Using Python with Cython

cs512

Step 1: Define a cython function in a .pyx file

Place the following inside myprime.pyx (note the use of typed function arguments and the use of cdef to define typed variables)

```
def primes(int kmax):
    cdef int n, k, i
    cdef int p[1000]
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result
```

Step 2: Define a setup.py file

```
from distutils.core import setup
from Cython.Build import cythonize

setup(
  name = 'My prime module',
  ext_modules = cythonize("myprime.pyx"),
)
```

Step 3: Compile the cython module

Execute from within a terminal:

```
python setup.py build_ext --inplace
```

Step4: Load the compiled module and test

```
import myprime
myprime.primes(1000000000);
```

Timing test

```
# Make sure the cython function is compiled (optional)
from subprocess import call
call(["python setup.py build_ext --inplace"], shell = True)

import numpy as np
import myprime
import time

def primes(kmax):
#    def  n, k, i
    p = np.zeros(1000)
    result = []
    if kmax > 1000:
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result


start_time = time.time();
primes(1000000000);
print("--- Python function: %s seconds ---" % (time.time() - start_time));
```

```
start_time = time.time();
myprime.primes(1000000000);
print("--- Cython function: %s seconds ---" % (time.time() - start_time));
```

Output

```
--- Python function: 0.185851097107 seconds ---
--- Cython function: 0.00182819366455 seconds ---
```

Efficient indexing

Array lookups and assignments and C/Python types conversions degrade performance. The [] operator still uses full Python operations and should be replaced to access the data buffer directly at C speed. To do this we need to type the contents of ndarray objects using memoryview. memoryviews are C structures that hold a pointer to the data of a NumPy array as well as additional metadata. They can be indexed by C integers allowing fast access to NumPy array data. No data is copied when creating a memory view.

Example of declaring a memoryview of integers:

```
cdef int [:] foo       # 1D memoryview
cdef int [:, :] foo    # 2D memoryview
cdef int [:, :, :] foo # 3D memoryview
```

Example of using memoryviews to compute A*a+B*b+c:

```
import numpy as np

def compute(int[:, :] array_1, int[:, :] array_2, int a, int b, int c):

    cdef Py_ssize_t x_max = array_1.shape[0]
    cdef Py_ssize_t y_max = array_1.shape[1]

    assert tuple(array_1.shape) == tuple(array_2.shape) # array_1.shape is a C array

    result = np.zeros((x_max, y_max), dtype=np.intc)
    cdef int[:, :] result_view = result    # <==== memoryview of output

    cdef Py_ssize_t x, y

    for x in range(x_max):
        for y in range(y_max):
            result_view[x, y] = array_1[x, y] * a + array_2[x, y] * b + c

    return result
```

Array lookups are slowed down by index validity checks (bounds and non-negative checks). These can be disabled by adding in the beginning of the file:

```
cimport cython
@cython.boundscheck(False)  # Deactivate bounds checking
@cython.wraparound(False)   # Deactivate negative indexing.
@cython.nonecheck(False)    # Deactivate test that a variable is not set to none
```

To have Cython infer C types of variables automatically (in case you forget to type variables) use:

```
# cython: infer_types=True
```

Using multiple threads

Cython supports OpenMP for multiple thread computations. When using elementwise operations it is easy to distribute work among multiple threads.

```
# distutils: extra_compile_args=-fopenmp
# distutils: extra_link_args=-fopenmp

import numpy as np
cimport cython
from cython.parallel import prange

# Define a fused type to support multiple array types (int, double, and long long)
ctypedef fused my_type:
    int
    double
    long long

@cython.boundscheck(False)
@cython.wraparound(False)

def compute(my_type[:, ::1] array_1, my_type[:, ::1] array_2, my_type a, my_type b, my_type c):

    cdef Py_ssize_t x_max = array_1.shape[0]
    cdef Py_ssize_t y_max = array_1.shape[1]

    assert tuple(array_1.shape) == tuple(array_2.shape)

    if my_type is int:
        dtype = np.intc
    elif my_type is double:
        dtype = np.double
    elif my_type is cython.longlong:
        dtype = np.longlong

    result = np.zeros((x_max, y_max), dtype=dtype)
    cdef my_type[:, ::1] result_view = result

    cdef Py_ssize_t x, y

    # We use prange here and nogil (release GIL - global interpreter lock)
    for x in prange(x_max, nogil=True):
        for y in range(y_max):
            result_view[x, y] = array_1[x, y] * a + array_2[x, y] * b + c

    return result
```

When using a Jupyter notebook add in the beginning:

```
%%cython --force
# distutils: extra_compile_args=-fopenmp
# distutils: extra_link_args=-fopenmp
```

Improving performance

- To test performance bottlenecks execute:

  ```
  cython myprime.pyx -a # generates myprime.html
  ```

  Open myprime.html in a browser. Lines highlighted in yellow are still using Python. Try to eliminate yellow lines and especially inside loops.

- Replace python functions with C functions. E.g.:

  ```
  from math import exp   ==>   from libc.math cimport exp
  ```

- Use `cdef` for functions you call frequently. E.g.:

  ```
  def primes(int kmax):  ==>   cdef primes(int kmax):
  ```

- Add compiler directives to cancel runtime tests:

  ```
  #cython: boundscheck=False, wraparound=False, nonecheck=False
  ```

- Add compiler flags. E.g. add `--ffast-math` in setup.py:

  ```
  from distutils.core import setup
  from distutils.extension import Extension
  from Cython.Distutils import build_ext

  ext_modules=[ Extension("myprime",
                          ["myprime.pyx"],
                          libraries=["m"],
                          extra_compile_args = ["-ffast-math"])]

  setup(
    name = "myprime",
    cmdclass = {"build_ext": build_ext},
    ext_modules = ext_modules)
  ```

Debugging

- To check what Cython is doing (e.g. if you do not get faster execution) execute in a terminal:

```
cython myfile.pyx -a
```

then open the html file that is generated and check it.

```
Generated by Cython 0.28.6

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that

Raw output: myprime.c

+01: def primes(int kmax):
 02:     cdef int n, k, i
 03:     cdef int p[1000]
+04:     result = []
+05:     if kmax > 1000:
+06:         kmax = 1000
+07:     k = 0
+08:     n = 2
+09:     while k < kmax:
+10:         i = 0
+11:         while i < k and n % p[i] != 0:
+12:             i = i + 1
+13:         if i == k:
+14:             p[k] = n
+15:             k = k + 1
+16:             result.append(n)
+17:         n = n + 1
+18:     return result
 19:
```