Class 3 (Monday 24 October)

Contents

These tasks are designed to be worked on in the practical class on Monday 24 October.

In this class, we'll be running code on a GPU using Cuda. To see if you have a device available, you can run:

```
from numba import cuda
cuda.detect()
```

If you don't have a suitable device on your own computer, you should use Google Colab this week: you can use a GPU in Colab by selecting **Runtime -> Change Runtime Type** and selecting **GPU**.

During this class, you may wish to use the GPU accelerated evaluation of particle sums section of the lecture notes, where a similar example is worked through using a radial basis function kernel.

Background

In lots of applications, it is useful to calculate the sum

$$\sum_j c_j g(\mathbf{x}, \mathbf{y}_j),$$

where g is a "kernel" function, $\mathbf{x}$ is a point in $\mathbb{R}^3$, $\mathbf{y}_0, \dots, \mathbf{y}_{n-1}$ are (known) points in $\mathbb{R}^3$, and $c_0, \dots, c_{n-1}$ are (known) values in $\mathbb{C}$.

In this class, we're going to use the acoustic Green's function

$$g(\mathbf{x}, \mathbf{y}) = \mathrm{e}^{-\mathrm{i}k|\mathbf{x}-\mathbf{y}|} / \left(4\pi\,|\mathbf{x}-\mathbf{y}|\right),$$

where k is the wavenumber of the wave. This is the acoustic wave due to a point source: if there are point sources at points $\mathbf{y}_0, \dots, \mathbf{y}_{n-1}$ of sizes $c_0, \dots, c_{n-1}$, then the sum above can be used to compute the magnitude of a (time-harmonic) acoustic wave at each point.

Plotting some waves

There is a point source with wavenumber 10 at the point $(-1.2, 0, 0)$ with magnitude 1. The following code plots a slice through the wave due to this source in the plane $z = 0$ with $0 \leqslant x \leqslant 3$ and $-\frac{3}{2} \leqslant y \leqslant \frac{3}{2}$.

```
import numpy as np
import matplotlib.pylab as plt

k = 10.


def g(x, y):
    """Evaluate real part of the acoustic Green's function."""
    return math.cos(k * np.linalg.norm(x - y)) / 4 / np.pi / np.linalg.norm(x
- y)


sources = np.array([[-1.2, 0., 0.]])
magnitudes = np.array([1.0])

img_size = 250
values = np.empty((img_size, img_size), dtype="complex128")

xmin = 0
xmax = 3
ymin = -1.5
ymax = 1.5

# plt.imshow interprets data as the colour of pixels starting at the top left
then
# row by row. For example, if an image was 5 pixels wide, the order of the
pixels
# would be:
# 0 1 2 3 4
# 5 6 7 8 9
# etc
# Due to this ordering, the y values here might at first glance appear to be
backwards
for i in range(img_size):
    y = ymax + (ymin - ymax) * i / (img_size - 1)
    for j in range(img_size):
        x = xmin + (xmax - xmin) * j / (img_size - 1)
        point = np.array([x, y, 0])
        v = 0
        for m, s in zip(magnitudes, sources):
            v += m * g(s, point)
        values[i, j] = v

plt.imshow(values, extent=[xmin, xmax, ymin, ymax])
plt.show()
```

Adapt this code so that it plots the real part of a wave due to two point sources with magnitude 1 at the points $(-1.2, 0.5, 0)$ and $(-1.2, -0.5, 0)$.

Adapt this code so that it plots the real part of a wave due to four point sources with random magnitudes between 0 and 1 at the points random points in the area $x = -1.2$, $-1 \leqslant y \leqslant 1$, $-1 \leqslant z \leqslant 1$.

GPU acceleration

Write a new version of the code for four sources that runs on a GPU using Numba's Cuda functionality. You should use blocks of 16 by 4 threads (I picked 4 as this is the number of sources), and an appropriately sized grid so that there is a thread for each point you want to compute the wave at.

You may use the function `rbf_evaluation_cuda` from the lecture notes section GPU accelerated evaluation of particle sums) as inspiration for your function. The function in the lecture notes uses the following features of `numba.cuda` that we didn't use in the lecture:

- `cuda.shared.array` creates a shared array. This array can then be used by threads in the same block.
- `cuda.grid` returns the current thread's absolute position in entire grid of threads. For a two-dimensional
- `cuda.syncthreads` synchronises all the threads in the same block. This allows you to ensure that all the threads are ready to perform the next operation at the same time (which is important as a GPU will perform best if all the threads in a block are performing the same operation).

- `cuda.threadIdx.x` and `cuda.threadIdx.y` get the position of the current thread in the current block of threads.

You may wish to create an array of points that you want to evaluate the wave at rather than computing `x` and `y` inside the loops, you could do this either by using two for loops to generate the points or by using `np.mgrid` and `ravel` as done in the GPU accelerated evaluation of particle sums) section.

Create a plot using single precision floating points numbers using your Cuda-accelerated function and a plot using double precision numbers using the standard Python code above. Visually compare the two plots: can you see any differences?

Comparing GPU and CPU acceleration

Write a version of the code that uses `numba.njit(parallel=True)` and `numba.prange` to create the plot in parallel on a CPU. (You may want to use the function `rbf_evaluation` from the lecture notes section GPU accelerated evaluation of particle sums) as inspiration for your function.)

Time your GPU and CPU functions. (For your GPU function, you might want to copy a small array to the device (eg `a = cuda.to_device(np.array([1.]))`) before you start timing to be sure that the time waiting for the GPU to become available on Colab is not included in your timing.) Which is faster?

Extension task

Time your two functions for higher and low numbers of points at which you compute the wave. Create a plot showing the time taken for the two functions as you vary the number of points.