Neural Process Lett (2016) 43:445–458
DOI 10.1007/s11063-015-9430-9

Stacked Autoencoders Using Low-Power Accelerated Architectures for Object Recognition in Autonomous Systems

Joao Maria[1] · Joao Amaro[1] · Gabriel Falcao[1] · Luís A. Alexandre[2]

Published online: 12 May 2015

Abstract This paper investigates low-energy consumption and low-power hardware models and processor architectures for performing the real-time recognition of objects in power-constrained autonomous systems and robots. Most recent developments show that convolutional deep neural networks are currently the state-of-the-art in terms of classification accuracy. In this article we propose to use of a different type of deep neural network—stacked autoencoders—and show that within a limited number of layers and nodes, for accommodating the use of low-power accelerators such as mobile GPUs and FPGAs, we are still able to achieve both classification levels not far from the state-of-the-art and a high number of processed frames per second. We present experiments using the color CIFAR-10 dataset. This enables the adaptation of the architecture to a live feed camera. Another novelty equally proposed for the first time in this work suggests that the training phase can also be performed in these low-power devices, instead of the usual approach that uses a desktop CPU or a GPU to perform this task and only runs the trained network later on the FPGA. This allows incorporating new functionalities as, for example, a robot performing runtime learning.

Keywords Deep learning · Neural networks · Stacked autoencoder · Parallel computing · FPGAs · Mobile GPUs · OpenCL · Low-power · Autonomous systems

✉ Gabriel Falcao
gff@co.it.pt

Joao Maria
jmaria@co.it.pt

Joao Amaro
jamaro@co.it.pt

Luís A. Alexandre
luis.alexandre@ubi.pt

[1] Department of Electrical and Computer Engineering, Instituto de Telecomunicações, University of Coimbra, R. Silvio Lima, 3030-290 Coimbra, Portugal

[2] Department of Informatics and Instituto de Telecomunicações, University of Beira Interior, R. Marques d'Ávila e Bolama, 6201-001 Covilhã, Portugal

1 Introduction

Over the last years, deep neural networks (DNNs) have established as the state-of-the-art in terms of classification performance on many different tasks [7,11,12]. In particular, convolutional neural networks (CNNs) have assumed greater and greater importance [7], since they have shown performances 30–80 % superior when benchmarking against 7 typical datasets commonly used to assess these algorithms.

Against what was considered the best approach in the recent past, they have shown that using several layers can lead to superior performance [5,6,15,19]. Such use of multiple representation stages can be achieved using CNNs or other types of DNNs such as stacked denoising autoencoders (SDAE). Also impactful, in order to obtain superior classification performance, is the number of samples currently used to train these algorithms. They surpass the dozens to hundreds of thousands, which has considerably increased the computational complexity required to train these networks for achieving good performance.

The fact that these models are computationally intensive to train has encouraged the porting of these algorithms for execution on graphics processing unit (GPU) devices [21]. This allowed concurrent execution of different parts of the neural network either at training or classification phases, thus accelerating the long processing times. However, top performer GPUs, which are mainly desktop accelerators coupled to a host CPU, have reached power and heat dissipation walls, as the number of stream processors included on a single die has risen to thousands [14]. Also, power, heat dissipation and physical limitations in the chip limit the frequency of operation of these devices to values around 1 GHz.

There have been previous attempts at implementing deep learning architectures on FPGAs, but to the best of our knowledge, the high costly training phase was always performed first on a separate machine, either recurring to CPUs or GPUs to perform that computation, and the trained model was then implemented on the FPGA [8,10].

Also, the computational power of mobile GPUs in smartphones and tablets opens new possibilities for portable processing power, mainly in the area of computer vision [25]. In fact these platforms are equipped with a variety of sensors and cameras suitable for this type of application.

In this paper we propose the use of stacked autoencoders (SAEs) in low-power mobile GPUs and FPGAs to perform the real-time classification of objects. Instead of a traditional approach to improve on the state-of-the-art regarding classification accuracy, this work aims at reaching a sub-optimal classification performance, by proposing solutions that are capable of achieving those performances in real-time running in low-power devices. Among the multiple applications that can benefit from such use of deep neural networks, we find robots and other types of autonomous vehicles that are limited to severe low-power constraints. We used a parallel computing language and framework—OpenCL—to develop kernels for concurrent execution on these accelerators [9]. We have parallelized both the training and classification phases of the process, which allows the robot to perform the training of newly acquired datasets during runtime. Although we can find in the literature a vast set of works describing the implementation of neural networks on FPGAs, for the best of our knowledge the inclusion of the training phase on an FPGA has never been reported before.

We achieved 10 fps on the training phase and more importantly, real-time performance during classification, with 119 fps while classifying the CIFAR-10 color dataset. In the end, the approach proposed in this work is capable of achieving classification performances comparable to the mid level of the Kaggle table [16], and above the accuracy obtained from processing raw pixels as the input data [17], while demanding power consumption levels ranging from 6.6 to 16 W, which makes them suitable for being incorporated in autonomous

systems. Moreover, the proposed solution is scalable to future devices that expectedly should have more hardware resources and processing cores available [22], allowing more frames per second to be processed or more complex deep neural networks to be developed.

2 Sub-optimal Neural Networks: The Stacked Autoencoder

We are interested in using deep learning for object recognition. One of the simplest methods consists of using a series of autoencoders, stacked on top of each other.

An autoencoder (AE) is a restricted version of an MLP that has one hidden and one output layer, such that the weight matrix of the last layer is the transposed of the weight matrix of the hidden layer (clamped weights) and the number of output neurons is equal to the number of inputs.

In fact, an AE is trying to obtain at its output the values present in the input. Since the hidden layer is usually of a smaller size that the input layer, the network has to be able to represent the input data in some compressed way.

The process of training the AE, can be formalized in the following way. The j-th input value can be represent by x_j, the weight matrix components by $\{W_{ij}\}$, and the input size by n, with $i = 1, \ldots, n_h$ and $j = 1, \ldots, n$, where n_h is the number of hidden layer neurons. The hidden layer neurons output, called the encoding, is obtained with $h_i = s(a_i)$ where

$$a_i = b_i + \sum_{j=1}^{n} W_{ij} x_j, \tag{1}$$

b_i is the bias of the hidden layer neuron i and $s(\cdot)$ is the sigmoid function. The ouput layer values, or the decoding, is given by

$$\hat{x}_j = s(\hat{a}_j) = s\left(c_j + \sum_{i=1}^{n_h} W_{ij}^T h_i\right), \tag{2}$$

where c_j is the bias of the output layer neuron j. A possible cost function to use for the training algorithm is

$$C(\hat{\mathbf{x}}, \mathbf{x}) = \sum_{k=1}^{n} (\hat{x}_k - x_k)^2 \,. \tag{3}$$

When the sigmoid is used as the activation function, the weight update is done with:

$$W_{ij} = W_{ij} - \eta \sum_{k=1}^{n} \left[(\hat{x}_k - x_k)\hat{x}_k(1 - \hat{x}_k) \left(h_i + W_{ik} h_i (1 - h_i) x_j \right) \right], \tag{4}$$

$$b_i = b_i - \eta \sum_{k=1}^{n} \left[(\hat{x}_k - x_k)\hat{x}_k(1 - \hat{x}_k) W_{ik} h_i (1 - h_i) \right] \tag{5}$$

and

$$c_j = c_j - \eta(\hat{x}_j - x_j)\hat{x}_j(1 - \hat{x}_j) \,. \tag{6}$$

This process of adjusting the AE's weights in an unsupervised manner is called pre-training (Fig. 1).

The stacked autoencoder (SAE) is built by first pre-training several AEs such that the first learns to approximate the inputs from the dataset, the second learns to approximate the

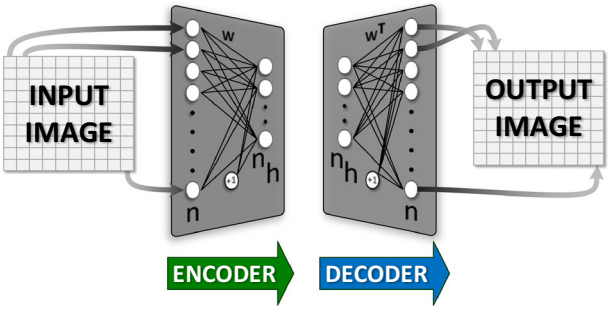

Fig. 1 Pre-training process of the first autoencoder

hidden representations of the first and so on. The output layer is not an AE but a regular MLP layer and has as many neurons as there are classes in the problem. We use the softmax as the activation function of the output layer. So, for the output layer neuron i, its output is given by

$$f(a_i) = \frac{e^{a_i}}{\sum_{k=1}^{L} e^{a_k}}, \tag{7}$$

where L represents the number of classes (and output layer neurons) and a_i is the activation of neuron i obtained using an expression similar to (1) but where the x_i are replaced by the h_i and the b_i by the respective hidden-layer biases.

3 OpenCL Parallelism for Neural Networks

3.1 The OpenCL Programming Framework

A cross-platform parallel computing framework such as OpenCL opens a broad range of possible applications. Currently supported in x86 and ARM CPUs, desktop and mobile GPUs, several APUs and FPGAs [9], the OpenCL programming framework provides the means to easily port an existing code into any compatible device [22], provided there is a software development kit (SDK) for that desired platform. The OpenCL framework links a host to one or more OpenCL devices, forming a single heterogeneous computational system [13]. The framework is structured in the following manner:

1. *Platform layer* The platform layer supports the host program, finding available OpenCL devices and their capabilities and then creating a connection through a context environment (Fig. 4).
2. *Runtime* The runtime component allows the host program to manipulate context environments once they have been created, sending kernels and command queues to the device.
3. *Compiler* From the OpenCL kernels the compiler produces program executables. The OpenCL C programming language implemented by the compiler supports a subset of the ISO C99 language with extensions for parallelism [13].

A parallel implementation of a standard sequential algorithm, as described in Fig. 2, can induce a considerable speedup on the overall processing time. In the sequential algorithm the calculation is performed one row at a time, and a computationally expensive control check is performed at the end of every loop. In the parallel algorithm, the parallel function

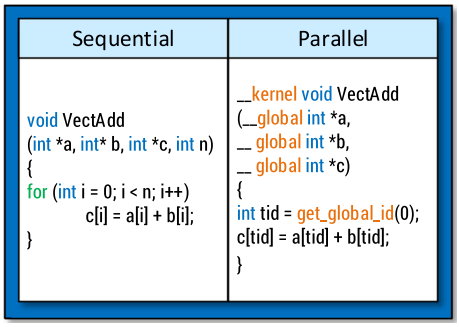

Fig. 2 Traditional sequential processing *versus* parallel processing

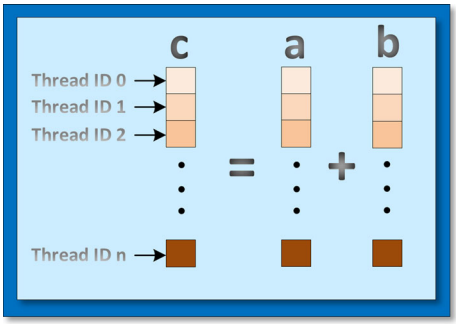

Fig. 3 Multithread parallelism on a vector addition computation (in OpenCL and through this text a work-item defines a computing thread)

(called kernel) is launched **n** times, equal to the expected number of loops in the sequential version, and the calculations are performed simultaneously on all vector points, by distinct work-items (i.e., computing threads) [9] as depicted in Fig. 3.

3.2 OpenCL Kernels for Neural Network Parallelism

To enable the processing parallelism on the SAE described in Sect. 2, three OpenCL kernels (special functions that run on the OpenCL compatible devices) were developed (Fig. 4):

1. *Feed-Forward* Linked to the feed-forward phase of the training algorithm, this kernel sends the data through the network and computes the sigmoidal activation function. The parallel kernel is launched across two dimensions, for a total of *HiddenNodes* × *BatchSize* simultaneous work-items (OpenCL threads) for the encoder computation, and *VisibleNodes* × *BatchSize* for the decoder. Section 3.2.1 presents a detailed description of this phase.

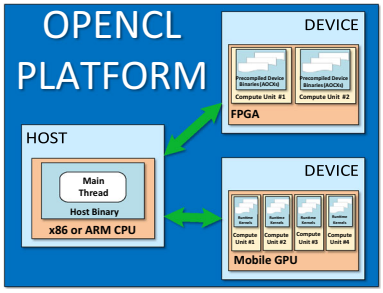

Fig. 4 OpenCL platform model comprised of a host CPU and one or more devices

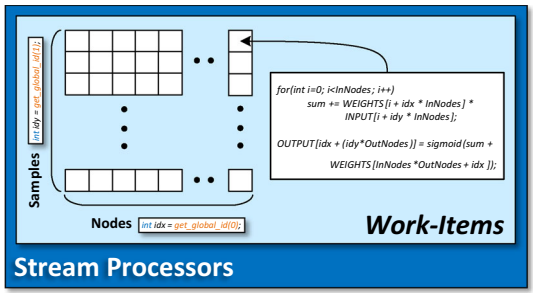

Fig. 5 Feed forward work-items spread across two dimensions

2. *Back Propagation—Output Layer* Computes the reconstruction error on the output (decoder) layer and the gradient-based back-propagation algorithm for that same layer, launching *VisibleNodes* simultaneous work-items. Section 3.2.2 presents a detailed description of this phase.
3. *Back Propagation—Hidden Layer* Since the back-propagation on the hidden (encoder) layer is dependent on the gradient from the decoder layer rather than the reconstruction error, a third kernel was developed for that purpose. The back-propagation on the hidden layer is then launched on *HiddenNodes* simultaneous work-items. Section 3.2.3 presents a detailed description of this phase.

3.2.1 Feed-Forward

After the weights and required batch from the dataset are loaded to the OpenCL device's global memory, the feed-forward phase can begin. In this phase, one particular work-item is responsible for the activation of one input image from the batch, in one of the output nodes from the selected layer. On each of the work-items, the weighted sum is computed over a loop with the input nodes size. A bias for the output node is added to the weighted sum and an activation sigmoid function produces the final output.

This feed-forward kernel has the original image as input for the first AE, with the extracted features from one AE serving as the input for the next AE in the network, culminating on the full SAE network. A visual representation of the work-items/dimension can be seen in Fig. 5.

3.2.2 Back Propagation: Output Layer

When the feed-forward passage ends on a batch, an output is obtained for each of the images in the batch, with the same size as the input. The back-propagation kernel on the output layer performs a pixel-by-pixel comparison of input and output, resulting in a reconstruction error. Each work-item computes the reconstruction error and gradient-based back-propagation in one of the visible nodes of the output layer, for all the images in the batch.

The partial derivative for the weights is then calculated via the gradient, the value for the bias is obtained directly from it, and with the value for the weights also being dependent on the output from the encoder. When all the samples have been processed, a mean of the gradient is computed, due to the batch training method. A visual representation of the work-items/dimension can be seen in Fig. 6.

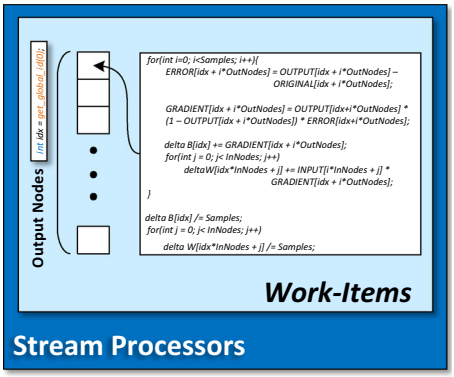

Fig. 6 Back propagation work-items for the output layer (decoder)

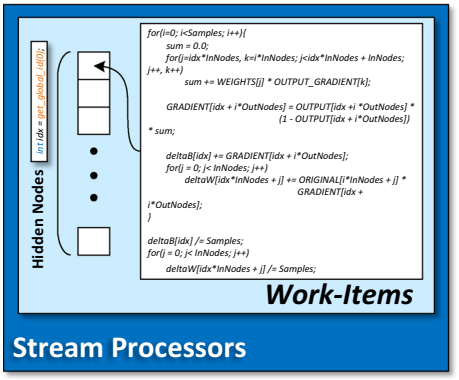

Fig. 7 Back propagation work-items for the hidden layer (encoder)

3.2.3 Back Propagation: Hidden Layer

For the hidden layer, the gradient is obtained from the gradient of the output layer, rather than, as before, from the reconstruction error. This is the main reason for the development of this separate kernel, as it mimics the previous back-propagation kernel in the remaining computations. It is also launched across only one dimension, equal to the number of hidden nodes.

The product of the weights of this layer and the output gradient is summed across 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 computationally easy operation. A visual representation of the work-items/dimension can be seen in Fig. 7.

4 Experimental Results

4.1 The CIFAR-10 Dataset

The CIFAR-10 dataset consists of RGB images of 32 by 32 pixels, each containing one photograph from ten distinct classes: airplane, automobile, bird, cat, deer, dog, frog, horse,

Table 1 Hardware overview of the computing platforms

Platform	Host CPU	OpenCL Device	Device memory
GPU	Intel i7 4790K	NVIDIA GTX Titan	6 GB GDDR5
mGPU	Qualcomm Krait 400	Qualcomm Adreno 330	2 GB LPDDR3*
FPGA	Intel i7 2600k	Altera Stratix V D5	2 × 4 GB DDR3

* Shared memory (between Host and OpenCL device)

Table 2 Cost and power consumption for the OpenCL devices, as per indicated manufacturer data

Platform	Process technology (nm)	Price (USD)	Power (W)
GPU	28	1000	250
mGPU	28	475	10
FPGA	28	4000	30

ship and truck. The dataset is divided into a training set with 50,000 images and a test set with 10,000 images. Each set has an equal distribution of elements from each one of the ten classes. A full discussion of the dataset and the data itself can be obtained online [18].

4.2 Apparatus

The computing platforms used in these experiments are stated in Table 1, with further specifications presented in Sects. 4.2.1 and 4.2.2. The desktop GPU is used only for reference purposes, as our focus is mainly on low-power devices.

The OpenCL devices are manufactured using the same 28nm process design technology. Predictably, even though the low-power alternatives present similar power consumption levels, the desktop GPU has an estimated power consumption an order of magnitude higher. Regarding the purchase costs, the prices differ nearly an order of magnitude between low-power solutions. The mobile GPU also costs only half as much as the desktop version, as seen in Table 2.

The OpenCL devices' throughput performance during the training duration of the SAE is barely affected by the disparity of the host platforms. It was verified via the profiling tool, that the percentage of total computational time on the OpenCL device was 99.86 %, with the host CPU running idle most of the time.

4.2.1 mGPU

The Adreno 330 GPU shares a unified global memory with the Krait CPU, using the remaining space from the 2 GB of LP-DDR3 memory, with up to 12.8 GB/s memory bandwidth [23]. The processing core of the Adreno 330 is composed of four compute units (CUs), each with 32 stream processors (SPs), providing 128 SPs in total.

For testing purposes, a developing platform from Qualcomm was used, the DragonBoard [24], with a Snapdragon 800 SoC, comprised of an ARMv7 Krait 400 CPU at 2.15 GHz and the OpenCL device, the Adreno 330 GPU clocked at 450MHz with 2 GB of shared LP-DDR3 at 1600 MHz. This platform is currently running Android 4.3—Jelly Bean.

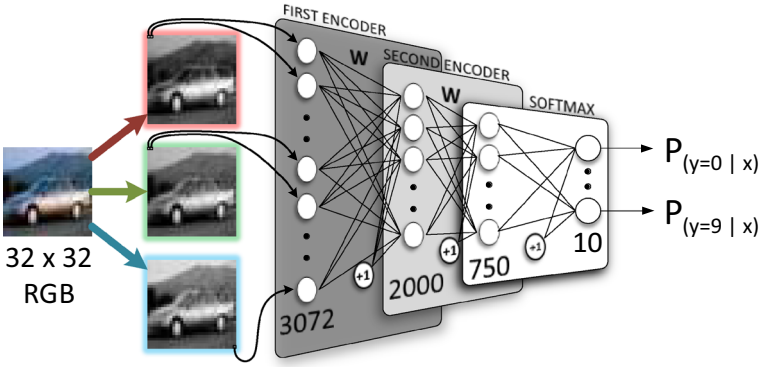

Fig. 8 Topology of the stacked autoencoder for the CIFAR-10 dataset

4.2.2 FPGA

One of the current FPGAs from Altera with OpenCL support is the Stratix V GS D5 [1]. This device has been developed for digital signal processing (DSP) and integrates 3180 18x18, high-performance, variable-precision multipliers, 36 full-duplex 14.1 Gbps transceivers, along with 457000 logic elements, 172600 adaptive logic modules and 690,400 registers. The memory interface allows for up to six independent banks of DDR3 SDRAM on a 72-bit data bus, with connection to the Host made via an 8-lane PCIe 3.0 bus with up to 10 GB/s sustained bandwidth.

The FPGA host system has an Intel i7 2600k at 3.4GHz, with 2 × 4 GB DDR3 of memory, running CentOS release 6.4. The FPGA board is a Nallatech PCIe 385N Stratix V D5 [20], populated with 2 × 4 GB of DDR3 at 1600MHz. The FPGA is used in conjunction with the Altera SDK compiler for OpenCL, version 13.1, in compliance to the 1.0 version of the OpenCL standard [2,4]. This produces a high-level description of the architecture for reconfiguring the FPGA substract, without the specific need of a long development time solution based on hardware description languages, such as Verilog or VHDL [3].

These OpenCL-based descriptions of the architecture allow the developer to manipulate several parameters at programming level, namely: (i) the number of compute units (CU), which are hardware replications of the system for achieving data-parallelism; (ii) loop unrolling that eliminates branch conditional verifications at the end of loops, thus accelerating execution time; and (iii) single instruction multiple data (SIMD) vectorized hardware processing that applies the same instruction to distinct data elements. The best results described in this section were achieved using two CUs on the feed-forward kernel (the one that is used more often), one CU for the other kernels, a loop unroll factor of two and no SIMD vectorization, since the FPGA resources were exhausted by the first two optimizations. These results occupy 88 % of the FPGA resources and process each epoch in 16.87 s.

4.3 Training Hyper-parameters

The training hyper-parameters defined for our SAE consist of a network of size 3072-2000-750-10, deemed the appropriate size for problem reduction, using a training batch of 64 images and an initial learning rate set at 0.01. An overview of the network topology is described in Fig. 8.

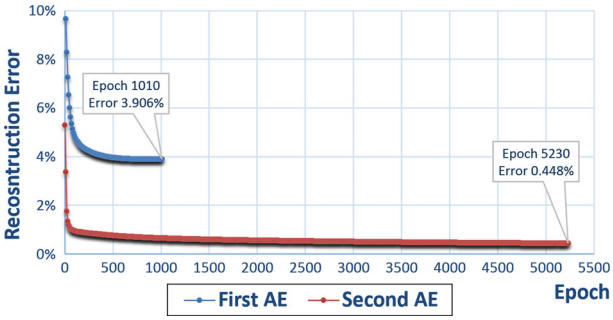

Fig. 9 SAE reconstruction error as function of the number of epochs

Table 3 Final SAE training time with a batch size of 64 images and initial learning rate equal to 0.01

Platform	First AE training time	Second AE training time	Total training time*
GPU	36m38s	2h37m42s	3h14m20s
mGPU	2h21m00s	14h47m26s	17h08m26s
FPGA	6h26m30s	39h04m33s	45h31m03s

* Lower is better

4.4 Evaluating the Neural Network

As we trained the SAE using the CIFAR-10 dataset, several performance metrics were recorded for each of the AEs: the reconstruction error on the validation set, the number of epochs and corresponding duration, amounting in the end to the SAE total training time.

The progression of the reconstruction error for the SAE can be seen in Fig. 9. By training the first AE during 1010 epochs, we achieved a reconstruction error of 3.906 % for the first AE. The second AE was trained during another 5230 epochs with a final reconstruction error of 0.448 %. Since the algorithm remains the same and the weights were initialized with the same random seed generator, the error is constant across both platforms.

In Table 3 we evaluate the training time in both platforms. In the end, the mobile GPU produced the fastest results, training the SAE 3× faster than the FPGA.

The maximum valued output of the network on the Softmax decided the estimated classification, varying from 1 to 0, with 1 being total certainty of the result. A variety of classification outputs were analyzed, along with a graphical output of the estimated classification as a function of the expected labels, all in Figs. 10, 11, 12. We studied cases of correct classification with high degree of probability as seen in Fig. 10. Some cases close to being misclassified are presented in Fig. 11 and finally samples of misclassified images are represented in Fig. 12. A classification accuracy of 46.51 % was obtained over the 10000 unprocessed test samples of the CIFAR-10 dataset.

4.5 Throughput and Energy Analysis

A metric for throughput performance, used loosely in our work, is the amount of *frames-per-second* (FPS) we can process, where a *frame* represents a sample from the dataset either

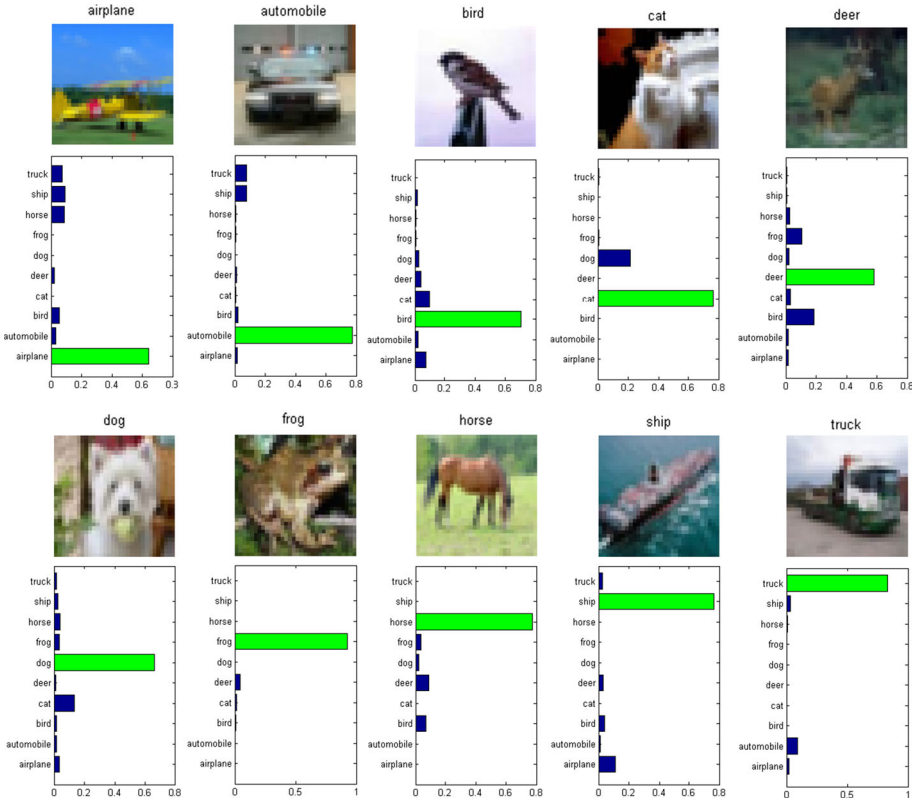

Fig. 10 Some of the images correctly classified (from CIFAR-10)

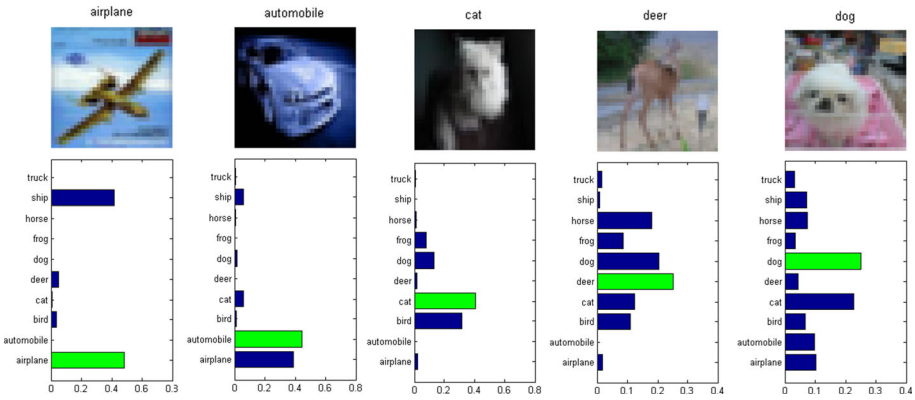

Fig. 11 Images that were close to being misclassified (from CIFAR-10)

being trained with, or classified. Since our goal is to produce a solution for robotics and other low-power applications (used for instance in computer vision), the achievable value of FPS is important to a possible application where a live camera feed replaces the dataset samples as the network's input. Concluding, we use this metric as a reference to the ability of our implementation to cope with real-time object classification.

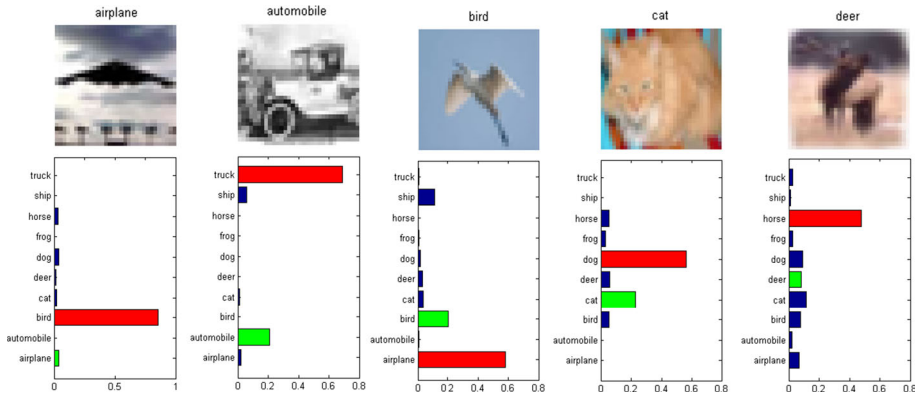

Fig. 12 A collection of misclassified images (from CIFAR-10)

Table 4 Running time and throughput performance while training the first AE with a batch size of 64 images

Platform	Feed forward (s)	Back propagation (s)	Epoch total (s)	Training throughput (FPS)*
GPU	0.15	1.64	1.79	36
mGPU	1.27	5.43	6.70	10
FPGA	3.08	13.79	16.87	4

* Higher is better

Table 5 Running time and throughput performance during the classification of a batch of 64 images

Platform	First AE (ms)	Second AE (ms)	Classification total (ms)	Classification throughput (FPS)*
GPU	79	21	100	640
mGPU	459	81	540	119
FPGA	1170	248	1418	45

* Higher is better

The training results for the first AE can be observed in Table 4. The first AE was used as comparison for these measurements considering it is the largest and most computationally demanding part of the SAE. This is in fact due to the nature of our SAE, reducing in size as the network deepens.

After the training process, the SAE is ready to classify the provided test samples. The decoder's feed forward and all back propagation is now withdrawn from the computation, leaving the network with only the encoder from each AE. From such reduced computation we can obtain a measurement of classification throughput, i.e., how many images we can classify in a second, as shown in Table 5.

For the power consumption analysis, we first measured the average static consumption of the entire system (Host + Device) and then launched the application, measuring the dynamic average power (Load − Idle), over the SAE training time. The results are shown in Table 6.

Table 6 Total SAE training time and energy consumption

Platform	Total training time	Average power (W)	Energy consumption (kWh)*
GPU	3h14m20s	247	0.800
mGPU	17h08m26s	6.6	0.113
FPGA	45h31m03s	16.0	0.728

* Lower is better

Table 7 Throughput per power ratio for all computing platforms

Platform	Training (FPS/W)	Classification (FPS/W)*
GPU	0.14	2.59
mGPU	1.45	18.03
FPGA	0.24	2.81

* Higher is better

By combining throughput performance and average power we were able to measure throughput per power ratio, which shows a metric for energetic efficiency of these systems as depicted in Table 7.

5 Conclusions

In this paper we show for the first time, to the best of our knowledge, the training phase of a deep neural network, a stacked autoencoder (SAE), performed directly on low-power devices, namely an FPGA and a mobile GPU. Although the time necessary to complete the training process in these devices is extensive, the overall energy consumption is lower than the traditional desktop GPU. With a training phase 3× quicker compared to the FPGA, the mobile GPU still manages to have a total energy consumption 6.4× lower than the FPGA, and 7.1× lower than its desktop counterpart. Since the average power during training remains low in both mobile GPU and FPGA, the utilization of these solutions in low-power constrained scenarios is thus shown adequate by this work.

As for the classification phase, since our efforts were towards a SAE implementation applicable in low-power devices, our accuracy of 46.51 % remains below the current state-of-the art. With this sub-optimal approach based on the SAE, we have achieved a throughput capable of real-time classification on both low-power platforms, with 45 FPS on the FPGA and 119 FPS on the mobile GPU, even though somewhat far from the 640 FPS of the desktop GPU. Regarding the mobile GPU, a future implementation can be linked to the platform's camera using an Android interface, providing the capture and classification of images in real-time for a myriad of applications. The purchase cost remains a major drawback from FPGAs and makes the usage of the more affordable and readily available mobile GPU a valid alternative.

The mobile GPU and FPGA are then in a class of low-power devices that allow computationally demanding algorithms to be performed directly on autonomous vehicles, robots and other low-power demanding applications. As technology progresses and more powerful FPGAs and mobile GPUs with more hardware resources are developed, we aim at creating state-of-the-art networks, such as convolutional neural networks (CNNs), running entirely on those devices and achieving top results in both energy savings and classification accuracy.

Acknowledgments This work has been supported by Instituto de Telecomunicações and Fundação para a Ciência e a Tecnologia (FCT) under grant UID/EEA/50008/2013.

References

1. Altera: Stratix V FPGA Overview. http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html
2. Altera: Altera SDK for OpenCL: Optimization Guide (2013) http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
3. Altera: design tools: VHDL (2013) http://www.altera.com/support/examples/vhdl/vhdl.html
4. Altera: SDK for OpenCL (2014) http://www.altera.com/products/software/opencl/opencl-index.html
5. Bengio Y (2009) Learning deep architectures for ai. Foundations and trends®. Mach Learn 2(1):1–127
6. Bengio Y, Lamblin P, Popovici D, Larochelle H et al (2007) Greedy layer-wise training of deep networks. Adv Neural Inf Process Syst 19:153
7. Ciresan DC, Meier U, Schmidhuber J (2012) Multi-column deep neural networks for image classification. In: IEEE conference on computer vision and pattern recognition (CVPR), Providence, pp 3642–3649
8. Dundar A, Jin J, Gokhale V, Martini B, Culurciello E (2013) Accelerating deep neural networks on mobile processor with embedded programmable logic. In: Neural information processing systems conference (NIPS)
9. Falcao G, Silva V, Sousa L, Andrade J (2012) Portable ldpc decoding on multicores using opencl (applications corner). IEE Signal Process Mag 29(4):81–109. doi:10.1109/MSP.2012.2192212
10. Farabet C, Martini B, Akselrod P, Talay S, LeCun Y, Culurciello E (2010) Hardware accelerated convolutional neural networks for synthetic vision systems. In: IEEE international symposium on circuits and systems (ISCAS), pp 257–260
11. Gong Y, Jia Y, Leung T, Toshev A, Ioffe S (2013) Deep convolutional ranking for multilabel image annotation. CoRR abs/1312.4894
12. Goodfellow IJ, Bulatov Y, Ibarz J, Arnoud S, Shet V (2013) Multi-digit number recognition from street view imagery using deep convolutional neural networks. CoRR abs/1312.6082
13. Group K (2012) The OpenCL specification Version 1.2. https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
14. Hardavellas N (2011) The rise and fall of dark silicon. USENIX 37(2):7–17
15. Hinton G, Deng L, Dahl GE, Mohamed A, Jaitly N, Senior A, Vanhoucke V, Nguyen P, Sainath T, Kingsbury B (2012) Deep neural networks for acoustic modeling in speech recognition. IEEE Signal Process Mag 29(6):82–97
16. Kaggle: Public Leaderboard: CIFAR-10: Object Recognition in Images (2013) http://www.kaggle.com/c/cifar-10/leaderboard
17. Krizhevsky A (2009) Learning multiple layers of features from tiny images. Tech. rep
18. Krizhevsky A, Nair V, Hinton G CIFAR-10 Dataset. http://www.cs.toronto.edu/kriz/cifar.html
19. Krizhevsky A, Sutskever I, Hinton G (2012) ImageNet classification with deep convolutional neural networks. In: Advances in neural information processing systems 25 (NIPS'2012)
20. Nallantech: PCIe-385N - Altera Stratix V D5 (2012) http://www.nallatech.com/PCI-Express-FPGA-Cards/pcie-385n-altera-stratix-v-fpga-computing-card.html
21. Oh KS, Jung K (2004) Gpu implementation of neural networks. Pattern Recognit 37(6):1311–1314
22. Owaida M, Falcao G, Andrade J, Antonopoulos C, Bellas N, Purnaprajna M, Novo D, Karakonstantis G, Burg A, Lenne P (2015) Enhancing design space exploration by extending CPU/GPU specifications onto FPGAs. ACM Trans Embed Comput Syst 14(2):33
23. Qualcomm: Snapdragon 800 (2013). http://www.qualcomm.com/snapdragon/processors/800
24. Qualcomm: Snapdragon 800 DragonBoard (2013). http://mydragonboard.org/db8074/
25. Wang G, Xiong Y, Yun J, Cavallaro JR (2013) Accelerating computer vision algorithms using opencl framework on the mobile GPU: a case study. In: IEEE international conference on acoustics, speech and signal processing (ICASSP)