Assignment 4 - Time-dependent problems

Below are the packages and some convenient toll functions used in the assignment.

In [1]:
```python
import time
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import coo_matrix
from scipy.sparse.linalg import spsolve
from numba import cuda, float64, prange, njit

# the wanted solution to when does the center reach u=1
t_expect = 0.424011387033


# to estimate the efficiency of different methods
class Timer:
    def __enter__(self):
        self.start = time.perf_counter()
        return self

    def __exit__(self, *args):
        self.end = time.perf_counter()
        self.interval = self.end - self.start


# a tool: if there is remaider quotient + 1
def div_ceil(dividend, divisor):
    quotient = dividend // divisor
    remainder = dividend % divisor

    if remainder:
        return quotient + 1
    else:
        return quotient

def middle_element(arr,N):
    if N % 2 == 1:
        # If the length is odd, return the middle element
        return arr[N**2//2]
    else:
        # If the length is even, return the average of the two middle elements
        return (arr[(N**2)//2 + N//2- 1] + arr[(N**2)//2+N//2]) / 2.0



# generate the t0 plate (1D)
def t_0_generator(N):
    t_0 = np.zeros((N, N), dtype=np.float64)
    t_0[:, 0] = 5
    t_0.resize((N**2,))
    return t_0
```

explicit time-stepping (Forward Euler)

Forward-Euler is used in the explicit part. Methods with and without CUDA are both implemented in this part.

As the explict methods are sensitive to the time step, time step which is too large often raises unstability.

To make the whole process stable, CFL condition should be satisfied.

2D CFL condition gives that:

$$\sqrt{\left(\frac{c_x \Delta t}{\Delta x}\right)^2 + \left(\frac{c_y \Delta t}{\Delta y}\right)^2} \leq 1$$

which means when

$$u_{i,j}^{n+1} = u_{i,j}^n + \alpha \Delta t \left(\frac{u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n - 4u_{i,j}^n}{\Delta x^2} \right)$$

time step Δt and Δx should satisfy:

$$\Delta t \leq \frac{1}{4} \Delta x^2$$

In this assignment as the plate is a square plate with side lenth $L = 2$, thus:

$$\Delta t \leq \frac{1}{N^2}$$

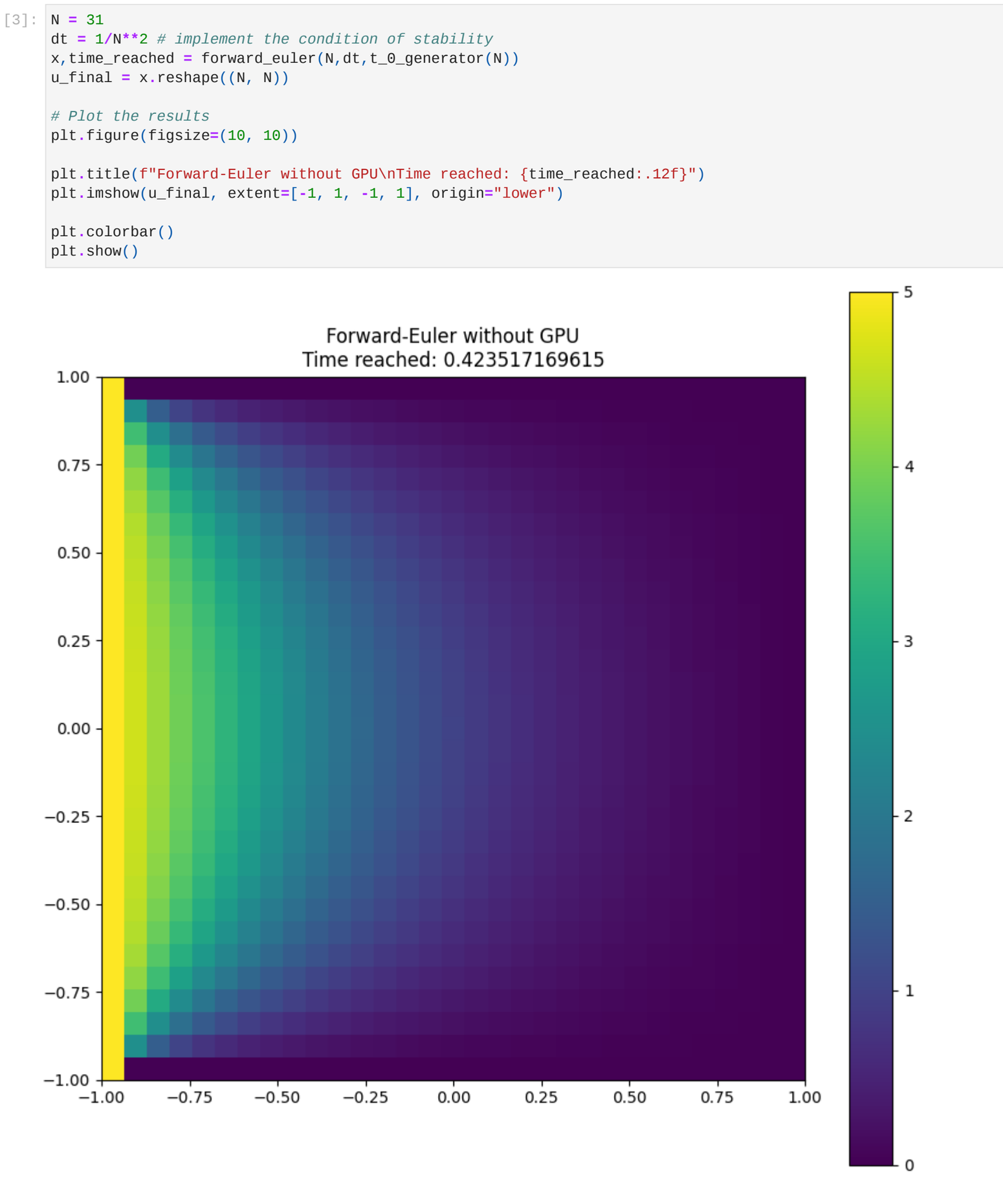
without cuda

`@numba.njit` and `numba.prange` is used to accelerate the computation. Thus, the work in the for loop are carried out in parallel.

In [2]:
```python
@njit
def forward_euler(N,dt,x):
    u_new = x.copy()
    k = dt / (4 / ((N-1)**2))

    u = np.zeros((N, N), dtype=np.float64)
    centre = N**2 // 2
    counter = 0
    time_reached = 0

    while u_new[centre] <= 1:
        u = u_new.copy()
        for i in prange(1, N - 1):
            for j in range(1, N - 1):
                u_new[i * N + j] = (1 - 4 * k) * u[i * N + j] + k * (
                    u[(i - 1) * N + j]
                    + u[(i + 1) * N + j]
                    + u[i * N + j + 1]
                    + u[i * N + j - 1]
                )
        counter += 1

    time_reached = counter * dt

    return u_new,time_reached
```

Below is the sample use of the function with the grid $31 * 31$ and $\Delta t = 1/(31)^2$. It retures the final plate and the time used when the mid of the plate hits 1 Celsius degree.

In [3]:
```python
N = 31
dt = 1/N**2 # implement the condition of stability
x,time_reached = forward_euler(N,dt,t_0_generator(N))
u_final = x.reshape((N, N))

# Plot the results
plt.figure(figsize=(10, 10))

plt.title(f"Forward-Euler without GPU\nTime reached: {time_reached:.12f}")
plt.imshow(u_final, extent=[-1, 1, -1, 1], origin="lower")

plt.colorbar()
plt.show()
```



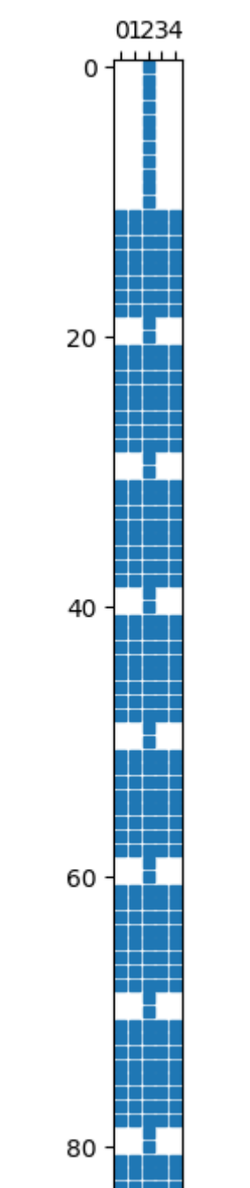
with cuda

In this section, CUDA is used to further accelerate the computation. For that the GPU can run more threads than CPU in parallel, it is more efficient to carry out calculations on GPU when dealing with a considerable large matrix (when the matrix is small, the two are likely to be close in effeciency).

In [3]:
```python
def explicit_matrix_generator_GPU(N, dt):
    size = N**2

    nelements = 5 * size - 16 * N + 16
    d_u = 2 / (N-1)

    k = dt / (d_u**2)

    row_ind = np.zeros(nelements, dtype=np.float64)
    col_ind = np.zeros(nelements, dtype=np.float64)
    data = np.zeros(nelements, dtype=np.float64)

    count = 0

    for i in range(N):
        for j in range(N):
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = i * N + j
                col_ind[count] = 2
                data[count] = 1
                count += 1
            else:
                row_ind[count : count + 5] = i * N + j

                col_ind[count] = 2
                col_ind[count + 1] = 0
                col_ind[count + 2] = 4
                col_ind[count + 3] = 1
                col_ind[count + 4] = 3

                data[count] = -4 * k + 1
                data[count + 1 : count + 5] = 1 * k

                count += 5

    return coo_matrix((data, (row_ind, col_ind)), shape=(size, 5)).tocsc()
```

The matrix generator above produces a $N * 5$ matrix whose shape is shown as below. This structure makes it easier to load data to the shared memory.

In [5]:
```python
plt.figure(figsize=(10, 10))
plt.spy(explicit_matrix_generator_GPU(10,0.01), markersize=5)
plt.show()
```

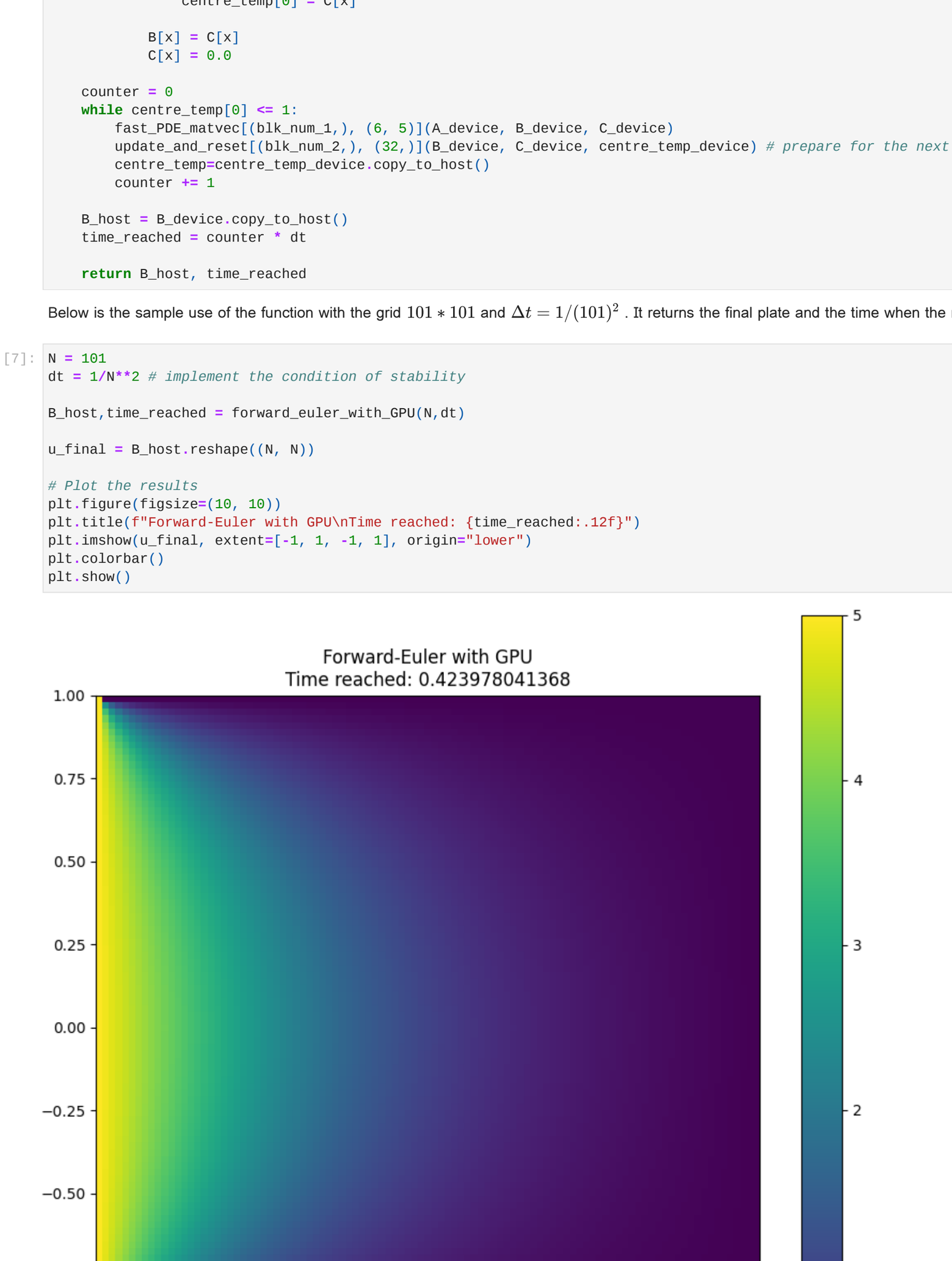


Below is the method. In the first kernel, the shape of each block is set to $6 * 5$ which can deal with 6 points, and it can automatically change the shape of the grid according to the number of grids. The details of the kernels are explained in the comments.

In [4]:
```python
def forward_euler_with_GPU(N, dt):
    N_square = N**2
    N_square_5 = 5 * N_square
    N_square_last_N = N_square - N

    centre = N_square // 2

    blk_num_1=div_ceil(N_square, 6) # calculate the block num in the first kernel
    blk_num_2=div_ceil(N_square, 32) # calculate the block num in the second kernel

    x = t_0_generator(N)
    A = explicit_matrix_generator_GPU(N, dt).toarray()
    b = np.zeros(N_square, dtype=np.float64)
    centre_temp = np.zeros(1, dtype=np.float64)

    # explicitly send the data to device(GPU)
    A_device = cuda.to_device(A)
    B_device = cuda.to_device(x)
    C_device = cuda.to_device(b)
    centre_temp_device = cuda.to_device(centre_temp)

    # a and b is the coeffient from y=a*x**3+b*x which pass the points:
    # (-N,-N),(-1,-1),(0,0),(1,1),(N,N)
    # thus improving the efficiency by removing 'if's
    a=(N-2)/6
    b=(8-N)/6

    @cuda.jit
    def fast_PDE_matvec(A, B, C):
        # load data from global memory to shared memory
        sA = cuda.shared.array(shape=(6, 5), dtype=float64)
        sB = cuda.shared.array(shape=(6, 5), dtype=float64)

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

        sA[tx, ty] = A[x, ty] # load data from A
        cuda.syncthreads()

        if x * 5+y <= N_square_5:
            if (x + tx <= N) or (x >= N_square_last_N):
                if ty == 2:
                    sB[tx, ty] = B[x]
                elif ty == 0 or ty == 1 or ty == 3 or ty == 4:
                    sB[tx, ty] = 0
            else:
                sB[tx, ty] = B[x +int(a*(ty-2)**3+b*(ty-2))] # to avoid using too much ifs

            cuda.syncthreads()

            temp = sA[tx, ty] * sB[tx, ty]
            cuda.atomic.add(C, x, temp) # avoid collision
            cuda.syncthreads()

    @cuda.jit
    def update_and_reset(B, C, centre_temp):
        # prepare for the next loop
        x = cuda.grid(1)
        if x < B.size:
            if x==centre:
                centre_temp[0] = C[x]

            B[x] = C[x]
            C[x] = 0.0

    counter = 0
    while centre_temp[0] <= 1:
        fast_PDE_matvec[(blk_num_1,), (6, 5)](A_device, B_device, C_device)
        update_and_reset[(blk_num_2,), (32,)](B_device, C_device, centre_temp_device) # prepare for the next loop
        centre_temp=centre_temp_device.copy_to_host()
        counter += 1

    B_host = B_device.copy_to_host()
    time_reached = counter * dt

    return B_host, time_reached
```

Below is the sample use of the function with the grid $101 * 101$ and $\Delta t = 1/(101)^2$. It returns the final plate and the time when the mid of the plate hits 1 Celsius degree.

In [7]:
```python
N = 101
dt = 1/N**2 # implement the condition of stability

B_host,time_reached = forward_euler_with_GPU(N,dt)

u_final = B_host.reshape((N, N))

# Plot the results
plt.figure(figsize=(10, 10))
plt.title(f"Forward-Euler with GPU\nTime reached: {time_reached:.12f}")
plt.imshow(u_final, extent=[-1, 1, -1, 1], origin="lower")
plt.colorbar()
plt.show()
```



Comparison (CUDA vs no CUDA)

In [8]:
```python
N = 101
dt = 1/N**2 # implement the condition of stability
print("GPU: ")
%timeit forward_euler_with_GPU(N, dt)
print("CPU: ")
%timeit forward_euler(N,dt,t_0_generator(N))
```

```
GPU:
1.5 s ± 799 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
CPU:
208 ms ± 33.2 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

In [9]:
```python
N = 225
dt = 1/N**2 # implement the condition of stability
print("GPU: ")
%timeit forward_euler_with_GPU(N, dt)
print("CPU: ")
%timeit forward_euler(N,dt,t_0_generator(N))
```

```
GPU:
4.62 s ± 521 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
CPU:
5.37 s ± 819 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

In [10]:
```python
N = 301
dt = 1/N**2 # implement the condition of stability
print("GPU: ")
%timeit forward_euler_with_GPU(N, dt)
print("CPU: ")
%timeit forward_euler(N,dt,t_0_generator(N))
```

```
GPU:
8.22 s ± 555 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
CPU:
17.2 s ± 729 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

It can be seen that when N is small, CPU is faster than GPU. When the N become large, GPU is faster. Thus, I will take the advantage of the both methods to optimize the explicit method with `forward_euler_optimized(N,dt)` :

In [5]:
```python
def forward_euler_optimized(N,dt):
  if N>223:
    u,t=forward_euler_with_GPU(N,dt)
    return u,t
  else:
    u,t=forward_euler(N,dt,t_0_generator(N))
    return u,t
```

Now the new method is able to choose from the two methods automatically depending on the size N.

In [12]:
```python
N = 101
dt = 1/N**2 # implement the condition of stability

B_host,time_reached = forward_euler_optimized(N,dt)

u_final = B_host.reshape((N, N))

# Plot the results
plt.figure(figsize=(10, 10))
plt.title(f"Forward-Euler(optimized)\nTime reached: {time_reached:.12f}")
plt.imshow(u_final, extent=[-1, 1, -1, 1], origin="lower")
plt.colorbar()
plt.show()
```

Convergence of explicit method (Forward-Euler)

In [27]:
```python
N_list_explicit = np.linspace(11, 271, 14)
list_time_explicit = []
for n in N_list_explicit:
    N = int(n)
    dt = 1 / N**2  # implement the condition of stability

    u, time_reached = forward_euler_optimized(N, dt)
    list_time_explicit.append(time_reached)
```

The graph below shows that the result converges well as N become larger and larger:

In [36]:
```python
# Plotting
plt.figure(figsize=(15, 6))
plt.plot(N_list_explicit, list_time_explicit, marker='o', linestyle='-', color='b',label='Calculation Time')
plt.axhline(y=t_expect, color='r', linestyle='-',label='expected time')

# Annotating each data point with its time value
for i, txt in enumerate(list_time_explicit):
    plt.annotate(f"{txt:.5f}", (N_list_explicit[i], list_time_explicit[i]), textcoords="offset points", xytext=(0,10), ha='center')

plt.xlabel('the number of N')
plt.ylabel('time')
plt.title('the result of the time versus the number of N')
plt.legend()
plt.grid(True)
plt.show()
```

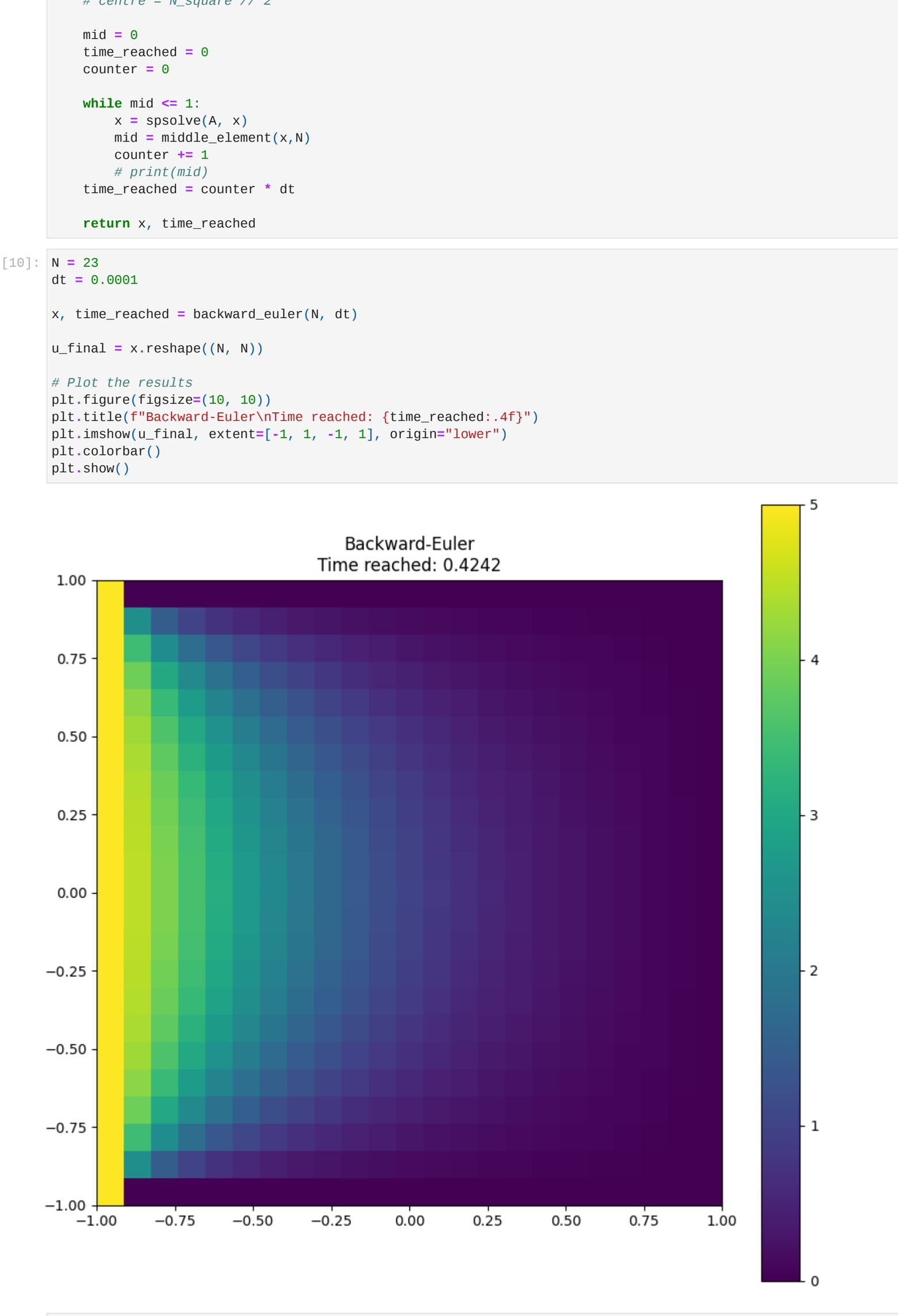
implicit time-stepping (Backward Euler)

For implicit method such as Backward-Euler, they are not sensitive to the size of time step. However, it comes with complex computation to solve the linear equtions, small time steps will greatly increase the running time. Thus, in the following sections, I will use 0.0001 as a fix time step to control the excution time in a reasonable range

In [2]:
```python
def implicit_matrix_generator(N, dt):
    size = N**2
    nelements = 5 * size - 16 * N + 16
    d_u = 2 / (N-1)
    k = dt / (d_u**2)

    row_ind = np.zeros(nelements, dtype=np.float64)
    col_ind = np.zeros(nelements, dtype=np.float64)
    data = np.zeros(nelements, dtype=np.float64)

    count = 0

    for i in range(N):
        for j in range(N):
            if i == 0 or i == N - 1 or j == 0 or j == N - 1:
                row_ind[count] = col_ind[count] = i * N + j
                data[count] = 1
                count += 1
            else:
                row_ind[count : count + 5] = i * N + j

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

                data[count] = 4 * k + 1
                data[count + 1 : count + 5] = -1 * k

                count += 5

    return coo_matrix((data, (row_ind, col_ind)), shape=(size, size)).tocsr()
```

In [3]:
```python
def backward_euler(N, dt):
    x = t_0_generator(N)
    A = implicit_matrix_generator(N, dt)
    # odd = N % 2

    # N_square = N**2

    # centre = N_square // 2

    mid = 0
    time_reached = 0
    counter = 0

    while mid <= 1:
        x = spsolve(A, x)
        mid = middle_element(x,N)
        counter += 1
        # print(mid)
    time_reached = counter * dt

    return x, time_reached
```

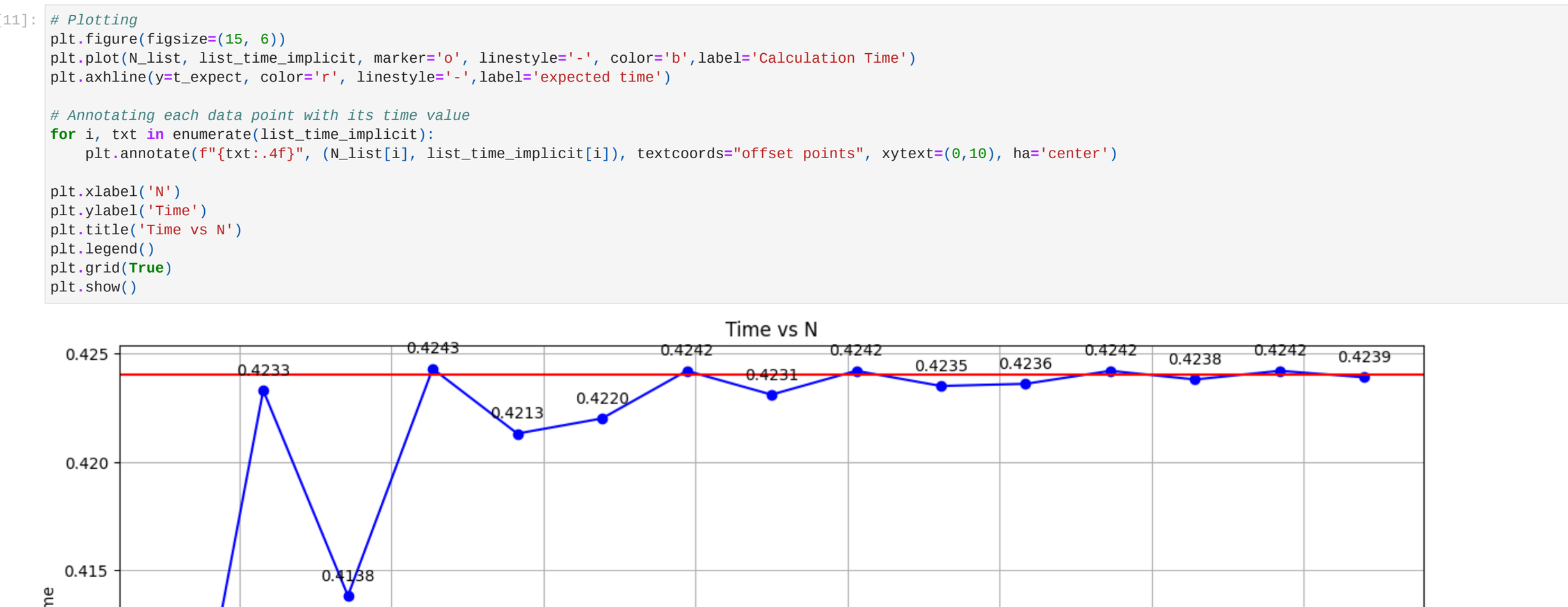
In [10]:
```python
N = 23
dt = 0.0001

x, time_reached = backward_euler(N, dt)

u_final = x.reshape((N, N))

# Plot the results
plt.figure(figsize=(10, 10))
plt.title(f"Backward-Euler\nTime reached: {time_reached:.4f}")
plt.imshow(u_final, extent=[-1, 1, -1, 1], origin="lower")
plt.colorbar()
plt.show()
```



In [7]:
```python
N_list = np.linspace(3, 42, 15)
list_time_implicit = []
for n in N_list:
    N = int(n)
    dt = 0.0001  # implement the condition of stability

    u, time_reached = backward_euler(N, dt)
    list_time_implicit.append(time_reached)
```

Below is the graph of the result calculated by Backward-Euler. Compared to the Forward-Euler method, it experiences oscillation, but still converges well to the expected result.

In [11]:
```python
# Plotting
plt.figure(figsize=(15, 6))
plt.plot(N_list, list_time_implicit, marker='o', linestyle='-', color='b',label='Calculation Time')
plt.axhline(y=t_expect, color='r', linestyle='-',label='expected time')

# Annotating each data point with its time value
for i, txt in enumerate(list_time_implicit):
    plt.annotate(f"{txt:.4f}", (N_list[i], list_time_implicit[i]), textcoords="offset points", xytext=(0,10), ha='center')

plt.xlabel('N')
plt.ylabel('Time')
plt.title('Time vs N')
plt.legend()
plt.grid(True)
plt.show()
```

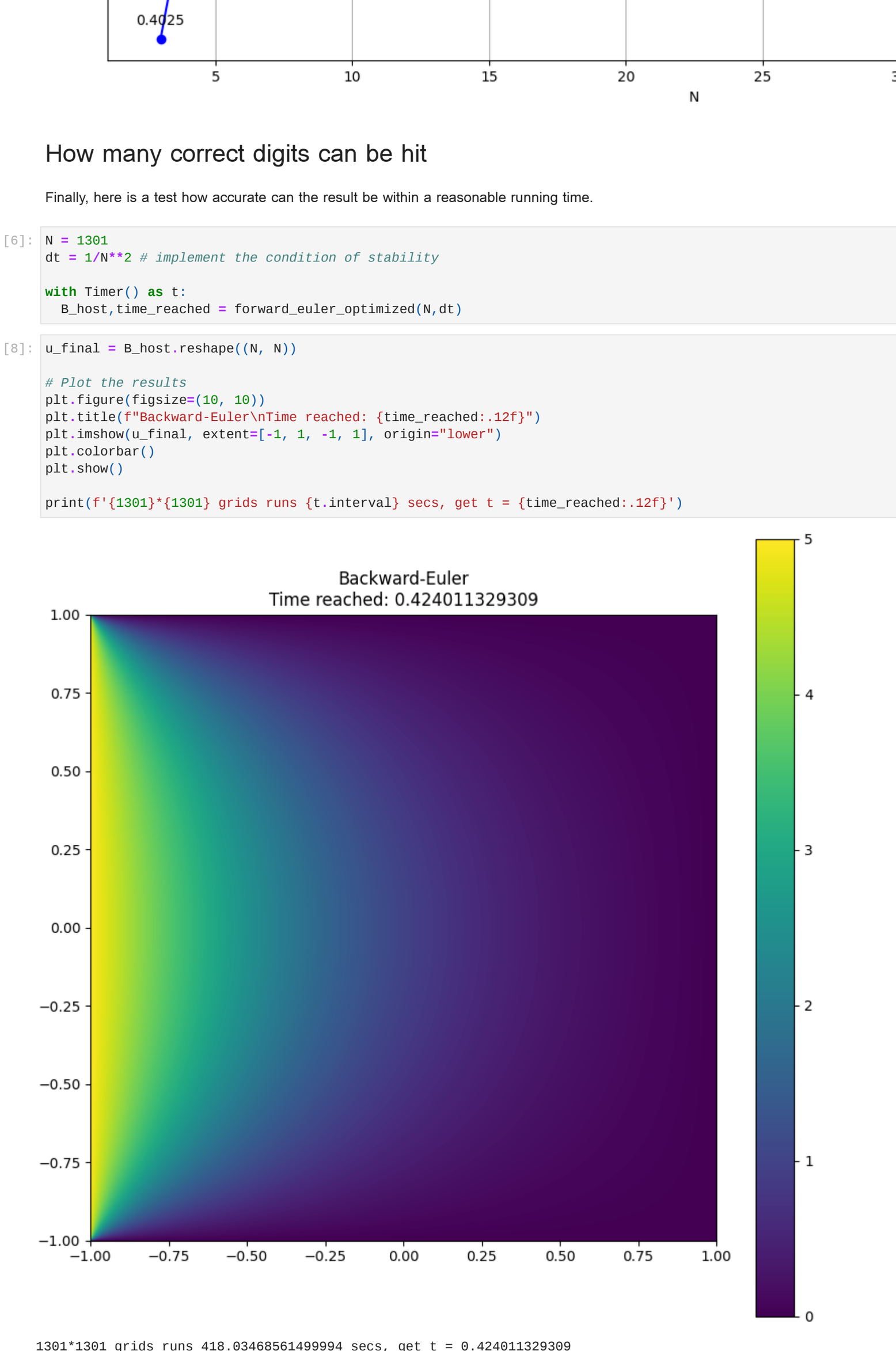


How many correct digits can be hit

Finally, here is a test how accurate can the result be within a reasonable running time.

In [6]:
```python
N = 1301
dt = 1/N**2 # implement the condition of stability

with Timer() as t:
  B_host,time_reached = forward_euler_optimized(N,dt)
```

In [8]:
```python
u_final = B_host.reshape((N, N))

# Plot the results
plt.figure(figsize=(10, 10))
plt.title(f"Backward-Euler\nTime reached: {time_reached:.12f}")
plt.imshow(u_final, extent=[-1, 1, -1, 1], origin="lower")
plt.colorbar()
plt.show()

print(f'{1301}*{1301} grids runs {t.interval} secs, get t = {time_reached:.12f}')
```




```
1301*1301 grids runs 418.03468561499994 secs, get t = 0.424011329309
```

After running for 418 secs, 7 digits are matched to the expected result which is 0.4240113