

# Transfer Entropy optimization using CUDA/OpenCL and Python. First deliverable

May, 2<sup>nd</sup> 2014

This text is a small guide of the first delivery. We will start with a small introduction about the files included in the .zip, and what are they used for. After that, we will check the requisites needed in our computer to execute the program, and we will give a short explanation about how to test the scripts, and how to call them from different files. To complete this readme text, we will include some performance graphics, in order to compare the computation time of the previous Matlab version against this python version.

This first delivery, includes seven files: three python scripts, three CUDA files, and one makefile.

- testKNN\_call\_multiGPU.py and testRSAll\_call\_multiGPU.py are sample files. They can be used to check the correctness of the algorithm and measure computation times.
- python\_to\_c.py is the most important one. It takes care of casting python types into C types, so that the .cu files can be called from python.
- The makefile is used to compile .cu source files into .so dynamic libraries.
- The CUDA files are like those in the previous library, but containing small changes so that functions can be callable from python.

The requisites to check this program are pretty low. We only need to have installed on our computer a NVIDIA card with a CUDA Toolkit, python and the numpy package. This program was tested under CUDA v5.5, python v2.6/2.7. and numpy v1.8.

Before running the program, it is needed to compile the .cu files. To do that, we need to open the makefile, and check if the nvcc compiler is in the specified directory (by default `CUDA_PATH = /usr/local/cuda/bin/`). Then type `make` on the command line and a new .so file will be created.

At this point, we can already call one of the sample scripts. They contain all the data needed to execute the code inside the .cu files, printing on screen the results and the time employed to execute all the program. These scripts are executables, so they can be called from the command line.

In case we want to call these functions from a different python script, we need to include some extra lines to make it work. At the top of the script, we have to import numpy package, and also type this line `"from python_to_c import *"`, where we import the functions of this file. Then we can include one of the following lines to invoke the range search or the nearest neighbours searches:

```
bool = testRSAll_call_multiGPU(npointsrange, pointset,
queryset, vecradius, thelier, nchunkspergpu, pointsdim,
datalengthpergpu, gpuid)
```

```
bool = testKNN_call_multiGPU(indexes, distances, pointset,
queryset, kth, thelier, nchunkspergpu, pointsdim,
signallengthpergpu, gpuid)
```

These two lines return a value which indicates if any error happened during the execution of the CUDA Kernels. If `bool == 0`, something went wrong, and the results will be incorrect. To make a right call, `vecradius`, `pointset` and `queryset` must be numpy arrays, of type `float32`, and they must be filled before making the call. `npointsrange` and `indexes`, must be initialized as numpy arrays of type `int32`, and `distances` as type `float32`. These arrays can be created with `numpy.empty` or `numpy.zeros` and reused for different calls. They will contain the solution after calling the previous lines. The other values are standard `int`, and work as in the matlab files.

We thought that you would find interesting some additional information on the performance, so we compared the CUDA + Matlab code against the CUDA + Python code. Even though, we do not pursue to give very accurate measures, or use them to calculate the real timing difference between these two methods.

All the tests were ran under a CentOS 6 distribution, working on an Intel Xeon 5650, and an Nvidia GTX 780 Graphics card. We used a random pointset between 0-1, and searched 5 neighbors between 8 dimension points. We also execute every test three times, so that we can show how both codes work along several executions (we are not providing the average time here, but the time of during these three executions).

In the first test (Figure 1), we used a signal with a chunksize of 800.000 points, and 10 chunks. In the second execution (Fig. 2), the chunksize was reduced to a size of 40.000 points, and the number of chunks was increased to 100. Finally, for the last execution, we used an even smaller chunksize, of just 4096 points, but increased the number of chunks up to 5000.

It is worth to mention that python code is faster than matlab code, as you can see on the graphics above. Of course, you will probably would like to perform some more tests in order to collect meaningful measures for your application. Please give us some feedback and let us know if this also happens in your tests.

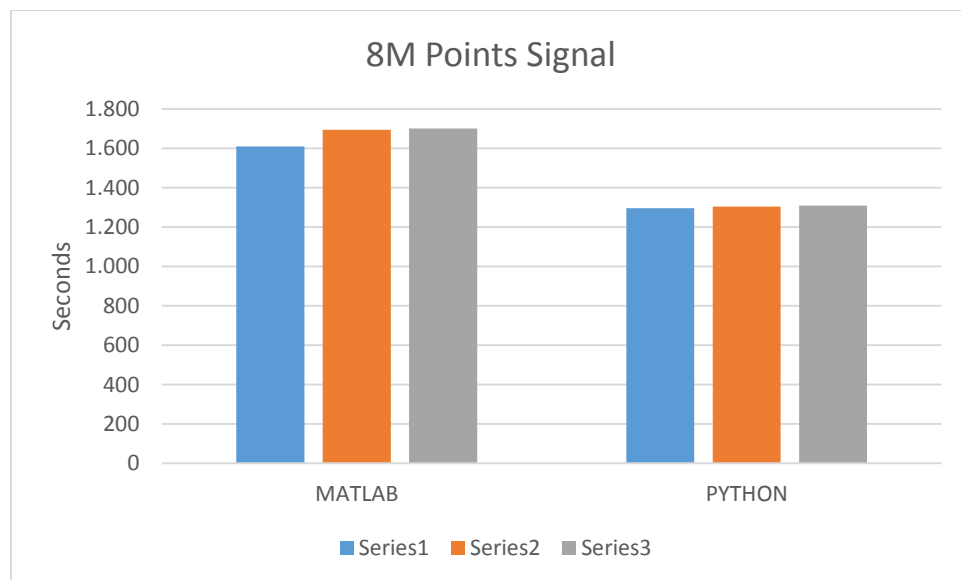


Fig.1.Performance of Matlab and Python versions (Chunksize: 800.000, Number of chunks: 10)

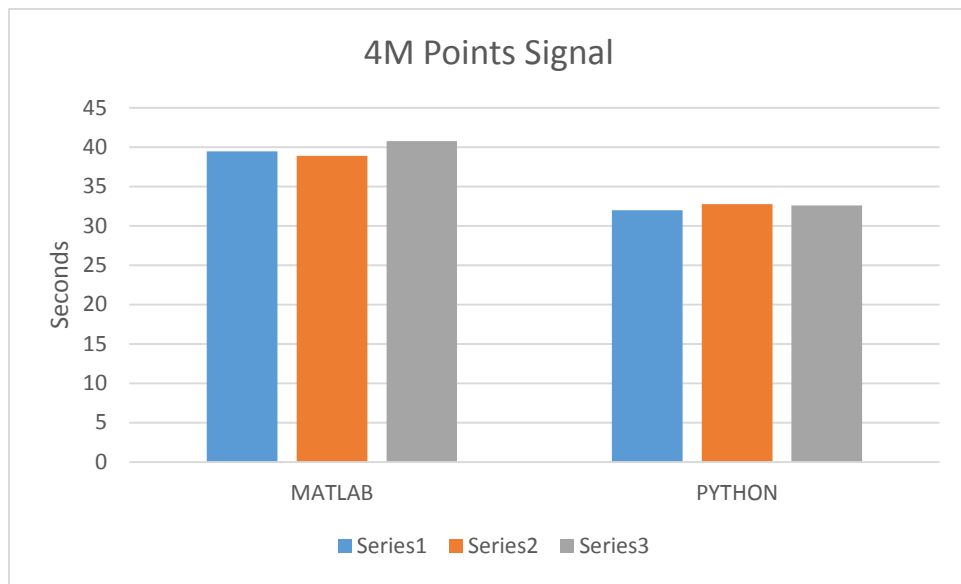


Fig.2.Performance of Matlab and Python versions (Chunksize: 40.000, Number of chunks: 100)

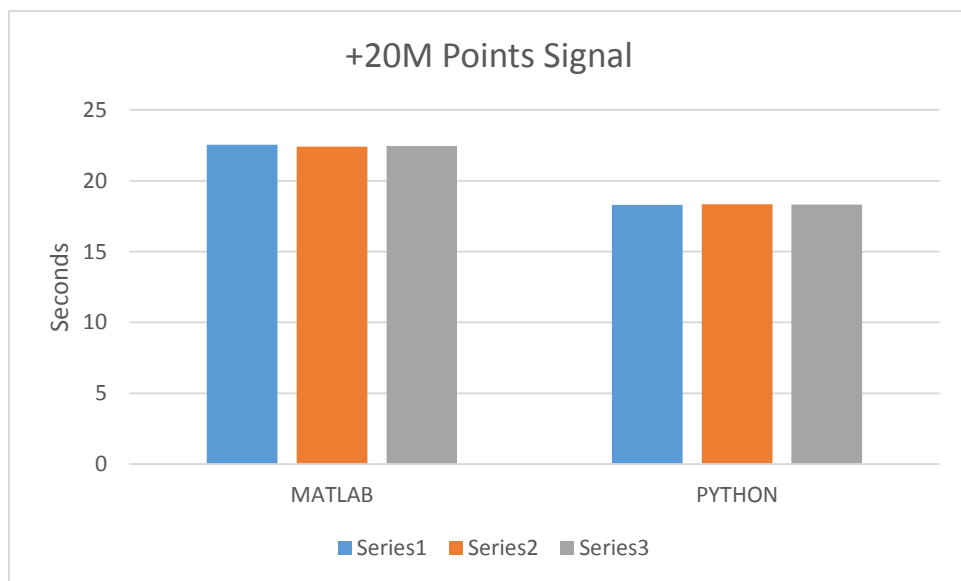


Fig.3.Performance of Matlab and Python versions (Chunksize: 4096, Number of chunks: 5000)