GPU Accelerated Poisson solver

A comparison of PyTorch, Cupy and multi-GPU (Cupy) for solving the Poisson equation

Sebastian Miles, Bingzhou Xie

Supervisor:
Lars Davidson

CHALMERS

Chalmers University of Technology
February 16, 2025

1 Abstract

The purpose of this project is to accelerate a CPU Poisson equation solver with a GPU. The basis of our code is the vectorized Poisson Solver by Lars Davidson [1]. This report documents the steps taken to convert the existing CPU Poisson solver to run efficiently on a GPU. We try two different libraries for this: PyTorch and Cupy and compare their differences and conclude the one that works the best for this particular problem. Finally we also try comparing with multiple GPUs for a single problem. Comparing CPU and PyTorch, there was roughly a 6x speedup with PyTorch. Between PyTorch and Cupy there was a better time complexity with Cupy, where Cupy had support for efficient CSR format. In our test with a 1000×1000 grid, Cupy was running 9x faster than PyTorch.

1.1 Contributions

- Sebastian Miles: Sections 2-5 & 7
- Bingzhou Xie: Sections 6 & 8

2 Solver

When solving the Poisson equation, our main interest is finding an appropriate method for solving the linear matrix equation

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

where A is a sparse, symmetric and positive-definite matrix.

From the initial code by lada, we implemented our own solvers. We started with the Conjugate Gradient (CG) method. After noticing a very slow convergence rate for large grid sizes, we decided to try preconditioned CG. In particular a Jacobi preconditioner, which is done by extracting only the diagonal of the matrix and leaving everything else to zero. With a tolerance of 10^{-10} on a 100×100 grid the original CG converged after 1173 iterations, while the preconditinoed CG converged after 18 iterations.

2.1 Conjugate Gradient (CG)

Suppose that

$$P = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$$

is a set of n vectors such that $\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0$ for all $i \neq j$, where $\mathbf{A}$ is the same as in equation 1. Then P forms a basis on $\mathbb{R}^n$. This means that we can express a

solution $\mathbf{x} = \hat{\mathbf{x}}$ of $\mathbf{Ax} = \mathbf{b}$ as

$$\hat{\mathbf{x}} = \sum_{i}^{n} c_i \mathbf{p}_i.$$

Left multiplying by A and then by $\mathbf{p}_k^T$ yields

$$\mathbf{A}\hat{\mathbf{x}} = \sum_{i}^{n} c_i \mathbf{A}\mathbf{p}_i \implies \mathbf{p}_k^T \mathbf{A}\hat{\mathbf{x}} = \sum_{i}^{n} c_i \mathbf{p}_k^T \mathbf{A}\mathbf{p}_i = c_k \mathbf{p}_k^T \mathbf{A}\mathbf{p}_k.$$

This means that

$$c_k = \frac{\mathbf{p}_k^T \mathbf{A}\hat{\mathbf{x}}}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}.$$

The last equality holds because $\mathbf{p}_i^T \mathbf{A}\mathbf{p}_j = 0$ for all $i \neq j$ as part of our premise. This gives us a brief idea for solving equation 1: Find the set of vectors P, then compute the coefficients c_k and then rebuild $\hat{\mathbf{x}}$ from the linear combination.

Consider the quadratic function

$$f : \mathbb{R}^n \to \mathbb{R}$$

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A}\mathbf{x} - \mathbf{x}^T \mathbf{b}.$$

We note that the gradient is equal to

$$\nabla f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$$

and that $\hat{\mathbf{x}}$ is in fact a minimizer to this function, since that the hessian matrix is positive definite

$$\mathbf{H}(f(x)) = \mathbf{A}.$$

By the Gradient descent method, f decreases the fastest in the direction of the negative gradient $-\nabla f(\mathbf{x})$. Thus, starting with an initial value of $\mathbf{x}_0$ we set the first vector to the negative gradient $\mathbf{p}_0 = \mathbf{b} - \mathbf{A}\mathbf{x_0}$. Let $\mathbf{r}_k$ be the residue at the k-th step:

$$\mathbf{r}_k = \mathbf{b} - \mathbf{A}\mathbf{x}_k$$

To enforce the conjugation constraint on $\mathbf{p_k}$ we set it to

$$\mathbf{p}_k = \mathbf{r}_k - \sum_{i<k} \frac{\mathbf{p}_i^T \mathbf{A}\mathbf{r}_k}{\mathbf{p}_i^T \mathbf{A}\mathbf{p}_i} \mathbf{p}_k.$$

Continuing along, the next iterations are

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$$

where α_k is some scalar. Let $g(\alpha_k) = f(\mathbf{x}_k + \alpha_k \mathbf{p}_k)$. We can construct a heuristically good scalar with $g'(\alpha_k) \to 0$ as $k \to \infty$.

$$
\begin{aligned}
g(\alpha_k) &= \frac{1}{2}(\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{A}(\mathbf{x}_k + \alpha_k \mathbf{p}_k) - (\mathbf{x}_k + \alpha_k \mathbf{p}_k)^T \mathbf{b} \\
\implies g'(\alpha_k) &= \frac{1}{2}\mathbf{x_k^T A p_k} + \frac{1}{2}\mathbf{p_k^T A x_k} + \alpha_k \mathbf{p}_k^T \mathbf{A p_k} - \mathbf{p_k^T b} \\
g'(\alpha_k) \to 0 \iff \alpha_k \mathbf{p_k^T A p_k} &\to \mathbf{p_k^T}(b - \frac{1}{2}\mathbf{A x_k}) - \frac{1}{2}\mathbf{x_k^T A p_k} \\
&= \mathbf{p_k^T}(b - \frac{1}{2}\mathbf{A x_k}) - \frac{1}{2}(\mathbf{x_{k+1}} - \alpha_\mathbf{k}\mathbf{p_k}))^\mathbf{T}\mathbf{A}\left(\frac{\mathbf{x_{k+1}} - \mathbf{x_k}}{\alpha_\mathbf{k}}\right) \\
&= \mathbf{p_k^T}(b - \mathbf{A x_k}) - \frac{\mathbf{x_{k+1}^T A}}{2\alpha_k}(x_{k+1} - x_k) + \frac{\mathbf{p_k^T A}}{2}(\mathbf{x_{k+1}} - \mathbf{x_k}) \\
&\approx \mathbf{p_k^T}(b - \mathbf{A x_k}) = \mathbf{p_k^T} r_k
\end{aligned}
$$

After dividing by $\mathbf{p}_k^T \mathbf{A p}_k$ on both sides, we end up with the scalar

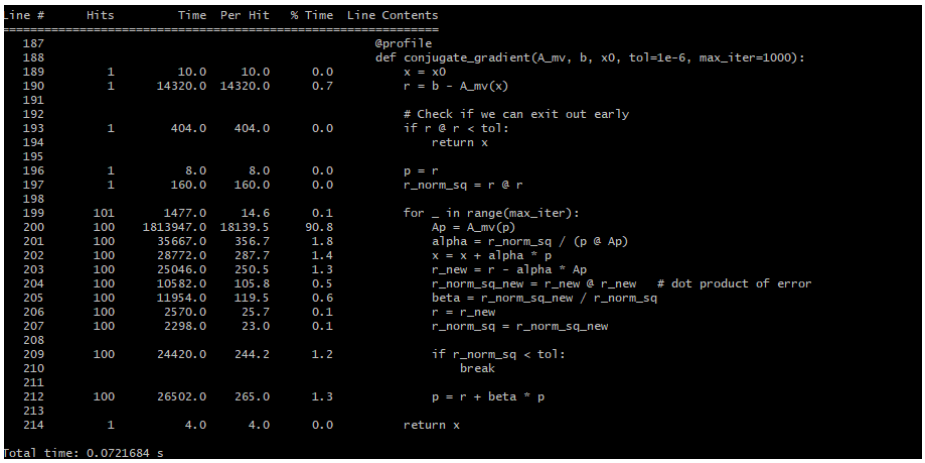
$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A p}_k}.$$

2.2 Jacobi Preconditioned Conjugate Gradient

A preconditioning $\mathbf{M}$ means that we transform the matrix into $\mathbf{M^{-1}A}$ which is more suitable for numerical calculations. A metric commonly used for this is the condition number $\kappa(\mathbf{A})$. The metric measures how much the output of $\mathbf{Ax}$ can change for a small change in $\mathbf{x}$. The goal of preconditioning is reducing the condition number, that is $\kappa(\mathbf{M}^{-1}\mathbf{A}) < \kappa(\mathbf{A})$. The Jacobi preconditioner is one of the more simpler preconditions. It is defined as $\mathbf{M} = \text{diag}(\mathbf{A})$. Note that the inverse $\mathbf{M}^{-1}$ is a diagonal matrix with the diagonal $M_{i,i}^{-1} = \frac{1}{\mathbf{M}_{i,i}}$. So the transformation is computationally cheap.

3 Profiling

The line_profiler package in python is a useful tool for profiling the code line by line. We started by profiling on the CPU, and came to the conclusion that the main time sink was within the CG method. In figure 2 the heaviest line is #200 which takes up 90.8% of the time. The function A_mv(p) is a matrix vector product and it would be a good idea to parallize this operation.



```
Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
   187                                           @profile
   188                                           def conjugate_gradient(A_mv, b, x0, tol=1e-6, max_iter=1000):
   189         1         10.0     10.0      0.0      x = x0
   190         1      14320.0  14320.0      0.7      r = b - A_mv(x)
   191
   192                                               # Check if we can exit out early
   193         1        404.0    404.0      0.0      if r @ r < tol:
   194                                                   return x
   195
   196         1          8.0      8.0      0.0      p = r
   197         1        160.0    160.0      0.0      r_norm_sq = r @ r
   198
   199       101       1477.0     14.6      0.1      for _ in range(max_iter):
   200       100    1813947.0  18139.5     90.8          Ap = A_mv(p)
   201       100      35667.0    356.7      1.8          alpha = r_norm_sq / (p @ Ap)
   202       100      28772.0    287.7      1.4          x = x + alpha * p
   203       100      25046.0    250.5      1.3          r_new = r - alpha * Ap
   204       100      10582.0    105.8      0.5          r_norm_sq_new = r_new @ r_new   # dot product of error
   205       100      11954.0    119.5      0.6          beta = r_norm_sq_new / r_norm_sq
   206       100       2570.0     25.7      0.1          r = r_new
   207       100       2298.0     23.0      0.1          r_norm_sq = r_norm_sq_new
   208
   209       100      24420.0    244.2      1.2          if r_norm_sq < tol:
   210                                                       break
   211
   212       100      26502.0    265.0      1.3          p = r + beta * p
   213
   214         1          4.0      4.0      0.0      return x

Total time: 0.0721684 s
```

Figure 1: Line profiler on the CG function

4 PyTorch Implementation

From the CPU poisson solver by lada, we converted all of the numpy code to PyTorch. With this implementation we could easily swap between a CPU or CUDA device with a single boolean variable. We decided to put all of the PyTorch tensors on the CUDA device as there would be no time spent moving data between the CPU and GPU.

This paralellized a lot of functions, but most importantly the matrix vector product. At first we had a $(n_i, n_j) = (60, 60)$ grid, there was no significant speedup. We then tried a larger grid of 200x200 where we timed 59s on CPU and 9.73s on the GPU. This is about a 6x speedup.

For larger grids CG did not converge very well, so we preconditioned it with a Jacobi Preconditioner as in section 2.2. The convergence was much faster now, so to accurately measure speed we simply solve the equation around 100 times. Our results for comparing the CPU and the PyTorch is in figure 2. For small grid sizes, the CPU dominates and then as the grid size increases the power of the GPU is clearly visible. Note also that the plot has a logarithmic scale, we had a speedup of 8.1x with Pytorch compared to CPU on the finest grid.

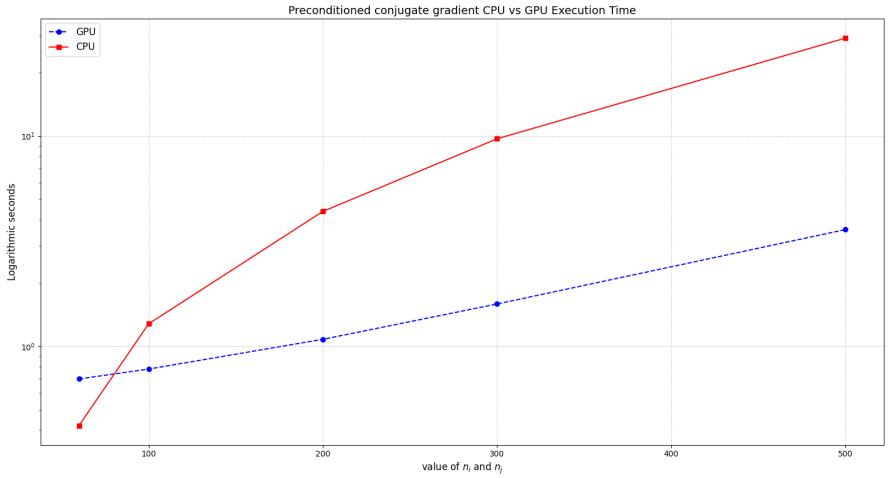


Figure 2: Computation time for cpu and gpu for various grid square grid sizes

A major flaw with PyTorch is that for the GPU version there is little support for sparse matrices, in particular at the time of writing, there is only support for Coordinate list (COO) matrix format on the GPU. Unfortunately, this format has a slow matrix vector product as it iterates through every element in the matrix for each element in the product. A format such as Compressed Sparse Row (CSR) format would efficiently compute the product with a better time complexity. Although the GPU beats the CPU there is still room for improvement.

5 Cupy Implementation

Cupy was easier to implement as Cupy is very similar to Numpy and Numpy is what the original code by Lada used. There is one major difference between the PyTorch and Cupy implementation and that is Cupy supports the Compressed Sparse Row format. When doing the Preconditioned CG algorithm the main bottleneck is the matrix dot product, but now we can efficiently compute it and we plot the PyTorch vs Cupy results in figure 3.

Cupy is performing better than PyTorch for every grid size that we tried. On the largest grid size of 1000×1000 we had a 9x speedup. Most importantly we can clearly see that Cupy has a better time complexity than the PyTorch implementation. This is due to the different sparse matrix formats discussed in an earlier section. This should be one polynomial degree time complexity difference as COO iterates every element while CSR iterates only the necessary elements.

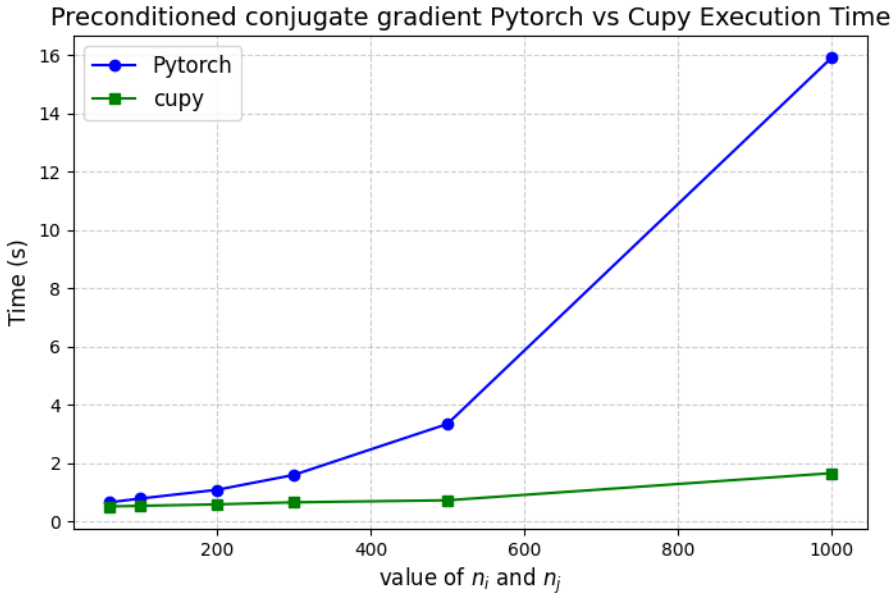


Figure 3: Computation time for PyTorch and Cupy for various grid square grid sizes

6 Multi-GPU Cupy

To achieve parallelization in the multi-GPU implementation, we employ domain decomposition. The computational domain is divided into subdomains, each assigned to a different GPU. This allows the solver to run concurrently on multiple GPUs, with each GPU handling a portion of the domain.

6.1 Subdomain and Ghost Cells

Decomposition is performed along the dimension by distributing the rows of the grid to available GPUs. For a domain of size ni×nj, distributed across n GPUs, each GPU processes approximately (ni/n) × nj grid points. The subdomains are represented by instances of the GPUDomain class, which encapsulates the data and operations specific to each subdomain.

To ensure proper communication between subdomains, we introduce ghost cells [2]. Ghost cells are additional cells surrounding each subdomain that store a copy of the boundary data from neighboring subdomains4. These cells prompt the exchange of boundary information and maintain consistency across the global domain. When we initialize the subdomain of GPU, we need to allocate memory space for both subdomains and ghost cells. The number of ghost cells depends on the size of the finite difference stencil.

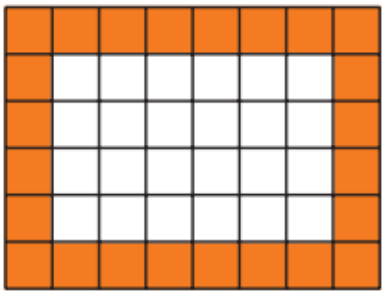


Figure 4: The subdomain(white part) and the ghost cell(orange part)

In the program, we use a 5 point stencil [3], which requires one layer of ghost cell on some side of the subdomain.

The Poisson equation in 2D can be discretized using the following formula:

$$u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} - 4u_{i,j} = 0$$

This formula represents the relationship between a central point $u_{i,j}$ and its four neighboring points $(u_{i+1,j})$, $(u_{i-1,j})$, $(u_{i,j+1})$, and $(u_{i,j-1})$ in a uniform 2D grid. It need ghost cells to handle boundary conditions and ensure the accuracy of finite difference formulas near boundaries. Ghost cells are also an important tool for implementing parallelization.

6.2 Exchange boundary data

After every iteration of the solver, the boundary data of each neighboring subdomain need to be exchanged to update the ghost cells. All neighboring subdomain pairs should be traversed and the boundary data from one subdomain should be copied to the ghost cell of its neighbor subdomain, which ensures all the ghost cell have been synchronized with the latest data.

To optimize the inter-GPU communication, we use CuPy's asynchronous copy functions (cupy.cuda.runtime.memcpy_async). These functions enable efficient data transfer between different GPU devices, minimizing the overhead of boundary exchange.

In the solver ghost cells are treated as part of the subdomain. But we need to treat the edges of the subdomain that lie on the edge of the grid differently. The ghost cells of those subdomains are expanding beyond the original grid. When we need to exchange boundary data, these ghost cells expanding beyond the grid are updated according to their own boundary conditions.

By managing the ghost cells and boundary exchange process, the multi-GPU implementation ensures seamless communication between subdomains and the consistency of the solution across the whole grid.

6.3 Result and Conclusion

The performance impact of the multi-GPU implementation is significant, especially for large problem sizes. For the case with 1000×1000 grid and maximum iteration count of 500 (maxit = 500), the version with one A40 GPU uses about 10.75s while the version with 2 A40 GPUs use about 6s.

The multi-GPU implementation shows the scalability of the Poisson solver and the ability to leverage the power of multiple GPUs to solve computationally intensive problems. By efficiently using domain decomposition, ghost cells, and optimized inter-GPU communication, we can significantly reduce the overall runtime and enable the solver to handle larger problem sizes in a fraction of the time compared to a single-GPU approach.

7 Problems encountered

Initially after converting to PyTorch there was a problem that went unnoticed for a while. The code was running and we were able to benchmark GPU versus CPU. However, the residual term

$$\epsilon = \mathbf{b} - \mathbf{A}\mathbf{x}_\mathbf{k}$$

converged badly and plateaud after about 800 CG iterations with a big residue. The fact that the residue term did not decrease led us thinking the conjugate gradient method was not good enough. Therefore we attempted to solve this by trying other solvers and still it would not converge. It was not until a meeting with Lada, that we realized the issue was within the initialization of the stiffness matrix $\mathbf{A}$ and not the actual solver. After fixing this issue, the solution converged as expected.

However, when the grid size was large such as 1000x1000, CG was very unstable and slow to converge. The solution to this was to implement a preconditioner. We tried a Jacobi preconditioner and it solved the stability issue.

8 Implementation Improvements

Several key improvements were implemented to optimize the GPU-accelerated solver.

8.1 Sparse Matrix Operations

We implemented efficient sparse matrix construction and operations using PyTorch's sparse tensor capabilities. There are some key improvements.

- Efficient creation of sparse matrices using COO format
- Optimized matrix-vector multiplication using torch.sparse.mm

- Masked operations for boundary conditions to avoid unnecessary computations

8.2 Memory Management

Directly load data to GPU help us to minimize the data transfer between CPU and GPU.

```
x2d = torch.zeros((ni+1,nj+1), device=device)
```

This code also ensure the efficient use of some in-place to reduce memory allocation.

References

[1] L. Davidson, "pypoisson," 2024. [Online]. Available: https://www.tfd.chalmers.se/~lada/pyPoisson.html

[2] Z. H. Ma, L. Qian, D. M. Causon, H. B. Gu, and C. G. Mingham, "A cartesian ghost-cell multigrid poisson solver for incompressible flows," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 2, pp. 230–246, 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.2967

[3] Wikipedia contributors, "Five-point stencil — Wikipedia, the free encyclopedia," 2025, [Online; accessed 31-January-2025]. [Online]. Available: https://en.wikipedia.org/wiki/Five-point_stencil

Code

```
import torch
import numpy as np
import time
import socket
from scipy.sparse import spdiags, eye
import line_profiler as lp

profile_on = False
gpu = True

if profile_on:
    profile = lp.LineProfiler()
else:
    # If we are not profiling then define profile to
    ↪ just passthrough
    def profile(func):
```

```
16              return func
17
18  if gpu and torch.cuda.is_available():
19      device = torch.device('cuda')
20  elif not gpu:
21      device = torch.device('cpu')
22  else:
23      print("Cuda is not available!")
24      exit()
25
26  print(f"Using device: {device}")
27
28
29  def setup_case():
30      global convergence_limit_u, dist, fx, fy, imon,
            ↪ jmon, maxit, \
31      ni, nj, nsweep_u, solver_u, sormax, u_bc_east,
            ↪ u_bc_east_type, u_bc_north, u_bc_north_type,
            ↪  u_bc_south, u_bc_south_type, u_bc_west, \
32      u_bc_west_type, urf_u, viscos, vol, x2d, xp2d, y2d
            ↪ , yp2d, device
33
34      ############ section 4 fluid properties ############
35      viscos = 1/10
36
37      ############ section 5 relaxation factors
            ↪ ############
38      urf_u = 0.5
39
40      ############ section 6 number of iteration and
            ↪ convergence criterira ############
41      maxit = 100
42      sormax = 1e-20
43
44      solver_u = 'cg'  # Using PyTorch's conjugate
            ↪ gradient solver
45      nsweep_u = 50
46      convergence_limit_u = 1e-6
47
48      ############ section 7 monitoring point ############
49      imon = ni-10
50      jmon = int(nj/2)
51
52      ############ section 10 boundary conditions
            ↪ ############
53      # Boundary conditions for u (converted to tensors)
```

```
    u_bc_west = torch.zeros(nj, device=device)
    u_bc_east = torch.zeros(nj, device=device)
    u_bc_south = torch.zeros(ni, device=device)
    u_bc_north = torch.ones(ni, device=device)  # =1 at north boundary, y=1

    u_bc_west_type = 'd'
    u_bc_east_type = 'd'
    u_bc_south_type = 'd'
    u_bc_north_type = 'd'

    return

def init():
    print('hostname: ', socket.gethostname())

    global x2d, y2d, xp2d, yp2d, dist, fx, fy, vol, areaw, areas

    # Distance to nearest wall
    ywall_s = 0.5*(y2d[0:-1,0] + y2d[1:,0])
    dist_s = yp2d - ywall_s.unsqueeze(1)
    ywall_n = 0.5*(y2d[0:-1,-1] + y2d[1:,-1])
    dist_n = ywall_n.unsqueeze(1) - yp2d

    dist = torch.minimum(dist_s, dist_n)

    # West face coordinate
    xw = 0.5*(x2d[0:-1,0:-1] + x2d[0:-1,1:])
    yw = 0.5*(y2d[0:-1,0:-1] + y2d[0:-1,1:])

    del1x = ((xw-xp2d)**2 + (yw-yp2d)**2)**0.5

    del2x = ((xw-torch.roll(xp2d, 1, dims=0))**2 +
             (yw-torch.roll(yp2d, 1, dims=0))**2)**0.5
    fx = del2x/(del1x + del2x)

    # South face coordinate
    xs = 0.5*(x2d[0:-1,0:-1] + x2d[1:,0:-1])
    ys = 0.5*(y2d[0:-1,0:-1] + y2d[1:,0:-1])

    del1y = ((xs-xp2d)**2 + (ys-yp2d)**2)**0.5
    del2y = ((xs-torch.roll(xp2d, 1, dims=1))**2 +
             (ys-torch.roll(yp2d, 1, dims=1))**2)**0.5
    fy = del2y/(del1y + del2y)

```

```
    # Area calculations
    areawy = torch.diff(x2d, dim=1)
    areawx = -torch.diff(y2d, dim=1)
    areasy = -torch.diff(x2d, dim=0)
    areasx = torch.diff(y2d, dim=0)

    areaw = torch.sqrt(areawx**2 + areawy**2)
    areas = torch.sqrt(areasx**2 + areasy**2)

    # Volume calculation
    ax = torch.diff(x2d, dim=1)
    ay = torch.diff(y2d, dim=1)
    bx = torch.diff(x2d, dim=0)
    by = torch.diff(y2d, dim=0)

    areaz_1 = 0.5*torch.abs(ax[0:-1,:]*by[:,0:-1] - ay
        ↪ [0:-1,:]*bx[:,0:-1])
    areaz_2 = 0.5*torch.abs(ax[1:,:]*by[:,1:] - ay
        ↪ [1:,:]*bx[:,1:])
    vol = areaz_1 + areaz_2

    # Boundary coefficients
    as_bound = areas[:,0]**2/(0.5*vol[:,0])
    an_bound = areas[:,-1]**2/(0.5*vol[:,-1])
    aw_bound = areaw[0,:]**2/(0.5*vol[0,:])
    ae_bound = areaw[-1,:]**2/(0.5*vol[-1,:])

    return areaw, areawx, areawy, areas, areasx,
        ↪ areasy, vol, fx, fy, aw_bound, ae_bound,
        ↪ as_bound, an_bound, dist

@profile
def solve_2d(phi2d, aw2d, ae2d, as2d, an2d, su2d, ap2d
    ↪ , tol_conv, nmax):
    # Convert inputs to PyTorch tensors
    phi = torch.flatten(phi2d)
    aw = torch.flatten(aw2d)
    ae = torch.flatten(ae2d)
    as1 = torch.flatten(as2d)
    an = torch.flatten(an2d)
    ap = torch.flatten(ap2d)
    su = torch.flatten(su2d)

    # Build sparse tensor
    n = ni * nj
    indices = []
```

```
    values = []

    # Main diagonal
    diag_idx = torch.arange(n, device=device)
    indices.append(torch.stack([diag_idx, diag_idx]))
    values.append(ap)

    # West coefficient (aw)
    i = torch.arange(1, n, device=device)
    j = i - 1
    mask = (i % nj != 0)
    i, j = i[mask], j[mask]
    indices.append(torch.stack([i, j]))
    values.append(aw[1:][mask])

    # East coefficient (ae)
    i = torch.arange(0, n-1, device=device)
    j = i + 1
    mask = ((i+1) % nj != 0)
    i, j = i[mask], j[mask]
    indices.append(torch.stack([i, j]))
    values.append(ae[:-1][mask])

    # South coefficient (as)
    i = torch.arange(nj, n, device=device)
    j = i - nj
    indices.append(torch.stack([i, j]))
    values.append(as1[nj:])

    # North coefficient (an)
    i = torch.arange(0, n-nj, device=device)
    j = i + nj
    indices.append(torch.stack([i, j]))
    values.append(an[:-nj])

    # Concatenate all indices and values
    indices = torch.cat(indices, dim=1)
    values = torch.cat(values)

    # Create sparse tensor
    A = torch.sparse_coo_tensor(indices, values, (n,n)
       ↪ , device=device)

    # Define matrix-vector product function for
       ↪ conjugate gradient
    @profile
```

```
183     def mv(v):
184         return torch.sparse.mm(A, v.unsqueeze(1)).
               ↪ squeeze(1)
185 
186     # Conjugate gradient solver
187     @profile
188     def conjugate_gradient(A_mv, b, x0, tol=1e-10,
           ↪ max_iter=1000):
189         x = x0
190         r = b - A_mv(x)
191         p = r
192         r_norm_sq = r @ r
193 
194         for _ in range(max_iter):
195             Ap = A_mv(p)
196             alpha = r_norm_sq / (p @ Ap)
197             x = x + alpha * p
198             r_new = r - alpha * Ap
199             r_norm_sq_new = r_new @ r_new
200             beta = r_norm_sq_new / r_norm_sq
201             r = r_new
202             r_norm_sq = r_norm_sq_new
203 
204             if r_norm_sq < tol:
205                 break
206 
207             p = r + beta * p
208 
209         return x
210 
211     # Initial guess
212     x0 = torch.zeros_like(su)
213 
214     # Solve using conjugate gradient
215     phi = conjugate_gradient(mv, su, x0, tol=tol_conv,
           ↪  max_iter=nmax)
216 
217     # Reshape solution back to 2D
218     phi2d = phi.reshape(ni, nj)
219 
220     # Calculate residual
221     resid = torch.norm(mv(phi) - su)
222 
223     return phi2d, resid
224 
225 
```

```
226 def save_data(u2d):
227     print('save_data called')
228     # Convert tensor to numpy before saving
229     u_numpy = u2d.cpu().numpy()
230     np.save('u2d_saved', u_numpy)
231     return
232 
233 # Main execution
234 if __name__ == "__main__":
235     # Load grid data and convert to tensors
236     datax = torch.from_numpy(np.loadtxt("x2d.dat")).to
        ↪ (device)
237     x = datax[0:-1]
238     ni = int(datax[-1].item())
239 
240     datay = torch.from_numpy(np.loadtxt("y2d.dat")).to
        ↪ (device)
241     y = datay[0:-1]
242     nj = int(datay[-1].item())
243 
244     # Initialize tensors on GPU
245     x2d = torch.zeros((ni+1,nj+1), device=device)
246     y2d = torch.zeros((ni+1,nj+1), device=device)
247 
248     x2d = x.reshape(ni+1,nj+1)
249     y2d = y.reshape(ni+1,nj+1)
250 
251     # Compute cell centers
252     xp2d = 0.25*(x2d[0:-1,0:-1] + x2d[0:-1,1:] + x2d
        ↪ [1:,0:-1] + x2d[1:,1:])
253     yp2d = 0.25*(y2d[0:-1,0:-1] + y2d[0:-1,1:] + y2d
        ↪ [1:,0:-1] + y2d[1:,1:])
254 
255     # Initialize solution variables as tensors
256     u2d = torch.ones((ni,nj), device=device) * 1e-20
257 
258     setup_case()
259     areaw, areawx, areawy, areas, areasx, areasy, vol,
        ↪  fx, fy, aw_bound, ae_bound, as_bound,
        ↪ an_bound, dist = init()
260 
261 @profile
262 def coeff():
263     """GPU version of coefficient calculation"""
264     # Initialize viscosity tensors on GPU
```

```
265     visw = torch.ones((ni+1,nj), device=device) *
            ↪ viscos
266     viss = torch.ones((ni,nj+1), device=device) *
            ↪ viscos
267
268     # Initialize volume tensors
269     volw = torch.ones((ni+1,nj), device=device) * 1e
            ↪ -10
270     vols = torch.ones((ni,nj+1), device=device) * 1e
            ↪ -10
271
272     # Calculate volumes and diffusion coefficients
273     volw[1:,:] = 0.5 * torch.roll(vol, -1, dims=0) +
            ↪ 0.5 * vol
274     diffw = visw[0:-1,:] * areaw[0:-1,:]**2 / volw
            ↪ [0:-1,:]
275
276     vols[:,1:] = 0.5 * torch.roll(vol, -1, dims=1) +
            ↪ 0.5 * vol
277     diffs = viss[:,0:-1] * areas[:,0:-1]**2 / vols
            ↪ [:,0:-1]
278
279     # Set coefficients
280     aw2d = diffw
281     ae2d = torch.roll(diffw, -1, dims=0)
282     as2d = diffs
283     an2d = torch.roll(diffs, -1, dims=1)
284
285     # Zero out boundary coefficients
286     as2d[:,0] = 0
287     an2d[:,-1] = 0
288
289     # Initialize source terms
290     su2d = torch.zeros((ni,nj), device=device)
291     sp2d = torch.zeros((ni,nj), device=device)
292
293     if iter == 0:
294         print('aw[5,5],ae,as,an', aw2d[5,5].item(),
                ↪ ae2d[5,5].item(),
295               as2d[5,5].item(), an2d[5,5].item())
296
297     return aw2d, ae2d, as2d, an2d, su2d, sp2d
298
299 @profile
300 def calcu(su2d, sp2d, aw2d, ae2d, as2d, an2d):
```

```
301     """GPU version of u-momentum source terms
            ↪ calculation"""
302 
303     # Add sources and modify source terms
304     su2d, sp2d = modify_u(su2d, sp2d)
305 
306     # Calculate ap coefficient
307     ap2d = aw2d + ae2d + as2d + an2d - sp2d
308 
309     # Apply under-relaxation
310     ap2d = ap2d / urf_u
311     su2d = su2d + (1-urf_u) * ap2d * u2d
312 
313     return su2d, sp2d, ap2d
314 
315 @profile
316 def bc(su2d,sp2d,phi_bc_west,phi_bc_east,phi_bc_south,
        ↪ phi_bc_north\
317       ,phi_bc_west_type,phi_bc_east_type,
            ↪ phi_bc_south_type,phi_bc_north_type):
318 
319     su2d = torch.zeros((ni,nj), device=device)
320     sp2d = torch.zeros((ni,nj), device=device)
321 
322     #south
323     if phi_bc_south_type == 'd':
324         sp2d[:,0] = sp2d[:,0] - viscos*as_bound
325         su2d[:,0] = su2d[:,0] + viscos*as_bound*
                ↪ phi_bc_south
326 
327     #north
328     if phi_bc_north_type == 'd':
329         sp2d[:,-1] = sp2d[:,-1] - viscos*an_bound
330         su2d[:,-1] = su2d[:,-1] + viscos*an_bound*
                ↪ phi_bc_north
331 
332     #west
333     if phi_bc_west_type == 'd':
334         sp2d[0,:] = sp2d[0,:] - viscos*aw_bound
335         su2d[0,:] = su2d[0,:] + viscos*aw_bound*
                ↪ phi_bc_west
336 
337     #east
338     if phi_bc_east_type == 'd':
339         sp2d[-1,:] = sp2d[-1,:] - viscos*ae_bound
```

```
340         su2d[-1,:] = su2d[-1,:] + viscos*ae_bound*
                ↪ phi_bc_east
341
342     return su2d, sp2d
343
344 @profile
345 def modify_u(su2d, sp2d):
346     # Add a point source/volume of 100 at x = 1.5 and
            ↪ y = 0.5
347     xx = 1.5
348
349     i1 = torch.argmin(torch.abs(xx-xp2d[:,1]))
350     yy = 0.5
351     j1 = torch.argmin(torch.abs(yy-yp2d[1,:]))
352     su2d[i1,j1] = su2d[i1,j1] + 100*vol[i1,j1]
353
354     return su2d, sp2d
355
356 @profile
357 def main_loop(u2d):
358     cumulative_time = 0
359     # Main iteration loop
360     for iter in range(maxit):
361         start_time_iter = time.time()
362         start_time = time.time()
363         # Compute coefficient matrices
364         aw2d, ae2d, as2d, an2d, su2d, sp2d = coeff()
365
366         # Apply boundary conditions for u2d
367         su2d, sp2d = bc(su2d, sp2d, u_bc_west,
                ↪ u_bc_east, u_bc_south, u_bc_north,
368                         u_bc_west_type, u_bc_east_type
                            ↪ , u_bc_south_type,
                            ↪ u_bc_north_type)
369
370         # Calculate source terms for u momentum
                ↪ equation
371         su2d, sp2d, ap2d = calcu(su2d, sp2d, aw2d,
                ↪ ae2d, as2d, an2d)
372
373         # Solve for u using conjugate gradient method
374         if maxit > 0:
375             u2d, residual_u = solve_2d(u2d, aw2d, ae2d
                    ↪ , as2d, an2d, su2d, ap2d,
376                                        convergence_limit_u
                                           ↪ , nsweep_u)
```

```
377
378         # Print computation time for u
379         print(f"{'time u: '}{time.time()-start_time:.2
                ↪ e}")
380         cumulative_time += time.time()-start_time
381
382         # Monitor convergence
383         if iter % 10 == 0:
384             print(f"\n--iter:{iter:d}, residual:{
                    ↪ residual_u:.2e}\n")
385             print(f"\nmonitor iteration:{iter:4d}, u:{
                    ↪ u2d[imon,jmon].item(): .2e}\n")
386
387         # Calculate maximum velocity for stability
                ↪ monitoring
388         umax = torch.max(u2d).item()
389         print(f"\n---iter: {iter:2d}, umax: {umax:.2e
                ↪ }\n")
390
391         # Print iteration timing
392         print(f"time one iteration: {time.time()-
                ↪ start_time_iter:.2e}")
393
394         # Check convergence
395         if residual_u < sormax:
396             break
397     return cumulative_time
398
399 total_time = main_loop(u2d)
400
401 # Print and save profiler results
402 if profile_on:
403     profile.print_stats()
404     with open('profile_results.txt', 'w') as f:
405         profile.print_stats(stream=f)
406 # Save final results
407 save_data(u2d)
408 print('program reached normal stop')
409 print(f"{'Cumulative time: '}{total_time}")
```