```
#!/usr/bin/env python
"""
Vector addition using PyOpenCL.
"""

import time

import pyopencl as cl
import pyopencl.array
import numpy as np

# Select the desired OpenCL platform; you shouldn't need to change this:
NAME = 'NVIDIA CUDA'
platforms = cl.get_platforms()
devs = None
for platform in platforms:
    if platform.name == NAME:
        devs = platform.get_devices()

# Set up a command queue; we need to enable profiling to time GPU operations:
ctx = cl.Context(devs)
queue = cl.CommandQueue(ctx,
properties=cl.command_queue_properties.PROFILING_ENABLE)

# Define the OpenCL kernel you wish to run; most of the interesting stuff you
# will be doing involves modifying or writing kernels:
kernel = """ #pragma OPENCL EXTENSION cl_khr_fp64 : enable
__kernel void func( int  a, int b, __global double* x, __global double* c) {
    unsigned int i = get_global_id(0);

        c[i] =   c[i]+a*cos(x[i])+b*sin(x[i]);

}
"""

# Load some random data to process. Note that setting the data type is
# important; if your data is stored using one type and your kernel expects a
# different type, your program might either produce the wrong results or fail to
# run.  Note that Numerical Python uses names for certain types that differ from
# those used in OpenCL. For example, np.float32 corresponds to the float type in
# OpenCL:
N = 4
a_max = 5
```

```
b_max = 7
a = np.random.randint(0,a_max,N)
b = np.random.randint(0,b_max,N)
dx = 0.05
x_max = 6.0
x = np.arange(0.0,x_max,dx)
xcopy = x
k=np.zeros_like(x)
# We can use PyOpenCL's Array type to easily transfer data from numpy arrays to
# GPU memory (and vice versa):
a_gpu = cl.array.to_device(queue, a)
b_gpu = cl.array.to_device(queue, b)
k_gpu = cl.array.to_device(queue, k)
c_gpu = cl.array.zeros(queue, x.shape, x.dtype)

# Launch the kernel; notice that you must specify the global and locals to
# determine how many threads of execution are run. We can take advantage of Numpy to
# use the shape of one of the input arrays as the global size. Since our kernel
# only accesses the global work item ID, we simply set the local size to None:
prg = cl.Program(ctx, kernel).build()

for i in xrange(0,N):
        x_gpu = cl.array.to_device(queue,x)
        prg.func(queue, x.shape, None, np.uint32(a[i]), np.uint32(b[i]) ,x_gpu.data, c_gpu.data)
        x = x+xcopy

# Retrieve the results from the GPU:
c = c_gpu.get()

y=np.zeros_like(x)

x = xcopy
for i in xrange(0, N):
    y += a[i]*np.cos((i+1)*x)+b[i]*np.sin((i+1)*x)

print 'input (a):    ', a
print 'input (b):    ', b
print 'numpy (a+b):  ', y
print 'opencl (a+b): ', c

# Compare the results from the GPU with those obtained using Numerical Python;
# this should print True:
print 'equal:        ', np.allclose(y, c)
```

```
# Here we compare the speed of performing the vector addition with Python and
# PyOpenCL. Since the execution speed of a snippet of code may vary slightly at
# different times depending on what other things the computer is running, we run
# the operation we wish to time several times and average the results:

M = 4
times = []
for f in xrange(M):
    start = time.time()
    for i in xrange(0, N):
        y += a[i]*np.cos((i+1)*x)+b[i]*np.sin((i+1)*x)
    times.append(time.time()-start)

print 'python time:  ', np.average(times)


times = []
for f in xrange(M):
    start = time.time()
    for i in xrange(0,N):
        x_gpu = cl.array.to_device(queue,x)
        prg.func(queue, x.shape, None, np.uint32(a[i]), np.uint32(b[i]) ,x_gpu.data, c_gpu.data)
        x = x+xcopy
    times.append(time.time()-start)
print 'opencl time:  ', np.average(times)

# Notice that the
# that of the PyOpenCL code for very short arrays. This is because data
# transfers between host memory and that of the GPU are relatively slow. Try
# gradually increasing the number of elements in a and b up to 100000 and see
# what happens.




#References: Class notes, Khronos, StackOverflow.
```