On the evaluation of energy-efficient deep learning using stacked autoencoders on mobile GPUs

G. Falcao*, L. A. Alexandre†, J. Marques*, X. Frazao†, J. Maria*
*Instituto de Telecomunicações, Department of Electrical and Computer Engineering, University of Coimbra, Portugal
†Instituto de Telecomunicações, Department of Informatics, University of Beira Interior, Portugal

***Abstract*—Over the last years, deep learning architectures have gained attention by winning important international detection and classification challenges. However, due to high levels of energy consumption, the need to use low-power devices at acceptable throughput performance is higher than ever. This paper tries to solve this problem by introducing energy efficient deep learning based on local training and using low-power mobile GPU parallel architectures, all conveniently supported by the same high-level description of the deep network. Also, it proposes to discover the maximum dimensions that a particular type of deep learning architecture—the stacked autoencoder—can support by finding the hardware limitations of a representative group of mobile GPUs and platforms.**

***Index Terms*—Parallel processing, Mobile GPU, Low-power, Energy savings, Deep Learning, Stacked Autoencoders**

I. Introduction

To address both the increasing size of training datasets and corresponding high computational cost, modern deep learning approaches of neural networks have been turning towards the cooperative use of GPU clusters [1]. However, training can still take hours, days or even weeks to complete.

While the current trend in machine learning is using convolutional neural networks (CNNs), such current state-of-the-art implementations tend to consume high levels of energy in order to produce the expected results, which directly impacts the processing costs of big data and creates constraints in their utilization in low-power-driven autonomous vehicles/robots, that consume at least one order of magnitude less energy while guaranteeing equally competitive throughput and classification error performance, when compared to desktop GPUs or CPUs.

In this paper we propose a scalable parallel solution for stacked autoencoder (SAE) architectures in mobile GPUs, that allow providing to small autonomous robots/vehicles deep learning capabilites. The paper builds upon [2] as a first step towards the implementation of more complex approaches to deep learning, such as CNNs, so as to understand the possible gains in terms of energy savings, as well as comprehend the limitations at hardware and software levels.

The goal is to conciliate the performance of deep learning applications, such as object detection and classification, with real-time execution capabilities at low-energy consumption budgets and discover the associated hardware constraints.

Currently, several frameworks allow the training and evaluation of deep learning models, (e.g. Theano [3]). However, these do not allow to change all aspects of the algorithm for the proposed experiments, which required the development of code to support higher control degrees over several aspects of execution and model parallelization.

II. Deep Learning and Stacked Autoencoders

The use of more than two hidden layers in neural network supervised learning was seen as unnecessary until recently [4]. The exceptions were the neocognitron [5] and the convolutional neural networks (CNNs) [6], both developed mostly for visual tasks with the main issue being the difficulty in training several hidden layers using standard back-propagation: there were problems with adjusting the weights as the depth increased (vanishing gradients) [7].

After that, it has become a major trend in machine learning (deep learning is currently the state-of-the-art approach in multiple domains), when the efforts by Hinton and co-workers [8] resulted in the ability to train deep neural networks (DNNs), namely Deep Belief Networks (DBNs). At the same time, other groups proposed a way to train deep networks based on stacking autoencoders[9].

The potential advantages that come with using deep learning are the possibility of having increasingly more abstract levels of representation, reusing the intermediate level representations across different tasks and also obtaining a more compact and efficient representation for certain types of problems [10].

A. Stacked autoencoders

An autoencoder (AE) is a network that tries to produce at the output what is presented in the input [2]. The most basic AE is a multi-layered perceptron that has one hidden and one output layer, such that the weight matrix of the last layer is the transpose of the weight matrix of the hidden layer (clamped weights) and the number of output neurons is equal to the number of inputs. An AE is trained in an unsupervised manner (no class information is used).

To obtain a deep architecture using AEs they are stacked on top of each other such that the output of an AE is the input for the next one. This stacking can produce a deep network: the SAE. The SAE is obtained as follows: first pre-train several AEs such that the first learns to approximate the inputs from the dataset, the second learns to approximate the hidden representations of the first and so on. A final layer of neurons is placed on top of the AE that is the output layer and will have as many neurons as there are classes in the problem (e.g. a softmax layer). The training is then performed for all layers in a supervised manner (called fine-tuning).

2377-5750/17 $31.00 © 2017 IEEE
DOI 10.1109/PDP.2017.98

Algorithm 1: Training Phase

```
1: Load training set from disk
2: if load_checkpoint = true then
3:     Load weights from previous checkpoint
4: else
5:     Generate random weights
6: end if
7: Initialize OpenCL
8: for layer = 0 to number_of_layers: do
9:     Allocate INPUT, OUTPUT and WEIGHTS buffers
10:    Allocate ERROR and GRADIENT buffers
11:    for batch = 0 to number_of_batches: do
12:        {Parallel Encoder's Feed-Forward}
13:        INPUT ⇐ Host, WEIGHTS ⇐ Host
14:        Enqueue the Feed-Forward parallel kernel (HiddenNodes × BatchSize
           work-items)
15:        Compute the encoder's feed-forward phase on the OpenCL device
16:        Host ⇐ OUTPUT
17:        {Parallel Decoder's Feed-Forward}
18:        Decoder INPUT ⇐ Encoder OUTPUT
19:        Enqueue the Feed-Forward parallel kernel (VisibleNodes × BatchSize
           work-items)
20:        Compute the decoder's feed-forward phase on the OpenCL device
21:        Host ⇐ OUTPUT
22:        {Parallel Decoder's Back-Propagation}
23:        Enqueue the Back-Propagation - Output Layer parallel kernel (VisibleNodes
           work-items)
24:        Compute the encoder's Back-Propagation phase on the OpenCL device
25:        Host ⇐ ERROR
26:        {Parallel Encoder's Back-Propagation}
27:        Decoder GRADIENT ⇐ Encoder GRADIENT
28:        Enqueue the Back-Propagation - Hidden Layer parallel kernel
           (HiddenNodes work-items)
29:        Compute the decoder's back-propagation phase on the OpenCL device
30:        Host ⇐ GRADIENT
31:        Update weights for the next epoch
32:    end for
33:    Release all buffers
34: end for
```

III. Hardware parallelism for Neural Networks

A. Mapping parallel OpenCL kernels on the device

For the parallel development of the training phase, three OpenCL kernels were created. The first one relates to the feed-forward algorithm, sending the data through the network, layer-by-layer, and computing the results. The second kernel computes the AE reconstruction error at the output layer and begins the gradient-based back-propagation algorithm. The back-propagation, as the feed-forward, has data-dependencies from the previous layer. Since the back-propagation for the hidden layer is dependent on the gradient calculations from the output layer, this results in a third kernel for that purpose. The training phase is described in Algorithm 1.

After the training process, the SAE is ready to classify the provided test samples. The decoder's feed-forward and all back-propagation are now withdrawn from the computation, leaving the network with only the encoder from each AE. This phase is described in Algorithm 2.

1) Feed-forward: When the samples from the dataset and weights for that layer are loaded to the device's global memory, the initial phase is started by sending data through the network. The kernel is launched on the device across two dimensions, the first being equal to the output nodes of the current layer and the second relative to the amount of samples from the dataset. This means that one particular work-item is responsible for one output node when all the input nodes

Algorithm 2: Testing Phase

```
1: Load training set from disk
2: Load weights from training phase
3: Initialize OpenCL
4: for layer = 0 to number_of_layers: do
5:     Allocate INPUT, OUTPUT and WEIGHTS buffers
6:     for batch = 0 to number_of_batches: do
7:         {Parallel Encoder's Feed-Forward}
8:         INPUT ⇐ Host, WEIGHTS ⇐ Host
9:         Enqueue the Feed-Forward parallel kernel (HiddenNodes × BatchSize
           work-items)
10:        Compute the encoder's feed-forward phase on the OpenCL device
11:        Host ⇐ OUTPUT
12:    end for
13: end for
14: Compute final classification accuracy
```

from one sample go through it. Inside the kernel, a weighted sum is computed in a loop, over all the layer input nodes and respective weights for that particular output node, computing the overall sum of that product. An activation function (the sigmoid function), is then applied to that sum plus the bias of that output node. This kernel is valid for both the encoder and decoder phase of the AE, the only difference being the input varying between the original image for the encoder layer and the encoder output for the decoder layer.

2) Back-propagation (output layer): After computing the feed-forward across the AE (encoder, then decoder), the resulting decoder output is of the same size as the encoder's input. We then have the possibility of calculating a reconstruction error. The kernel developed for this phase calculates that error and then computes the gradient descent on the back-propagation. Since we are batch training the network, this time the kernel is launched only on one dimension, as opposed to two dimension like the feedforward phase, so as to fit into the device's memoryThe algorithm inside the kernel then loops over all dataset samples, computing the reconstruction error and gradient for each sample. The partial derivative for the weights is then calculated via the gradient. The value for the bias is obtained directly from the gradient, with the weights also being dependent on the output from the previous hidden layer. When all the samples have been processed, the mean of the gradient is needed due to the batch training.

3) Back-propagation (hidden layer): The kernel used for the back propagation in the hidden layer is close to that of the output layer. We do not have a reconstruction error for this layer but we are dependent on the gradient calculated in the output layer. The kernel is then launched with one dimension, the size of the hidden layer output nodes. The product of the weights of this layer and the output gradient is summed across the input nodes, with the resulting sum replacing the error in the previous algorithm, finally obtaining the gradient for this layer. The kernel then proceeds to compute the partial derivatives as described in the output layer kernel. When the back propagation for this hidden layer comes to an end, the partial derivatives are then copied to the host where a simple loop updates the weights and bias, this being a fast and low computationally demanding operation. In order to implement the aforementioned parallel kernels, we developed parallel

kernels for mobile GPUs with optimizations that are identified in the next subsections.

B. Mobile GPU specific high-level memory optimizations

For the mobile GPU case, the memory embedded in the system on chip (SoC), present in smartphones with ARM CPUs and mobile GPUs, differs from regular OpenCL devices. Usually, on conventional desktop GPUs, there is a host memory and a separate memory, directly on the device's (GPU) board. These systems require memory transactions (copies, reads and writes) between the host and device, usually via the PCI-e bus linking them together, so the data is accessible on the faster device's memory. For SoC implemented in smartphone and similar devices, a single memory is available and thus shared by host and device. The memory transactions between host and device are therefore unnecessary, as the memory space is the same across both of them.

1) Shared memory: An algorithmic limitation with impact in the utilization of mobile resources consists of the need of floating-point calculation to be performed on input data and weights product between host and device. To ensure an implementation with zero-copy buffers, allocation of said buffers must be first performed via a call to `clCreateBuffer` with the flag `CL_MEM_ALLOC_HOST_PTR`, resulting in a buffer visible by both the host CPU and GPU OpenCL device. This ensures the buffer is automatically memory aligned to the device, and that an unnecessary copy and data duplication is not performed at a later stage in the pipeline. After the allocation is complete, the buffer can be mapped to a host pointer with `clEnqueueMapBuffer` and filled with the necessary data to be processed. The buffer can then be returned to the device's control via `clEnqueueUnmapMemObject`, after which the kernel is launched.

This process is necessary, since buffers created on the host side via `malloc()` cannot be mapped to the device's memory space and, furthermore, buffers created with the `CL_MEM_USE_HOST_PTR` flag and then linked to an existing host side pointer will still result in a time expensive copy and in data duplication.

IV. Experimental Results on Low-power Architectures

The goal of the experiments was three-fold: 1) validate the implementations in all devices; 2) allow comparing energy consumption between the tested platforms; and 3) find their hardware limitations namely due to memory and processing capabilities. For this we have chosen a well known dataset, the MNIST [11]. The five computing platforms used in these experiments are listed in Table I.

The desktop GPU is used only for reference, since our focus is on low-power devices. The training hyper-parameters defined for the SAE consist of a training batch of 64 images and an initial learning rate of 0.45 on a network of size $784-500-250-10$. For this particular SAE we achieved a classification error of 1.47% training during 1500 epochs.

TABLE I
Computing Platforms. Mobile devices have shared RAM.

Platform	CPU	GPU
refGPU	i7 4770k, 32GB	GTX Titan, 6GB
mGPU1	ARMv7 Krait 400,	Adreno 330, 2GB
mGPU2	ARMv7-A Krait 450	Adreno 420, 3GB
mGPU3	ARMv7 Krait 400	Adreno 330, 3GB
mGPU4	ARMv8-A Cortex-A57	Adreno 430, 4GB

A variety of reconstruction and classification outputs were analyzed, along with a graphical output of the estimated classification as a function of the expected labels varying from digit 0 to 9, and we present a few cases with a high degree of probability (higher than 0.9) in Fig 1. It should be noted that since the algorithm remains equal and weights are initialized with the same random seed generator, the error is constant in all platforms.

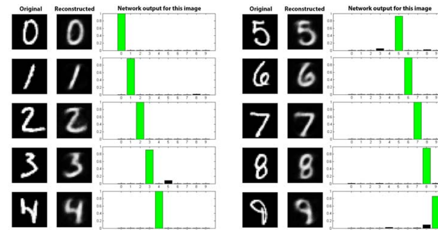

Fig. 1. Some of the images correctly classified (from MNIST).

For the energy consumption analysis in Table II we kept the same SAE architecture using 1 epoch. These measurements scale linearly with the number of epochs. Power consumption was calculated measuring the idle requirements of the entire system (host and device) and then launching the application, measuring the power difference (load - idle) over the SAE execution time, using a power meter for the desktop refGPU and the PowerTutor [12] application for the remaining devices.

TABLE II
Execution time of 1 epoch and energy consumption on a network of size 784-500-250-10 (*lower is better)

Device	Exec. Time (min|sec)	Average Power (W)	Energy Consump. (Wh)*	Energy Consump. (vs GPU)*
refGPU	54s	247	3.7050	-
mGPU1	13m25s	0.242	0.0541	1.46%
mGPU2	11m33s	0.230	0.0436	1.18%
mGPU3	12m15s	0.317	0.0593	1.61%
mGPU4	10m58s	0.140	0.0256	0.69%

Table II shows that regarding energy consumption, mobile devices are clearly better than the reference desktop refGPU, achieving the same results while consuming only from 0.69% to 1.61% of the energy, which can be attributed to both the

optimizations performed and the hardware, since mobile GPUs are far more energy efficient than the desktop counterpart. Considering the energy-efficiency point of view, mobile devices clearly outperform the refGPU, despite taking 15 times more time to complete the same task. It should be noted, however, that using the power meter to measure the average power for both smartphone platforms (mGPU1 and mGPU2) we achieve approximately 3.4W, which represents the power required by the entire development platform. Nonetheless, using those values as basis for energy consumption calculation would give 0.7603Wh and 0.6451Wh, respectively. Even for such worst case scenarios, mobile devices still require only 20% of the energy of the desktop GPU.

For the hardware limitations analysis we test an increasing number of neurons for the first hidden layer until a maximum is reached (i.e., device kills the process), thus achieving the maximum weights that each device can train using its GPU. Table III indicates the maximum dimensions achieved.

TABLE III
EXECUTION TIME OF 1 EPOCH AND MAXIMUM NUMBER OF WEIGHTS FOR EACH MOBILE DEVICE

Device	Execution Time	First Hidden Layer Neurons	Number of Weights
mGPU1	1h51m24s	3150	3263010
mGPU2	3h50m19s	5950	6161010
mGPU3	3h10m44s	5000	5177760
mGPU4	3h58m13s	7250	7506510

As a term of comparison, the refGPU ran the SAE for each mobile devices' largest architecture in 5m43s, 9m13s, 7m40s and 11m40s, respectively. As is normal when using SAE, the number of weights was calculated using the following formula:

$$(784 + 1) \times N + (N + 1) \times 250 + (250 + 1) \times 10 \qquad (1)$$

where 784 is the number of inputs, N is the number of neurons from the first hidden layer, 250 the number of neurons from the second hidden layer and 10 is the number of outputs.

Although Table III shows that due to hardware limitations the devices perform significantly slower for very large neural networks, they run fast small to medium sized networks (as seen in Table II), albeit execution times are higher than they normally would in desktop GPUs. However, energy consumption savings make up for such higher execution times.

To further grasp hardware limitations results, there are several factors that need to be considered: first, mobile GPUs do not have dedicated memory, so the memory that is available is small and managed by the SoC, varying between devices; also, even using the same SoC, results can vary by simply using different OS versions that can implement different resource management policies; and finally, we have to consider that mobile devices only recently started supporting OpenCL, so these implementations have still margin to progress. With the expected advances of hardware and new OpenCL implementations, OpenCL capabilities in mobile devices will likely improve considerably in the near future.

V. CONCLUSIONS

This work presented energy-efficient training and testing of deep neural networks of the SAE type on mobile smartphones and low-power GPUs. We addressed implementation details and experimental analysis by comparing the energy consumption of 5 different and representative embedded architectures. We have found the limits in terms of the maximum deep neural network size that fits their restricted hardware resources.

Although not as fast as on a desktop GPU, the training on mobile GPUs uses less than 2% energy than it would on the desktop counterpart, opening room to the processing of compute-intensive algorithms directly on autonomous vehicles, robots and other low-power applications.

Moreover, this study paves the way for technology progression, as mobile GPUs with more hardware resources are developed. This may include state-of-the-art networks, such as CNNs, running exclusively on low-power devices, achieving top results in terms of energy savings and classification accuracy as well. Additionally, the use of approaches such as Deep compression [13] to improve speed and reduce storage needs can further contribute to this goal.

Finally, we have made the OpenCL source code available (https://montecristo.co.it.pt/pdp17) to the community that wishes to replicate these experiments.

VI. ACKNOWLEDGEMENT

This work was supported by FCT and Instituto de Telecomunicações under grant UID/EEA/50008/2013.

REFERENCES

[1] A. Coates, B. Huval *et al.*, "Deep learning with COTS HPC systems," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 1337–1345.
[2] J. Maria, J. Amaro *et al.*, "Stacked autoencoders using low-power accelerated architectures for object recognition in autonomous systems," *Neural Processing Letters*, vol. 43, no. 2, pp. 445–458, 2015.
[3] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.
[4] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314.
[5] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, 1980.
[6] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time-series," 1995.
[7] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
[8] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006.
[9] Y. Bengio, P. Lamblin *et al.*, "Greedy layer-wise training of deep networks," in *In NIPS*. MIT Press, 2007.
[10] Y. Bengio, "Learning deep architectures for AI," *Found. Trends Mach. Learn.*, vol. 2, no. 1, pp. 1–127, Jan. 2009.
[11] Y. LeCun. (2014) MNIST Dataset. http://yann.lecun.com/exdb/mnist/. Accessed: 2015-04-15.
[12] "PowerTutor: A Power Monitor for Android-Based Mobile Platforms," http://ziyang.eecs.umich.edu/projects/powertutor, Accessed: 2015-04-15.
[13] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2015.