Optimization of GPU and CPU Acceleration for Neural Networks Layers Implemented in Python

Radu Dogaru, Ioana Dogaru
Natural Computing Laboratory,
Dept. of Applied Electronics and Information Eng.
University "Politehnica" of Bucharest, Romania
radu_d@ieee.org, ioana.dogaru@upb.ro

Abstract—**Many neural architectures including RBF, SVM, FSVC classifiers, or deep-learning solutions require the efficient implementation of neurons layers, each of them having a given number of m neurons, a specific set of parameters and operating on a training or test set of *N* feature vectors having each a dimension *n*. Herein we investigate how to allocate the computation on GPU kernels and how to better optimize the problem parameters (neural structure and training set size) as well as the GPU parameters in order to maximize the acceleration (relative to a CPU implementation). It is shown that by maximizing the load (number of threads on each computational GPU core) and by a proper allocation of the GPU global memory, very large speedups (100-250 times) with respect to the CPU implementation can be achieved while using the convenient NUMBA Python package supporting CUDA programming of GPU. Consequently, it is shown that given a problem to be posed to a neural network a convenient decomposition of the network can be done in order to allocate optimally the parts of the computation to the GPU in order to maximize efficiency. Also, for CPU implementations it was found that Intel's MKL library (called from NUMPY package) can offer efficient implementation of neural layers, comparable to what is achieved using GPU.**

Keywords — neural networks; high performance computing; graphical processing unit (GPU), radial basis functions.

I. INTRODUCTION

Numerous practical problems arise where big data (images, complex signals, time series, etc) have to be automatically classified using neural networks or other similar constructs. Since computations in such networks involve billion of basic arithmetic operations it is worth investigating how to make this computation efficient, particularly when relatively cheap GPUs are readily available on most actual computational platforms. Numerous authors investigated this aspect in the last years, proposing various solutions [1][2]. Usually speedups (over CPU implementations) of up to 1 order of magnitude are reported for commonly available GPU [1][2]. There are many possibilities to map computations associated to neural layers into GPU, for instance by allowing threads (distributed to the multiple GPU cores) to execute synaptic multiplications as discussed in [3] where a need for "map-reduce" unit to collect results from all parallel threads was identified as a limiting factor leading to the proposal of a specialized hardware and its FPGA implementation [4]. While this solution proves fast and efficient in terms of power and acceleration, our focus herein is on widely available GPUs. Herein we describe a methodology and results for optimizing the efficiency while aiming to get the most from a given GPU unit. Particularly, we discuss a "high productivity" solution, namely the implementation in Python 2.7, using the NUMBA package from the Anaconda 4.0 distribution[1].

Section II provides the method proposed herein to map our problem (computation of the outputs for a neural layer with a given structure and parameters) into GPU. Our choice was to process batches of *N* input samples in order to avoid map-reduce operation after kernel computations. Such map-reduce operations (synaptic combiners) are now included in the kernels running as parallel threads in the GPU. Section III investigates how to choose the parameters of both GPU and neural structure in order to maximize efficiency. It is shown that a basic neural layer (BNL) can be designed such that speedups (in comparison to CPU) and GPU efficiency are maximized. Consequently, if the desired neural network layer is larger than the optimized BNL it can be split in a convenient number of such optimized units in order to get efficient processing. It is shown that given a GPU unit, there is an optimal set of parameters (the size M of the BPL, where M^2 corresponds to the number of threads running the basic kernel, as well as the block size parameter B) to get the best performance and speedup. Section IV investigates the influence of n (number of inputs or feature vector size) showing that in order to achieve a good efficiency, a large value of n , e.g. $n>100$ is desirable for GPU implementations. This situation corresponds well to the situation of image processing, as it often appears in deep learning neural structures. Section V investigates other possibilities to get efficient implementation, using Intel MKL libraries available in recent NUMPY distributions. The paper is closed with the concluding remarks section.

II. MAPPING NEURAL LAYERS INTO THE GPU – A PROGRAMMER MODEL

A. Neural network layers and parameters

Figure 1 represents both the neural model and its mapping into GPU. The CPU mapping is simpler since it computes the output of each neuron (corresponding to one thread on GPU)

[1] https://www.continuum.io/blog/developer-blog/anaconda-4-release

978-1-5386-2059-5/17/$31.00

sequentially. In the following we will call this representation a ***basic neural layer*** **(BNL)** and the aim is to optimize the speed of computing its output **H** when a given matrix of samples **X** (Samples) is given. The following parameters (also depicted in Figure 1) are considered next:

i) ***n* is the size of the feature vector** (for instance in a classic handwritten digit image recognition problem (MNIST) n=28x28=784 pixels;

ii) ***N* is the number of samples** to be considered for a parallel computation. As indicated next, a large number of it (up to the GPU board limits) will make the BNL efficient to compute (high acceleration with respect to CPU implementation);

iii) ***m* is the number of neurons on the layer**, this is typically optimized in the learning process such that it will produce the best accuracy (tradeoff between under-fitting and over-fitting). For reasons of optimizing the GPU computation the variable M is defined (let call it GPU layer size) such that $M^2 = mN$ with an **asymmetry index** As also to be considered. Using the above notations: $m = M / As$ and $N = M \cdot As$. Other, GPU specific, parameter is the **block size** B. As indicated next its careful optimization would provide the best efficiency of the implementation.

In order to implement any arbitrary size neural layer one would have to decompose it into optimal BNLs to be fed into GPU. Smaller size layers can be computed as well but as indicated in the next, the efficiency (acceleration) cannot reach its maximal value. For instance, assume that one has to implement a network with 8000 nodes running for 10000 samples. For the given GPU the optimal BNL was found to be *M=n=N=4096*. Consequently the network will be split in 2x3=6 BNL in order to cover both the number of hidden nodes and given samples.

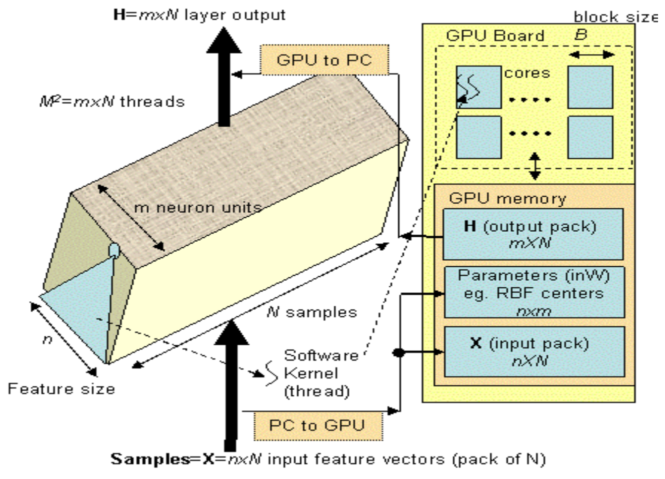


Fig. 1. Basic neural layer and its GPU mapping

The two following formulae (denoted as RBF and MUL) for a given neuron are implemented and considered next:

$$y^{RBF}{}_{i,j} = rbf\left(\sum_{k=1}^{n} \left| X_{k,i} - inW_{k,j} \right| \right) \qquad (1)$$

where $rbf(d) = \max(0, 1 - d/(2.5066 \cdot raza))$ (2)

is the triangular RBF function proposed in conjunction with the FSVC and SFSVC fast learning classifiers [5] while the next is the traditional, sigmoidal neuron, used in many deep-learning multi-layer perceptron structures:

$$y_{i,j} = \tanh\left(\sum_{k=1}^{n} X_{k,i} \cdot inW_{k,j} \right) \qquad (3)$$

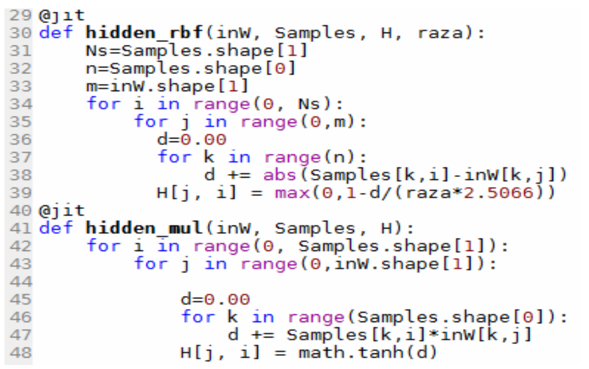
In the above **X** and **inW** are matrices of input samples and respectively weights (or RBF centroid parameters) as shown in Fig.1. In practical implementations one should also consider *Mem* which represents the GPU allocated memory in order to implement a BNL. As shown in Fig.1 and given a 8 byte floating point representation (float64) it follows that:

$$Mem = 8(n \cdot m + N \cdot n + N \cdot m) \qquad (4)$$

It is a limiting factor, given that on the same memory unit some other variables used by the graphics of the operating system are present. For the boards used herein using $Mem < 0.6$ Gbytes was possible without memory allocation errors at runtime.

B. *CPU implementation in Python using NUMBA package and JIT*

As shown in previous work [6] the NUMBA package (now merged with the deprecated NUMBAPRO supporting GPU) offers a high productivity approach to accelerate computation on CPU using the JIT (just in time compiler) implemented with the @jit decorator (Figure 2). Therefore we use this approach and GPU speedups are computed with respect to the above Python implementation of the BNL on CPU. Comparisons for the same neural layer parameters with our previous C++ implementation of FSVC [5] running as .MEX under Octave indicates that such JIT based Python/NUMBA implementations are around 3 times slower. Consequently, for a better comparison with other CPU and GPU implementations the effective running times are also given next.

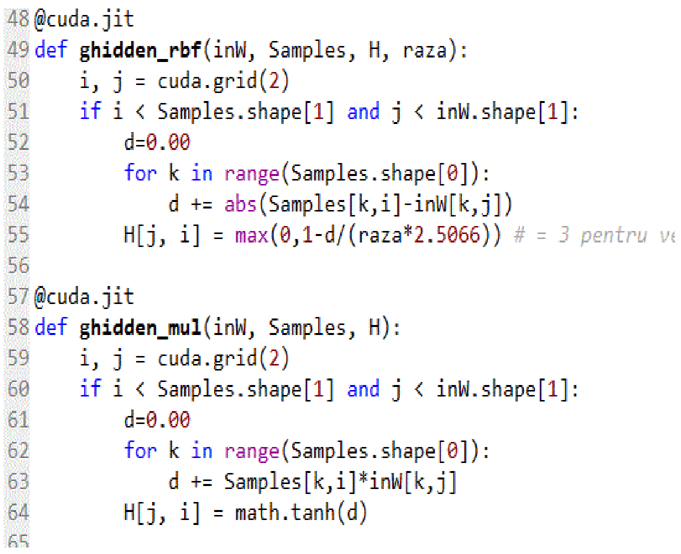


```
@jit
def hidden_rbf(inW, Samples, H, raza):
    Ns=Samples.shape[1]
    n=Samples.shape[0]
    m=inW.shape[1]
    for i in range(0, Ns):
        for j in range(0,m):
          d=0.00
          for k in range(n):
              d += abs(Samples[k,i]-inW[k,j])
          H[j, i] = max(0,1-d/(raza*2.5066))
@jit
def hidden_mul(inW, Samples, H):
    for i in range(0, Samples.shape[1]):
        for j in range(0,inW.shape[1]):

            d=0.00
            for k in range(Samples.shape[0]):
                d += Samples[k,i]*inW[k,j]
            H[j, i] = math.tanh(d)
```

Fig. 2. RBF and MUL – type neuronal layer implemented on CPU in Python using JIT from NUMBA

C. *GPU implementation in Python using NUMBA package*

As seen in Fig.3 the NUMBA package offers a very straight forward and elegant way to pass from CPU implementation to a GPU implementation of the software kernel (thread to be executed on the GPU cores, thus implementing the mapping indicated in Figure 1). The specific JIT compiler is activated by the decorator @cuda.jit



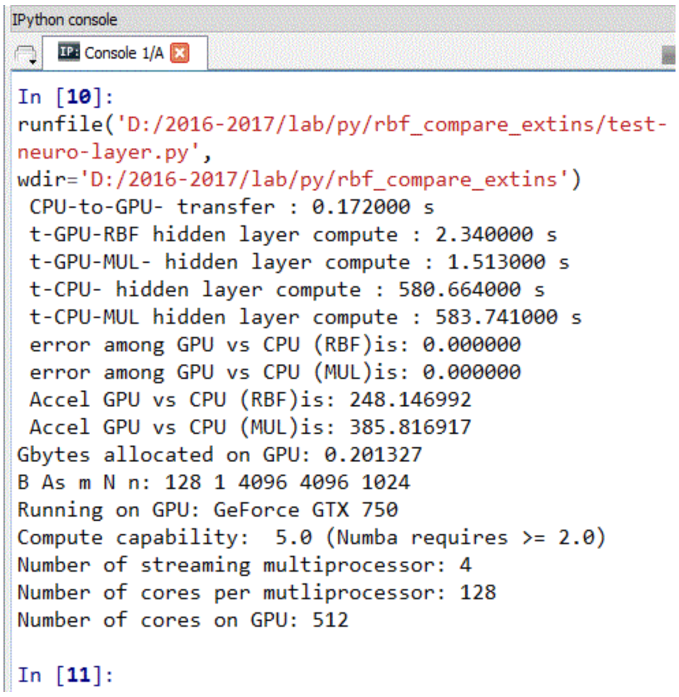
```python
@cuda.jit
def ghidden_rbf(inW, Samples, H, raza):
    i, j = cuda.grid(2)
    if i < Samples.shape[1] and j < inW.shape[1]:
        d=0.00
        for k in range(Samples.shape[0]):
            d += abs(Samples[k,i]-inW[k,j])
        H[j, i] = max(0,1-d/(raza*2.5066)) # = 3 pentru ve

@cuda.jit
def ghidden_mul(inW, Samples, H):
    i, j = cuda.grid(2)
    if i < Samples.shape[1] and j < inW.shape[1]:
        d=0.00
        for k in range(Samples.shape[0]):
            d += Samples[k,i]*inW[k,j]
        H[j, i] = math.tanh(d)
```

Fig. 3. GPU implementation of computational kernel for both RBF and MUL type of neural layers using NUMBA.

D. *Optimization of parameters for maximal acceleration*

The following running example (Figure 4) indicates how our optimization program in Python works; It evaluates first the PC-to-GPU transfer time for all 3 matrices indicated in Fig.1. Usually this time is less than 10% of the GPU computation time and it will be not included in the speedup evaluation. Both GPU and CPU implementations are called for the same neural layer structure, parameters and dataset (input samples) while computing times are reported (for GPU they include the GPU-to-PC transfer of the output data – matrix **H**).

All parameters (as described in Section II.A) are also listed in the above example corresponding to the best performance obtained on the GeForce GTX 750 board. Errors between GPU and CPU implementations are also computed indicating (zero value) that both implementations are functionally equivalent. The above BNL has 4096 hidden units, input vectors are 32x32 (i.e. *n=1024*) pixel images, and it operates on a block of 4096 samples. The whole computation takes 2.3 seconds (RBF implement being slower than the MUL one) indicating a speedup of more than 248. Indeed it is quite close to the number of cores (512 in this case) but if one consider that Python JIT on CPU is 3 times slower than a C++ implementation, it follows that in fact an acceleration of 80 times is achieved with respect to a standard C++ compiler.



```
In [10]:
runfile('D:/2016-2017/lab/py/rbf_compare_extins/test-
neuro-layer.py',
wdir='D:/2016-2017/lab/py/rbf_compare_extins')
 CPU-to-GPU- transfer : 0.172000 s
 t-GPU-RBF hidden layer compute : 2.340000 s
 t-GPU-MUL- hidden layer compute : 1.513000 s
 t-CPU- hidden layer compute : 580.664000 s
 t-CPU-MUL hidden layer compute : 583.741000 s
 error among GPU vs CPU (RBF)is: 0.000000
 error among GPU vs CPU (MUL)is: 0.000000
 Accel GPU vs CPU (RBF)is: 248.146992
 Accel GPU vs CPU (MUL)is: 385.816917
Gbytes allocated on GPU: 0.201327
B As m N n: 128 1 4096 4096 1024
Running on GPU: GeForce GTX 750
Compute capability:  5.0 (Numba requires >= 2.0)
Number of streaming multiprocessor: 4
Number of cores per mutliprocessor: 128
Number of cores on GPU: 512

In [11]:
```

Fig. 4. Running of the program to evaluate various GPU/CPU implementations of the neural layers.

E. *Computing platforms*

Two configurations were used, denoted next as PC1 and PC2 (both using 64bit OS on Intel processors). The programming environment on both platforms is Anaconda 4.0 running Python 2.7.

i) **PC1** is a desktop with Pentium Dual Core CPU, E7500 @ 2.93GHz, 2 GB of RAM, running Windows 7. The GPU board is 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[2]);

ii) **PC2** is a laptop computer with quad-core Pentium N3540 CPU @2.166 Ghz, 4GB of RAM and with GPU of type GeForce 820M Compute capability: 2.1; 96 cores per GF117 GPU (operated at 625 Mhz). GPU board memory is 2GByte (14.4 Gbyte/s).

In all cases discussed next, the following instructions are used to determine the GPU block-dim and grid-dim: blockdim=(B,B) blocksize=(8,8) griddim=(N/blockdim[0], m/blockdim[1]). As seen next, with a proper tuning of M and B one can increase the speedup for a given GPU.

III. MAXIMIZING THE EFFICIENCY FOR A GIVEN SIZE OF THE FEATURE VECTOR

In the following we are interested to maximize efficiency (speedup compared to CPU). Consequently several runs were done with various *M* and *B* values. The results are summarized in the following tables were filling with " * " indicates that an

[2] http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-750/specifications

error occurs (GPU threads or GPU memory). In all cases within this section $n = 28x28 = 784$, corresponding to the MNIST problem. If not other specified, asymmetry index *As*=1.

A. Laptop GPU (Geforce 820M)

Table I reports the speedups (GPU/CPU) for PC2 in the case of implementing RBF neurons (where synapses are computed as absolute value of differences). In the most efficient case (*M=2048, B=128*) the effective computing time for the corresponding basic neural layer (BNL) was $t^{RBF} = 2.4$ seconds. Similar optimization results but with slightly better speedups were obtained for the MUL (multiplicative synapse) neurons. Generally it was observed that GPU implementation of distances (RBF neurons) is about 1.5 times slower than implementing sum of products (multiplicative synapses) while in the CPU there is no significant difference among RBF and MUL types of neurons.

TABLE I. OPTIMIZING B AND M FOR 820M GPU (RBF NEURONS) – WHERE CELL CONTENTS REPRESENTS SPEEDUPS WHILE * DENOTES DIFFERENT ERRORS RELATED TO UNPROPER LOADING OF GPU

M \ B	256	128	64	32	16
2048	*M	71	*A	*A	*A
1024	38	31	40	*A	*A
512	3.6	9.3	14.5	18	*A
256	1.09	2.16	4.2	5.5	3.5
128	*Y	0.6	0.8	1.2	1.42
64	*Y	*Y	0.7	0.8	1.28

From the above it is clear that in order to get the maximal efficiency one needs to maximize the dimension of the layer *M* up to the limit supported by the given GPU board (in the case of *n*=784, *M* cannot be larger than 2048. A speedup of 71 is obtained in this case. Note that best accelerations for any choice of *M* are obtained in this case for griddim(16,16).

The influence of *As* was investigated by taking values from 1 to 16 . No significant changes were observed in speedups, thus indicating that one may use this coefficient to best configure the BNL to match the specific needs of the neural network to be implemented (e.g. similar speedups are obtained when using $m \times N = 2048x2048$ or $m \times N = 512x8192$).

B. PC GPU (GTX 750)

Instead of speedups, Table II reports BNL execution times for PC1 in the case of implementing both RBF and MUL neurons. Execution times are given in each cell as (t^{RBF}, t^{MUL}) .

As in the previous case, the best efficiency is obtained using the maximal *M* allowed by the GPU board (in this case *M=4096*) indicating a number of threads 4 times larger than in the case of 820M. Corresponding speedups are near 300 as seen in Fig. 4. For PC1 in the most efficient case the average time per kernel (there are M^2 threads to compute) is 0.12 μs while in the case of 820M GPU it was $2.4/(2048 \times 2048) = 0.572$ μs i.e. 4.7 times slower. In this case (PC1) the griddim(32,32) is favored. Again, MUL layers are a bit more efficient to compute on GPU (1.5 times better speeds) than RBF layers.

TABLE II. OPTIMIZING B AND M FOR GTX 750 GPU (EXECUTION TIMES ARE GIVEN IN SECONDS)

M \ B	512	256	128	64	32	16
4096	2, 1.23	2, 1.34	2, 1.31	*A	*A	*A
2048	1.1, 0.76	0.71, 0.46	0.71, 0.5	0.7, 0.46	*A	*A
1024	1, 0.6	0.43, 0.3	0.32, 0.26	0.31, 0.26	0.34, 0.26	*A
512	0.95, 0.6	0.43, 0.31	0.27, 0.22	0.22, 0.18	0.22, 0.18	0.22, 0.18

IV. OPTIMIZING SPEEDUPS FOR A GIVEN SIZE OF THE FEATURE VECTOR

In order to efficiently use a given GPU one has to choose a dimension *M* as large as it can be supported by the GPU board. Choosing smaller dimensions results in lower speedups and thus less efficient realizations. In the following we are interested to see how to optimize GPU computations in order to achieve efficiency for arbitrary size *n* of the feature vector. Figure 5 shows the evolution of the t_{syn} (synaptic computing time) in the RBF case for *(M,B)* optimized as indicated above. The synaptic computing time is determined using the following formula (where it is assumed that times allocated to compute the output nonlinearity can be neglected since they represent a fraction of *1/n* from the total number of synapses): $t_{syn} = t_{GPU}^{RBF-BNL} / (nM^2)$ and is expressed in nanoseconds. It represents the effective computation time allocated to the basic unit (synapse) in a neural layer which is either a multiplication or an absolute value difference.

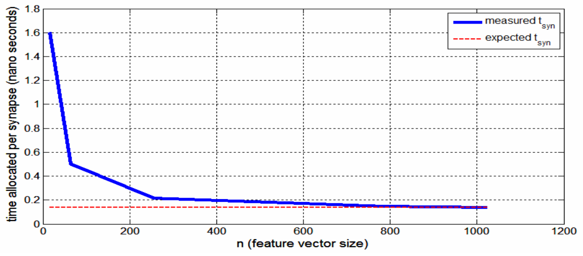


Fig. 5. Dependence on *n* for the measured synaptic time (GTX-750 GPU, *M*=4096, *B*=128). Note that a good value (around 0.2 nanoseconds) is obtained for *n>200*. Small number of inputs do not give very good efficiencies of the GPU (speedups in comparison to CPU fall to 1 and even less).

Although a constant value for the t_{syn} was expected (global execution time of a kernel would be proportional with n), as seen from Fig. 5 this is not the case, and the most efficient computation takes place for largest n values (admissible for a choice of M). For small n values (<10) larger t_{syn} may be the effect of neglecting the output nonlinearity, but this does not stand for bigger n values. One explanation may be a more efficient usage of accesses to the global memory on the GPU board when n is large. A similar measurement for the case of using CPU only (the RBF neuron implemented with @jit on PC2) provides a completely different plot, with best synaptic times achieved for a small number n (but large values when n increases) as seen in Fig.6. This indicates that for GPUs a larger n allows a more efficient realization.

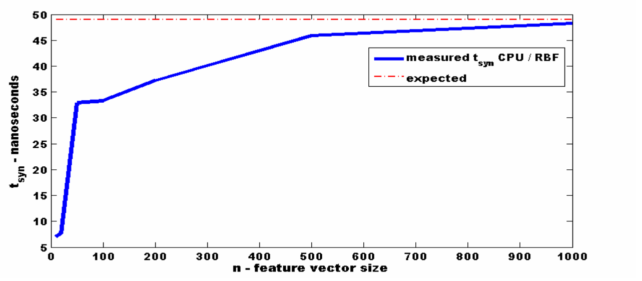


Fig. 6. Dependence on n for the measured synaptic time (PC2 CPU @jit, M=4096). Note that a good value (around 7 nanoseconds) is obtained for $n<20$. With large n synaptic time saturates up to around 49 nanoseconds.

The graph in Fig. 5 was plotted for the case $M=4096$ (GTX-750 GPU). If one needs more inputs n, in order to avoid CUDA errors, M decreases as shown in table III (generated using the same GPU as for Fig. 5). As seen in Table III, the efficiency is always better for a very large number of inputs (e.g. last column corresponds to a 256x256 input image).

TABLE III. SYNAPTIC TIMES WITH RESPECT TO NUMBER N OF INPUTS

M,B	4096, 128	4096, 128	4096, 128	4096, 128	4096, 128	2048, 64	1024, 32	512, 16
n	16	64	256	784	1024	4096	16384	65536
GPU time (s) per BNL	0.43	0.54	0.93	1.95	2.36	2.3	2.23	2.31
Synaptic time (ns)	1.60	0.5	0.22	0.15	0.14	0.133	0.13	0.134

The synaptic time t_{syn} is a good indication of the absolute performance, allowing an implementation solution to be compared to any other given in the literature. Consequently one should tailor its algorithm parameters accordingly when expecting high efficiency aiming to obtain as much smaller as possible t_{syn}.

V. Accelerating CPU implementation using optimized math libraries

In the above examples CPU implementations of neurons (RBF or MUL) were done using @jit (just in time compiler) decorator of the NUMBA package. In our previous work from 2015 [6] using this option for implementing cellular nonlinear options proved to be the best choice and then compared favorably in terms of speed with a linear algebra implementation (using NUMPY). Since 2016, the optimized Intel® MKL (Math Kernel Library) was integrated in NUMPY [7][8][9] providing efficient implementation on Intel processors. In order to test how efficient this library is for our problem we simply implemented the neural layer (MUL) as the following line:

```
Hm=np.tanh(np.dot(np.transpose(inW), Samples))
```

Where **inW** and **Samples** (**X**) are the matrices of parameters and samples (as indicated in Fig.1) and **Hm** is the output of the BNL. The rest of the parameters of the BNL are the same as in previous chapters. As results in Fig. 7 indicate, excellent synaptic times are now achieved using the PC2 (laptop) configuration. Note the similarity with GPU in dependence on n, with better efficiency for larger n.

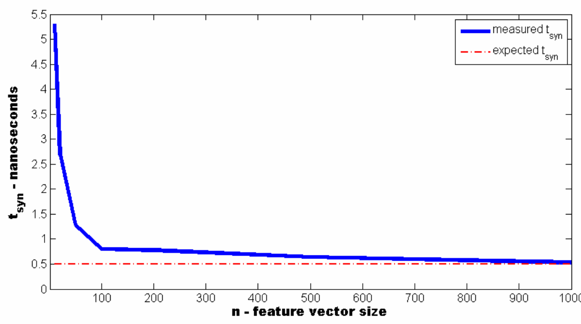


Fig. 7. Dependence on n for the measured synaptic time (PC2 CPU using NUMPY with Intel® MKL, M=4096). Note that very good values (around 0.5 nanoseconds) is obtained for $n>200$. As in the GPU implementation, for small n values the synaptic time gets larger (worse) approaching the value obtained for small n on the same CPU using @jit.

Note that for large number of inputs (n), the CPU implementation is almost 100 times faster than what was achieved using @jit decorator. And, comparing with results in Fig.5 it follows that our best GPU performs only 2.5 times faster than CPU with MKL. While for the MUL neuron the above Python line resolves elegantly the MKL-based implementation, much work has still to be done in order to exploit the MKL facilities for RBF neurons. Our preliminary results indicates a possible solution with synaptic times only 1.2 times slower than for MUL layers.

Further exploration of the MKL facilities is worth to be further investigated, particularly when neural networks should be implemented without GPU support. Table IV gives "synaptic" times in nanoseconds (times allocated to the basic add and multiply operation) using NUMPY (with MKL support) in multiplying two square matrices with 1000x1000 size each.

TABLE IV. COMPUTING TIMES FOR DIFFERENT DATA-TYPES USING NUMPY (WITH MKL) IN PYTHON

Data type	float16	**float32**	float64	int16	int32	int64
$t_{syn}(ns)$	17.8	**0.094**	0.43	7.86	7.98	8.6

The results in this table indicates that MKL optimized library works only for *float32* or *float64* types and indicates that in order to obtain the highest efficiency *float32* type would be a better choice (note that all our previous experiments were done using *float64* data type). Indeed, redoing the CPU MKL example with *float32* type gives $t_{syn} \cong 0.25ns$, which is only 2 times slower than what was achieved on GTX750 GPU (see Table III).

VI. CONCLUDING REMARKS

The main conclusion of our study is that an empirical investigation on how the GPU can be used with a maximal efficiency using a high productivity programming approach (Python with adequate GPU / CPU computation packages) succeeds following some very simple rules of thumb:

i) Define the computational kernels such that the "map-reduce" (i.e. sum of weighted inputs or distance computations) part of the computation is executed on the kernel, thus avoiding the limitation observed in [3][4] for GPUs;

ii) Use a value $M^2=m*N$ (of threads to be distributed on the parallel GPU cores) as large as supported by the available GPU board and find the smallest block size *B* to accommodate it; Consequently the number of computational kernels is M^2, each of them being independent on the other and thus with no need for a "map-reduce" unit. Both the input **X** (a block of $n*N$ samples) and the result of the output layer (let call it **H**) will be available in the GPU memory, thus reducing at minimum the need of PC(host)-to-GPU transfers. For any GPU board there is a maximal *n* (feature vector size) optimizing the best speedup (e.g. 1024 for GTX-750), while using small *n* in general results in a less efficient solution. For GPU, the method proposed herein is ideal for neural layers processing images or other big-data signals with a network size bigger than the minimal unit (BNL) optimized as indicated. For larger networks, one can split them in convenient BNL slices each having *M* size (padding with 0-valued weights the unused parts of BNL when the network sizes are not multiples of *m* and *N*). The above approach results generates simple and portable Python code easy to integrate into more complex tools capable to solve the training and use of such large networks operating on very large data-sets.

For CPU acceleration it is worth investigating if linear algebra libraries are available. In particular, for Intel processors that were available on our platforms, using Intel MKL proves to be a very good option, giving an acceleration (when compared to JIT compiled version of the code) comparable to what our medium/low GPU provided at less costs. The following table gives a synthesis of accelerations obtained for implementing RBF layers on different platforms and in the case of CPU using different libraries and compilers.

TABLE V. SYNAPTIC TIMES AND ACCELERATIONS (COMPARED TO @JIT) FOR GPU AND CPU PLATFORMS

Computing platform	GPU (GTX-750) with NUMBA	CPU with Intel MKL (float 32) under NUMPY	CPU .MEX (compiled with Gcc)	CPU (JIT – NUMBA)
t_{syn}	0.13	0.33	4.6	46
Speedup (with respect to Gcc)	35	14	1	0.1

Further possibilities to improve efficiency will be considered in future research, including a study on how various data types may help improving efficiency (for GPU, using fixed point computations instead of 8 byte floating point types while for CPU MKL libraries with float32 type proved to offer the best choice). Also testing other Python packages (e.g. MINPY[3], THEANO[4]) supporting transparent GPU programming would be part of future research.

REFERENCES

[1] Javier Lopez-Fandino, Dora B. Heras, Francisco Arg uello, "Efficient Classification of Hyperspectral Images on Commodity GPUs using ELM-based Techniques", July 2014. Available: https://citius.usc.es/sites/default/files/publicacions/PDPTA14-CiTIUS_template.pdf

[2] S. Scanzio, S. Cumani, R. Roberto, F. Mana, and P. Laface. Parallel implementation of artificial neural network training for speech recognition. Pattern Recognition Letters, 31(11):1302-1309, 2010.

[3] G.M. Stefan and M. Malita, "Can One-Chip Parallel Computing Be Liberated From Ad Hoc Solutions? A Computation Model Based Approach and Its Implementation", in Proceedings of the 18th International Conference on Computers (part of CSCC '14), Advances in Information Science and Applications - volume II, Santorini Island, Greece, 2014, pp. 582–597.

[4] G.M. Stefan, C. Bira, R. Hobincu, M. Malita,"FPGA-Based Programmable Accelerator for Hybrid Processing", in ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY Volume 19, Numbers 1-2, 2016, 148–165

[5] R. Dogaru and I. Dogaru, "A super fast vector support classifier using novelty detection and no synaptic tuning," 2016 International Conference on Communications (COMM), Bucharest, 2016, pp. 373-376.

[6] R. Dogaru and I. Dogaru, "A Low Cost High Performance Computing Platform for Cellular Nonlinear Networks Using Python for CUDA," 2015 20th International Conference on Control Systems and Computer Science, Bucharest, 2015, pp. 593-598.

[7] Intel® MKL for Python; Available: https://software.intel.com/en-us/blogs/python-optimized

[8] Anaconda with Intel® MKL; Available: - https://www.continuum.io/blog/news/latest-anaconda-innovation-combines-intel-mkl-enhance-analytics-performance-7x

[9] A.A. Fedotov, V.N. Litvinov, A.F. Melik-Adamyan, "Speeding up numerical calculations in Python",in Russian Supercomputing Days 2016; Available: http://russianscdays.org/files/pdf16/26.pdf

[3] https://media.readthedocs.org/pdf/minpy/stable/minpy.pdf
[4] http://deeplearning.net/software/theano/