14_Optimizing_Loops_with_Numba

January 20, 2016

 Thanks: Gilbert Forsyth for help writing the notebooks. NSF for support via CAREER award 1149784. [@LorenaABarba](https://twitter.com/LorenaABarba)

This notebook complements the interactive CFD online module **12 steps to Navier-Stokes**, addressing the issue of high performance with Python.

0.1 Optimizing Loops with Numba

You will recall from our exploration of array operations with NumPy that there are large speed gains to be had from implementing our discretizations using NumPy-optimized array operations instead of many nested loops.

Numba is a tool that offers another approach to optimizing our Python code. Numba is a library for Python which turns Python functions into C-style compiled functions using LLVM. Depending on the original code and the size of the problem, Numba can provide a significant speedup over NumPy optimized code.

Let's revisit the 2D Laplace Equation:

```
In [27]: from mpl_toolkits.mplot3d import Axes3D
         from matplotlib import cm
         import matplotlib.pyplot as plt
         import numpy as np

         ##variable declarations
         nx = 81
         ny = 81
         c = 1
         dx = 2.0/(nx-1)
         dy = 2.0/(ny-1)

         ##initial conditions
         p = np.zeros((ny,nx)) ##create a XxY vector of 0's

         ##plotting aids
         x = np.linspace(0,2,nx)
         y = np.linspace(0,1,ny)

         ##boundary conditions
         p[:,0] = 0              ##p = 0 @ x = 0
         p[:,-1] = y              ##p = y @ x = 2
         p[0,:] = p[1,:]              ##dp/dy = 0 @ y = 0
         p[-1,:] = p[-2,:]        ##dp/dy = 0 @ y = 1
```

Here is the function for iterating over the Laplace Equation that we wrote in Step 9:

```
In [17]: def laplace2d(p, y, dx, dy, l1norm_target):
             l1norm = 1
             pn = np.empty_like(p)

             while l1norm > l1norm_target:
                 pn = p.copy()
                 p[1:-1,1:-1] = (dy**2*(pn[2:,1:-1]+pn[0:-2,1:-1])+dx**2*(pn[1:-1,2:]+pn[1:-1,0:-2]))/(
                 p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))/(2*(dx**2+dy**2))
                 p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))

                 p[:,0] = 0                ##p = 0 @ x = 0
                 p[:,-1] = y                ##p = y @ x = 2
                 p[0,:] = p[1,:]                ##dp/dy = 0 @ y = 0
                 p[-1,:] = p[-2,:]        ##dp/dy = 0 @ y = 1
                 l1norm = (np.sum(np.abs(p[:])-np.abs(pn[:])))/np.sum(np.abs(pn[:]))

             return p
```

Let's use the `%%timeit` cell-magic to see how fast it runs:

```
In [28]: %%timeit
         laplace2d(p, y, dx, dy, .00001)

1 loops, best of 3: 206 us per loop
```

Ok! Our function `laplace2d` takes around 206 micro-seconds to complete. That's pretty fast and we have our array operations to thank for that. Let's take a look at how long it takes using a more 'vanilla' Python version.

```
In [29]: def laplace2d_vanilla(p, y, dx, dy, l1norm_target):
             l1norm = 1
             pn = np.empty_like(p)
             nx, ny = len(y), len(y)

             while l1norm > l1norm_target:
                 pn = p.copy()

                 for i in range(1, nx-1):
                     for j in range(1, ny-1):
                         p[i,j] = (dy**2*(pn[i+1,j]+pn[i-1,j])+dx**2*(pn[i,j+1]-pn[i,j-1]))/(2*(dx**2+d

                 p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))/(2*(dx**2+dy**2))
                 p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))

                 p[:,0] = 0                ##p = 0 @ x = 0
                 p[:,-1] = y                ##p = y @ x = 2
                 p[0,:] = p[1,:]                ##dp/dy = 0 @ y = 0
                 p[-1,:] = p[-2,:]        ##dp/dy = 0 @ y = 1
                 l1norm = (np.sum(np.abs(p[:])-np.abs(pn[:])))/np.sum(np.abs(pn[:]))

             return p

In [30]: %%timeit
         laplace2d_vanilla(p, y, dx, dy, .00001)
```

```
10 loops, best of 3: 32 ms per loop
```

The simple Python version takes 32 milli-seconds to complete. Let's calculate the speedup we gained in using array operations:

```
In [35]: 32*1e-3/(206*1e-6)
```

```
Out[35]: 155.33980582524273
```

So NumPy gives us a 155x speed increase over regular Python code. That said, sometimes implementing our discretizations in array operations can be a little bit tricky.

Let's see what Numba can do. We'll start by importing the special function decorator `autojit` from the `numba` library:

```
In [36]: from numba import autojit
```

To integrate Numba with our existing function, all we have to do it is prepend the `@autojit` function decorator before our `def` statement:

```
In [38]: @autojit
         def laplace2d_numba(p, y, dx, dy, l1norm_target):
             l1norm = 1
             pn = np.empty_like(p)

             while l1norm > l1norm_target:
                 pn = p.copy()
                 p[1:-1,1:-1] = (dy**2*(pn[2:,1:-1]+pn[0:-2,1:-1])+dx**2*(pn[1:-1,2:]+pn[1:-1,0:-2]))/(2
                 p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))/(2*(dx**2+dy**2))
                 p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))

                 p[:,0] = 0                ##p = 0 @ x = 0
                 p[:,-1] = y                ##p = y @ x = 2
                 p[0,:] = p[1,:]                ##dp/dy = 0 @ y = 0
                 p[-1,:] = p[-2,:]        ##dp/dy = 0 @ y = 1
                 l1norm = (np.sum(np.abs(p[:])-np.abs(pn[:])))/np.sum(np.abs(pn[:]))

             return p
```

The only lines that have changed are the `@autojit` line and also the function name, which has been changed so we can compare performance. Now let's see what happens:

```
In [39]: %%timeit
         laplace2d_numba(p, y, dx, dy, .00001)
```

```
1 loops, best of 3: 137 us per loop
```

Ok! So it's not a 155x speed increase like we saw between vanilla Python and NumPy, but it is a non-trivial gain in performance time, especially given how easy it was to implement. Another cool feature of Numba is that you can use the `@autojit` decorator on non-array operation functions, too. Let's try adding it onto our vanilla version:

```
In [41]: @autojit
         def laplace2d_vanilla_numba(p, y, dx, dy, l1norm_target):
             l1norm = 1
             pn = np.empty_like(p)
             nx, ny = len(y), len(y)
```

```
        while l1norm > l1norm_target:
            pn = p.copy()

            for i in range(1, nx-1):
                for j in range(1, ny-1):
                    p[i,j] = (dy**2*(pn[i+1,j]+pn[i-1,j])+dx**2*(pn[i,j+1]-pn[i,j-1]))/(2*(dx**2+dy

            p[0,0] = (dy**2*(pn[1,0]+pn[-1,0])+dx**2*(pn[0,1]+pn[0,-1]))/(2*(dx**2+dy**2))
            p[-1,-1] = (dy**2*(pn[0,-1]+pn[-2,-1])+dx**2*(pn[-1,0]+pn[-1,-2]))/(2*(dx**2+dy**2))

            p[:,0] = 0                ##p = 0 @ x = 0
            p[:,-1] = y                ##p = y @ x = 2
            p[0,:] = p[1,:]                ##dp/dy = 0 @ y = 0
            p[-1,:] = p[-2,:]        ##dp/dy = 0 @ y = 1
            l1norm = (np.sum(np.abs(p[:])-np.abs(pn[:])))/np.sum(np.abs(pn[:]))

        return p
```

In [42]:
```
%%timeit
laplace2d_vanilla_numba(p, y, dx, dy, .00001)
```

```
1 loops, best of 3: 561 us per loop
```

561 micro-seconds. That's not quite the 155x increase we saw with NumPy, but it's close. And all we did was add one line of code.

So we have:

Vanilla Python: 32 milliseconds

NumPy Python: 206 microseconds

Vanilla + Numba: 561 microseconds

NumPy + Numba: 137 microseconds

Clearly the NumPy + Numba combination is the fastest, but the ability to quickly optimize code with nested loops can also come in very handy in certain applications.

In [ ]:

In [ ]:

In [42]:

In [1]:
```
from IPython.core.display import HTML
def css_styling():
    styles = open("../styles/custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[1]: <IPython.core.display.HTML at 0x36fbb10>

(The cell above executes the style for this notebook. We modified a style we found on the GitHub of CamDavidsonPilon, [@Cmrn_DP](https://twitter.com/cmrn_dp).)