

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.

$$\nabla^2 p = b \quad (1)$$

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

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

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.

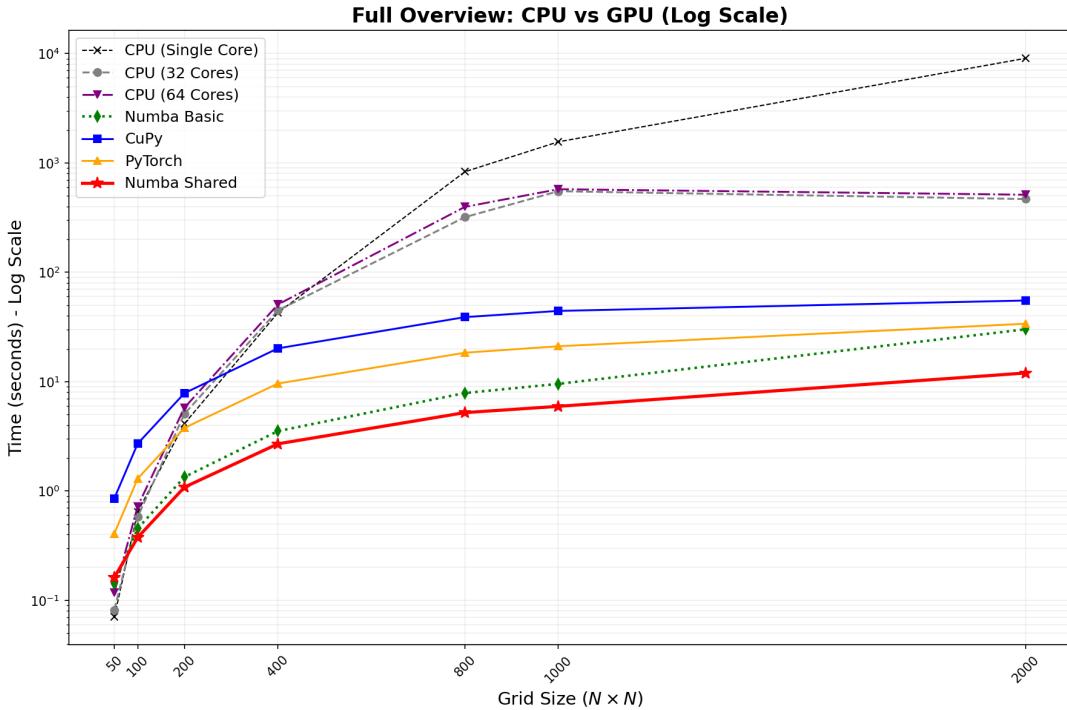


Figure 1: Full Performance Overview (Log Scale). Note the massive gap between CPU and GPU performance.

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.

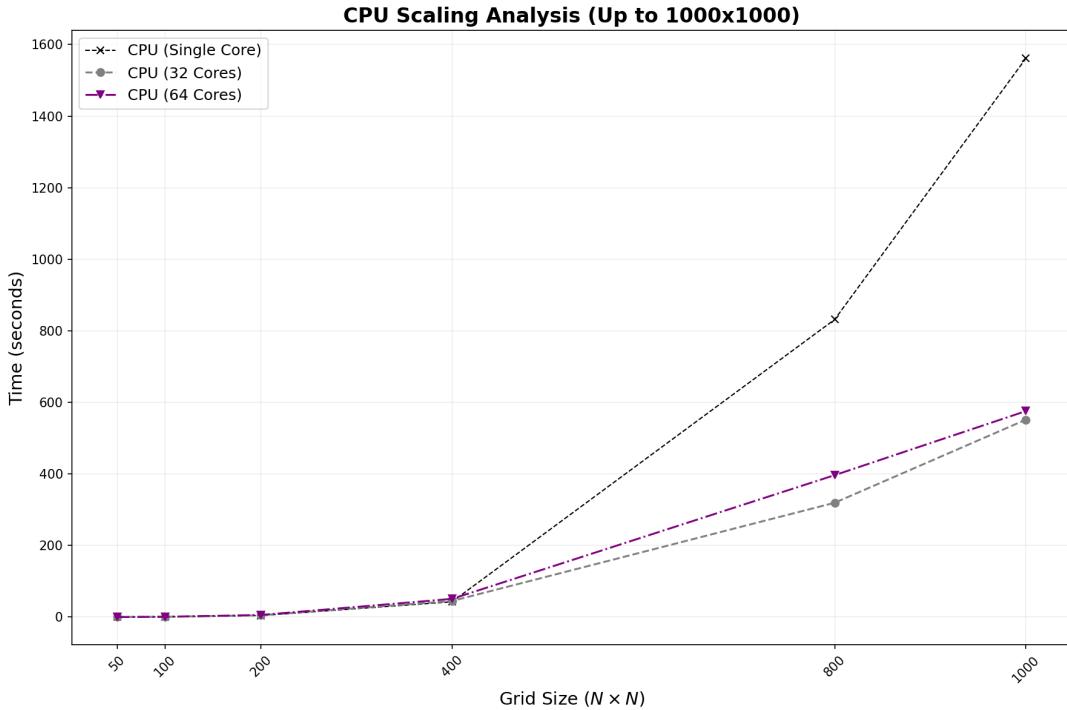


Figure 2: CPU Scaling Analysis. Increasing cores from 32 to 64 yields diminishing returns due to bandwidth saturation.

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.

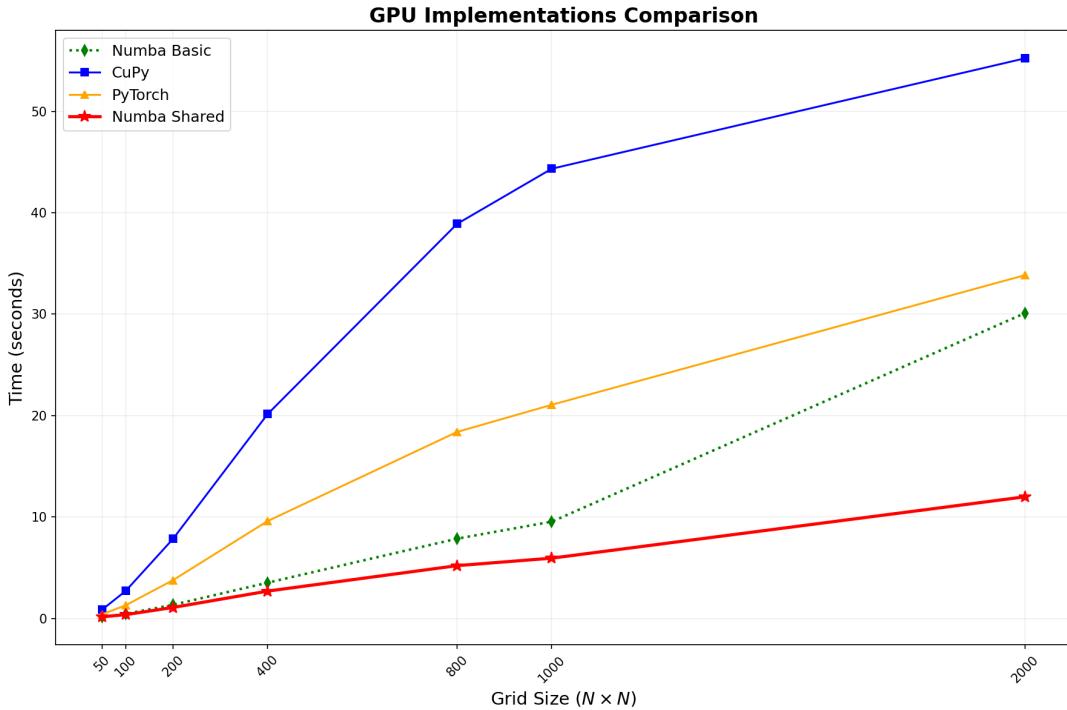


Figure 3: Detailed Comparison of GPU Implementations. Shared Memory optimization provides the fastest convergence.

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

```

1 import config
2 import time
3 from poisson_cpu import solve_cpu
4
5 def run_benchmark():
6     #             Benchmark
7     config.BENCHMARK_MODE = False
8
9     #
10
11     config.MAX_ITER = 200000
12
13     #           (      50x50      )
14     BASE_TOL = 1e-7
15     BASE_SIZE = 50
16
17     sizes = [50, 100, 200, 400, 800, 1000, 2000]
18
19     print(f"=====
20     ")
21     print(f"    CPU Benchmark (Dynamic Tolerance Mode)")
22     print(f"    Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
Tol_base / (Size/50)^2")
23     print(f"=====
24     ")
25     print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)':^10} | {'Status':^12}")
26     print("-" * 68)
27
28     for size in sizes:
29         # --+
30         #                                     4                               dx^2
31
32         scaling_factor = (size / BASE_SIZE) ** 2
33         dynamic_tol = BASE_TOL / scaling_factor
34
35         try:
36             #                         tol
37             _, _, _, iters, duration = solve_cpu(
38                 nx=size,
39                 ny=size,
40                 max_iter=config.MAX_ITER,
41                 tol=dynamic_tol
42             )
43
44             status = "Converged" if iters < config.MAX_ITER else "Max
45 Reached"
46
47             #
48             print(f"{size}x{size}<5} | {dynamic_tol:.1e} | {iters:^10}
49 | {duration:^10.4f} | {status:^12}")
50
51         except KeyboardInterrupt:
52             print("\nStopped by user.")

```

```

47         break
48     except Exception as e:
49         print(f"{size}x{size}:<5} |    ERROR      | {str(e)}")
50
51     print("-" * 68)
52
53 if __name__ == "__main__":
54     run_benchmark()

```

Listing 1: benchmark_cpu.py

```

1 import config
2 import time
3 import os
4 import numpy as np
5 #                                     poisson_cpu_parallel
6 #                                     poisson_cpu_advanced
7 from poisson_cpu_parallel import solve_cpu_parallel
8
9 def run_parallel_benchmark():
10    #                         Benchmark                  Dynamic Tolerance
11
12    config.BENCHMARK_MODE = False
13
14    # config.MAX_ITER = 200000
15
16    #                                     (
17    BASE_TOL = 1e-7
18    BASE_SIZE = 50
19
20    #
21    #
22    sizes = [50, 100, 200, 400, 800, 1000, 2000]
23
24    #
25    num_threads = os.environ.get('NUMBA_NUM_THREADS', 'Default')
26
27    print(f"=====
28    CPU PARALLEL Benchmark (Numba Multicore)")
29    print(f"    Threads Active: {num_threads}")
30    print(f"    Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
31    Tol_base / (Size/50)^2")
32    print(f"=====
33    # ---           1:                  'Tol Used'
34    # ---           1:                  'Tol Used'
35    print(f'{Grid}^10 | {Tol Used}^10 | {Steps}^10 | {Time (s)}^10 | {Status}^12")
36    print("-" * 68)
37
38    for size in sizes:
39        #

```

```

39     scaling_factor = (size / BASE_SIZE) ** 2
40     dynamic_tol = BASE_TOL / scaling_factor
41
42     try:
43         #
44         # --- 2: tol=dynamic_tol --- 1e-5
45         #
46
47         _, _, _, iters, duration = solve_cpu_parallel(
48             nx=size,
49             ny=size,
50             max_iter=config.MAX_ITER,
51             tol=dynamic_tol
52         )
53
54         status = "Converged" if iters < config.MAX_ITER else "Max
55 Reached"
56
57         # --- dynamic_tol
58         ---
59         print(f"{size}x{size:<5} | {dynamic_tol:.1e} | {iters:^10}
60 | {duration:^10.4f} | {status:^12}")
61
62     except TypeError as e:
63         print(f"{size}x{size:<5} | ERROR | {e}")
64         print("          : poisson_cpu_parallel.py")
65         solve_cpu_parallel('tol')
66         break
67     except Exception as e:
68         print(f"{size}x{size:<5} | ERROR | {str(e)}")
69
70     print("-" * 68)
71
72 if __name__ == "__main__":
73     run_parallel_benchmark()

```

Listing 2: benchmark_cpu_parallel.py

```

1 import config
2 import time
3 #           CuPy
4 from poisson_cupy import solve_cupy
5
6 def run_benchmark():
7     # --- 1. --- 1.
8     #           Benchmark
9     config.BENCHMARK_MODE = False
10
11     #
12
13     config.MAX_ITER = 200000
14
15     # --- 2. ( ) --- 2.
16     #           ( 50x50 )
17     BASE_TOL = 1e-7
18     BASE_SIZE = 50
19     #           ( )

```

```

20     sizes = [50, 100, 200, 400, 800, 1000, 2000]
21
22     print(f"
23 =====
24 ")
25     print(f"    CuPy Benchmark (Dynamic Tolerance Mode)")
26     print(f"    Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
27 Tol_base / (Size/50)^2")
28     print(f"
29 =====
30 ")
31     #           CPU
32     print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)':^10} |
33     {'Status':^12}")
34     print("-" * 68)
35
36     #           (Warmup)
37     # CuPy          K   e   r   n   e   l      50x50
38
39     try:
40         solve_cupy(nx=50, ny=50, max_iter=5, tol=1.0)
41     except Exception:
42         pass
43
44     for size in sizes:
45         # ---          ---
46         #           4           dx^2
47
48         scaling_factor = (size / BASE_SIZE) ** 2
49         dynamic_tol = BASE_TOL / scaling_factor
50
51         try:
52             #           tol
53             _, _, _, iters, duration = solve_cupy(
54                 nx=size,
55                 ny=size,
56                 max_iter=config.MAX_ITER,
57                 tol=dynamic_tol
58             )
59
60             status = "Converged" if iters < config.MAX_ITER else "Max
61 Reached"
62
63             #           (           CPU           )
64             print(f"{size}x{size}<5} | {dynamic_tol:.1e} | {iters:^10}
65             | {duration:^10.4f} | {status:^12}")
66
67         except Exception as e:
68             print(f"{size}x{size}<5} |    ERROR    | {str(e)}")
69
70     print("-" * 68)
71
72 if __name__ == "__main__":
73     run_benchmark()

```

Listing 3: benchmark_cupy.py

```

1 import config
2 import time

```

```

3 #                                     PyTorch
4 from poisson_pytorch import solve_pytorch
5
6 def run_torch_benchmark():
7     # --- 1.                               ---
8     config.BENCHMARK_MODE = False
9     config.MAX_ITER = 200000
10
11    # --- 2.          (           CPU/CuPy      ) ---
12    BASE_TOL = 1e-7
13    BASE_SIZE = 50
14
15    sizes = [50, 100, 200, 400, 800, 1000, 2000]
16
17    print(f"=====
18    PyTorch Benchmark (Dynamic Tolerance Mode)")
19    print(f"    Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
Tol_base / (Size/50)^2")
20    print(f"=====
21    {'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)':^10} |
{'Status':^12}")
22    print("-" * 68)
23
24    for size in sizes:
25        #
26        scaling_factor = (size / BASE_SIZE) ** 2
27        dynamic_tol = BASE_TOL / scaling_factor
28
29        try:
30            #          (Warmup): PyTorch
31            if size == sizes[0]:
32                solve_pytorch(nx=50, ny=50, max_iter=10, tol=1.0)
33
34            #
35            _, _, _, iters, duration = solve_pytorch(
36                nx=size,
37                ny=size,
38                max_iter=config.MAX_ITER,
39                tol=dynamic_tol # <---
40            )
41
42            status = "Converged" if iters < config.MAX_ITER else "Max
Reached"
43
44            print(f"{size}x{size}<5} | {dynamic_tol:.1e} | {iters:^10}
| {duration:^10.4f} | {status:^12}")
45
46        except Exception as e:
47            print(f"{size}x{size}<5} |     ERROR     | {str(e)}")
48
49    print("-" * 68)
50
51 if __name__ == "__main__":

```

52 run_torch_benchmark()

Listing 4: benchmark_torch.py

```
1 import config
2 import time
3 #           Numba
4 from poisson_numba import solve_numba
5
6 def run_numba_benchmark():
7     # --- 1. --- config.BENCHMARK_MODE = False
8     config.MAX_ITER = 200000
9
10    # --- 2. --- BASE_TOL = 1e-7
11    BASE_SIZE = 50
12
13    sizes = [50, 100, 200, 400, 800, 1000, 2000]
14
15    print(f"=====
16    =====")
17    print(f"      Numba (Basic) Benchmark (Dynamic Tolerance Mode)")
18    print(f"      Base Tol: {BASE_TOL} (at 50x50) | Logic: Tol_new =
19      Tol_base / (Size/50)^2")
20    print(f"      =====")
21    print(f"{'Grid':^10} | {'Tol Used':^10} | {'Steps':^10} | {'Time (s)':^10} |
22      {'Status':^12}")
23    print("-" * 68)
24
25    for size in sizes:
26        #
27        scaling_factor = (size / BASE_SIZE) ** 2
28        dynamic_tol = BASE_TOL / scaling_factor
29
30        try:
31            #           (Warmup): Numba             JIT
32            if size == sizes[0]:
33                solve_numba(nx=50, ny=50, max_iter=10, tol=1.0)
34
35            #
36            _, _, _, iters, duration = solve_numba(
37                nx=size,
38                ny=size,
39                max_iter=config.MAX_ITER,
40                tol=dynamic_tol # <---
41            )
42
43            status = "Converged" if iters < config.MAX_ITER else "Max
44 Reached"
45
46            print(f"{size}x{size:<5} | {dynamic_tol:.1e} | {iters:^10}
47 | {duration:^10.4f} | {status:^12}")
48
49        except Exception as e:
50            print(f"{size}x{size:<5} |     ERROR     | {str(e)}")
```

```

48     print("-" * 68)
49
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         "=====")
16     print(f"    Numba (Shared Memory) Benchmark")
17     print(f"    Tile Size: {TILE_SIZE}x{TILE_SIZE} | Shared Mem: {(TILE_SIZE+2)**2*4/1024:.2f} KB/block")
18     print(""
19         "=====")
20     print(f"{'Grid':<10} | {'Tol Used':<10} | {'Steps':<10} | {'Time (s)':<10} | {'Status':<12}")
21     print("-" * 68)
22
23     for n in grid_sizes:
24         nx, ny = n, n
25
26         # 1.
27         base_tol = 1e-7
28         tolerance = base_tol / ((n / 50.0) ** 2)
29
30         #           float32          GPU
31         p_host = np.zeros((ny, nx), dtype=np.float32)
32         b_host = np.zeros((ny, nx), dtype=np.float32)
33
34         #
35         b_host[int(ny / 4), int(nx / 4)] = 100
36         b_host[int(3 * ny / 4), int(3 * nx / 4)] = -100
37
38         # 3.          GPU
39         d_p_in = cuda.to_device(p_host)
40         d_p_out = cuda.to_device(p_host)
41         d_b = cuda.to_device(b_host)
42
43         # 4.
44         xmin, xmax = 0.0, 2.0
45         ymin, ymax = 0.0, 1.0
46         dx = (xmax - xmin) / (nx - 1)

```

```

46     dy = (ymax - ymin) / (ny - 1)
47     dx2, dy2 = dx**2, dy**2
48     div_term = 1.0 / (2 * (dx2 + dy2))
49
50     # 5.          Grid/Block
51     #           Shared Memory           Block
52     #                   TILE_SIZE
53     threads_per_block = (TILE_SIZE, TILE_SIZE)
54     blocks_per_grid = (math.ceil(nx / TILE_SIZE), math.ceil(ny / TILE_SIZE))
55
56     # 6.          (Warmup) -      JIT
57     poisson_shared_kernel[blocks_per_grid, threads_per_block](
58         d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
59     )
60     cuda.synchronize()
61
62     # 7.
63     start_time = time.time()
64     max_iter = 200000
65     converged = False
66     steps = 0
67
68     #           Host-Device           1000
69
70     check_interval = 1000
71
72     for i in range(0, max_iter, check_interval):
73         #           check_interval
74         for _ in range(check_interval):
75             poisson_shared_kernel[blocks_per_grid, threads_per_block
76             ](
77                 d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
78             )
79             #
80             d_p_in, d_p_out = d_p_out, d_p_in
81
82             cuda.synchronize()
83             steps += check_interval
84
85             #
86             p_curr = d_p_in.copy_to_host()
87             p_prev = d_p_out.copy_to_host() #
88             #           out
89             diff = np.max(np.abs(p_curr - p_prev))
90
91             if diff < tolerance:
92                 converged = True
93                 break
94
95             total_time = time.time() - start_time
96             status = "Converged" if converged else "Max Iter"
97
98             print(f"{'f'{n}x{n}':<10} | {tolerance:.1e} | {steps:<10} | {total_time:<10.4f} | {status:<12}")
99
100 if __name__ == "__main__":
101     if not cuda.is_available():

```

```

98     print("Error: CUDA not available.")
99 else:
100     run_benchmark()

```

Listing 6: benchmark_numba_shared.py

```

1 import numpy as np
2 import time
3 import config
4
5 #          tol=config.TOLERANCE
6 def solve_cpu(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER, tol=
config.TOLERANCE):
7     xmin, xmax = config.X_MIN, config.X_MAX
8     ymin, ymax = config.Y_MIN, config.Y_MAX
9     dx = (xmax - xmin) / (nx - 1)
10    dy = (ymax - ymin) / (ny - 1)
11
12    p = np.zeros((ny, nx))
13    pd = np.zeros((ny, nx))
14    b = np.zeros((ny, nx))
15
16    x = np.linspace(xmin, xmax, nx)
17    y = np.linspace(ymin, ymax, ny)
18
19    b[int(ny / 4), int(nx / 4)] = 100
20    b[int(3 * ny / 4), int(3 * nx / 4)] = -100
21
22    dx2 = dx**2
23    dy2 = dy**2
24    div_term = 1.0 / (2 * (dx2 + dy2))
25
26    start_time = time.time()
27    final_it = 0
28    final_error = 0.0
29
30    # print(f"CPU Baseline: Running {max_iter} steps (tol={tol:.1e})")
31    ...
32
33    for it in range(max_iter):
34        pd = p.copy()
35        p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
36                           (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
37                           b[1:-1, 1:-1] * dx2 * dy2) * div_term)
38
39        p[0, :] = 0; p[-1, :] = 0
40        p[:, 0] = 0; p[:, -1] = 0
41
42        #
43        #          tol      config.TOLERANCE
44        if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL ==
45        0:
46            final_error = np.abs(p - pd).max()
47            if final_error < tol:
48                final_it = it
49                # print(f"      CPU Converged at {it}, err={final_error
50                  :.2e}")
51                break
52        else:

```

```

50     final_it = max_iter
51
52     return x, y, p, final_it, time.time() - start_time

```

Listing 7: poisson_cpu.py

```

1 import numpy as np
2 import time
3 import config
4 from numba import jit, prange
5
6 @jit(nopython=True, parallel=True)
7 def poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny):
8     # prange                                         CPU
9     for y in prange(1, ny - 1):
10         for x in range(1, nx - 1):
11             p[y, x] = (((pd[y, x+1] + pd[y, x-1]) * dy2 +
12                           (pd[y+1, x] + pd[y-1, x]) * dx2 -
13                           b[y, x] * dx2 * dy2) * div_term)
14
15     return p
16
17 #          tol
18 def solve_cpu_parallel(nx=config.NX, ny=config.NY, max_iter=config.
19 MAX_ITER, tol=config.TOLERANCE):
20     xmin, xmax = config.X_MIN, config.X_MAX
21     ymin, ymax = config.Y_MIN, config.Y_MAX
22     dx = (xmax - xmin) / (nx - 1)
23     dy = (ymax - ymin) / (ny - 1)
24
25     p = np.zeros((ny, nx))
26     pd = np.zeros((ny, nx))
27     b = np.zeros((ny, nx))
28
29     x = np.linspace(xmin, xmax, nx)
30     y = np.linspace(ymin, ymax, ny)
31
32     b[int(ny / 4), int(nx / 4)] = 100
33     b[int(3 * ny / 4), int(3 * nx / 4)] = -100
34
35     dx2 = dx**2
36     dy2 = dy**2
37     div_term = 1.0 / (2 * (dx2 + dy2))
38
39     #          (Numba)
40     poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny)
41
42     start_time = time.time()
43     final_it = 0
44
45     for it in range(max_iter):
46         #
47         pd[:] = p[:]
48
49         #
50         poisson_step_parallel(p, pd, b, dx2, dy2, div_term, nx, ny)
51
52         #          (      CPU
53
54         )

```

```

51     p[0, :] = 0; p[-1, :] = 0
52     p[:, 0] = 0; p[:, -1] = 0
53
54     #
55     if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL == 0:
56         final_error = np.abs(p - pd).max()
57         if final_error < tol: # tol
58             final_it = it
59             # print(f"      CPU Parallel Converged at {it}")
60             break
61     else:
62         final_it = max_iter
63
64 return x, y, p, final_it, time.time() - start_time

```

Listing 8: poisson_cpu_parallel.py

```

1 import cupy as cp
2 import numpy as np
3 import time
4 import config
5
6 def solve_cupy(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER, tol=config.TOLERANCE):
7     """
8     GPU Implementation using CuPy (Drop-in replacement for NumPy)
9     """
10    # 1. ( CPU )
11    xmin, xmax = config.X_MIN, config.X_MAX
12    ymin, ymax = config.Y_MIN, config.Y_MAX
13    dx = (xmax - xmin) / (nx - 1)
14    dy = (ymax - ymin) / (ny - 1)
15
16    # 2. ( GPU )
17    # cp.zeros      np.      z      e      r      o      s      GPU
18    p = cp.zeros((ny, nx), dtype=cp.float32)
19    pd = cp.zeros((ny, nx), dtype=cp.float32)
20    b = cp.zeros((ny, nx), dtype=cp.float32)
21
22    #
23    b[int(ny / 4), int(nx / 4)] = 100
24    b[int(3 * ny / 4), int(3 * nx / 4)] = -100
25
26    #
27    dx2 = dx**2
28    dy2 = dy**2
29    div_term = 1.0 / (2 * (dx2 + dy2))
30
31    # 3. (Warmup)
32    # CuPy
33    #      K      e      r      n      e      l
34    p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
35                      (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
36                      b[1:-1, 1:-1] * dx2 * dy2) * div_term)
37    cp.cuda.Device().synchronize() #
38
39    print(f"CuPy: Starting simulation on GPU...")
start_time = time.time()

```

```

40     final_it = 0
41
42     # 4.
43     for it in range(max_iter):
44         pd = p.copy() #      GPU
45
46             #                         NumPy
47             p[1:-1, 1:-1] = (((pd[1:-1, 2:] + pd[1:-1, :-2]) * dy2 +
48                                (pd[2:, 1:-1] + pd[:-2, 1:-1]) * dx2 -
49                                b[1:-1, 1:-1] * dx2 * dy2) * div_term)
50
51             #
52             p[0, :] = 0; p[-1, :] = 0
53             p[:, 0] = 0; p[:, -1] = 0
54
55             #
56             if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL ==
57             0:
57                 # cp.max      cp.abs      GPU
58                 #                  < tol      CuPy
59                 #                           CPU
60                 diff = cp.max(cp.abs(p - pd))
61                 if diff < tol:
62                     final_it = it
63                     # print(f"      CuPy Converged at {it}, err={diff:.2e}
64                     #")
65                     break
66             else:
67                 final_it = max_iter
68
69     # 5.
70     # GPU
71
72     cp.cuda.Device().synchronize()
73     total_time = time.time() - start_time
74
75     # 6.          GPU      CPU (
76     p_cpu = cp.asarray(p)
77
78     return np.linspace(xmin, xmax, nx), np.linspace(ymin, ymax, ny),
79     p_cpu, final_it, total_time

```

Listing 9: poisson_cupy.py

```

1 import torch
2 import numpy as np
3 import time
4 import config
5
6 #                      tol                  config.TOLERANCE
7 def solve_pytorch(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER,
8 tol=config.TOLERANCE):
9     if not torch.cuda.is_available():
10         raise RuntimeError("CUDA not available")
11
12     device = torch.device('cuda')
13
14     xmin, xmax = config.X_MIN, config.X_MAX
15     ymin, ymax = config.Y_MIN, config.Y_MAX

```

```

15     dx = (xmax - xmin) / (nx - 1)
16     dy = (ymax - ymin) / (ny - 1)
17
18     p = torch.zeros((ny, nx), device=device, dtype=torch.float32)
19     b = torch.zeros((ny, nx), device=device, dtype=torch.float32)
20
21     b[int(ny / 4), int(nx / 4)] = 100
22     b[int(3 * ny / 4), int(3 * nx / 4)] = -100
23
24     dx2 = dx**2
25     dy2 = dy**2
26     div_term = 1.0 / (2 * (dx2 + dy2))
27
28     start_time = time.time()
29     final_it = 0
30
31     #           GPU (                                benchmark
32     # torch.cuda.synchronize()
33
34     for it in range(max_iter):
35         p_old = p.clone()
36
37         p[1:-1, 1:-1] = (((p_old[1:-1, 2:] + p_old[1:-1, :-2]) * dy2 +
38                            (p_old[2:, 1:-1] + p_old[:-2, 1:-1]) * dx2 -
39                            b[1:-1, 1:-1] * dx2 * dy2) * div_term)
40
41         p[0, :] = 0; p[-1, :] = 0
42         p[:, 0] = 0; p[:, -1] = 0
43
44         if not config.BENCHMARK_MODE and it % config.CHECK_INTERVAL == 0:
45             #
46             diff = torch.max(torch.abs(p - p_old)).item()
47
48             #
49             if diff < tol:
50                 final_it = it
51                 # print(f"      PyTorch Converged at {it}, err={diff:.2e}")
52             break
53         else:
54             final_it = max_iter
55
56     torch.cuda.synchronize()
57     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

```

1 from numba import cuda, float32
2 import numpy as np
3 import math
4 import time
5 import config
6
7 @cuda.jit
8 def poisson_kernel(p_out, p_in, b, dx2, dy2, div_term, nx, ny):
9     x, y = cuda.grid(2)
10    if x > 0 and x < nx - 1 and y > 0 and y < ny - 1:

```

```

11         p_out[y, x] = (((p_in[y, x+1] + p_in[y, x-1]) * dy2 +
12                         (p_in[y+1, x] + p_in[y-1, x]) * dx2 -
13                         b[y, x] * dx2 * dy2) * div_term)
14
15 def solve_numba(nx=config.NX, ny=config.NY, max_iter=config.MAX_ITER,
16                  tol=config.TOLERANCE):
17     if not cuda.is_available():
18         return None, None, None, None
19
20     # --- 1.          CPU          ---
21     xmin, xmax = config.X_MIN, config.X_MAX
22     ymin, ymax = config.Y_MIN, config.Y_MAX
23     dx = float32((xmax - xmin) / (nx - 1))
24     dy = float32((ymax - ymin) / (ny - 1))
25
26     p_host = np.zeros((ny, nx), dtype=np.float32)
27     b_host = np.zeros((ny, nx), dtype=np.float32)
28     b_host[int(ny / 4), int(nx / 4)] = 100
29     b_host[int(3 * ny / 4), int(3 * nx / 4)] = -100
30
31     dx2, dy2 = dx**2, dy**2
32     div_term = float32(1.0 / (2 * (dx2 + dy2)))
33
34     threads_per_block = (16, 16)
35     blocks_per_grid = (int(math.ceil(nx / 16)), int(math.ceil(ny / 16)))
36
37     start_time = time.time()
38     final_it = 0
39
40     # --- 2.          Context Manager          GPU
41     # ---                                     kernel
42
43     try:
44         with cuda.gpus[0]:
45             #
46             d_p_in = cuda.to_device(p_host)
47             d_p_out = cuda.to_device(p_host)
48             d_b = cuda.to_device(b_host)
49
50             #
51             for it in range(max_iter):
52                 poisson_kernel[blocks_per_grid, threads_per_block](
53                     d_p_out, d_p_in, d_b, dx2, dy2, div_term, nx, ny
54                 )
55                 #
56                 cuda.synchronize()
57
58                 #
59                 if not config.BENCHMARK_MODE and it % config.
60                 CHECK_INTERVAL == 0:
61                     p_curr = d_p_out.copy_to_host()
62                     p_prev = d_p_in.copy_to_host()
63                     diff = np.max(np.abs(p_curr - p_prev))
64
65                     if diff < tol:
66                         final_it = it
67                         #                                     d_p_in

```

```

65             d_p_in.copy_to_device(d_p_out)
66             break
67
68         #           ( GPU )
69         d_p_in, d_p_out = d_p_out, d_p_in
70     else:
71         final_it = max_iter
72
73     #
74     p_final = d_p_in.copy_to_host()
75
76 except Exception as e:
77     print(f"CRITICAL ERROR: {e}")
78     return None, None, None, None, None
79
80 return np.linspace(xmin, xmax, nx), np.linspace(ymax, ymin, ny),
p_final, final_it, time.time() - start_time

```

Listing 11: poisson_numba.py

```

1 from numba import cuda, float32
2 import numba          #                         numba.float32
3 import numpy as np
4
5 TILE_SIZE = 16
6 SHARED_Y = TILE_SIZE + 2
7 SHARED_X = TILE_SIZE + 2
8
9 @cuda.jit
10 def poisson_shared_kernel(p_out, p_in, b, dx2, dy2, div_term, nx, ny):
11     #           +
12     s_p = cuda.shared.array(shape=(SHARED_Y, SHARED_X), dtype=float32)
13
14     tx, ty = cuda.threadIdx.x, cuda.threadIdx.y
15     x, y = cuda.grid(2)
16
17     sx, sy = tx + 1, ty + 1
18
19     if x < nx and y < ny:
20         s_p[sy, sx] = p_in[y, x]
21     else:
22         s_p[sy, sx] = 0.0
23
24     if tx == 0 and x > 0:
25         s_p[sy, 0] = p_in[y, x - 1]
26     if tx == TILE_SIZE - 1 and x < nx - 1:
27         s_p[sy, sx + 1] = p_in[y, x + 1]
28     if ty == 0 and y > 0:
29         s_p[0, sx] = p_in[y - 1, x]
30     if ty == TILE_SIZE - 1 and y < ny - 1:
31         s_p[sy + 1, sx] = p_in[y + 1, x]
32
33     cuda.syncthreads()
34
35     if x > 0 and x < nx - 1 and y > 0 and y < ny - 1:
36         val = (((s_p[sy, sx+1] + s_p[sy, sx-1]) * dy2 +
37                  (s_p[sy+1, sx] + s_p[sy-1, sx])) * dx2 -
38                  b[y, x] * dx2 * dy2) * div_term)

```

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

Listing 12: poisson_numba_shared.py

```
1 #!/bin/bash
2 #SBATCH --job-name=cpu_scaling
3 #SBATCH --account=C3SE2025-2-17
4 #SBATCH --array=0-1
5 #SBATCH --nodes=1
6 #SBATCH --exclusive
7 #SBATCH --time=00:30:00
8 #SBATCH --partition=vera
9 #SBATCH --constraint=ZEN4           # <---
10 #SBATCH --output=logs/cpu_test_%A_%a.log
11 #SBATCH --error=logs/cpu_test_%A_%a.err
12
13 echo "Job ID: $SLURM_JOB_ID, Array Task ID: $SLURM_ARRAY_TASK_ID"
14 echo "Running on node: $SLURMD_NODELIST"
15
16 #
17 module purge
18 module load virtualenv/20.26.2-GCCcore-13.3.0 SciPy-bundle/2024.05-gfbf
19 -2024a
20
21 #
22 echo "Activating virtual environment..."
23 source $HOME/venvs/poisson_venv/bin/activate
24
25 #          Python
26 which python
27
28 #          Array ID
29 if [ "$SLURM_ARRAY_TASK_ID" -eq 0 ]; then
30     CORES=32
31     echo "Configuring for 32 Cores..."
32 else
33     CORES=64
34     echo "Configuring for 64 Cores..."
35 fi
36
37 export NUMBA_NUM_THREADS=$CORES
38 export OMP_NUM_THREADS=$CORES
39
40 #          Python
41 #          benchmark_cpu_advanced.py           NUMBA_NUM_THREADS
42 python -u benchmark_cpu_parallel.py    # <---           2           -u
43
44 echo "Done with $CORES cores test."
```

Listing 13: run_cpu_array.sh

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

```

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

10
11 echo "Job ID: $SLURM_JOB_ID"
12 echo "Running on node: $SLURMD_NODELIST"
13
14 # 1.      GPU      ( )
15 echo "-----"
16 echo "Checking GPU status..."
17 nvidia-smi
18 echo "-----"

19
20 # 2.      ( CPU      )
21 module purge
22 module load virtualenv/20.26.2-GCCcore-13.3.0 SciPy-bundle/2024.05-gfbf
      -2024a
23
24 # 2.      CUDA
25 # CuPy      libnvrtc. s o
26 #             CUDA 12 (   cupy-cuda12x)
27 module load CUDA/12.1.1
28 #
      : module load CUDA
29
30 # 3.
31 echo "Activating virtual environment..."
32 source $HOME/venvs/poisson_venv/bin/activate
33
34 # 4.      Python      CuPy
35 which python
36 #             cupy
37 python -c "import cupy; print(f'CuPy imported successfully! Device: {cupy.cuda.Device()}' )"
38
39 echo "-----"
40 echo "Starting CuPy Benchmark..."
41
42 # 5.      Python
43 #             -u
44 python -u benchmark_cupy.py
45
46 echo "Done."

```

Listing 14: run_cupy.sh

```

1 #!/bin/bash
2 #SBATCH --job-name=torch_bench
3 #SBATCH --account=C3SE2025-2-17      # ID
4 #SBATCH --time=00:15:00
5 #SBATCH --nodes=1
6
7 # ---          A100      ---
8 #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
26 -2024a
27
28 # 3.
29 echo "Activating virtual environment..."
30 source $HOME/venvs/poisson_venv/bin/activate
31
32 # 4.          (-u)
33 echo "Starting PyTorch Benchmark..."
34 python -u benchmark_torch.py
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
15 module purge
16 module load GCC/12.3.0
17 module load CUDA/12.1.1
18 module load Python/3.11.3-GCCcore-12.3.0
19
20 # 2.           gpu_env
21 # (           gpu_env           project
22 #                   )           project
23
24 # 3.           NUMBA_CUDA_DRIVER=/usr/lib64...
25 export NUMBA_CUDA_USE_NVIDIA_BINDING=1

```

```

26 unset CUDA_HOME
27 unset NUMBA_CUDA_DIR
28
29 # 4.          Benchmark
30 echo "Starting Numba Benchmark..."
31 python -u benchmark_numba.py
32
33 echo "Done."

```

Listing 16: run_numba.sh

```

1 #!/bin/bash
2 #SBATCH --job-name=numba_shared
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_shared_%j.log
9 #SBATCH --error=logs/numba_shared_%j.err
10
11 echo "Job ID: $SLURM_JOB_ID"
12
13 # 1.
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
20 source $HOME/project/gpu_env/bin/activate
21
22 # 3.          NVIDIA Bindings
23 export NUMBA_CUDA_USE_NVIDIA_BINDING=1
24 unset CUDA_HOME
25 unset NUMBA_CUDA_DIR
26
27 # 4.          Shared Memory Benchmark
28 echo "Starting Numba Shared Memory Benchmark..."
29 python -u benchmark_numba_shared.py
30
31 echo "Done."

```

Listing 17: run_numba_shared.sh