

Tutorial: when Numpy isn't fast enough...

A tutorial on using fortran/blas under the hood of your python program for a 6x speed pickup.

Posted by iamtrask on November 23, 2014

Summary: a demo on how to use fortran/blas libraries under the hood of your python program's vector operations to squeeze out extra speed over Numpy.

Yesterday, I posted a on how to use Apache Spark with GPUs from a notebook. To my joy, it reached the first page of Hacker News (while serving the Scala community!!!). Using Spark from one of the iPython notebooks has become a real passion of mine... and whereas yesterday I focused on Scala/JVM/GPU operations, today I want to offer a bit up to the scientific Python community. These discoveries are from studying a wonderful codebase by Radim Rehurek called Gensim... specifically the word2vec implementation.

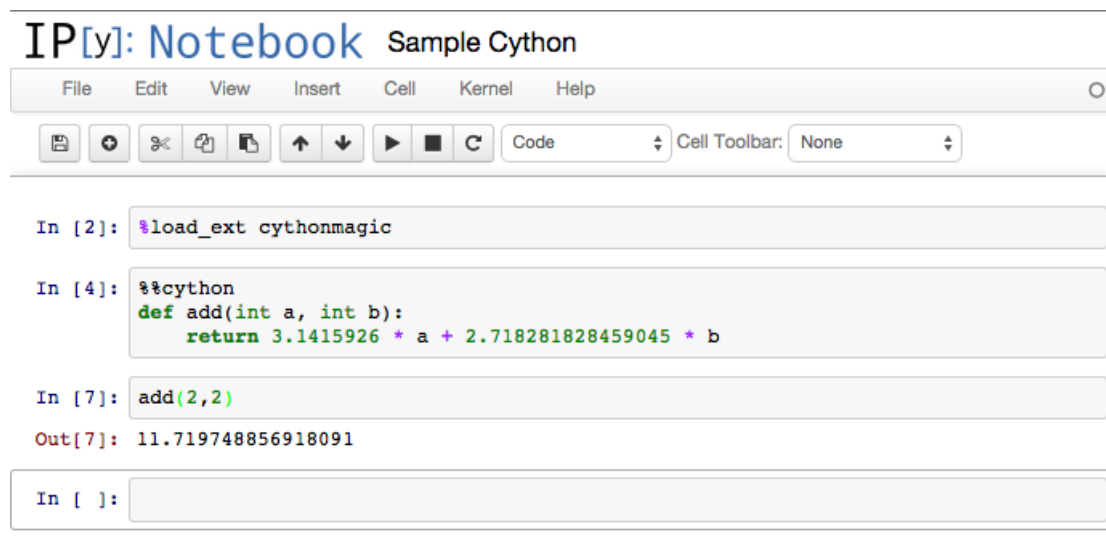
You might be wondering why I would cover CPU based speedups following GPU based... and the truth is that sometimes lighter weight optimiations are a better fit... especially when dealing with smaller batches of vectors at a time or when GPUs simply aren't available.

Part 1: iPython-Notebook Cython Magic

Install Numpy, Scipy, GFortran, Cython, and scikit-learn packages. I HIGHLY recommend sticking to easy_install, brew (apt-get), and pip. In my experience, macports has some real trouble with these packages. Also, of course, you need to have ipython notebook installed for these examples to work, but technically it can work for normal cython too.

With Cython you'll get the "Cython" magic as well. The following command should work in your

notebook.



```
IP[y]: Notebook Sample Cython
File Edit View Insert Cell Kernel Help
[Icons] Code Cell Toolbar: None

In [2]: %load_ext cythonmagic

In [4]: %%cython
def add(int a, int b):
    return 3.1415926 * a + 2.718281828459045 * b

In [7]: add(2,2)
Out[7]: 11.719748856918091

In [ ]:
```

Notice that you load the cython magic using "%load_ext cythonmagic" and then compile cython using "%cython" at the top of the cell containing cython code. You can then call your cython functions (or classes... etc) from python. It's a neat system. :)

Part 2: Scipy Fortran-Blas in Cython

Below you'll see the core code that we need to get our superfast blas operations. After the first few imports, you'll see a "cdef extern from" import from a file called voidptr.h. This file allows us to cast a numpy array to its pointer without copying any data.... a key part of the code. The contents of that file are also below.

```
In [7]: %%cython --annotate

import cython
import numpy as np
cimport numpy as np

from libc.math cimport exp
from libc.string cimport memset

from scipy.linalg.blas import fblas

REAL = np.float32
ctypedef np.float32_t REAL_t

cdef extern from "/Users/myname/Laboratory/voidptr.h":
    void *PyObject_AsVoidPtr(object obj)

ctypedef void (*scopy_ptr) (const int *N, const float *X, const int *incX, float *Y, const int *incY)
ctypedef void (*saxpy_ptr) (const int *N, const float *alpha, const float *X, const int *incX, float *Y, const int *incY)
ctypedef float (*sdot_ptr) (const int *N, const float *X, const int *incX, const float *Y, const int *incY)
ctypedef double (*dsdot_ptr) (const int *N, const float *X, const int *incX, const float *Y, const int *incY)
ctypedef double (*snrm2_ptr) (const int *N, const float *X, const int *incX) nogil
ctypedef void (*sscal_ptr) (const int *N, const float *alpha, const float *X, const int *incX) nogil

cdef scopy_ptr scopy=<scopy_ptr>PyObject_AsVoidPtr(fblas.scopy._cpointer) # y = x
cdef saxpy_ptr saxpy=<saxpy_ptr>PyObject_AsVoidPtr(fblas.saxpy._cpointer) # y += alpha * x
cdef sdot_ptr sdot=<sdot_ptr>PyObject_AsVoidPtr(fblas.sdot._cpointer) # float = dot(x, y)
cdef dsdot_ptr dsdot=<dsdot_ptr>PyObject_AsVoidPtr(fblas.sdot._cpointer) # double = dot(x, y)
cdef snrm2_ptr snrm2=<snrm2_ptr>PyObject_AsVoidPtr(fblas.snrm2._cpointer) # sqrt(x^2)
cdef sscal_ptr sscal=<sscal_ptr>PyObject_AsVoidPtr(fblas.sscal._cpointer) # x = alpha * x

cdef int ONE = 1
cdef REAL_t ONEF = <REAL_t>1.0

def pubDotty(syn0, syn1, size):
    cdef int lSize = size
    f = <REAL_t>dsdot(&lSize, <REAL_t *>(np.PyArray_DATA(syn0)), &ONE, <REAL_t *>(np.PyArray_DATA(syn1)))
    return f
```

[voidptr.h code on Github](#)

Next, you'll also see six function types and their implementations. There is a whole suite of these funky-named fortran functions in the [Scipy Blas/Fortran Documentation](#) I also write a simple dot-product function leveraging the dsdot (double dot product... as opposed to float) called pubDotty.

```

File Edit View Insert Cell Kernel Help
[Icons] Code Cell Toolbar: None

34:
35: def pubDotty(syn0,syn1,size):
36:     cdef int lSize = size
37:     f = <REAL_t>dscdot(&lSize,<REAL_t *>(np.PyArray_DATA(syn0)),&ONE,<
REAL_t *>(np.PyArray_DATA(syn1)),&ONE)
38:     return f

In [10]: np.random.seed(0)
x = np.zeros(32,dtype='f')
y = np.zeros(32,dtype='f')

x += 1
y += 3

In [11]: %timeit np.dot(x,y)
1000000 loops, best of 3: 1.33 µs per loop

In [12]: %timeit pubDotty(x,y,len(x))
1000000 loops, best of 3: 226 ns per loop

In [13]: pubDotty(x,y,len(x))
Out[13]: 96.0

In [14]: np.dot(x,y)
Out[14]: 96.0

```

In this example, I create two numpy vectors of length 32. (one full of ones and another full of threes). I then benchmark and show how the cython/fortran version is **5.8x faster**. It should be noted that this is still passing in a python object... this efficiency gain increases when everything stays in cython for several progressive operations.

← PREVIOUS POST

NEXT P →



Copyright © i am trask 2017