

Transfer Entropy optimization using CUDA/OpenCL and Python. Last deliverable.

June, 25th 2014

This small guide describes the final code release of the project, including Python and OpenCL components. Along this guide, we will analyze the files included in the .zip, and what are they used for. Also, we will check the requisites of a computer to execute our program. These are slightly different from the requisites needed in the previous releases. We will give a short explanation about how to test the scripts, which includes a small comparison test, and how to call them from different files. Finally, we also included some performance graphics, comparing the performance between the previous MATLAB+CUDA version, the Python+CUDA version and this new version using Python+OpenCL.

This second delivery includes six files: five python scripts and an OpenCL file.

- testKNN_call.py, testKNN_callCompare.py, testRSAll_call.py and testRSAll_callCompare.py are sample files. They can be used to measure computation times and to compare the results with the previous CUDA algorithm.
- clKnnLibrary.py contains the main function calls. In this script we use PyOpenCL to prepare the GPU for all the calculations running by the kernel.
- gpuKnnBF_kernel.cl are the heart of the program. It contains the translations from CUDA code to OpenCL, so our program could be executed at any graphic card which support this language.

As mentioned up above, the requisites to run the library have changed. We still need python and the numpy package, but instead the CUDA Toolkit, now we are going to need the OpenCL implementation and any kind of graphics card, such as NVIDIA, AMD or Intel. There is another package that we need too. We use PyOpenCL to communicate between python scripts and OpenCL, so we will need to install the distribution find in [1] in our computer.

Once we got all installed, we can check the functionality by executing the python files. testKNN_call.py and testRSAll_call.py execute the OpenCL version, print the results on the screen, and show the computation times. These two scripts can be executed right after installing all the previous requisites. In the case of testKNN_callCompare.py and testRSAll_callCompare.py, we also need to have the previous CUDA-based versions in the folder, so to be able to compare the results between OpenCL and CUDA. The procedure to install CUDA is the same than in the previous delivery. You have to compile the files, and include the .so and python_to_c.py in the same folder as the main scripts. Once you have done this, you can execute both scripts, check if both results match, and compare times between CUDA and OpenCL versions.

In case we want to call these functions from a different python script, we need to include some extra code lines to make it work. At the top of the script, we have to import numpy package, and also type this line `"from clKnnLibrary 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 neighbor searches:

```

correct = clFindRSAll(npointsrange, pointset, queryset,
vecradius, thelier, nchunkspergpu, pointsdim, signallengthpergpu,
gpuid)

correct = clFindKnn(indexes, distances, pointset, queryset,
kth, thelier, nchunkspergpu, pointsdim, signallengthpergpu, gpuid)

```

These two lines are pretty much the same than the equivalent ones on the first delivery, so all the tips which apply then, are useful in this case too. However, this is a small reminder of the paragraph in the previous delivery. If `correct == 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.

In addition, we include an execution time comparison between different versions. First, we compare MATLAB+CUDA, Python+CUDA and Python+OpenCL over the same card and the same conditions. Then we will a test of performance over an AMD card with the Python+OpenCL version.

The tests were run under a CentOS 6 distribution, working on an Intel Xeon 5650, and a Nvidia GTX 780 Graphics card. We used a random pointset between 0-1, and searched 5 or 10 neighbors between 8 or 10 dimension points. Each test was executed three times, so we can show how all codes work along several executions (we are not providing the average time here, but the total time of these three executions).

We also employed a 7870 AMD card, working with an Intel Q9550, running under a Fedora 20 distribution. The reason to choose this graphic card, it is because it has comparable power to the AMD 5870, which is the graphics card that can be found on the LOEWE-CSC supercomputer located in Frankfurt.

In the first test (Figure 1), we used a signal with a chunksize of 800.000 points and 10 chunks. In the second execution (Figure 2), the chunksize was reduced to a size of 40.000 points, and the number of chunks was increased to 100. The third one (Figure 3), has an even smaller chunksize, of just 4096 points, but an increased number of chunks up to 5000. All these tests search for 5 neighbors with 8 dimensions points. They are just the same tests that we ran in the previous deliverable, but now we add times with OpenCL both in Nvidia GTX 780 and the AMD 7870.

As we can see in the Figs. 1 and 2, Python + OpenCL performance is similar to Python + CUDA, but gives an interesting improvement over MATLAB + CUDA. It is worth to mention that NVIDIA has much more interest in CUDA than in OpenCL, so it was also expected that CUDA would run a little faster than OpenCL. This is what actually happens and can be checked in the figure.

We can not compare the results of the AMD 7870 directly against the results obtained with the NVIDIA GTX 780, because these are graphics cards with an important difference in power and price. In spite of this you can use this graph as a rough approximation of the time that would be needed to execute the algorithm over the 5870 cards in the cluster.

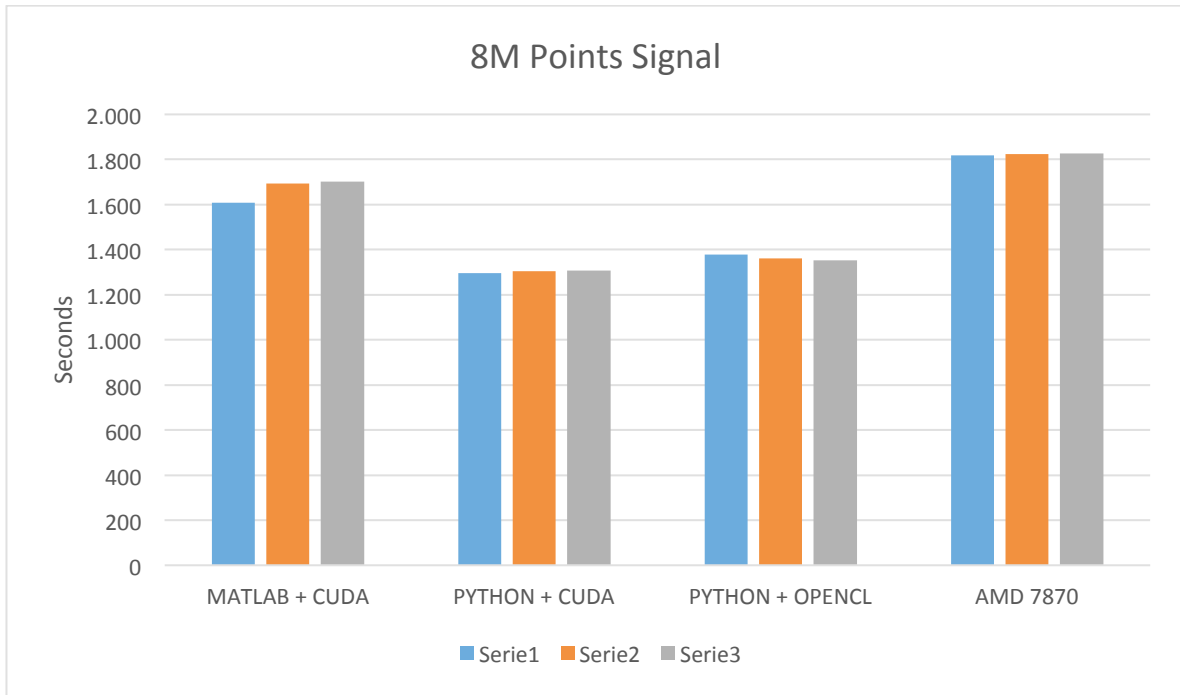


Fig.1. Performance different versions (Chunksize: 800.000, Number of chunks: 10, Neighbors: 5, Points Dimension: 8)

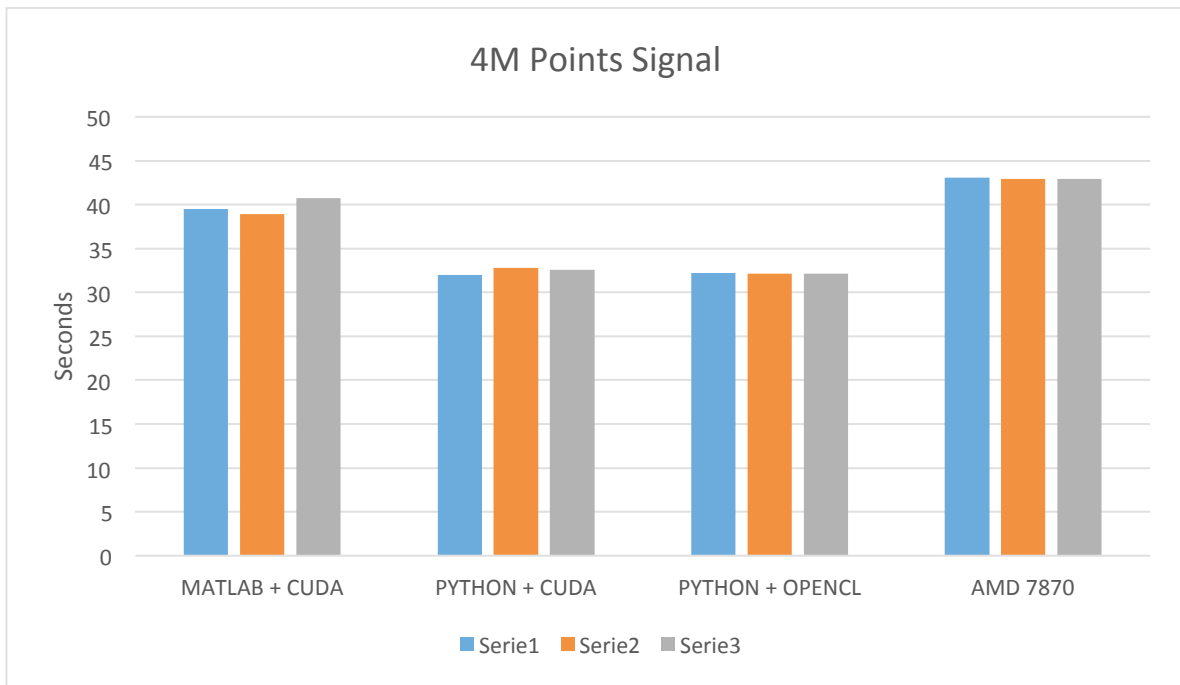


Fig.2. Performance different versions (Chunksize: 40.000, Number of chunks: 100, Neighbors: 5, Points Dimension: 8)

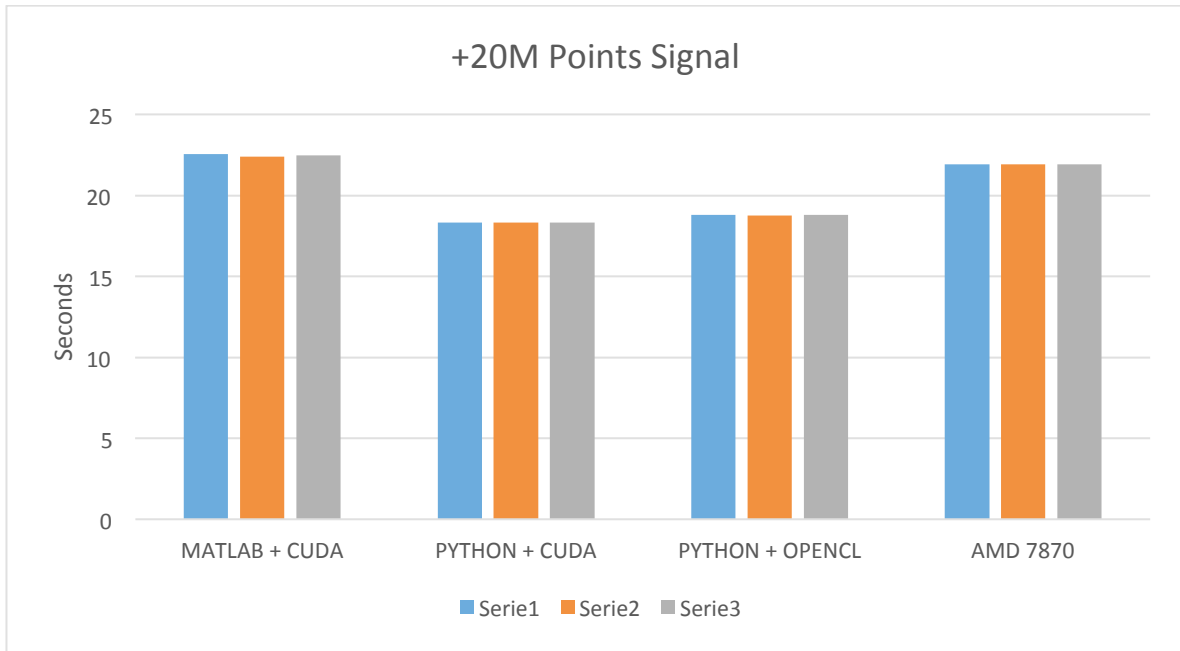


Fig.3.Performance different versions (Chunksize: 4.096, Number of chunks: 5000, Neighbors: 5, Points Dimension: 8)

Finally, we found it interesting to include a curious last test (Figure 4). In this case, we search for 10 neighbors in a pointset with 10 dimensional points, with a chunksize of 40.000 points, and 100 chunks. In this case, the 7870 beats the 780, which it is quite impressive because 780 cost three times more than the 7870.

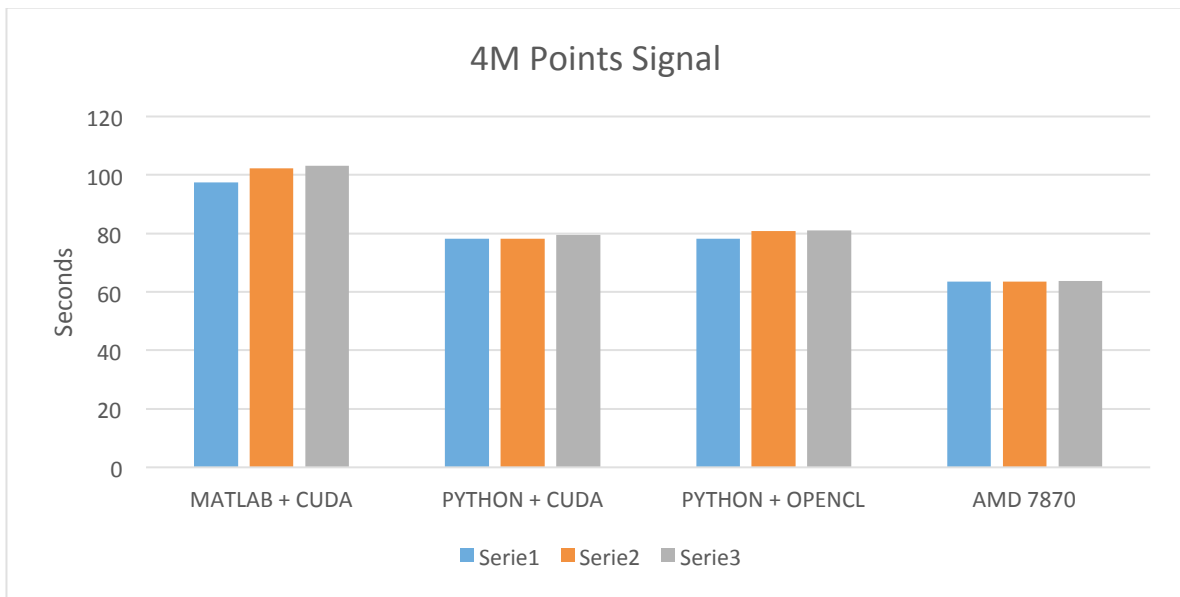


Fig.4.Performance different versions (Chunksize: 40.000, Number of chunks: 100, Neighbors: 10, Points Dimension: 10)

We encourage you to run your own tests, and let us know if everything goes right. Also, we will appreciate if you could give us some feedback about performance measurements with real signals, so we could know if we are on the right track with OpenCL.

There is one last tip about this deliverable. During the last days, we have made some testing with several GPUs working at the same time. We found out that if you execute the python scripts twice from command line, selecting a different GPU on each of them, you can have a tricky multi-GPU version of the code. In order to test this, you can duplicate any of the sample scripts, change the gpuid value and execute both scripts simultaneously from the command line.

[1] <http://mathematica.tic.de/software/pyopengl>