Class 2 (Monday 17 October)

Contents

These tasks are designed to be worked on in the practical class on Monday 17 October.

Experimenting with Numba

Let's start by looking at the matvec code we wrote last week.

```python
import numpy as np

def slow_matvec(matrix, vector):
    assert matrix.shape[1] == vector.shape[0]
    result = []
    for r in range(matrix.shape[0]):
        value = 0
        for c in range(matrix.shape[1]):
            value += matrix[r, c] * vector[c]
        result.append(value)
    return np.array(result)


# Example of using this function
matrix = np.random.rand(3, 3)
vector = np.random.rand(3)
print(slow_matvec(matrix, vector))
print(matrix @ vector)
```

Use `numba.njit` to tell Numba to just-in-time (JIT) compile this function.

Numba appears to be giving incorrect results for this function. This is because Numba interprets `value = 0` as "make an **integer** `value` that is equal to 0", then will not allow `value` to take non-integer values. To fix this, replace `value = 0` with `value = 0.0`.

Using matplotlib, make a plot that shows the time this function takes to compute a matrix-vector product with and without Numba acceleration. Add timings for the `faster_matvec` function that you wrote to this plot. The first time you call your function, it will need to do the JIT compilation: you may want to measure the time the first run takes separately.

`jit` vs `njit`

Add another line to your plot to shw the timings if you use `numba.jit` instead of `numba.njit`. Which is faster?

`numba.njit` will use "no Python mode", while `numba.jit` uses "Python compatibility mode". We would expect `numba.njit` to produce faster code, but `numba.jit` is able to compile a wider range of functions.

Parallel range

Replace any `range`s in your function with `numba.prange`: this will make your function use a parallel for loop. Compare the timings of your function with and without parellel ranges. How big does your matrix need to be before parallelisation becomes worth using?

Optimising your code

Take the fastest version of your function you've obtained to far. Is there anything else you can try doing to it to make it faster? Try a few things and see if you can get any more speed.

Compare the time your function takes to the time Numpy takes to multiply two matrices. How close to Numpy's speed can you get?