Speeding up Python code with Numba

Pawel Pomorski ppomorsk@sharcnet.ca

June 1, 2022

Overview

Python is a language widely used in scientific computing, thanks to easy to use syntax that permits fast code development.

However, it is an interpreted language, so it can be much slower than a compiled language.

When this becomes a concern, there are various ways to address it.

Could rewrite (parts of) code in a compiled language but that is time consuming.

Numba offers a way to convert Python code into compiled code automatically.

The strategy is to convert only those functions which are responsible for the bulk of the runtime (identified via profiling).

Only a subset of Python is currently supported by Numba.

Installing Numba

Run once to install

```
module load StdEnv/2020
module load python/3.8
virtualenv --no-download ~/ENV
source ~/ENV/bin/activate
pip install --no-index --upgrade pip
pip install --no-index numba
```

To use afterwards:

```
source ~/ENV/bin/activate
python code_with_numba.py
```

Numba syntax

To use Numba, often all you need to do is add the decorator

```
@jit
```

to the function you want to compile. Numba will attempt to compile what it can, but will run the rest in Python interpreter which can be slow.

To force Numba to compile everything, use

```
@jit(nopython=True)
```

To save typing, this is equivalent to

```
@njit
```

If there are no warnings or errors when you run the program, then Numba was able to compile the specified functions without any problem.

Numba compiling

First call to the function will include the compilation, so it will run longer than subsequent calls. Keep this in mind when doing timing. There is a precompile mode if this is an issue.

Numba will figure out the variable types to be used in compiling based on arguments provided when function is called.

It is possible to specify variable types for function arguments in the decorator, but it is better to let Numba figure them out on its own.

Euler 39 problem

If p is the perimeter of a right angle triangle with integral length sides, (a,b,c), there are exactly three solutions for p = 120.

(20,48,52), (24,45,51), (30,40,50)

For which value of p $\leq$ 1000, is the number of solutions maximised?

```
def find_num_solutions(p):
    n=0
    for a in range(1,p//2):
        for b in range(a,p//2):
            c=p-a-b
            if(a*a+b*b==c*c):
                n=n+1
    return n

# Run find_num_solutions for arguments
# up to 1000 and find maximum
```

Euler 39 problem with Numba

```
from numba import njit

@njit
def find_num_solutions(p):
    n=0
    for a in range(1,p//2):
        for b in range(a,p//2):
            c=p-a-b
            if(a*a+b*b==c*c):
                n=n+1
    return n
```

This runs 273 times faster.

Numba and Python modules

Numba has to know about a module to work with it

Trying to put unknown module calls inside functions compiled with Numba can slow your code down.

For example, Numba does not know about Pandas.

However, Numba is compatible with NumPy.

NumPy example - 2D diffusion equation

```
import numpy as np
def iterate(u,udt,fac,kappa):
# compute value after evolving over single time step
    for i in range(1,u.shape[0]-1):
        for j in range(1,u.shape[1]-1):
            udt[i,j]=fac*u[i,j]+ \
            kappa*(u[i+1,j]+u[i-1,j]+ \
            u[i,j+1]+u[i,j-1])
    u[:,:]=udt[:,:]

u=np.empty((1000,1000),dtype="float64")
udt=np.empty_like(u)
# initialize u
...
for i in range(100):
    iterate(u,udt,fac,kappa)
...
```

2D diffusion equation with Numba

```
from numba import njit
@njit
def iterate(u,udt,fac,kappa):
# compute value after evolving over single time step
    for i in range(1,u.shape[0]-1):
        for j in range(1,u.shape[1]-1):
            udt[i,j]=fac*u[i,j]+ \
            kappa*(u[i+1,j]+u[i-1,j]+ \
            u[i,j+1]+u[i,j-1])
    u[:,:]=udt[:,:]
```

The Numba version runs 309 times faster.

Diffusion with NumPy without loops

```
def iterate(u,udt,fac,kappa):
# udt[i,j]=fac*u[i,j]+ \
#             kappa*(u[i+1,j]+u[i-1,j] + \
#             u[i,j+1]+u[i,j-1])
    udt[1:-1,1:-1]=fac*u[1:-1,1:-1] + \
                kappa*(u[2:,1:-1]+u[0:-2,1:-1] + \
                u[1:-1,2:]+u[1:-1,0:-2])
    u[:,:]=udt[:,:]
```

This runs 157 times faster than Python with loops, but still 2.0 times slower than with Numba.

Note that compiling this no-loop function with numba actually results in somewhat slower execution (1.4 times slower). The no-loop function already uses compiled code, so compiling again decreases performance.

Problems with Numba

Codes shown so far works fine worked without issues.

However, Numba supports only a subset of Python (likely to grow with time).

An existing code that you are trying to convert to use Numba is likely to have features outside of the supported subset.

To use Numba in such a code, need to modify it to avoid features Numba does not understand.

No need to change the whole code, only those functions in which code spends a lot of time.

The difficulty of doing this will vary.

Particular problem area - lists

Lists in Python can be very complex objects. Numba has trouble dealing with complex lists.

Need to replace lists with NumPy arrays to avoid problems.

List operations will have to be translated into NumPy operations.

Can translate back from NumPy array to list when Numba operations are done.

Example 1: Deepcopy of list does not work in Numba

```
import copy
from numba import njit

@njit
def copyfunc(a):

    b=copy.deepcopy(a)
    b[0]=1000
    return b

a=[1,2,3,4]
c=copyfunc(a)
```

Does not compile.

Convert list to NumPy array

```
import copy
import numpy as np
from numba import njit

@njit
def copyfunc(a):
    b=np.copy(a)
    b[0]=1000
    return b

a=[1.0,2.0,3.0,4.0]

a=np.array(a) # convert to NumPy array
c=copyfunc(a)
c=c.tolist() # convert back to list
```

Example 2: Filtering a list of lists

```
>>> a=[  [1,1,1],[9,0,9],[9,9,9],[2,3,4] ]
>>> a=list(filter(lambda x: x != [9, 9, 9], a))
>>> print(a)
[[1, 1, 1], [9, 0, 9], [2, 3, 4]]
```

Numba has problems filtering list of lists

```
from numba import njit
@njit
def applyfilter(a):

    a=list(filter(lambda x: x != [9, 9, 9], a))

    return a

a=[[1,1,1],[2,2,2],[9,9,9],[2,3,4]]
a=applyfilter(a)
```

Output:

```
lambda.py:15: NumbaPendingDeprecationWarning:
...
TypeError: Failed in nopython mode pipeline ...
cannot reflect element of reflected container ...
```

Convert to NumPy array then filter

```
>>> a=[  [1,1,1],[9,0,9],[9,9,9],[2,3,4] ]
>>> a=np.array(a)
>>> print(a)
[[1 1 1]
 [9 0 9]
 [9 9 9]
 [2 3 4]]
>>> a=a[np.logical_or( \
... np.logical_or(a[:,0] !=9,a[:,1] != 9) \
... ,a[:,2] != 9)]
>>> print(a)
[[1 1 1]
 [9 0 9]
 [2 3 4]]
```

Filtering a list with Numba

```
import numpy as np
from numba import njit

@njit
def applyfilter(a):

    a=a[np.logical_or( \
        np.logical_or(a[:,0] !=9,a[:,1] != 9) \
        ,a[:,2] != 9)]

    return a

a=[[1,1,1],[2,2,2],[9,9,9],[2,3,4]]

a=np.array(a)
a=applyfilter(a)
a=a.tolist()
```

Parallel Numba

Automatic parallelization is possible with Numba

```
@njit(parallel=True)
```

Only works if code is parallelizable.

Compiling code for the GPU also possible.

This will be covered in Compute Ontario Summer School at the end of June

https://training.computeontario.ca/