HPC Assignment 2 - VCYQ6

by Juan Francisco Rodriguez Hernandez

Submission date: 08-Nov-2021 08:39AM (UTC+0000)

Submission ID: 162291052

File name: 801558_Juan_Francisco_Rodriguez_Hernandez_HPC_Assignment_2_-_VCYQ6_2485672_1352377719.pdf (4.35M)

Word count: 0

Assignment 2 - GPU Accelerated solution of Poisson problems

Candidate number: VCYQ6

Student number: 18018226

PDEs are equations involving of an unknown function u, its derivatives. Generally, PDEs cannot be solved using analytical means, except for a few cases. Therefore, iteritative methods are employed in an attempt to approximate the unknown function.

Our main contribution was focused on applying a highly parallelizable adaptation on the finite element assembly algorithm to the solution of the Poisson heat equation on the GPU. A typical GPU consists of up to 16 streaming multiprocessors. Each streaming mul- tiprocessor can have up to 48 active warps. A warp is a group of 32 threads to per- form calculations in SIMD (Single Instruction Multiple Data) fashion. Hence every streaming multiprocessor can perform up to 1536 calculations simultaneously. This architecture gives the GPU the capacity to perform up to 24576 calculations in parallel

This method, combined with a comprehensive optimization has provided a huge speed boost to the assembly part of the code, considerably lowering the assembly time from over 8 s to below 10−3 s.

Double-click (or enter) to edit

Imports

```
#import all the packages
import numpy as np
import math
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import spsolve

import numba
from numba import cuda, njit, prange, float64, float32

%matplotlib inline
from matplotlib import pyplot as plt
from matplotlib.pyplot import figure
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
```

Discretise Poisson function

We first need to create a solution to the following Poisson problem: $-\Delta u(x,y) = f(x,y)$.

Where $\Delta := u_{xx} + u_{yy}$ for $(x,y) \in \Omega \subset \Re^2$ and boundary conditions $u(x,y) = g(x,y)$ on $\Gamma = \delta\Omega$. We will take the domain Ω to be the unit square $\Omega := [0,1]^2$.

In order to solve this problem, we will define the grid points $x_i := ih$ and $y_j := jh$ with $i,j = 1,\ldots,N$ and $h = 1/(N+1)$. Hence we can now approximate:

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

Therefore, the Poisson problem becomes the following system of N^2 equations:

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

1. Implement a function discretise_poisson(f, g, N) that takes a Python callable f and g and the parameter N and returns a sparse CSR matrix A and the corresponding right-hand side b of the above discretised Poisson problem.

```
def discretise_poisson(f, g, N):
    """Generate the matrix and rhs associated with the discrete Poisson operator."""

    #number of non-zero elements
    nelements = 5 * N**2 - 16 * N + 16

    #array to create our sparse matrix
    row_ind = np.empty(nelements, dtype=np.float64)
    col_ind = np.empty(nelements, dtype=np.float64)
    data = np.empty(nelements, dtype=np.float64)

    #right hand-side vector
    b = np.empty(N * N, dtype=np.float64)

    h = 1/(N+1)

    count = 0
    for j in range(N):
        for i in range(N):

            #update the boundary values
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = col_ind[count] = j * N + i
                data[count] =  1
                b[j * N + i] = g(h*i,h*j)
                count += 1

            #update the central values
            else:
```

```
                row_ind[count : count + 5] = j * N + i
                col_ind[count] = j * N + i
                col_ind[count + 1] = j * N + i + 1
                col_ind[count + 2] = j * N + i - 1
                col_ind[count + 3] = (j + 1) * N + i
                col_ind[count + 4] = (j - 1) * N + i

                data[count] =  (4 * (N + 1)**2)
                data[count + 1 : count + 5] = (- (N + 1)**2)
                b[j * N + i] = f(h*i,h*j)

                count += 5

    A = coo_matrix((data, (row_ind, col_ind)), shape=(N**2, N**2)).tocsr()

    return A, b
```

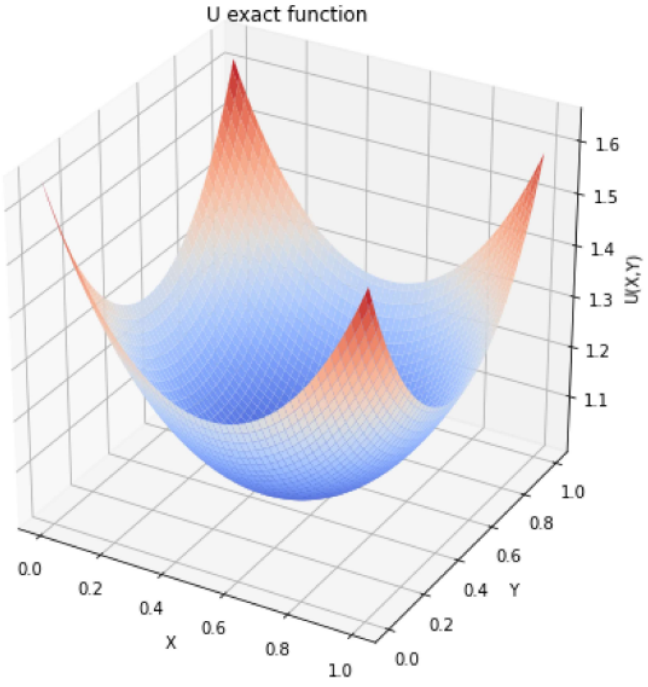
2. To verify our code we will employ the method of manufactured solutions. We will define $u(x, y)$ as $u_{exact}(x, y) = e^{(x-0.5)^2+(y-0.5)^2}$. Hence, if we calculate $-\Delta u_{exact}$ we will obtain $f(x, y)$.

```
def g(x, y):
  "Returns u exact"
  return np.exp(((x-0.5)**2)+((y-0.5)**2))

def f(x, y):
  "Returns the second derivative of u exact"
  return -((2*x-1)**2+(2*y-1)**2+4)*np.exp(((x-0.5)**2)+((y-0.5)**2))

def u_exact(N):
  "Returns u exact in matrix form"
  h = 1/(N+1)
  u = np.zeros(shape = (N,N))
  for i in range(N):
    for j in range(N):
      u[i,j] = g(h*i,h*j)
  return u
```

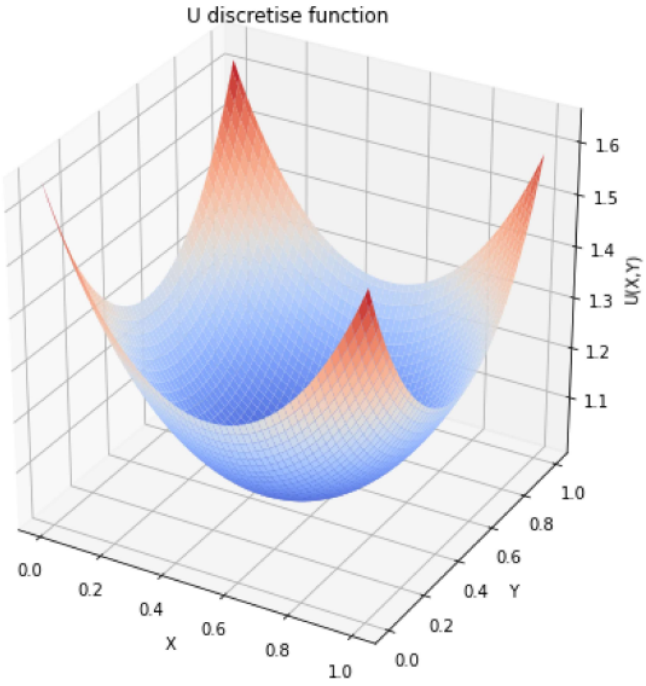
We will now plot u exact and the u obtained from our discretise Poisson function for N = 100 and compare them.

```
#Plot u exact for N = 100
N = 100
ticks= np.linspace(0, 1, N)
X, Y = np.meshgrid(ticks, ticks)
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
u = u_exact(N)
surf = ax.plot_surface(X, Y, u, antialiased=True, cmap=cm.coolwarm)
```

```
#Set the labels and title
ax.set_title('U exact function')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')

plt.show()
```



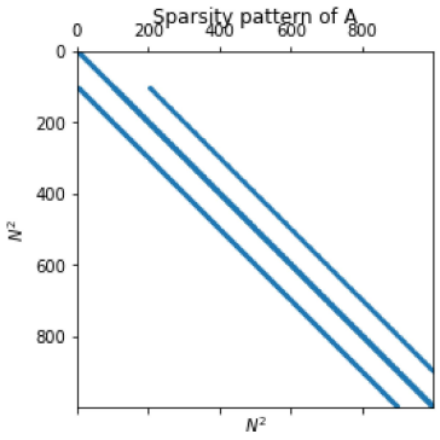

```
N = 100
A, b = discretise_poisson(f, g, N)
sol = spsolve(A, b)
u_discretise = sol.reshape((N, N))

#Plot u exact for N = 100
ticks= np.linspace(0, 1, N)
X, Y = np.meshgrid(ticks, ticks)
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
u = u_exact(N)
surf = ax.plot_surface(X, Y, u_discretise, antialiased=True, cmap=cm.coolwarm)

#Set the labels and title
ax.set_title('U discretise function')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')

plt.show()
```



We will now plot the sparsity pattern of our matrix A. This visualizes the non-zero values of the array.

```
#Plot the sparsity pattern of A
plt.figure(figsize=(8, 4))
plt.spy(A[:1000, :1000], markersize=1)
plt.xlabel('$N^2$')
plt.ylabel('$N^2$')
plt.title('Sparsity pattern of A')
```

```
Text(0.5, 1.05, 'Sparsity pattern of A')
```



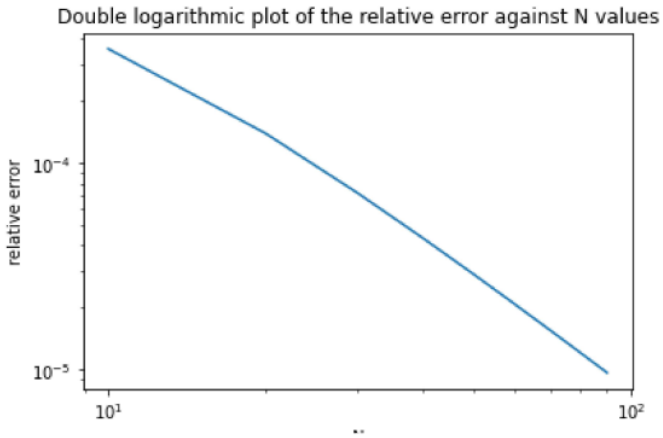
From the graph above, it can be observed the banded structure of the matrix in more detail. By using sparse matrices to store data that contains a large number of zero-valued elements can both save a significant amount of memory and speed proceses.

3. Finally, we will plot the relative error of your computed grid values $u(x, y)$ against the exact solution u_{exact} as N increases. The relative error at a given point is given by the following formula:

$$e_{rel} = \frac{|u(x_i,y_j) - u_{exact}(x_i,y_j)|}{|u_{exact}(x_i,y_j)|}$$

```
#Define our relative error function in terms of N
def e_rel(sol, exact, N):
  "Return the relative error between u and u_exact as a 1D numpy array"
  e = np.zeros(shape = (N,N))
  for i in range(N):
    for j in range(N):
      e[i][j]= abs(sol[i][j] - exact[i][j])/abs(exact[i][j])
  e_max = np.mean(np.ravel(e))
  return e_max


#Plot the error against different values of N
N = 100    # maximum value of N
n = np.array([i for i in range(10, N, 10)])    #list of the different values of N


def calculate_error_N(n):
  "Returns an array of the relative error "
  e = np.zeros(len(n))
  for i in range(len(n)):
    X = n[i]
    A, b = discretise_poisson(f, g, X)
    sol = spsolve(A, b)
    u_discretise = sol.reshape((X, X))
    u_ex = u_exact(X)
    e[i] = e_rel(u_discretise, u_ex, X)

  return e


#Plot the relative error against the different values of N
#We will use the double logarithmic plot
e_max_list = calculate_error_N(n)
plt.loglog(n, e_max_list)
plt.xlabel('N')
plt.ylabel('relative error')
```

```
plt.title('Double logarithmic plot of the relative error against N values')
plt.show()
```



From the above graph it can be observed that as N increases, the relative error between u_{exact} and $u_{discretise}$ decreases. 2

Iterative method

For the second task, we are going to implement an iterative scheme to solve the above Poisson equation. The idea behind iterative methods is to use $x_{current}$, a current approximation (or guess) for the true solution x of $Ax = b$, to find a new approximation x_{new}, where x_{new} is closer to x than $x_{current}$ is. Then, we use this new approximation as the current approximation to find the next, more accurate approximation.

Specifically, we will use Jacobi's method to solve the discrete Poisson problem. Although this method is not commonly used in practice, its potential usefulness has been reconsidered within parallel computing. To apply this method, we need to rewrite our problem in the following way:

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

In this example, we will set $u^k(x_i, y_j)$ as our approximation of $u(x_i, y_j)$ after the k-th iteration. We will set $u^k(x_i, y_j) = 0$ during the first iteration when $k = 0$. To get the following value, we use the following update rule:

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

If f is zero then the left-hand side $u(x_i, y_j)$ is just the average of all the neighbouring grid points. In other words, the value of u at the iteration $k + 1$ is just the average of all the values at iteration k plus the contribution from the right-hand side.

1. We define a function that returns two arrays with the central and boundary values.

```
def u_and_b(f, g, N):
  """Returns two arrays with the central and boundary values"""
  b = np.empty((N+2)**2, dtype=np.float32)
  u = np.empty((N+2)**2, dtype=np.float32)
  h = 1/(N+1)
  for j in range(N+2):
    for i in range(N+2):
    #update the boundary values
      if i == 0 or i == N + 1 or j == 0 or j == N + 1:
        b[j * (N+2) + i] = 0
        u[j * (N+2) + i] = g(i*h, j*h)

    #update the central values
      else:
        b[j * (N+2) + i] = f(i*h, j*h)
        u[j * (N+2) + i] = 0

  return u.astype('float32'), b.astype('float32')
```

2. Define a function that outputs a first approximation of u(x,y).

```
def discretise_poisson_iterative_scheme(u, b, N):
  """Returns u(x, y) after the first iteration"""
  u_k_1 = u
  u_k = u
  h = 1/(N+1)
  for j in range(1, N+1):
    for i in range(1, N+1):
      u_k_1[j * (N+2) + i] = 0.25 * (u_k[j * (N+2) + (i-1)] +
                                     u_k[j * (N+2) + (i+1)] +
                                     u_k[(j+1) * (N+2) + i] +
                                     u_k[(j-1) * (N+2) + i] +
                                     (h**2) * b[i + j*(N+2)])
    u_k = u_k_1
  return u_k.reshape((N+2),(N+2))
```

3. Define a function that iterates over u(x, y) k times.

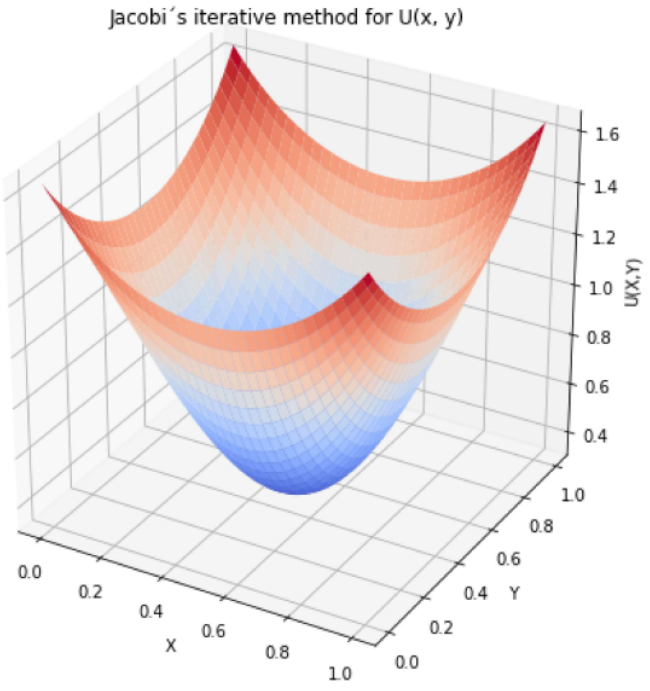
```
def discretise_poisson_iterate(u, b , N, itter):
  tmp = 0
  for i in range(itter):
    tmp = discretise_poisson_iterative_scheme(u,b,N)
  return tmp
```

4. Plot u(x, y) after 1000 iterations for N = 100.

```
# Set parameters
N = 100
iteration = 1000
```

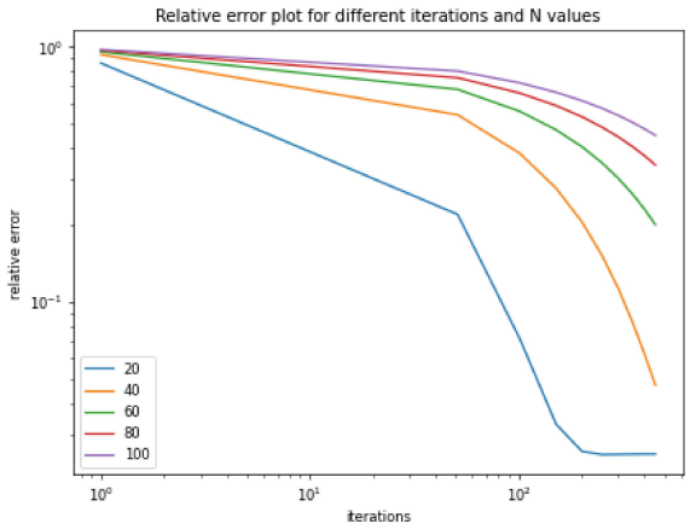
```
# Get the values for u and b for N
u, b = u_and_b(f, g, N)

# Get the updated value of u(x, y) after k iterations
c = discretise_poisson_iterate(u, b, N, iteration)


#Calculate the relative error
u_ex = u_exact(N+2)
error = e_rel(c, u_ex, N)
print("The relative error is", error)

#Plot u(x,y) for N = 100 and 1000 iterations
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N+2)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, c, antialiased=True, cmap=cm.coolwarm)
ax.set_title('Jacobi´s iterative method for U(x, y)')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')
plt.show()
```

```
The relative error is 0.2527742189950338
```



5. We will now plot the relative error for our u(x,y) using Jacobi's method for different iterations and N values.

```
def calculate_error_ite(n, iterations):
  """Returns the relative error between u and u exact"""
  e = np.zeros(shape = (len(n), len(iterations)))
  for i in range(len(n)):
    for j in range(len(iterations)):
      X = n[i]
      ites = iterations[j]
      u, b = u_and_b(f, g, X)
      d = discretise_poisson_iterate(u, b, X, ites)
      u_ex = u_exact(X+2)
      e[i][j] = e_rel(d, u_ex, X)
  return e


#Define our initial data
N = 101
max_ite = 501
n = np.array([i for i in range(20, N, 20)])
iterations = np.array([i for i in range(1, max_ite, 50)])    #array for iterations
errors = calculate_error_ite(n, iterations)


#Plot the relative error against the different values of N
#We will use the double logarithmic plot
figure(figsize=(8, 6), dpi=60)
for i in range(len(errors)):
  plt.loglog(iterations, errors[i], label = n[i])
  plt.xlabel('iterations')
  plt.legend()
  plt.ylabel('relative error')
  plt.title('Relative error plot for different iterations and N values')

plt.show()
```

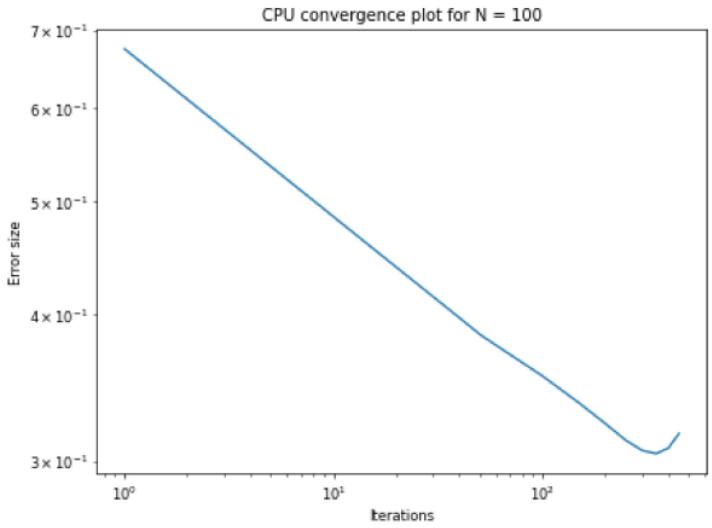


From the graph above it can be observed that as the number of iterations increases, the relative error between u_{exact} and u_{iter} decreases. However, as the number of N values increases, the relative error increases.

6. We will now plot the convergence rate of u_k and u_{k+1}. In order to do this, we will calculate the error between u_k and u_{k+1}, and plot it against the number of iterations for N = 100.

```
#Define our error function
def error_size(u, u1, N):
  "Returns the error of two arrays"
  e = np.zeros(shape = (N,N))
  for i in range(N):
    for j in range(N):
      e[i][j]= (u[i][j] - u1[i][j])
  e_max = np.max(np.ravel(e))
  return e_max


#Define our convergence function for the CPU
def calculate_convergence_CPU(N, iterations):
  "Returns the error of uk and uk1 for different iterations"
  e = np.zeros(shape = (len(iterations)))
  for i in range(len(iterations)):
    ite = iterations[i]
    u, b = u_and_b(f, g, N)
    c_k = discretise_poisson_iterate(u, b, N, ite)
    d_k = c_k.copy()
    c_k_1 = discretise_poisson_iterate(u, b, N, ite+1)
    d_k_1 = c_k_1.copy()
    e[i]= error_size(d_k_1.reshape((N+2, N+2)), d_k.reshape((N+2, N+2)), N)
  return e


e = calculate_convergence_CPU(N, iterations)


(



)

     ()


#Plot the convergence function for N = 100
#We will use the double logarithmic plot
figure(figsize=(8, 6), dpi=60)
plt.loglog(iterations, e)
plt.xlabel('Iterations')
plt.ylabel('Error size')
plt.title('CPU convergence plot for N = 100')
```

```
plt.show()
```



It can be observed that as the number of iteration increases, the error between u_k and u_{k+1} decreases and hence their values converges. In the following section, we will implement the same iterative scheme on the GPU.

GPU implementation

The main purpose of GPU Programming is computing with the use of a Graphic Processing Unit (GPU). This is achieved by using a GPU together with a CPU to accelerate the computations in applications that are normally by just the CPU.

1.The first step for GPU programming is to define our kernel. A kernel is a GPU function that does not return a value and is meant to be called from the CPU. It defines its own thread hierarchy when called.

```
@cuda.jit()
def kernel_iterative_scheme(u, b, u_k_1, N):
  """Kernel that updates values of u using Jacobis method"""
  h = 1/(N+1)
  j = cuda.grid(1)  # absolute position of the current thread in the entire grid of blocks

  # Load all the information into the shared memory
  local_u = cuda.const.array_like(u)           # First global buffer copied to shared memc
  local_u_k_1 = cuda.const.array_like(u_k_1)   # Second global buffer copied to shared mem
  local_b = cuda.const.array_like(b)           # Right hand side copied to shared memory

  # Sync all the threads
  cuda.syncthreads()
```

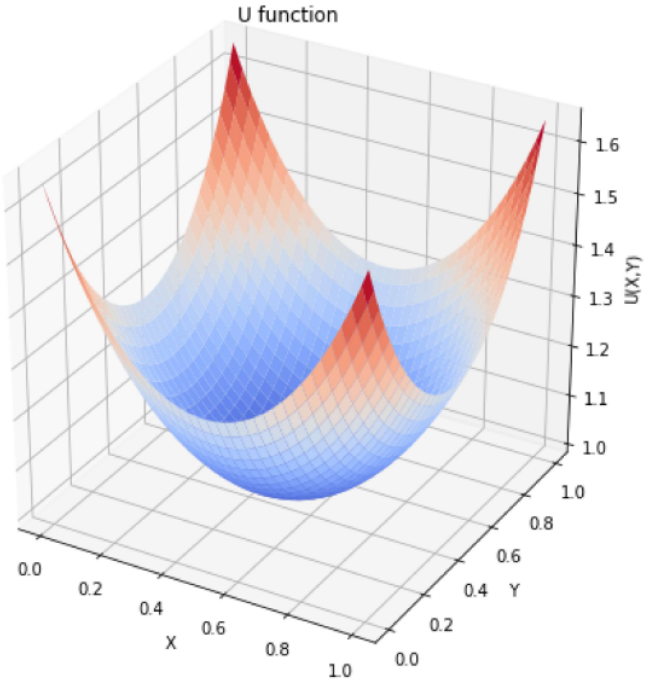
```
  # Initialise the update of the global buffers
  for i in range(1, N+1):

    # We do not update boundary values
    if j == 0 or j == N+1:
      return

    # We update central values
    else:
      local_u_k_1 = local_u
      local_u_k_1[j * (N+2) + i] = 0.25 * (local_u[j * (N+2) + (i-1)] +
                                           local_u[j * (N+2) + (i+1)] +
                                           local_u[(j+1) * (N+2) + i] +
                                           local_u[(j-1) * (N+2) + i] +
                                           (h**2) * local_b[i + j*(N+2)])
      local_u = local_u_k_1

      # Sync all the threads
      cuda.syncthreads()
```

2. In order to call the kernel, we need to specify the number of blocks per grid and threads per block. The product of these two will result in the total number of threads. Then we take the kernel function and index it with a tuple of these integers.

```
# Set the number of threads in a block
# It should be multiples of 16
TPB = 128

# Set the number of thread blocks in the grid
BPG = ((N+2)+TPB-1)//TPB


# Get the initial data
N = 100
iterations = 10000
u, b = u_and_b(f, g, N)

# Copy the data into the device memory
su = cuda.to_device(u)
su_k_1 = cuda.device_array(shape= (N + 2)**2)
sb = cuda.to_device(b)


# Iterate the kernel
for i in range(iterations):
  kernel_iterative_scheme[(BPG, 1), (TPB, 1)](su, sb, su_k_1, N)


# Copy the data from the device back to the host
arry = np.empty(shape= su.shape, dtype = su.dtype)
su.copy_to_host(arry)
```

```
su = arry
su = su.reshape((N+2, N+2))

#Calculate the relative error
u_ex = u_exact(N+2)
error = e_rel(su, u_ex, N)
print("The relative error is", error)

# Plot su
su = su.reshape((N+2, N+2))
fig = plt.figure(figsize=(8, 8))
ax = fig.gca(projection='3d')
ticks= np.linspace(0, 1, N+2)
X, Y = np.meshgrid(ticks, ticks)
surf = ax.plot_surface(X, Y, su1, antialiased=True, cmap=cm.coolwarm)
ax.set_title('U function')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('U(X,Y)')
plt.show()
```

```
The relative error is 0.0075241040157188795
```

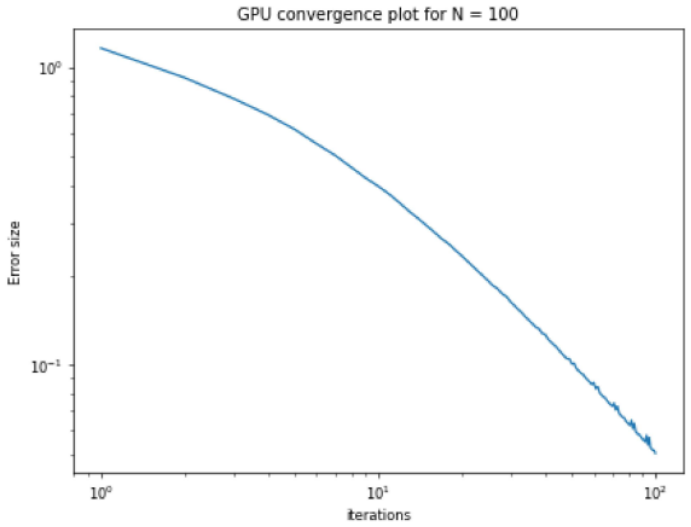


4. We will check for convergence of our GPU code for N=100. In order to do this, we will invoke the kernel twice u_k and u_{k+gap} and then calculate the error size.

```
# Define initial data
max_ite = 101
N = 100
```

```
gap = 10
iteration = np.array([i for i in range(1, max_ite)])    #array for iterations



#Define our convergence function for the GPU
def calculate_convergence_GPU(N, iterations, gap):
  "Returns the error of uk and uk1 for different iterations"

  e = np.zeros(len(iterations))
  for j in range(len(iterations)):
    iter = iterations[j]  # get the exact iteration from the array
    u, b = u_and_b(f, g, N) # set the initial values

    # Copy these values into the device memory
    su = cuda.to_device(u)
    su_k_1 = cuda.device_array(shape= (N + 2)**2)
    sb = cuda.to_device(b)

    # Iterate the kernel
    for i in range(iter):
      kernel_iterative_scheme[(BPG, 1), (TPB, 1)](su, sb, su_k_1, N)

    # Copy the data from the device back to the host
    arry = np.empty(shape= su.shape, dtype = su.dtype)
    su.copy_to_host(arry)
    su = arry
    su = su.reshape((N+2, N+2))

    su2 = su.copy() #make a copy of su

    u, b = u_and_b(f, g, N) # set the initial values again

    # Copy these values into the device memory
    su = cuda.to_device(u)
    su_k_1 = cuda.device_array(shape= (N + 2)**2)
    sb = cuda.to_device(b)

    # Iterate the kernel over the inital iteration plus the gap
    for i in range(iter+gap):
      kernel_iterative_scheme[(BPG, 1), (TPB, 1)](su, sb, su_k_1, N)

    # Copy the data from the device back to the host
    arry = np.empty(shape= su.shape, dtype = su.dtype)
    su.copy_to_host(arry)
    su = arry
    su = su.reshape((N+2, N+2))

    su1 = su.copy() #make a copy of su

    # Calculate the error between the values of uk and uk1
    e[j] = e_rel_1(su1, su2, N)

  return e
```

```
e = calculate_convergence_GPU(N, iteration, gap)


#Plot the convergence for N = 100
#We will use the double logarithmic plot
figure(figsize=(8, 6), dpi=60)
plt.loglog(iteration, e)
plt.xlabel('iterations')
plt.ylabel('Error size')
plt.title('GPU convergence plot for N = 100')

plt.show()
```



It can be observed again that as the number of iteration increases, the error between u_k and u_{k+1} decreases. Moreover, the shape of the CPU convergence graph is similar to the GPU convergence plot.

Benchmarking

In the following section, we are going to benchmark the CPU and GPU function for N = 100 dimensions and 500 iterations.

```
# Define initial values
N = 100
iterations = 501
u, b = u_and_b(f, g, N)

# Copy these values into the device memory
su = cuda.to_device(u)
```

```
su_k_1 = cuda.device_array(shape= (N + 2)**2)
sb = cuda.to device(b)
for i in range(1, iterations, 50):
  time_GPU = %timeit kernel_iterative_scheme[(BPG, 1), (TPB, 1)](su, sb, su_k_1, N)
```

```
The slowest run took 5.31 times longer than the fastest. This could mean that an int
10000 loops, best of 5: 187 µs per loop
10000 loops, best of 5: 189 µs per loop
The slowest run took 6.02 times longer than the fastest. This could mean that an int
1000 loops, best of 5: 193 µs per loop
The slowest run took 8.52 times longer than the fastest. This could mean that an int
1000 loops, best of 5: 180 µs per loop
The slowest run took 7.23 times longer than the fastest. This could mean that an int
10000 loops, best of 5: 187 µs per loop
1000 loops, best of 5: 181 µs per loop
1000 loops, best of 5: 184 µs per loop
10000 loops, best of 5: 187 µs per loop
1000 loops, best of 5: 181 µs per loop
10000 loops, best of 5: 185 µs per loop
```

```
time_CPU = %timeit discretise_poisson_iterate(u, b, N, iterations)
```

```
1 loop, best of 5: 27.7 s per loop
```

This shows that the GPU function is approxmately 145000 times faster than the CPU code.

HPC Assignment 2 - VCYQ6

GRADEMARK REPORT

FINAL GRADE

14/20

GENERAL COMMENTS

Instructor

Task1: 7/8 Wrong implementation, very few comments on results.

Task2: 7/12 There isn't memory optimization. There isn't comparison with exact solution.

PAGE 1

PAGE 2

PAGE 3

Comment 1

This is the same implementation made in class. You were asked to not define boundary points as unknowns

PAGE 4

PAGE 5

PAGE 6

PAGE 7

Comment 2

Rate of convergence?

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

Comment 3

What's su1?

PAGE 15

PAGE 16

PAGE 17