ID"
echo "Running on node: $SLURMD_NODELIST"

#
module purge
module load virtualenv/20.26.2-GCCcore-13.3.0 SciPy-bundle/2024.05-gfbf
    -2024a

#
echo "Activating virtual environment..."
source $HOME/venvs/poisson_venv/bin/activate

#         Python
which python

#         Array ID
if [ "$SLURM_ARRAY_TASK_ID" -eq 0 ]; then
    CORES=32
    echo "Configuring for 32 Cores..."
else
    CORES=64
    echo "Configuring for 64 Cores..."
fi

#
export NUMBA_NUM_THREADS=$CORES
export OMP_NUM_THREADS=$CORES

#         Python
#         benchmark_cpu_advanced.py                    NUMBA_NUM_THREADS

python -u benchmark_cpu_parallel.py   # <---            2             -u


echo "Done with $CORES cores test."
```

Listing 13: run_cpu_array.sh

```
#!/bin/bash
#SBATCH --job-name=cupy_bench
#SBATCH --account=C3SE2025-2-17       #             ID
```

```
#SBATCH --time=00:15:00                # CuPy
                      15
#SBATCH --nodes=1
#SBATCH --gpus-per-node=A100:1           # <---                    1
    GPU
#SBATCH --partition=vera               # Vera        (                GPU
    )
#SBATCH --output=logs/cupy_%j.log      #                  logs
#SBATCH --error=logs/cupy_%j.err

echo "Job ID: $SLURM_JOB_ID"
echo "Running on node: $SLURMD_NODELIST"

# 1.        GPU        (                            )
echo "------------------------------------------------"
echo "Checking GPU status..."
nvidia-smi
echo "------------------------------------------------"

# 2.              (          CPU             )
module purge
module load virtualenv/20.26.2-GCCcore-13.3.0 SciPy-bundle/2024.05-gfbf
    -2024a

# 2.                           CUDA
# CuPy        libnvrtc.         s         o
#                     CUDA 12 (                   cupy-cuda12x)
module load CUDA/12.1.1
#                                                : module load CUDA

# 3.
echo "Activating virtual environment..."
source $HOME/venvs/poisson_venv/bin/activate

# 4.        Python     CuPy
which python
#                     cupy
python -c "import cupy; print(f'CuPy imported successfully! Device: {
    cupy.cuda.Device()}')"

echo "------------------------------------------------"
echo "Starting CuPy Benchmark..."

# 5.        Python
#              -u
python -u benchmark_cupy.py

echo "Done."
```

Listing 14: run_cupy.sh

```
#!/bin/bash
#SBATCH --job-name=torch_bench
#SBATCH --account=C3SE2025-2-17        #        ID
#SBATCH --time=00:15:00
#SBATCH --nodes=1

# ---                   A100         ---
#SBATCH --gpus-per-node=A100:1
```

```
9
10 #SBATCH --partition=vera
11 #SBATCH --output=logs/torch_%j.log
12 #SBATCH --error=logs/torch_%j.err
13
14 echo "Job ID: $SLURM_JOB_ID"
15 echo "Running on node: $SLURMD_NODELIST"
16
17 # 1.
18 echo "------------------------------------------------"
19 echo "GPU Allocated:"
20 nvidia-smi --query-gpu=name,memory.total --format=csv
21 echo "------------------------------------------------"
22
23 # 2.              (            )
24 module purge
25 module load virtualenv/20.26.2-GCCcore-13.3.0 SciPy-bundle/2024.05-gfbf
      -2024a
26
27 # 3.
28 echo "Activating virtual environment..."
29 source $HOME/venvs/poisson_venv/bin/activate
30
31 # 4.              (-u                  )
32 echo "Starting PyTorch Benchmark..."
33 python -u benchmark_torch.py
34
35 echo "Done."
```

Listing 15: run_torch.sh

```
1 #!/bin/bash
2 #SBATCH --job-name=numba_bench
3 #SBATCH --account=C3SE2025-2-17
4 #SBATCH --time=00:15:00
5 #SBATCH --nodes=1
6 #SBATCH --gpus-per-node=A100:1
7 #SBATCH --partition=vera
8 #SBATCH --output=logs/numba_%j.log
9 #SBATCH --error=logs/numba_%j.err
10
11 echo "Job ID: $SLURM_JOB_ID"
12
13 # 1.                                Smoke Test
14 module purge
15 module load GCC/12.3.0
16 module load CUDA/12.1.1
17 module load Python/3.11.3-GCCcore-12.3.0
18
19 # 2.                                                gpu_env
20 # (       gpu_env           project
                                                )
21 source $HOME/project/gpu_env/bin/activate
22
23 # 3.
24 #                            NUMBA_CUDA_DRIVER=/usr/lib64...
25 export NUMBA_CUDA_USE_NVIDIA_BINDING=1
```

```
unset CUDA_HOME
unset NUMBA_CUDA_DIR

# 4.              Benchmark
echo "Starting Numba Benchmark..."
python -u benchmark_numba.py

echo "Done."
```

Listing 16: run_numba.sh

```
#!/bin/bash
#SBATCH --job-name=numba_shared
#SBATCH --account=C3SE2025-2-17
#SBATCH --time=00:15:00
#SBATCH --nodes=1
#SBATCH --gpus-per-node=A100:1
#SBATCH --partition=vera
#SBATCH --output=logs/numba_shared_%j.log
#SBATCH --error=logs/numba_shared_%j.err

echo "Job ID: $SLURM_JOB_ID"

# 1.
module purge
module load GCC/12.3.0
module load CUDA/12.1.1
module load Python/3.11.3-GCCcore-12.3.0

# 2.        GPU
source $HOME/project/gpu_env/bin/activate

# 3.                              NVIDIA Bindings
export NUMBA_CUDA_USE_NVIDIA_BINDING=1
unset CUDA_HOME
unset NUMBA_CUDA_DIR

# 4.        Shared Memory Benchmark
echo "Starting Numba Shared Memory Benchmark..."
python -u benchmark_numba_shared.py

echo "Done."
```

Listing 17: run_numba_shared.sh