Draft of paper: - published in Proceedings of the Workshop in **Information Technology and Bionics (Symposium in Memory of Tamás Roska), Budapest, 23–24 June 2015, ISBN 978-963-89880-3-4, pp. 23-27**

High Productivity Cellular Neural Network Implementation on GPUs using Python

Radu Dogaru, Ioana Dogaru
University "Politehnica" of Bucharest, Natural Computing Laboratory,
Applied Electronics and Information Engineering, Bucharest, Romania,
e-mail : radu_d@ieee.org, ioana.dogaru@upb.ro

***Abstract*— A convenient approach for implementing CNNs into CUDA-enabled GPU platforms is presented. It exploits the newest developments in both software (Continuum's Anaconda's Numba and Numbapro Python packages) and hardware (the use of parallel computation on GPU provided by the CUDA computing platform) to ensure high-performance, high-productivity and high-portability. The most important advantage here is the high-productivity while there is still room for improvements in terms of performance (speed-up). The implementation is low-cost one and requires only the purchase of a CUDA-enabled graphics card. Using little development effort and starting from a Python implementation of CNN for CPU only, we demonstrate that a GPU implementation results with a maximal speed of about 800 Mcells/second.**

Keywords*— *standard cellular neural network, computational modeling, Anaconda Accelerate Python compiler, CUDA programming, GPU computing

I. INTRODUCTION

Cellular Neural Networks (CNN) and the CNN-UM [1] found already a wide range of applications. In order to benefit by their intrinsic parallelism, suitable implementations have to be considered. While analogue implementations offer the full advantage of technology allowing building on-chip smart sensors with optical inputs, their development and implementation cost is still high. Fully parallel digital technologies (FPGA) were also considered [2][3]. They can offer very good performance (around 64000 Mcells/s reported in [5]) but the prices of a good FPGA may still be high (in the range of thousands of Euros).

On the other hand, developing high performance CNN applications in an academic environment with some low budget available is a major challenge. In such applications we are of course interested by high performance (speed) but it becomes more important to have a simple to understand implementation as a starting point in a educational and research process, in other words we are very much interested in *high productivity* i.e. little development costs. This is the major objective of the research presented herein. As target platform, the graphic cards containing GPUs with CUDA support from Nvidia [4] are a very convenient and low budget choice.

We are interested to develop a *to develop a HP³ (high performance, productivity and portability)* solution with freely available packages. While optimized CNN and other kind of cellular architectures implementations for GPU were already reported [5][6][7], their development usually requires deep programming skills and knowledge of various software technologies. Solutions often come after relatively long time development cycles. On the other hand, numerous scientists would like to have access to such computing facilities in a much transparent fashion, problem oriented, and not relying on much focus on the low level programming issues. In other words, **high-productivity** becomes an important issue, meaning a dramatic reduction of the development time from idea to implementation. Most major vendors of scientific computing software who already addressed the issue of high-productivity are now supporting transparent programming of GPU devices. Much effort is needed and research is still in its infancy for developing transparent and high-productivity programming of computationally more powerful platforms such as FPGA boards [8]. **High portability** of an implementation is also an important issue to be considered.

Our paper proposes such a high productivity approach to CNN implementations continuing some previous work [9][10] where reaction-diffusion CNNs (RD-CNN) were considered. While compared to a pure CUDA programming solution the performance is a little lower (about 800 Mcells/s in our case as compared to about 4000 Mcells/s in [5]) but the development time drastically reduces when resorting on writing the specific platform in Python 2.7 based on the Continuum's Anaconda Accelerate distribution. The novelty of this work stands in that it identified the most convenient available tools and the procedures to optimize and use them. It provides our experience with this particular CNN focus while very few other works report application of these novel programming tools and packages in other areas ([11] for time series modeling and [12] for implementing evolutionary algorithms).

Section II introduces the required software and hardware to build the modeling platform discussing performance and optimization issues. Section III introduces the particular CNN model, namely the standard CNN and its application in image processing. The specific implementations for both CPU and GPU are also presented in this section as well as some running

Draft of paper: - published in Proceedings of the Workshop in **Information Technology and Bionics (Symposium in Memory of Tamás Roska), Budapest, 23–24 June 2015, ISBN 978-963-89880-3-4, pp. 23-27**

examples, including a novel CNN template for character segmentation.

II. The Python-based Modelling and Simulation Platform

A. Software

Our main constraint was to use a freely available scientific computing language that is highly portable, highly efficient and supports transparently advanced parallel computing platforms such as GPUs but also capabilities to fully exploit the CPU (based on compilers). Among the various freely available scientific computing languages supporting transparent programming of GPUs we found that today Python, SciLab (via its sciGPGPU package[1]), R (via rpud[2]) and Julia offer such facilities. We decided to focus on Python since it is supported by an extremely large variety of packages including those emulating commands and libraries that are specific to usual scientific computing like SciLab, Octave or Matlab but also a wide diversity of packages opening interesting potential for further developments of our platform into particular application oriented solutions. In terms of learning cycle, only about 1 month took us to accommodate and write programs in Python language while having already some fluency in Matlab and C programming.

Besides being open source, Python comes with a variety of good IDEs allowing fast prototyping and profiling. A good tutorial on this topic is given in [13]. Herein we decided to use the Anaconda Accelerate distribution[3] with Spyder cross-platform IDE and Python 2.7. These are rather straightforward to install on both Windows XP and Windows 7 operating systems. The software support for CUDA-enabled GPU, includes in addition to specific drivers of the graphics board, the nVidia CUDA driver v6.0.1.

B. Hardware and optimization

The host computer used to report the results herein is an average PC from Dell with Pentium Dual Core CPU, E7500 @ 2.93GHz, 2 GB of RAM, running Windows.

The GPU used (GM107, optimized for low power) belongs to GeForce GTX 750 (using Maxwell architecture, compatibility 5.0, four 1020 MHz base clock multiprocessors, each having 128 cores with a total of 512 cores, 1GB GDDR5 DRAM with a bandwidth of 80 GB/s[3]). Any other GPU, with minimal compute capability 2 may be used. A minimal description of the CUDA computing model is reviewed here as it is needed by the programmer to understand the functionality:

In terms of hardware, GPU provides a certain number of *streaming multiprocessors* (SM) each one containing a certain number of *cores* (128 in our case) and a *shared memory* (on-chip, fast). The GPU *core* is the elementary physical computational device. All *cores* have access to the *global memory* (off-chip, slow) used to exchange data between threads and between GPU and the host CPU. While the implementation presented here uses global memory, further sophisticated optimization exploiting the speed of accessing shared memory can be done resulting in even better performance [7].

In terms of software, the basic functional unit is called a *kernel* (here the implementation of one-step CNN iteration) and the kernel is executed in parallel by a number of *threads*. In our case computing the output of each CNN cell is associated to a well defined (and indexed) *thread* .The total number of threads equals the number of cells. The kernel organizes its execution in a *grid* of *thread blocks*. Allocation of threads to cores is automatically done by GPU schedulers. In NUMBAPRO Python it is possible to define both the block and the grid size, in order to optimize performance. Particularly, not only the total number of threads per blocks but also their distribution (i.e. the choices of *n* and *m* in the Python variable *blockdim=(n,m)*) influence the core occupancy and therefore the speedup. As shown in [10] and confirmed by results in Table I a linear choice *(1,m)* gives better performance for the same number of threads per block. Optimal choices are depicted in gray shade within tables.

TABLE I. Optimization of the block size for various number of CNN cells (10000 iterations)

Cells / Block dim	128x128	256x256	512x512	1024x1024
(6,6)	8.87	2.47	2.69	2.71
(8,8)	8.77	2.22	2.40	2.28
(1,32)	8.77	2.19	1.30	1.27
(1,64)	8.77	2.19	1.26	1.21
(64,1)	8.77	3.79	3.87	3.85
(16,16)	8.87	2.96	3.00	2.97
(20,20)	8.77	2.88	3.18	3.24

It is observed (as reported in numerous other CUDA/GPU implementations) that better performance (1.21 ns/cell iterations or 826 Mcells/s) is obtained for large array sizes. In all the above cases the gridsize*=(M/m,N/n)* where *MxN* is the size of the CNN grid. The most convenient choice for **blockdim** is (1,64).

C. Comparison of various implementations

In order to have a good basis for comparison, we assume that in general, for simulating a cellular system on a NxM grid and a given number T of iterations the computational time is given as $t_{ef} = \lambda NMT$ where we identify the inverse of the cell speed λ (ns/cells and iterations = ns/cit): $\lambda \cong t_{ef} / NM$ as a valid measure of the computational time allocated to each cell. Alternatively the

[1] https://atoms.scilab.org/toolboxes/sciGPGPU
[2] http://www.r-tutor.com/gpu-computing
[3] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750/specifications

inverse of λ is denoted as speed (expressed in Mega cells/second – 1000 Mcell/sec is thus equivalent to 1ns/cit) as often cited in other works [5][6][7]. In a highly parallel platform (each cell assigned to a computational unit). In the following we will evaluate the efficiency of the algorithm for a given machine using the grid-size-independent λ value. This makes it easy to compare the implementation of the same model on various platforms. Three implementations of the model are considered herein (second and last are slight modifications of the first implementation):

i) The first package (NUMBA) is included in Anaconda[4] distribution from Continuum and the code was completely rewritten according to the methodology detailed in the next section. This code targets CPU only and it has the advantage of using a very convenient JIT compiler [14]; ii) Not using this decorator, the program executes in pure Python (first column in Table II) requiring huge amounts of computing time (much like when using loops in Matlab); iii) The third package NUMBAPRO (distributed freely for academics with Anaconda Accelerate[4]) supports convenient parallel computing while exploiting a CUDA compatible GPU when it is available on board. The results of running the same CNN system are summarized in Tables II and II (Blockdim=(1,64)):

TABLE II. SPEED COMPARISON OF VARIOUS PYTHON PACKAGES (T=1000 ITERATIONS, CNN SIZE = 128X128)

	Pure PYTHON (eliminate @autojit from NUMBA implementation	**NUMBA compiled with autojit (CPU)**	**NUMBA PRO compliled with cudajit (CUDA)**
Execution time (seconds)	167.59	1.187	**0.156**
Speed per cell (ns/cit)	10229	72.45	**9,52**
Speed in Mcells/s	0.0977	13.80	**105**
Speedup with respect to Pure Python implementation	1	**141.24**	**1074 (7.6 to Numba CPU)**

TABLE III. SPEED COMPARISON OF VARIOUS PYTHON PACKAGES (T=1000 ITERATIONS, CNN SIZE = 1024X1024)

	NUMBA compiled with autojit (CPU)	NUMBAPRO compliled with cudajit (CUDA)
Execution time (seconds)	50.26	1.42
Speed per cell (ns/cit)	47.93	1.35
Speed in Mcells/s	20.86	737 (35 times speedup)

Comparisons with previous C++ implementations indicate that NUMBA JIT provides very efficient use of the CPU (almost the same speed as resulting from C++), at very little programming effort (simply adding a decorator @autojit in front of the function definition).

When the GPU is used, a speedup of 35 times is achieved for large enough array size, when compared to the best possible CPU implementation (NUMBA). Although there may be some place to improve the GPU implementation as indicated in other works (shared memory, texture, occupation optimization etc.) [5][7][10] it is worth to mention the simplicity of translating the NUMBA code into the NUMBAPRO one targeting the GPU, detailed in the next section. Even for novice programmers the NUMBAPRO implementation is a very convenient, transparent, approach.

III. THE STANDARD CNN MODEL AND ITS IMPLEMENTATION

The same standard CNN Euler-based discrete time model as described in [5] was considered, where the forward part is computed only once while each of the iterations is running like a kernel on the GPU. Our actual implementation has only periodic boundary condition.

A. NUMBA implementation

The starting point for developing our implementation is a CPU-oriented Python program which can be easily adapted to support GPU computation. The general structure of our platform includes the following modules (as indicated in the next figure): i) **import modules** – here various classes from packages needed in the software are mentioned; ii) **cell definition** – this module is specific to the particular CNN that is implemented and has a list of parameters as arguments; iii) **constructing the initial states** for the CNN (usually, in our examples they are images that are either loaded or generated artificially – for instance some randomly assigned pixels in a central square etc.); iv) **preparation for the main loop** – this includes a storage buffer where snapshots from the dynamic simulation are stored for further visualization; v) **the main loop** – iterates on T, each iteration consisting in a call to the array of cells and for some particular moments saving the snapshots; vi) **data visualization** – produces the results in a format that is convenient for the researcher and reports on the execution time.

In the next, the most important part of the modules for NUMBA implementation targeting CPU are presented:

```
3 from numba import *
4 from numbapro import cuda  #--------CUDA -----
```

The above is an excerpt from the **import module**, useful in both CPU and GPU implementations. In additions other packages are imported here, such as NUMPY, MATPLOTLIB, offering useful functions for computing and visualizing data. Next, one of the most important modules is the **cell module** implementing the specific CNN model. It is quite easy to change it to accommodate various other models

[4] https://store.continuum.io/cshop/academicanaconda

```
34 @autojit
35 def cnn_core(X, Y, W, Xnew, Ynew, dt, Atemp): # Atemp is the template
36 # (X, Y) - actual CNN (state,output) W - INPUT array (B*u+z) computed once
37 # (Xnew, Ynew) - next iteration CNN (state,output), dt - time step-size (1 fo
38 # Atemp is the template
39     n = X.shape[0]
40     m = X.shape[1]
41     for j in range(n):
42         for i in range(m):
43             # specific implementation of the dynamics (can be used unchanged
44             Xnew[j, i] = (1-dt)*X[j,i]+dt*(W[j,i]+ \
45             Y[(j-1)%n,(i-1)%m]*Atemp[0,0]+Y[(j-1)%n,i]*Atemp[0,1]+Y[(j-1)%n,(
46             Y[i,(i-1)%m]*Atemp[1,0]+Y[i,i]*Atemp[1,1]+Y[i,(i+1)%m]*Atemp[1,2]
```

It is interesting to note the @autojit decorator (line 34), in its absence the function is interpreted and not compiled, resulting in a huge computational burden. While there are other possibilities to compile code in Python, the one offered by NUMBA, is the most natural, clear and transparent. The next module will have the same implementation for both CPU and GPU implementations and has the role to declare the simulation conditions.

```
61 # - main program
62 dt, ka  = 1, 2
63 Atemp=np.zeros((3,3), dtype=np.float64)
64 Atemp=np.array([[0, 1, 0],[1, 2.1, 1],[0, 1, 0]], dtype=np.float64)
65 # line "W=A-4" included later to compute the feed-forward CNN layer (A
66 # The above implements Btemp=[[0,0,0],[0,1,0],[0,0,0]] and z=-4
67 NN = 128  # image size (if not from file)
68 NM = NN   # square image
69 iter_max = 1000  # iterations
70 nsnp= 5          # number of snapshots
71 name='123sq.bmp' # file loaded as initial state and/or input image
```

Observe in line 62 the choice for time step zie (dt) and image magnification (ka) while the A template is defined in line 64. The B and z templates are defined later and the matrix W (feed-forward term) is computed once before the main loop. In the above one can set the number of iterations and the image selected for processing. A synchronous model is usually implemented. The next module represents **the main loop.** The CNN module is called each iteration and computes the new state Xnew and output Ynew. Time advance marks a new call of the *cnn_core* function where the initial states X,Y are copied from the previously generated new states. At certain iteration moments, states are saved into X_show and Y_show for further visualization of snapshots.

```
119 # Main loop
120 iter = 0
121 while iter < iter_max:
122  cnn_core(X, Y, W, Xnew, Ynew, dt, Atemp)
123  if sync==1:
124   np.copyto(X,Xnew)  #(+0 syncronous / none = asynchronous)
125   np.copyto(Y,Ynew)
126  else:
127   X = Xnew
128   Y = Ynew
129  if (iter +1) % test_mod == 0:
130   print "%5d, (elapsed: %f s)" % (iter, time.time()-timer)
```

Finally, another module that is common for both CPU and GPU implementations is the reporting and **data visualization module**, not shown here (basically it plots the images in X_show and Y_show) and reports various things.

B. NUMBAPRO implementation from the NUMBA template

In this case, most of the code used for the CPU implementation is preserved, and changes will be done only in the modules depicted in dashed lines. The **cell module** (the CUDA kernel) now becomes:

```
38 @cuda.jit(argtypes=[f8[:,:], f8[:,:], f8[:,:], f8[:,:], f8[:,:], f8, f8[:,:]]
39 def cnn_core(X, Y, W, Xnew, Ynew, dt, Atemp):
40     n = X.shape[0]
41     m = X.shape[1]
42     # thread distribution of the cell implementation
43     j = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x
44     i = cuda.threadIdx.y + cuda.blockIdx.y * cuda.blockDim.y
45     if (j >= 0 and j < n) and (i >= 0 and i < m) :  # GPU equivalent of for l
46             # specific implementation of the dynamics (can be used unchanged
47             Xnew[j, i] = (1-dt)*X[j,i]+dt*(W[j,i]+ \
48             Y[(j-1)%n,(i-1)%m]*Atemp[0,0]+Y[(j-1)%n,i]*Atemp[0,1]+Y[(j-1)%n,(
49             Y[j,(i-1)%m]*Atemp[1,0]+Y[j,i]*Atemp[1,1]+Y[j,(i+1)%m]*Atemp[1,2]
```

As compared to the CPU implementation the equations are preserved, only the function declaration and compiling decorator will indicate now the GPU as target device. Also the two indexes for a cell (i,j) are now allocated to identify an execution thread. The **main loop** is now preceded by some additional operations which are specific for GPU and its memory transfers to the CPU host. Also the **blockdim** and **griddim** needs to be specified here.

```
167 # Main loop
168 iter = 0
169 while iter < iter_max:
170  cnn_core[griddim, blockdim](dX, dY, dW, dXnew, dYnew
171  # prepare input data for the new iteration
172  dX = dXnew
173  dY = dYnew
174  # memory transfer from GPU to CPU
175  if (iter +1) % test_mod == 0:
176   dXnew.to_host()
177   dYnew.to_host()
178   print "%5d, (elapsed: %f s)" % (iter, time.time()-t
```

Finally the **main loop** depicted above includes the call to the *cnn_core* (running on GPU device) and the transfer of the newly computed state streams (dXnew, dYnew) associated to Xnew and Ynew matrices to the initial state for the next iteration (this operation is actually done inside the graphics card (using its global memory) then from time to time, transfers the results to CPU host using the *.to_host()* method in order to be visualized. Note that the translation from the NUMBA implementation (CPU only) to NUMBAPRO implementation is straightforward and requires a minimal number of changes only in the cell module and in the main loop module with the addition of the GPU initialization module.

C. Running Example (optimizing a template)

To get a glimpse of operating our CNN implementation, the results of the CNN dynamics are shown for a template that we propose here as a useful one in segmenting connected pixels (i.e. representing a handwritten character). A similar functional template was first proposed in the context of binary cellular automata in [16] but here it is adapted for the standard CNN model (in fact discrete-time CNN with dt=1). Such kind of templates (based on wave propagation) are particularly interesting to simulate into a high performance environment since they usually require a large number of iterations. In particular the number of time-steps needed to obtain a rectangle framing a given group of connected pixels is proportional to the size of the object represented by the pixels. A first simulation using the GPU implementation is given in Figure 1. while a simulation on CPU-only is given in Figure 2. The upper row displays the dynamics of the X (state) and the lower

row the dynamics of the output (Y). The image is (512x512) size and the central element of A is 2.01 (lower limit for this behavior). Note that about 600 iterations were needed to achieve squaring. For this particular case the speed was about 700 Mcells/second – meaning that all 1000 iterations were completed in 0.4 seconds.

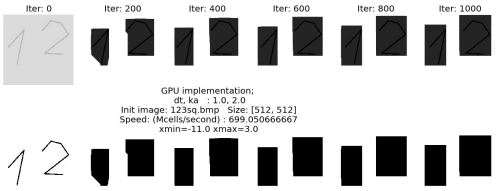

Fig. 1. A CNN simulation for dt=1, A=[0, 1, 0; 1, 2.01, 1; 0, 1, 0], B=[0, 0, 0; 0, 1, 0; 0, 0, 0] z=-4. The GPU simulation runs for 0.4 seconds.

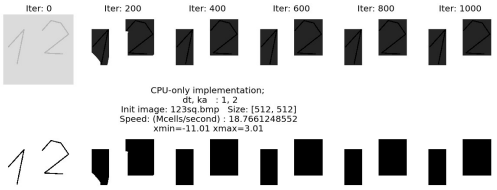

Fig. 2. Same conditions for CNN simulation as in Fig.1 but now the simulation is run on CPU only. Simulation speed is now 37 slower.

IV. Conclusions

We found Python and in particular its NUMBA and NUMBAPRO packages included in the Anaconda Accelerate distribution as the most convenient choice for implementing a CNN platform for CUDA-enabled GPU on commercially available graphic cards. Reasonable speeds (up to 800 Mcells/second) were achieved with very little effort for developing with an easy to program style, closer to the use of traditional scientific languages and thus allowing the researcher to focus on the specific modeling and simulation problem rather than on programming details.

The adopted technology allows a convenient and fast translation from CPU-only code (developed based on NUMBA) to GPU supported by CUDA graphic card (the one considered herein costs about 100 Euros). This makes it easy to offer a portable CNN implementation tailored to the available hardware. The platform is portable (not depending on the operating system and of the particular GPU) and consequently it is quite easy to provide a faster implementation when specific hardware is available. IDEs such as Spyder offer convenient facilities for editing, profiling and debugging the code at virtually no cost. Concluding, with a minimal investment we demonstrate that a HP^3 (high-performance, high-productivity and high-portability) modeling and simulation platform can be rapidly developed.

Compared to other solutions presented in the literature the speed is a bit lower (6 times lower compared to [5]). But the C-CUDA implementation requires more programming effort and a much longer learning curve, especially for scientists with no advanced programming skills. Running thousands of CNN iterations for usual-sized images in fractions of seconds is convenient for further embedding in more complex applications (using the CNN-UM concept) but also for investigation and discovery of new templates.

Further research will consider optimization of the CUDA implementations but also expansions to consider more complex applications using the CNN-UM concept (scripts defining CNN processing flow). Also expanding the platform to other more general cellular nonlinear networks considered in [15] will be considered.

References

[1] T. Roska and L. O. Chua, "The CNN universal machine: An analogic array computer," *IEEE Trans. Circuits Syst. II*, vol. 40, pp. 163–173, Mar. 1993.

[2] Z. Nagy and P. Szolgay, "Configurable Multi-Layer CNN-UM Emulator on FPGA", Proceedings 7 IEEE Int. Workshop on Cellular Neural Networks and their Applications (2002) 164 – 171.

[3] Vörösházi, A. Kiss, Z. Nagy and P. Szolgay, "Implementation of embedded emulated-digital CNN-UM global analogic programming unit on FPGA and its application," in Int. J. Circ. Theor. Appl., Wiley, 2008, vol. 36, pp. 589–603.

[4] J. Nickolls, I, Buck, M. Garland, 2008. Scalable ParallelProgramming with CUDA. ACM Queue, 6, 2, 40-53.

[5] E. Lászlo, P. Szolgay and Z. Nagy, "Analysis of a GPU based CNN implementation", Cellular Nanoscale Networks and Their Applications (CNNA), 2012 13th International Workshop on, Turin, 29-31 Aug. 2012

[6] D. De Donno, A, Esposito, L. Tarricone, L. Catarinucci, "Introduction to GPU Computing and CUDA Programming: A Case Study on FDTD [EM Programmer's Notebook]," Antennas and Propagation Magazine, IEEE , vol.52, no.3, pp.116,122, June 2010.

[7] K.V. Kalgin, "Implementation of algorithms with a fine-grained parallelism on GPUs", Numerical Analysis and Applications, Vol.4, No.1, pp 46-55, 2011.

[8] Fleming, K.; Hsin-Jung Yang; Adler, M.; Emer, J., "The LEAP FPGA operating system," Field Programmable Logic and Applications (FPL), 2014 24th International Conference on , vol., no., pp.1,8, 2-4 Sept. 2014.

[9] R. Dogaru and Ioana Dogaru, "A Low Cost High Performance Computing Platform for Cellular Nonlinear Networks using Python for CUDA, in Proceedings of 20'th International Conference on Control Systems and Computer Science (CSCS20), Bucharest, Romania, May 27-29, 2015, in press.

[10] G.V. Stoica, R. Dogaru, C.E. Stoica, " Speeding-up Image Processing in Reaction-Diffusion Cellular Neural Networks using CUDA-enabled GPU Platforms",International Conference on Electronics, Computers and Artificial Intelligence, Bucharest, Oct. 2014, Vol. 2, pp. 39-42 (also on IEEE Xplore)

[11] Signoretto, Marco; Frandi, Emanuele; Karevan, Zahra; Suykens, Johan A.K., "High level high performance computing for multitask learning of time-varying models," Computational Intelligence in Big Data (CIBD), 2014 IEEE Symposium on , vol., no., pp.1,6, 9-12 Dec. 2014

[12] Zubanovic, D.; Hidic, A.; Hajdarevic, A.; Nosovic, N.; Konjicija, S., "Performance analysis of parallel master-slave Evolutionary strategies (μ,λ) model python implementation for CPU and GPGPU," Information

and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on , vol., no., pp.1609,1613, 26-30 May 2014.

[13] H. P. Langtangen, "How to access Python for doing scientific computing", 2014, online http://hplgit.github.io/edu/accesspy/accesspy.pdf last time accessed January 2015.

[14] Prasad A Kulkarni, "Jit compilation policy for modern machines", In ACM SIGPLAN Notices, volume 46, pages 773–788. ACM, 2011.

[15] R. Dogaru, Systematic design for emergence in cellular nonlinear networks – with applications in natural computing and signal processing, Springer-Verlag, Berlin Heidelberg, 2008.

[16] R. Dogaru, I. Dogaru, M. Glesner, A smart sensor architecture based on emergent computation in an array of outer-totalistic cells , in Proceedings of SPIE Volume: 5839 Bioengineered and Bioinspired Systems II, Editor(s): Ricardo A. Carmona, Gustavo Liñán-Cembrano, pp. 254-263, 2005.