Solving Poisson problems using discrete methods.

September 26, 2023

Kelon Franklin
Student Number - 18020884

Contents

[1]:
```
import numpy as np
import math
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import spsolve
import timeit
import time

import matplotlib.pyplot as plt
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

import numba
from numba import cuda
```

0.1 1. Introducing discretisation and the Poisson problem

This notebook deals with solving the Poisson problem in two dimensions with an arbitrary f, and boundary conditions defined by another function g. For this notebook, these two functions will be considered as Python callables. The Poisson problem takes the following form.

$$-\Delta u(x,y) = f(x,y)$$

with $\Delta := u_{xx} + u_{yy}$ for $(x,y) \in \Omega \subset \mathbb{R}^2$ and boundary conditions $u(x,y) = g(x,y)$ on $\Gamma := \partial\Omega$. For our investigation we're considering the unit square domain $\Omega := [0,1]^2$. Grid points in this plane are defined by $x_i = ih$ and $y_j = jh$, where $i,j = 1,2,....,N$ and h is the grid spacing denoted by $h = \frac{1}{N+1}$.

To solve this problem we apply the discretised formula of the Laplace operator Δ:

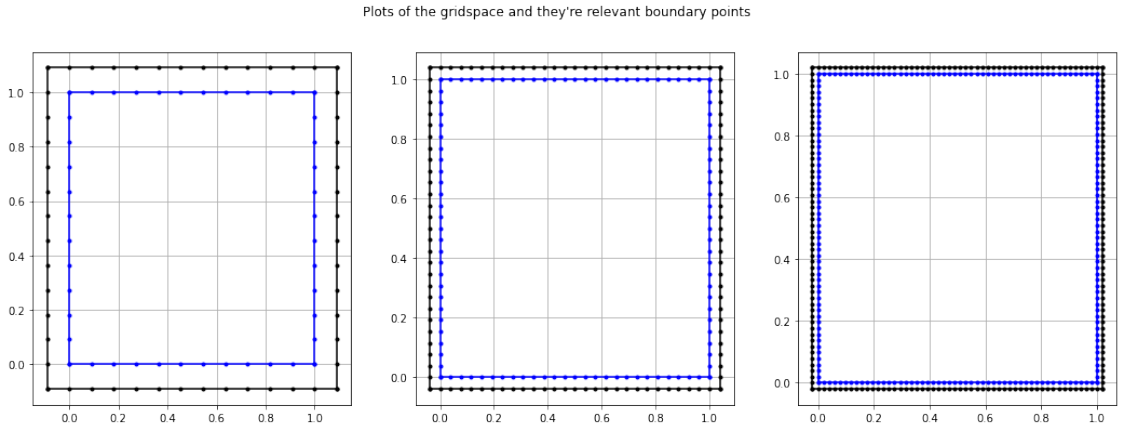
$$-\Delta u(x_i, y_j) \approx \frac{1}{h^2}(4u(x_i, y_j) - u(x_{i-1}, y_j) - u(x_{i+1}, y_j) - u(x_i, y_{j-1}) - u(x_i, y_{j+1})).$$

So, our problem is now discretised, using (x_i, y_j)'s neighbouring points to calculate it's relevant value. When one of the neighbouring points is found at the boundary of the domain, then the value of u within the domain is replaced by the value of the point given by the boundary conditions.

[2]:
```
fig = plt.figure(figsize=(18,6))


def grid_plot(N):
  """
  Plotting unit square grids with different point spacings, depending on N the␣
 ↪number of points along each axes.
  Also plots the positions of boundary points outside of the unit square that␣
 ↪affect edge cases in the iterative Poisson problem.
  ============================================================
  Input:

  N - the number of points along each edge of the unit square separated by grid␣
 ↪spacing h=1/(N+1).
  ============================================================
  Output:

  Plots a grid of the unit square and its relavant boundary points.
  ============================================================
  """
  h = 1/(N+1)

  # unit square separated into 1/N+1 points.
  points = np.arange(-h,1+2*h,h)

  plt.plot(points,np.full(len(points),-h), '.-k')
  plt.plot(points,np.full(len(points),1+h), '.-k')
  plt.plot(np.full(len(points),-h), points, '.-k')
  plt.plot(np.full(len(points),1+h), points, '.-k')

  # grid points that we're consider based on the allowed values for i and j.
  points = np.arange(0,(N+1)*h+h,h)

  plt.plot(points,np.zeros(len(points)), '.-b')
  plt.plot(points,np.ones(len(points)), '.-b')
  plt.plot(np.zeros(len(points)), points, '.-b')
  plt.plot(np.ones(len(points)), points, '.-b')

  plt.grid()
```

```
N= np.array([10,25,50])

for i in range(len(N)):
  fig.add_subplot(1,3,i+1)
  grid_plot(N[i])

plt.suptitle("Plots of the gridspace and they're relevant boundary points")
```

[2]: `Text(0.5, 0.98, "Plots of the gridspace and they're relevant boundary points")`



The above plot shows the domain we're working with and discretising in blue, and the positions of the boundary points in black.

0.2 2. Solving the discretised Poisson problem using Sparse Matrices

Now that the problem has been arranged, we can look to the use of sparse matrices to solve it.
By describing our system as a set of N^2 equations, where each equation focuses on a single point in the domain (x_i, y_j) and it's relevant neighbouring points.
The system is governed by the equation retrieved from the original Poisson problem:

$$\frac{1}{h^2}(4u(x_i, y_j) - u(x_{i-1}, y_j) - u(x_{i+1}, y_j) - u(x_i, y_{j-1}) - u(x_i, y_{j+1})) = f(x_i, y_j)$$

This set of equations can be organsied into a linear system consisting of a sparse matrix A of dimensions (N^2, N^2), a vector x of dimensions N^2 where each element in the vector corresponds to an element in the domain, and a vector b of dimensions N^2 whose elements contain evaluations of the function f at each of the points in the domain.
Domain points are translated into vector form through the relation $u_{i,j} = x_{jN+i}$, and each row contains information unique to the $(jN + i)^{th}$ point in the domain. Here we will implement a function to accurately discretise the domain depending on the grid spacings defined by h and use the function *scipy.sparse.linalg.spsolve* to solve the linear system.

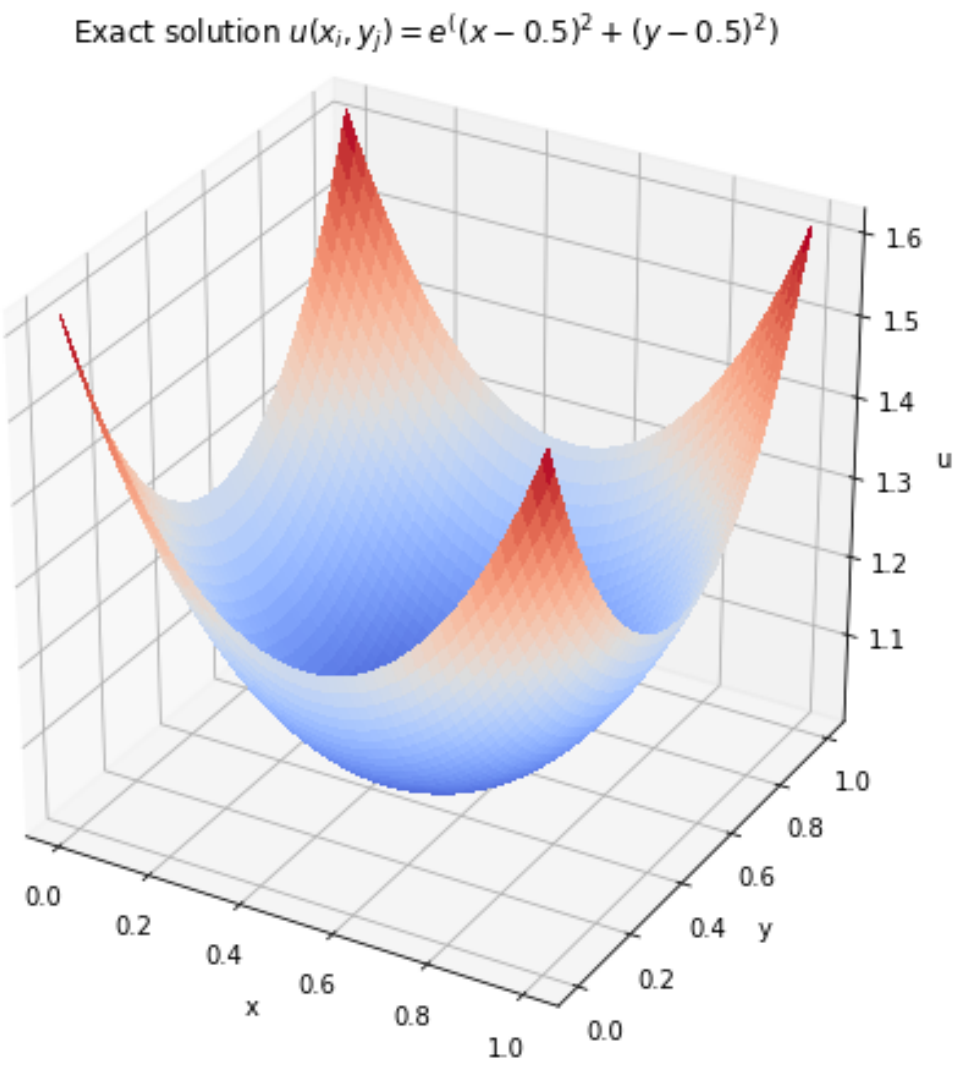
[3]:
```
def u_exact(i,j,N):
    """
    The manufactured solution for u in the Poisson problem that is valid for␣
 ↪all points in the domain.
    ============================================================
    Inputs:

    i - the x coordinate of the current point on the grid.
    j - the y coordinate of the current point on the grid.
    ============================================================
    Output:

    u - the manufactured solution of the point (i,j).
    ============================================================
    """

    h = 1/(N+1) # discreisation dimension of the [0,1]^2 domain

    u = np.exp(((i*h-0.5)**2) + ((j*h-0.5)**2))

    return u

def f(u,i,j,N):
    """
    The total iterative formula for the Laplace operator acting on the function␣
 ↪u in Cartesian coordinates.
    ============================================================
    Inputs:

    i - the x coordinate for the current point on the grid.
    j - the y coordinate for the current point on the grid.
    u - the manufactured solution of u.
    ============================================================
    Output:

    f - the RHS of the Poisson problem evaluated at point (i,j).
    ============================================================
    """

    h = 1/(N+1) # discreisation dimension of the [0,1]^2 domain

    f = (-4) * np.exp((i*h-0.5)**2 + (j*h-0.5)**2) * ((i*h)**2 - (i*h) +␣
 ↪(j*h)**2 - (j*h) + 1.5)

    return f

def Poisson_discretised(f,g,N):
```

```
    """
    Generate the Sparse Matrix and associated vector containing evaluated RHS␣
↪terms for a Poisson problem.
    ============================================================
    Inputs:

    f - The RHS of the Poisson problem.
    g - The boundary condition at the edge of the unit square domain.
    N - the spacing of points in the interior subsquare of points within the␣
↪total domain.
    ============================================================
    Outputs:

    A - Coordinate form of Sparse Matrix for the current problem.
    b - RHS vector of the problem.
    ============================================================
    """

    h = 1/(N+1) # discreisation dimension of the [0,1]^2 domain

    nelements = (5 * (N**2) - (4 * N)) # the number of non-zero elements

    row_ind = np.empty(nelements, dtype=np.float64) # row and column locations␣
↪of the non-zero entries in the matrix A.
    col_ind = np.empty(nelements, dtype=np.float64)
    data = np.empty(nelements, dtype=np.float64) # non-zero elements in the␣
↪matrix A

    b = np.empty((N) ** 2, dtype=np.float64) # RHS vector which is 1 for␣
↪interior values and 0 for boundary values.

    count = 0

    for j in range(1,N+1):
      for i in range(1,N+1):

        row_ind[count] = (j-1)*N + i-1 # defining the row index for this i,j
        col_ind[count] = (j-1)*N + i-1 # defining the column point␣
↪corresponding to u_ij, each subsequent element for this row, contains the␣
↪terms in Delta u(x_i,y_j)

        data[count] = 4/(h**2) # u_ij elements have coefficient -4/h^2 where h␣
↪= 1/(N-1)

        b[(j-1)*N + (i-1)] = f(g,i,j,N)
```

```
        count += 1

        if (i+1) > N:
          # condition for the right point being at the boundary condition

          b[(j-1)*N + (i-1)] += g((i+1),j,N)/(h**2)

        else:
          row_ind[count] = (j-1)*N + (i-1)
          col_ind[count] = (j-1)*N + (i-1+1)
          data[count] = -1/(h**2)
          count += 1

        if (i-1) < 1:
          # condition for the left point being at the boundary condition

          b[(j-1)*N + (i-1)] += g((i-1),j,N)/(h**2)

        else:
          row_ind[count] = (j-1)*N + (i-1)
          col_ind[count] = (j-1)*N + (i-1-1)
          data[count] = -1/(h**2)
          count += 1

        if (j+1) > N:
          # condition for the top point being at the boundary condition

          b[(j-1)*N + (i-1)] += g(i,(j+1),N)/(h**2)

        else:
          row_ind[count] = (j-1)*N + (i-1)
          col_ind[count] = (j-1+1)*N + (i-1)
          data[count] = -1/(h**2)
          count += 1


        if (j-1) < 1:
          # condition for the bottom point being at the boundary condition

          b[(j-1)*N + (i-1)] += (g(i, (j-1),N)/(h**2))

        else:
          row_ind[count] = (j-1)*N + (i-1)
          col_ind[count] = (j-1-1)*N + (i-1)
          data[count] = -1/(h**2)
          count += 1
```

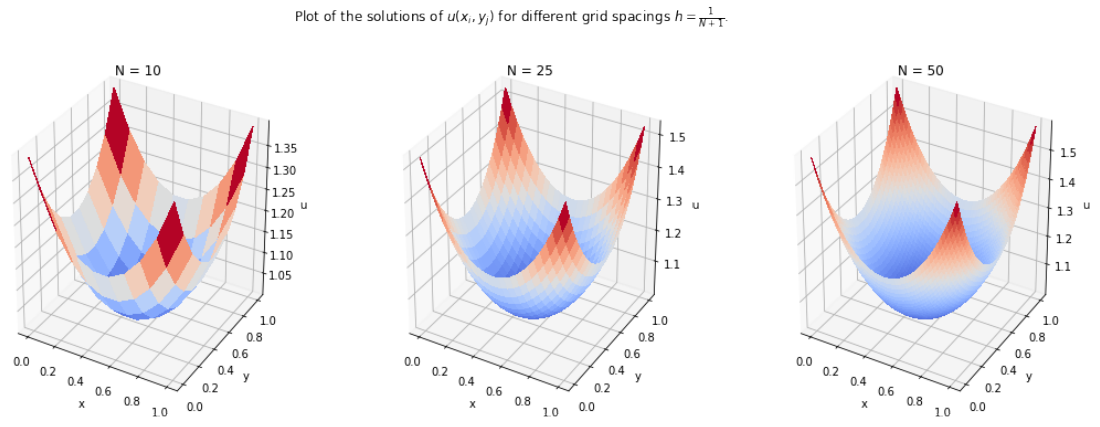
```
    return coo_matrix((data, (row_ind, col_ind)), shape=(N**2, N**2)).tocsr(), b

# We have a grid of unknowns in the NxN grid of u_ij elements.
# We map this grid of unknowns into a column vector x that contains the␣
 ↪elements of the unknowns. The (j*N+i)th element of x corresponds to the␣
 ↪(i,j)th element in the grid containing u values.
# Since we map N^2 unknowns into the vector x, the matrix A must be of␣
 ↪dimensions N^2 * N^2.
```

[4]:
```
# plotting the exact solution of the problem.
N = 100
u_true = np.zeros((N,N))

for j in range(1,N+1):
  for i in range(1,N+1):
    u_true[i-1,j-1] = u_exact(i,j,N)

fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, u_true, antialiased=False, cmap=cm.coolwarm)
ax.set_title(r'Exact solution $u(x_{i},y_{j}) = e^((x-0.5)^2 + (y-0.5)^2)$',␣
 ↪pad=20)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('u')
plt.show()
```



[5]:
```
# visual plot of the solutions
%matplotlib inline

N_values = np.array([10,25,50])
fig = plt.figure(figsize=(18, 6))
i = 1
for N in N_values:
  A, b = Poisson_discretised(f,u_exact,N)
  u_sol = spsolve(A,b)

  u_sol = u_sol.reshape((N, N))

  ax = fig.add_subplot(1,3,i,projection='3d')
```

```
  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('u')
  ax.set_title('N = {}'.format(N))
  ticks= np.linspace(0, 1, N)
  X, Y = np.meshgrid(ticks, ticks)
  surf = ax.plot_surface(X, Y, u_sol, antialiased=False, cmap=cm.coolwarm)
  i += 1

plt.suptitle(r'Plot of the solutions of $u(x_{i},y_{j})$ for different grid␣
 ↪spacings $h = \frac{1}{N+1}$.')
plt.show()
```

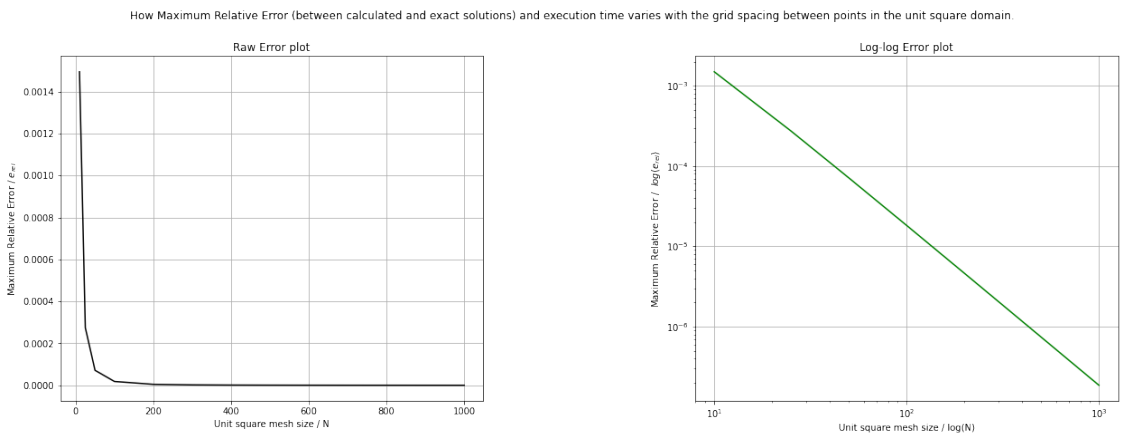


Above we have plots of the solutions of u, for different values of N. We can see that as the grid spacing gets smaller, the solution plane becomes more smoothly defined.

[6]:
```
# using a sparse matrix solver to solve the problem.

N_values = np.array([10,25,50,100,200,300,400,500,600,700,800,900,1000]) # the␣
 ↪dimensions I want to test the discretised function for.
e_rel = []
times = []

for N in N_values:

  A, b = Poisson_discretised(f,u_exact,N) # creating solutions
  u_sol = spsolve(A, b)
  u_sol = u_sol.reshape((N,N))

  # calculating and plotting the relative error between our sparse linear␣
 ↪solver solutions and the manufactured solution u_exact.
```

```
  u_true = np.zeros((N,N))

  for j in range(1,N+1):
    for i in range(1,N+1):
      u_true[i-1,j-1] = u_exact(i,j,N)

  err = np.amax(np.abs(u_sol - u_true)/np.abs(u_true)) # calculating and␣
↪storing errors
  e_rel.append(err)
```

[7]:
```
%matplotlib inline

N_values = np.array([10,25,50,100,200,300,400,500,600,700,800,900,1000])

fig = plt.figure(figsize=(21,7))

plt.suptitle('How Maximum Relative Error (between calculated and exact␣
↪solutions) and execution time varies with the grid spacing between points in␣
↪the unit square domain.')

ax1 = fig.add_subplot(1,2,1)
ax1.plot(N_values, e_rel, '-k')
ax1.set_title('Raw Error plot')
ax1.set_ylabel(r'Maximum Relative Error / $e_{rel}$')
ax1.set_xlabel('Unit square mesh size / N')
ax1.grid()

ax2 = fig.add_subplot(1,2,2)
ax2.loglog(N_values, e_rel, '-g')
ax2.set_title('Log-log Error plot')
ax2.set_ylabel(r'Maximum Relative Error /  $log(e_{rel})$')
ax2.set_xlabel('Unit square mesh size / log(N)')
ax2.grid()

plt.subplots_adjust(wspace=0.5)
plt.show()
```



After executing the sparse linear solver on my discretisation of the problem, we can see that, as expected, the relative error decreases rapidly as the grid spacings decrease, and the discrete system is closer approximated to the continuous system.
From the second graph we can see that the relationship between the relative error, e_{rel} and the mesh size N has an inverse relationship of the form.

$$e_{rel} \propto e^{\frac{1}{log(N)}}$$

0.3 3. Iterative Poisson solver accelerated with GPU computing

An alternative method of solving the Poisson problem is the Jacobi iterative method. This method follows on from the sparse system and is of the form:

$$u^{k+1}(x_i, y_j) = \frac{1}{4}\Big(h^2 f(x_i, y_j) + u^k(x_{i-1}, y_j) + u^k(x_{i+1}, y_j) + u^k(x_i, y_{j-1}) + u^k(x_i, y_{j+1}))\Big)$$

This method is a five-point average of the current iteration of grid-points used to update each individual grid-point in the next iteration.
To implement this method in Python, we will be looking at the ways in which CUDA can be used to parallelise the basic operations that will be applied to each element in the array of solutions.
The possibility for us to accelerate this iterative method using GPU units is available because of the simple computations that are being repeated over multiple points. These computations will allows us to utilise the architecture of the GPU maximally, resulting in parallelisation of our algorithm and speed up of our execution times for the entire solving process.

0.3.1 3.1 GPU Acceleration utilising GPU device memory

In this section, we will be utilising the device memory of the GPU to accelerate the solving process. Device memory is memory unique to the GPU, in which we can store the data that we're processing on the GPU units. Using device memory over the CPU memory can increase the execution times of any processes run on our data because of the reduction in overheads of data having to be pulled from the CPU memory, which is a long and timely process. Device memory keeps the data we require at hand for quick use across the GPU.

[8]:
```
@cuda.jit
def Poisson_device_kernel(u_k, u_kp1, f_array, N):
  """
  GPU Kernel to update the grid points using an interative.
  ================================================
  Inputs:

  u_k - an array of current values of u(i,j) on the grid.
  u_kp1 - an array to hold the next iteration of u(i,j) values in the grid.
  f_array - an array containing the evaluated RHS for all interior subsquare␣
 ↪points (all i and j between [1,N]) in the domain.
  N - number of grid points along the subsquare axes.
  ================================================
  Outputs:

  Updates u_kp1 based on the current values of u_k in the grid.
  ================================================
  """

  h = 1/(N+1)

  i, j = cuda.grid(2) # calling the global indices of the data.

  #  implementation of the iterative scheme

  if 0 < i < (N+1) and 0 < j < (N+1): # defining each neighbouring element from␣
 ↪the previous iteration used to update the element of the current iteration.

    up = u_k[i, j+1]
    down = u_k[i,j-1]
    left = u_k[i-1,j]
    right = u_k[i+1,j]

    u_kp1[i,j] = 0.25*((h**2)*f_array[i-1,j-1] + up + down + left + right)

  # Functions for creating relevant boundary and RHS arrays.

def initial_conditions(g,N):
  """
  Creates an array that holds the boundary conditions of the grid domain, and␣
 ↪the initialisation values for interior points.
  ================================================
  Inputs:

  g - Python callable function: boundary condition function.
  N - number of points along the x and y axes between the boundaries.
  ================================================
```

```
  Outputs:

  b - boundary condition array.
  ============================================
  """

  b = np.zeros((N+2,N+2), dtype=np.float32)

  for i in range(N+2): # initialising boundary conditions along each boundary.
    b[i,0] = g(i,0,N)
    b[i,N+1] = g(i,N+1,N)

  for j in range(N+2):
    b[0,j] = g(0,j,N)
    b[N+1,j] = g(N+1,j,N)

  return b


def RHS_array(f,N):
  """
  Creates an array holding the RHS of the Poisson problem for each interior,␣
 ↪subsquare point we're updating.
  ============================================
  Inputs:

  f - Python callable function: RHS function of the Poisson problem.
  N - number of points along the x and y axes excluding boundary points.
  ============================================
  Outputs:

  arr - array containing evaluations of the RHS Poisson problem function f at␣
 ↪interior points on the grid.
  ============================================
  """

  array = np.zeros((N,N))

  for i in range(N):
    for j in range(N):
      array[i,j] = f(u_exact,i+1,j+1,N) # calculating the function at each grid␣
 ↪point

  return array

def iterate_device_kernel(boundary_function, RHS_function, N, iterations,␣
 ↪check):
```

```
    """
    Iterates of the Poisson solving kernel to update the solutions across the␣
↪current grid.
    ================================================
    Inputs:

    boundary_function - Python callable function: determines the conditions at␣
↪the boundary points.
    RHS_function - Python callable function: determines the f function on the␣
↪Poisson problem.
    N - number of points along the x and y axes between the boundaries.
    iterations - number of times the CUDA kernel will be called to execute the␣
↪operation.
    check - the interval between iterations for which convergence between the␣
↪previous and current iteration will be checked.
    ================================================
    Outputs:

    updated_u - the final solution for u after the iterative scheme has been␣
↪implemented for the specified number of iterations.
    e_rel - the relative errors calculated between u_k and u_kp1 every 'check'␣
↪iterations.
    times - time taken to run the kernel for the current iteration.
    ================================================
    """

    # Initialising arrays.

    u_k = initial_conditions(boundary_function, N)
    f_array = RHS_array(RHS_function, N)

    u_k_device = cuda.to_device(u_k) # initialising grid.
    f_array_device = cuda.to_device(f_array) # initialising RHS at all grid points
    u_kp1_device = cuda.to_device(u_k) # initialising array for updated values

    # Initialising CUDA parameters.

    bd = 32 # number of threads in a block
    gd = (u_k_device.shape[0] + bd - 1) // bd # dimension of the grid of of␣
↪blocks.

    # Executing iterative scheme.

    # creating arrays for storing the error between consecutive iterations, which␣
↪is checked every 'check' iterations.
```

```
  u_true = np.zeros((N,N))

  for j in range(1,N+1):
    for i in range(1,N+1):
      u_true[i-1,j-1] = u_exact(i,j,N)

  e_rel = []
  convergence = []
  times = []

  for run in range(iterations):

    if run%2 == 0:
      Poisson_device_kernel[(gd, gd),(bd, bd)](u_k_device, u_kp1_device,␣
↪f_array_device, N)
      if run%check == 0:
        u_kp1 = u_kp1_device.copy_to_host() # copying k and k+1 iterations of u␣
↪back to the host for error and convergence calculations
        u_k = u_k_device.copy_to_host()

        err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true)/np.abs(u_true))
        conv = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_k[1:N+1,1:N+1])/np.
↪abs(u_k[1:N+1,1:N+1]))
        e_rel.append(err)
        convergence.append(conv)

    else:
      Poisson_device_kernel[(gd, gd),(bd, bd)](u_kp1_device, u_k_device,␣
↪f_array_device, N)
      if run%check == 0:
        u_kp1 = u_kp1_device.copy_to_host() # copying k and k+1 iterations of u␣
↪back to the host for error and convergence calculations
        u_k = u_k_device.copy_to_host()

        err = np.amax(np.abs(u_k[1:N+1,1:N+1] - u_true)/np.abs(u_true))
        conv = np.amax(np.abs(u_k[1:N+1,1:N+1] - u_kp1[1:N+1,1:N+1])/np.
↪abs(u_kp1[1:N+1,1:N+1]))
        e_rel.append(err)
        convergence.append(conv)

  return np.array(e_rel),  np.array(convergence)
```

[9]:
```
%matplotlib inline

N = np.array([64, 128, 256, 512])
iterations = 45000
check = 450
```

```
x = np.arange(0,iterations,check)

fig = plt.figure(figsize=(24,7))
ax1 = fig.add_subplot(1,2,1)
ax3 = fig.add_subplot(1,2,2)

for n in N:
  device_error, device_conv = iterate_device_kernel(u_exact, f, n, iterations,␣
 ↪check)
  ax1.plot(x, device_error, label='N={}'.format(n))
  ax3.plot(x, device_conv, label='N={}'.format(n))

ax1.set_xlabel('Iteration')
ax1.set_ylabel(r'Maximum Relative Error $\frac{|u^{k+1} -␣
 ↪u_{exact}|}{|u_{exact}|}$')
ax1.set_title('Raw error plot')
ax1.grid()

ax3.set_xlabel('Iteration')
ax3.set_ylabel(r'Convergence Rate between consecutive iterations␣
 ↪$\frac{|u^{k+1} - u^{k}|}{|u^{k}|}$')
ax3.set_title('Convergence Rate')
ax3.grid()
ax3.legend()

plt.suptitle('Error and convergence rate for the Poisson problem GPU␣
 ↪implementation using device memory only.')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:144:
RuntimeWarning: divide by zero encountered in true_divide
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:144:
RuntimeWarning: divide by zero encountered in true_divide
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:144:
RuntimeWarning: divide by zero encountered in true_divide
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:144:
RuntimeWarning: divide by zero encountered in true_divide
```

[9]:
```
Text(0.5, 0.98, 'Error and convergence rate for the Poisson problem GPU
implementation using device memory only.')
```

In comparison to the error graphs of the linear algebra method of solving the Poisson system, we see that relative error decreases at various rates depending on the dimensions of the grid. For small N very few iterations are needed for relative error to converge towards zero, however for $N = 512$, much more iterations that those run are required for our calculated solution to converge.
The convergence figure also shows us that for values $N = 512$ the convergence is very unsteady for fewer iterations, showing that the solution is quite unstable until after 20000 iterations, after which there is a steady convergence rate and decrease in error.

[10]:
```
# Writing functions to record the number of iterations taken for a certain mesh␣
↪size N before the maximum relative error of the operation falls below

def GPU_min_error(boundary_function, RHS_function, N, bd):
  """
  Determining the minimum number of iterations and the execution time required␣
↪to repeat the iterative scheme and update the grid to have a relative error␣
↪below 1%
  ========================================================
  Inputs:

  boundary_function - Python callable function defining the boundary conditions␣
↪at the edges of the domain.
  RHS_function - Python callable function defining the RHS, f(x_i,y_j) of the␣
↪Poisson problem.
  N - the number of points along the x and y axes between the boundaries.
  bd - threads per block
  ========================================================
  Outputs:

  iters - the number of iterations required for relative error to become lower␣
↪than 1%.
  interval - time taken for iterative scheme to decrease error below 1%.
  """
```

```
 u_k_device = cuda.to_device(initial_conditions(boundary_function, N))
 u_kp1_device = cuda.to_device(initial_conditions(boundary_function, N))
 f_array_device = cuda.to_device(RHS_array(RHS_function, N))

 gd = (u_k_device.shape[0] + bd - 1) // bd # dimension of the grid of of␣
↪blocks.

 # Executing iterative scheme.

 # creating arrays for storing the error between consecutive iterations, which␣
↪is checked every 'check' iterations.

 u_true = np.zeros((N,N))

 for j in range(1,N+1):
   for i in range(1,N+1):
     u_true[i-1,j-1] = u_exact(i,j,N)

 iters = 0

 # initialising error for the while loop
 u_kp1 = u_kp1_device.copy_to_host()

 err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true) / np.abs(u_true))

 # defining the start time before the algorithm runs

 start = time.time()

 while (err > 0.01):
   if iters%2 == 0:
     Poisson_device_kernel[(gd,gd),(bd,bd)](u_k_device, u_kp1_device,␣
↪f_array_device, N) # update grid
     iters += 1

     u_kp1 = u_kp1_device.copy_to_host()
     err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true) / np.abs(u_true))
     continue
   else:
     Poisson_device_kernel[(gd,gd),(bd,bd)](u_kp1_device, u_k_device,␣
↪f_array_device, N)
     iters += 1

     u_kp1 = u_k_device.copy_to_host()
     err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true) / np.abs(u_true))
     continue
```

```
  interval = time.time() - start

  return iters, interval
```

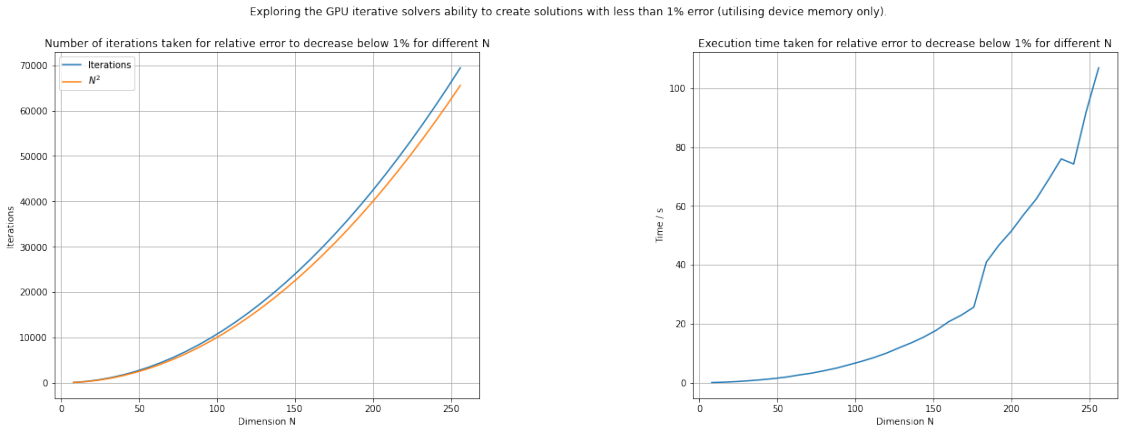
[11]:
```
GPU_iters = []
GPU_times = []

N_values = np.arange(8, 257, 8)
bd = 32

for N in N_values:
  iters, t = GPU_min_error(u_exact, f, N, bd)
  GPU_iters.append(iters)
  GPU_times.append(t)
```

[12]:
```
# printing iters and time
print(GPU_iters)
print(GPU_times)
```

```
[80, 298, 651, 1138, 1761, 2517, 3408, 4434, 5594, 6889, 8318, 9882, 11580,
13413, 15380, 17483, 19720, 22091, 24596, 27236, 30009, 32921, 35964, 39142,
42455, 45905, 49486, 53199, 57055, 61042, 65166, 69420]
[0.043561458587646484, 0.15984797477722168, 0.37584590911865234,
0.6559290885925293, 0.9943475723266602, 1.3971436023712158, 1.9198534488677979,
2.605602979660034, 3.2189059257507324, 4.054590225219727, 4.946115493774414,
6.072371006011963, 7.235923767089844, 8.520415544509888, 10.019538402557373,
11.813687562942505, 13.537022590637207, 15.505188226699829, 17.821309328079224,
20.730321884155273, 22.925487279891968, 25.648353338241577, 40.90994644165039,
46.63993263244629, 51.438491106033325, 57.09246897697449, 62.349313497543335,
69.02837419509888, 75.97434115409851, 74.25058937072754, 91.95466923713684,
106.89694476127625]
```

[13]:
```
%matplotlib inline

fig = plt.figure(figsize=(21,7))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.plot(N_values, GPU_iters, label='Iterations')
ax1.plot(N_values, N_values**2, label=r'$N^2$')
ax1.set_xlabel('Dimension N')
ax1.set_ylabel('Iterations')
ax1.set_title('Number of iterations taken for relative error to decrease below␣
 ↪1% for different N')
ax1.legend()
ax1.grid()
```

```
ax2.plot(N_values, GPU_times)
ax2.set_xlabel('Dimension N')
ax2.set_ylabel('Time / s')
ax2.set_title('Execution time taken for relative error to decrease below 1% for␣
 ↪different N')
ax2.grid()

plt.suptitle('Exploring the GPU iterative solvers ability to create solutions␣
 ↪with less than 1% error (utilising device memory only).')
plt.subplots_adjust(wspace=0.5)
plt.show()
```



This above plots display the error and convergence rates of the GPU accelerated function. For smaller grid spacings and better approximated domains (to the continuous domain), we see that the error has significantly increased, and is decreasing at a much slower rate than for those with larger grid spacings. While errors for coarser domains rapidly tend to zero, the largest domain tested for, $N = 1000$ struggled to breach below a relative error of $e_{rel} = 0.6$.
However, the convergence rates for the various grains of domains shows that our solutions, even for the large N, are converging towards a solution.
Looking at the convergence rate plot we see that for each of the dimensions of grid spacing I've plotted for, each of these curves start with a different convergence, and they all quickly plateau at a maximum accuracy after a certain number of iterations.
Similarly, when looking at the raw error plot against the number of iterations it's interesting to see that for $N = 100$, there's an intial peak in relative error before a steady decrease.
For further investigation, I'm going to plot the maximum number of iterations required for a certain mesh size N to have negligible error, and also I am going to plot the number of iterations (for various N) required before a steady convergence is achieved.
Could also test the effect on execution time with varying thread block sizes, and the time taken to reach less than 1% error. - can measure execution time and number of iterations.

0.3.2 3.2 GPU Acceleration utilising thread block shared memory

Alternatively to utilising device memory, one can make use of shared memory in a thread block. Shared memory is unique to each thread block in the GPU architecture, and here a massive problem can be sped up by uploading data relevant to a specific thread block into its shared memory. This saves the threads from having to access global memory and instead utilising their shared memory. Writing CUDA code to utilise shared memory is especially tricky because of the usage of different indices relative to the global array being operated on and the thread specific array being operated on.

[14]:
```
# bd = 32
# bd2 = bd + 2
# @cuda.jit
# def Poisson_shared_kernel(u_k, u_kp1, f_array, N):
#     """
#     GPU Kernel to update the grid points using an interative.
#     ================================================
#     Inputs:

#     u_k - an array of current values of u(i,j) on the grid.
#     u_kp1 - an array to hold the next iteration of u(i,j) values in the grid.
#     f_array - an array containing the evaluated RHS for all interior␣
 ↪subsquare points (all i and j between [1,N]) in the domain.
#     N - number of grid points along the subsquare axes.
#     ================================================
#     Outputs:

#     Updates u_kp1 based on the current values of u_k in the grid.
#     ================================================
#     """

#     h = 1/(N+1)

#     # defining local CUDA arrays to store the data required for each thread␣
 ↪block.

#     shared_u_k = cuda.shared.array((bd2, bd2), numba.float32)

#     # for corner points my shared array works on the bottom corner of the␣
 ↪global u_k array.
#     # shifting the indices of this thread block to align with points we want␣
 ↪to update, by defining the interior points of the shared array␣
 ↪(0<tx<bd+1,0<ty<bd+1) to be filled
#     # with the corresponding global indices ranging from (0<i<bd+1,0<j<bd+1).
#     # Then for the edge cases filling in the i,j==0 and i,j==bd+1 files of␣
 ↪the shared_u_k array.
#     # We can use when i == bx*bd-1 we input these elements into tx == 0.
```

```
#     # if tx == bd+1 then we input the elements at global position i == bx*bd␣
 ↪+ bd+1
#     # if ty == 0 then we input the elements at global position j == by*bd-1
#     # if ty == bd+1 then we input elements are global position j == by*bd +␣
 ↪bd+1

#     tx = cuda.threadIdx.x # thread locations relative to its thread block
#     ty = cuda.threadIdx.y

#     bx = cuda.blockIdx.x # which block in the grid
#     by = cuda.blockIdx.y

#     gx = cuda.gridDim.x # number of blocks in the grid
#     gy = cuda.gridDim.y

#     bdx = cuda.blockDim.x # number of threads along the x dimension in a block
#     bdy = cuda.blockDim.y # number of threads along the y dimension in a block

#     i, j = cuda.grid(2) # calling the global indices of the data.

#     # filling interior points in the shared array

#     if i < N and j < N:
#       for tx in range(bx):
#         for ty in range(by):
#           shared_u_k[tx,ty] = u_k[i,j]

#           cuda.syncthreads()


#     u_k[i,j] = 0.25 * ((h**2)*f_array[i,j] + shared_u_k[tx+1,ty+1-1] +␣
 ↪shared_u_k[tx+1,ty+1+1] + shared_u_k[tx+1+1,ty+1] + shared_u_k[tx+1-1,ty+1])

#     cuda.syncthreads()

#     # In the shared memory I was trying to upload all the u_k values required␣
 ↪for updates however I think because we use a single set of thread and global␣
 ↪indices this
#     # method of updating won't be possible when updating the final matrix.
#     # Check at the boundaries of the global memory because the shared arrays␣
 ↪might be trying to draw values from outside the global memory.

# def iterate_shared_kernel(boundary_function, RHS_function, N, iterations,␣
 ↪check):
#   """
```

```
#   Iterates of the Poisson solving kernel to update the solutions across the␣
 ↪current grid.
#   ================================================
#   Inputs:

#   boundary_function - Python callable function: determines the conditions at␣
 ↪the boundary points.
#   RHS_function - Python callable function: determines the f function on the␣
 ↪Poisson problem.
#   N - number of points along the x and y axes between the boundaries.
#   iterations - number of times the CUDA kernel will be called to execute the␣
 ↪operation.
#   check - the interval between iterations for which convergence between the␣
 ↪previous and current iteration will be checked.
#   ================================================
#   Outputs:

#   updated_u - the final solution for u after the iterative scheme has been␣
 ↪implemented for the specified number of iterations.
#   e_rel - the relative errors calculated between u_k and u_kp1 every 'check'␣
 ↪iterations.
#   times - time taken to run the kernel for the current iteration.
#   ================================================
#   """
#   # Initialising arrays.

#   u_k = initial_conditions(boundary_function, N)
#   f_array = RHS_array(RHS_function, N)
#   u_kp1 = initial_conditions(boundary_function, N)

#   u_k_device = cuda.to_device(u_k) # initialising grid.
#   f_array_device = cuda.to_device(f_array) # initialising RHS at all grid␣
 ↪points
#   u_kp1_device = cuda.to_device(u_kp1) # initialising array for updated values

#   block_uk_dev = cuda.to_device(np.ones((block_dim2, block_dim2))) #␣
 ↪debugging array

#   # Initialising CUDA parameters.

#   bd = 32 # number of threads in a block
#   gd = (u_kp1_device.shape[0] + bd - 1) // bd # dimension of the grid of of␣
 ↪blocks.

#   # Executing iterative scheme.
```

```
#   # creating arrays for storing the error between consecutive iterations,␣
 ↪which is checked every 'check' iterations.

#   u_true = np.zeros((N,N))

#   for j in range(1,N+1):
#     for i in range(1,N+1):
#       u_true[i-1,j-1] = u_exact(i,j,N)

#   e_rel = []
#   times = []


#   for run in range(iterations):

#     if run%2 == 0:
#       Poisson_shared_kernel[(gd, gd),(bd, bd)](block_uk_dev, u_k_device,␣
 ↪u_kp1_device, f_array_device, N)
#     else:
#       Poisson_shared_kernel[(gd, gd),(bd, bd)](block_uk_dev, u_kp1_device,␣
 ↪u_k_device, f_array_device, N)

#     if run%check == 0:

#       t = timeit.Timer(lambda: Poisson_shared_kernel[(gd, gd),(bd,␣
 ↪bd)](block_uk_dev, u_k_device, u_kp1_device, f_array_device, N),␣
 ↪globals=globals())
#       time = t.timeit(number=1)

#       current_iter = u_kp1_device.copy_to_host() # copying k and k+1␣
 ↪iterations of u back to the host for error and convergence calculations

#       err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true)/np.abs(u_true)) #np.
 ↪linalg.norm((current_iter[1:N+1,1:N+1] - u_true), np.inf) / np.linalg.
 ↪norm(u_true, np.inf)
#       e_rel.append(err)
#       times.append(time)


#       #u_updated = u_kp1_device.copy_to_host() # used for actual algorithm
#       u_updated = block_uk_dev.copy_to_host() # debugging
#       print(u_updated.shape)
#       print(u_updated)

#       fig = plt.figure(figsize=(8, 8))
#       ax = fig.gca(projection='3d')
```

```
#       ticks= np.linspace(0, 1, 34)
#       X, Y = np.meshgrid(ticks, ticks)
#       surf = ax.plot_surface(X, Y, u_updated, antialiased=False, cmap=cm.
 ↪coolwarm)
#       plt.show()


#   return u_updated, np.array(e_rel), np.array(times)
```

[15]:
```
# N = 10
# u_k_dev = cuda.to_device(initial_conditions(u_exact, N))
# u_kp1_dev = cuda.to_device(initial_conditions(u_exact, N))
# f_array_dev = cuda.to_device(RHS_array(f, N))

# bd = 2 # number of threads in a block
# bd2 = bd + 2
# gd = (u_kp1_dev.shape[0] + bd - 1) // bd # dimension of the grid of of blocks.

# # Poisson_device_kernel[(gd,gd),(bd,bd)](u_k_dev, u_kp1_dev, f_array_dev, N)
# # u_kp1 = u_kp1_dev.copy_to_host()

# # print(u_kp1)

# Poisson_shared_kernel[(gd,gd),(bd,bd)](u_k_dev, u_kp1_dev, f_array_dev, N)
# # test = test_dev.copy_to_host()
# # print(test)
# u_kp1 = u_kp1_dev.copy_to_host()
# print(u_kp1)
```

[16]:
```
# fig = plt.figure(figsize=(8, 8))
# ax = fig.gca(projection='3d')
# ticks= np.linspace(0, 1, N)
# X, Y = np.meshgrid(ticks, ticks)
# surf = ax.plot_surface(X, Y, u_kp1[1:N+1,1:N+1], antialiased=False, cmap=cm.
 ↪coolwarm)
# plt.show()
```

[17]:
```
# N = 64
# iterations = 10000
# u_k_dev = cuda.to_device(initial_conditions(u_exact, N))
# u_kp1_dev = cuda.to_device(initial_conditions(u_exact, N))
# f_array_dev = cuda.to_device(RHS_array(f, N))

# bd = 16 # number of threads in a block
# bd2 = bd + 2
# gd = (u_kp1_dev.shape[0] + bd - 1) // bd # dimension of the grid of of blocks.
```

```
# #test = cuda.to_device(np.ones((bd2,bd2)))

# #Poisson_shared_kernel[(gd,gd),(bd,bd)](test, u_k_dev, u_kp1_dev,␣
 ↪f_array_dev, N)

# for run in range(iterations):
#   if run%2 == 0:
#     Poisson_shared_kernel[(gd,gd),(bd,bd)](u_k_dev, u_kp1_dev, f_array_dev, N)
#   else:
#     Poisson_shared_kernel[(gd,gd),(bd,bd)](u_kp1_dev, u_k_dev, f_array_dev, N)
```

[18]:
```
# u_kp1 = u_kp1_dev.copy_to_host()
# print(u_kp1)
```

[19]:
```
# test = test.copy_to_host()
# print(test.shape)

# fig = plt.figure(figsize=(8, 8))
# ax = fig.gca(projection='3d')
# ticks= np.linspace(0, 1, test.shape[0])
# X, Y = np.meshgrid(ticks, ticks)
# surf = ax.plot_surface(X, Y, test, antialiased=False, cmap=cm.coolwarm)
# plt.show()
```

[20]:
```
# %matplotlib inline
# u_kp1 = u_kp1_dev.copy_to_host()

# fig = plt.figure(figsize=(8, 8))
# ax = fig.gca(projection='3d')
# ticks= np.linspace(0, 1, N)
# X, Y = np.meshgrid(ticks, ticks)
# surf = ax.plot_surface(X, Y, u_kp1[1:N+1,1:N+1], antialiased=False, cmap=cm.
 ↪coolwarm)
# plt.show()
```

0.3.3 3.3 CPU Accelerated Iterative scheme

To compare the efficiency of our GPU implemented Jacobi iterative solver, below I will execute the same iterative method using CPU parallelisation and use the results to compare the executions times and accuracies of the two schemes on the CPU and GPU.
Using numba.njit decorator I can run my callable function in nopython mode and parallelise across CPU cores using the 'parallel=True' argument.

[21]:
```
# Creating a function that takes three arrays, u_k , f and u_kp1 and executes␣
 ↪the iterative update once on the array u_kp1 using values from the arrays␣
 ↪u_k and f.
@numba.njit(parallel=True)
```

```
def CPU_Poisson_jacobi(u_k, u_kp1, f_array, N):
  """
  CPU implementation of the Jacobi iterative scheme for solving the Poisson␣
 ↪problem.
  ==========================================================
  Inputs:

  u_k - an array containing the current iterations values at each point (i,j):␣
 ↪this array contains the boundary conditions at the edges of the unit square␣
 ↪defined by g(i,j)
  u_kp1 - an array of the same dimensions at u_k (and the domain) to hold the␣
 ↪values of the (k+1)th iteration of the scheme.
  f_array - an array containing the evaluated RHS for all interior subsquare␣
 ↪points (all i and j between [1,N]) in the domain.
  N - number of grid points along the subsquare axes.
  ==========================================================
  Outputs:

  u_kp1 - the updated grid solutions for u after one iteration of the scheme.
  """

  h = 1/(N+1)

  # Updating using an iterative Jacobi method.

  for j in numba.prange(1,N+1):
    for i in range(1,N+1):

      up = u_k[i, j+1]
      down = u_k[i,j-1]
      left = u_k[i-1,j]
      right = u_k[i+1,j]

      u_kp1[i,j] = 0.25*((h**2)*f_array[i-1,j-1] + up + down + left + right)

  return u_kp1
```

[22]:
```
# Iterating the CPU implementation.
def CPU_min_error(boundary_function, RHS_function, N):
  """
  Iterating the CPU implementation of the Jacobi iterative method for solving␣
 ↪the Poisson problem.
  ==========================================================
  Inputs:

  boundary_function - Python callable function used to define the boundary␣
 ↪values of the domain.
```

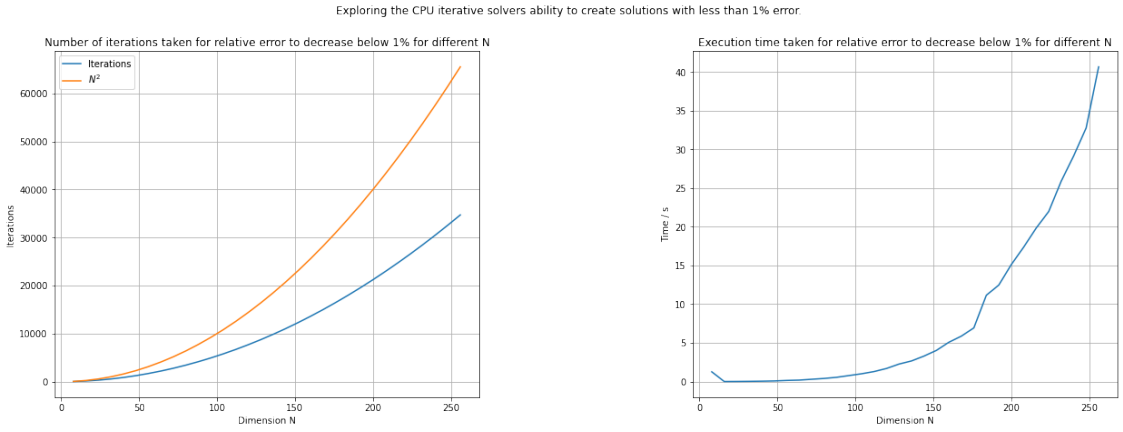
```
    RHS_function - Python callable function used to define the RHS of the Poisson problem, f.
    N - number of grid points along the subsquare axes.
    iterations - number of iterations we're going to run the Jacobi update for.
    check - how often we check the convergence rate of our solutions.

    Outputs:

    iters - number of iterations required for the solution to obtain a relative error below 1%.
    interval - the time taken for the solver to iterate until it has a relative error below 1%.
    """

    u_k = initial_conditions(boundary_function, N)
    u_kp1 = initial_conditions(boundary_function, N)
    f_array = RHS_array(RHS_function, N)

    u_true = np.zeros((N,N))

    for j in range(1,N+1):
      for i in range(1,N+1):
        u_true[i-1,j-1] = u_exact(i,j,N)

    iters = 0

    err = np.amax(np.abs(u_kp1[1:N+1,1:N+1] - u_true) / np.abs(u_true))

    start = time.time()

    while (err > 0.01):
      u_k = CPU_Poisson_jacobi(u_k, u_kp1, f_array, N)
      iters += 1

      err = err = np.amax(np.abs(u_k[1:N+1,1:N+1] - u_true) / np.abs(u_true))

    interval = time.time() - start

    return iters, interval
```

[23]:
```
CPU_iters = []
CPU_times = []

N_values = np.arange(8, 257, 8)

for N in N_values:
  iters, t = CPU_min_error(u_exact, f, N)
```

```
  CPU_iters.append(iters)
  CPU_times.append(t)
```

```
/usr/local/lib/python3.7/dist-packages/numba/np/ufunc/parallel.py:363:
NumbaWarning: The TBB threading layer requires TBB version 2019.5 or later i.e.,
TBB_INTERFACE_VERSION >= 11005. Found TBB_INTERFACE_VERSION = 9107. The TBB
threading layer is disabled.
  warnings.warn(problem)
```

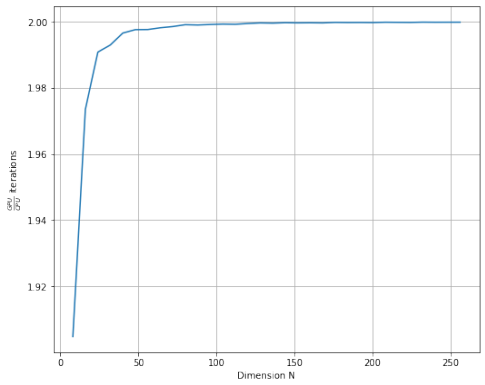
[24]:
```
%matplotlib inline

fig = plt.figure(figsize=(21,7))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.plot(N_values, CPU_iters, label='Iterations')
ax1.plot(N_values, N_values**2, label=r'$N^2$')
ax1.set_xlabel('Dimension N')
ax1.set_ylabel('Iterations')
ax1.set_title('Number of iterations taken for relative error to decrease below␣
 ↪1% for different N')
ax1.legend()
ax1.grid()

ax2.plot(N_values, CPU_times)
ax2.set_xlabel('Dimension N')
ax2.set_ylabel('Time / s')
ax2.set_title('Execution time taken for relative error to decrease below 1% for␣
 ↪different N')
ax2.grid()

plt.suptitle('Exploring the CPU iterative solvers ability to create solutions␣
 ↪with less than 1% error.')
plt.subplots_adjust(wspace=0.5)
plt.show()
```

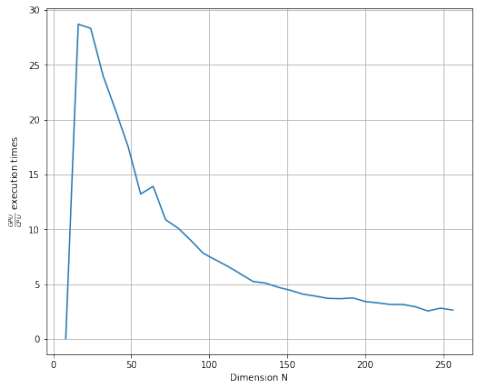


[25]:
```
print(CPU_iters)
print(CPU_times)
```

```
[42, 151, 327, 571, 882, 1260, 1706, 2219, 2799, 3446, 4161, 4943, 5792, 6709,
7692, 8743, 9862, 11047, 12300, 13620, 15007, 16462, 17984, 19573, 21230, 22954,
24745, 26602, 28529, 30523, 32585, 34712]
[1.25773286819458, 0.005571126937866211, 0.013277530670166016,
0.027378320693969727, 0.0477752685546875, 0.07977628707885742,
0.14534878730773926, 0.18729233741760254, 0.2964177131652832,
0.4015378952026367, 0.5495421886444092, 0.7766482830047607, 1.0044758319854736,
1.2898039817810059, 1.690758228302002, 2.257784128189087, 2.6607415676116943,
3.2876734733581543, 4.025485277175903, 5.07915186882019, 5.871351003646851,
6.934739112854004, 11.1489839553833, 12.468968152999878, 15.112051486968994,
17.387606382369995, 19.840806245803833, 21.97132706642151, 25.888665676116943,
29.1613130569458, 32.77235293388367, 40.66468596458435]
```

[30]:
```
# plotting the difference between CPU execution times and GPU execution times.

CPU_iters = np.array(CPU_iters)
CPU_times = np.array(CPU_times)
GPU_iters = np.array(GPU_iters)
GPU_times = np.array(GPU_times)

fig = plt.figure(figsize=(21,7))
ax1 = fig.add_subplot(1,2,1)
ax2 = fig.add_subplot(1,2,2)

ax1.set_xlabel('Dimension N')
ax1.set_ylabel(r'$\frac{GPU}{CPU}$ iterations')
ax1.plot(N_values, GPU_iters/CPU_iters)
ax1.grid()

ax2.set_xlabel('Dimension N')
ax2.set_ylabel(r'$\frac{GPU}{CPU}$ execution times')
ax2.plot(N_values, GPU_times/CPU_times)
ax2.grid()

plt.subplots_adjust(wspace=0.5)
plt.show()
```





[31]:
```
print(GPU_iters/CPU_iters)
print(GPU_times/CPU_times)
```

```
[1.9047619  1.97350993 1.99082569 1.99299475 1.99659864 1.99761905
 1.99765533 1.99819739 1.99857092 1.99912943 1.99903869 1.99919077
 1.99930939 1.99925473 1.99947998 1.99965687 1.9995944  1.99972843
 1.9996748  1.99970631 1.99966682 1.99981776 1.99977758 1.99979564
 1.99976448 1.9998693  1.99983835 1.99981204 1.99989484 1.99986895
 1.99987724 1.99988477]
[ 0.03463491 28.69221552 28.30691327 23.95797375 20.813019   17.51326934
 13.20859626 13.91195719 10.85935753 10.09765273  9.00042908  7.81868851
  7.2036813   6.60597708  5.92606219  5.23242564  5.08768787  4.7161582
  4.42712073  4.08145344  3.90463579  3.69853183  3.66938787  3.74048053
  3.40380597  3.28351515  3.14247882  3.14174806  2.93465651  2.54620185
  2.80586107  2.62874143]
```

From the above plots we can see that surprisingly, for my current implementation, the GPU execution times and number of iterations required to minimise error are significantly larger than that of the CPU. In the GPU accelerated iterative scheme, the number of iterations is almost double of that in the CPU accelerated scheme.
Similarly, for smaller dimensions like $N = 8, 16, 24$ we have the GPU scheme taking up 28 times as long to gain a solution with less than 1% error. This would suggest that the convergence rate of the CPU scheme is much steadier and consistently decreases with iterations.

0.4 Conclusion

While investigating the implementation of a Jacobi scheme to solve a complex Poisson problem using callable function to define the derivative and the boundary conditions, I have found that the discretisation of the unit domain we work with and the usage of a Sparse Linear Solver provided by the scipy library provide the most accurate solutions.
However, when using an iterative scheme on the discretised domain to find solutions, I've found that CPU parallelisation can find accurate solutions more quickly than executing operations on the GPU using device memory.
This is quite a surprising observation, and I believe some of the increase in time comes due to the way I've defined my iterative functions, using two extra if conditions within the while loop

compared to no if conditions used when iterating the CPU solver. The CPU solver requires ~2 times less iterations to converge to a solution with an error of less than 1% compared to the GPU solver, and ultimately does to ~5 times quicker.
Unfortunately, during this investigation I couldn't figure out the implementation of using the GPU's shared thread block memory in order to further acclerate the operations.

[ ]: