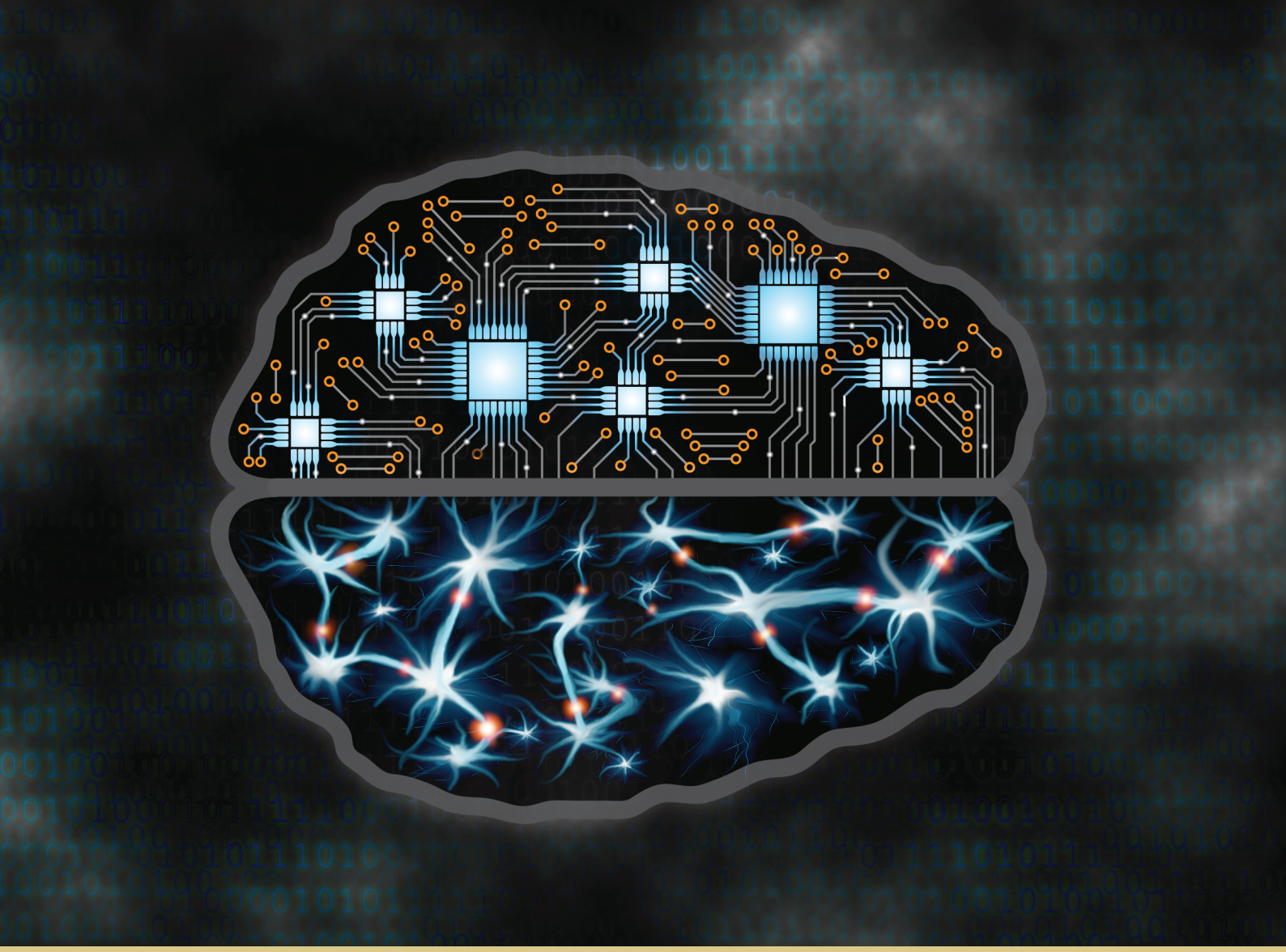

João António Batista Maria

Accelerating the Processing of Neural Networks for Object Recognition Using Low-power Devices

Dissertação de Mestrado em
Engenharia Electrotécnica e de Computadores

Setembro de 2014

· U C ·

Universidade de Coimbra

Aceleração do Funcionamento de Redes Neuronais para Reconhecimento de Objectos em Plataformas de Baixa Potência

João António Batista Maria

Dissertação para obtenção do Grau de Mestre em

Engenharia Electrotécnica e de Computadores

Orientador: Doutor Gabriel Falcão Paiva Fernandes
Co-Orientador: Doutor Luís Filipe Barbosa de Almeida Alexandre

Júri

Presidente: Doutor Vítor Manuel Mendes da Silva
Orientador: Doutor Gabriel Falcão Paiva Fernandes
Vogais: Doutor Fernando Manuel dos Santos Perdigão

Setembro de 2014

Agradecimentos

A vontade de avançar para uma dissertação de mestrado com o Professor Gabriel Falcão veio desde o 1º ano deste curso. Foi aí que primeiro tive contacto com a sua capacidade de educar, de abrir portas à nossa imaginação e nos motivar para darmos sempre o nosso melhor. Ao longo de todo este percurso académico, e em especial com a dissertação de mestrado, beneficiei do seu apoio e orientação, sem nunca impor barreiras de tempo ou paciência. Esteve sempre a meu lado, a acompanhar-me a cada passo e a dar-me sempre metas mais além, num constante desafio que me motivou mais e mais, e que tanto me fez crescer como aluno, como também como pessoa. Por tudo isto lhe estou eternamente grato, foi um orientador de mestrado e da vida.

Apesar da distância que nos separou, todo este trabalho seria impensável sem a experiência e apoio do Professor Luís Alexandre. A ele tenho a agradecer a sua disponibilidade para ensinar, arranjando forma de ultrapassar os obstáculos com que nos fomos deparando, sempre com paciência e um grande positivismo ao longo de todo o acompanhamento que me deu, numa área que me era desconhecida. Quero também agradecer ao Xavier Frazão, pelas bases de onde parti para a minha dissertação, e que estarão sempre ligadas a este projeto.

Agradeço à minha família, em especial aos meus pais, avós e sogros, por me terem suportado ao longo de todos estes anos académicos, por nunca terem duvidado de mim, por me terem apoiado em todas as minhas decisões. Graças a eles tive assim a oportunidade de explorar todo o meu potencial, com uma base de tranquilidade e apoio interminável.

Não há palavras que descrevam o incomensurável apoio da minha namorada, Nádia, ao longo de todos estes anos. Desde me motivar quando as coisas não corriam tão bem, até me tratar do jantar e não me deixar morrer à fome, devo-lhe tudo. O seu amor, carinho e entusiasmo, deu-me energia para vencer este desafio, para todos os dias dar o melhor de mim. A ti te agradeço por estes anos, mas acima de tudo por acreditares em mim mais do que eu mesmo.

Por fim, mas de todo não por último, agradeço também a todos os meus amigos e colegas. Graças à vossa amizade e apoio, estes anos passaram num instante, oferecendo memórias inesquecíveis. Foram muitas as noitadas de trabalho e diversão que tivemos juntos, cresci com vocês, e com vocês partilhei os anos mais importantes da minha vida. Obrigado por tudo.

Abstract

Over the last years, deep learning architectures have gained attention by winning some of the most important international detection and classification competitions, but this comes at a cost: these models are computationally expensive and have been recently ported to Graphics Processing Units (GPUs) to allow faster deployment. However, desktop GPUs have their own shortcomings and seem to be quickly approaching the limits of power and heat dissipation walls, imposing high levels of energy consumption. This implies high deployment costs in applications that process big data volumes on a permanent basis, and also the inability to use these architectures, for example, in autonomous systems such as vehicles and robots, which can hardly provide low power supplies. Therefore, this thesis proposes another shift of paradigm, this time from GPU-based deep learning approaches to an Field-Programmable Gate Array (FPGA)-based context. We show how to implement a particular type of deep learning architecture, the Stacked Autoencoder (SAE), and compare both accuracy and energy consumption levels achieved against similar implementations both on desktop and mobile GPUs. The results show that similar classification and error performances can be obtained using the SAE proposed solution, with paid dividends in energy savings. Also important is the fact that the proposed SAE architecture is scalable, and FPGAs and mobile GPUs have probably better progress margin than desktop GPUs. These results also pave the way for adopting low-power devices in energy-constrained applications for big data classification.

Keywords

Deep Learning, Neural Networks, Stacked Autoencoder, Parallel Computing, FPGAs, GPUs, OpenCL

Resumo

Ao longo dos últimos anos, as arquitecturas de *deep learning* têm vindo a ganhar destaque ao vencerem algumas das mais importantes competições internacionais de detecção e classificação, mas isso tem um preço: estes modelos são computacionalmente exigentes e foram recentemente portados para execução em placas gráficas de modo a permitir o seu rápido desenvolvimento. Contudo, as placas gráficas de *desktop* têm as suas limitações e parecem estar rapidamente a aproximar-se dos limites de potência e de barreiras de dissipação de calor suportáveis, com elevados níveis de consumo energético. Isto implica elevados custos de desenvolvimento em aplicações que processem um elevado volume de dados em regime permanente, assim como a incapacidade para usar estas arquitecturas, por exemplo, em sistemas autónomos como veículos e *robots*, que dificilmente conseguem fornecer uma fonte de baixa potência com autonomia. Assim, esta tese propõe uma mudança de paradigma, de uma abordagem a *deep learning* baseada em placas gráficas para um contexto baseado em *Field-Programmable Gate Arrays (FPGAs)*. Mostramos como implementar uma arquitectura de *deep learning* em particular, o *Stacked Autoencoder (SAE)*, e uma comparação tanto ao nível da precisão como de consumo energético obtidos face a implementações similares em placas gráficas de *desktop* e de plataformas móveis. Os resultados mostram que desempenhos semelhantes na classificação e erro podem ser atingidos usando a solução de SAE proposta, com dividendos pagos em poupança energética. Também é importante o facto de que a arquitectura de SAE proposta é escalável, e que as FPGAs e placas gráficas móveis têm provavelmente uma maior margem de progresso do que as placas gráficas de *desktop*. Estes resultados abrem também o caminho à adopção de dispositivos de baixa potência para classificação de grandes volumes de dados, em aplicações com elevadas restrições energéticas.

Palavras Chave

Deep Learning, Redes Neuronais, Stacked Autoencoder, Computação Paralela, FPGAs, GPUs, OpenCL

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AE	Autoencoder
ACE	Asynchronous Compute Engine
AMD	Advanced Micro Devices
ANN	Artificial Neural Network
API	Application Programming Interface
ARM	Advanced RISC Machine
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CU	Compute Unit
CUDA	Compute Unified Device Architecture
DBN	Deep Belief Network
DDR	Double-Data Rate
DNN	Deep Neural Network
DSP	Digital Signal Processing
FPGA	Field-Programmable Gate Array
FPS	Frames-per-second
GCN	Graphics Core Next
GDDR	Graphics Double-Data Rate Random Access Memory
GPC	Graphics Processing Cluster

GPU	Graphics Processing Unit
HLS	High-level Synthesis
LP	Low-Power
MC	Memory Controller
MLP	Multi-Layer Perceptron
MNIST	Mixed National Institute of Standards and Technology
NN	Neural Network
OpenCL	Open Computing Language
PCIe	Peripheral Component Interconnect Express
RBM	Restricted Boltzmann Machine
RTL	Register Transfer Level
SAE	Stacked Autoencoder
SDAE	Stacked Denoising Autoencoder
SDK	Software Development Kit
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SMX	Streaming Multiprocessor
SoC	System On Chip
SP	Streaming Processor
SVHN	Street View House Numbers

1

Introduction

Contents

Associated to the processing of increasingly larger amounts of (big) data, machine learning and perception models aim at solving more complex and challenging tasks with lower classification errors. The number of samples used to train the algorithms now surpasses the hundreds of thousands, which poses severe constraints regarding the time and processing power necessary to train the networks.

Recently, deep learning architectures have gained some momentum because they have shown superior performance in some of the most important international image, sound / voice detection and classification competitions [10–12]. These typically deal with the automatic recognition of objects in images, whether these objects are in effect traffic signs, digits, objects or animals, and have been won by research teams exploiting deep neural networks of the convolutional type [10]. The current trend in machine learning / perception presently exploits the use of multiple representation levels, which can be achieved using deep belief networks, Stacked Denoising Autoencoder (SDAE) or Convolutional Neural Networks (CNNs), among others.

However, such current state-of-the-art implementations are known to consume high energy levels in order to produce the expected results, which directly impacts the processing costs of big data and also creates constraints in their utilization in autonomous vehicles / robots. Moreover, some of the powerful parallel computing devices under utilization, namely Graphics Processing Units (GPUs), are reaching power- and heat-dissipation walls [13] (also known as utilization wall). Therefore, low power architectures and corresponding energy-saving strategies are required at this point of neural networks development.

1.1 Motivation

In this thesis we propose Stacked Autoencoder (SAE) architectures for reconfigurable Field-Programmable Gate Array (FPGA) substrates, as a first step towards the implementation of more complex approaches to deep learning, such as CNNs. Even though modern FPGAs support a high number of hardware resources, the proposed approach of investigating a simpler Neural Network (NN) is justified by the fact that implementing, for example, a CNN with the complexity of the largely adopted framework [14], would require an FPGA with orders of magnitude more resources than those provided by current state-of-the-art devices.

We propose to lower the N-dimensionality representation of the problem and associated computational complexity of the parallel architecture developed, allowing for sub-optimal results albeit making it more tractable and thus able to cope with the existing available hardware resources of modern FPGAs. These low-power FPGA architectures

consume at least one order of magnitude less energy and are still able to provide real-time throughput and competitive classification error performance, when compared to existing clusters of other high-performance computational resources such as GPUs or Central Processing Units (CPUs). The objective is to conciliate the quality of object recognition with faster or even real-time execution capabilities at low-energy consumption budgets.

The main problems identified are the limited hardware resources available in recent FPGAs to support this type of NN-based algorithms; the bandwidth bottleneck to access global memory; and the long development times associated with Register Transfer Level (RTL) design / development. The former problems can be addressed by developing new algorithms based on less complex Autoencoder (AE) networks. The latter can be overcome using new High-level Synthesis (HLS) tools that are very effective for designing and prototyping hardware systems for reconfigurable devices in short periods of time [15].

1.2 Objectives

In this thesis we show for the first time how we can train a type of NN, designated as the SAE, on low-power FPGAs architectures. In particular, we propose:

i) to develop a SAE architecture based on low-power processing using FPGA devices for classifying huge datasets that includes the training phase. For the best of our knowledge, these long training periods have never been processed on these devices before (they are usually processed on the GPU). To exemplify these scenarios we develop solutions for processing the well-known Mixed National Institute of Standards and Technology (MNIST) dataset.

and *ii)* to perform a power performance analysis by comparing the power and energy efficiency of these algorithms in several computing platforms, from desktop and mobile GPUs to FPGAs: we present experiments illustrating not only the accuracy obtained using these SAE architectures, but also the execution times and the respective power and energy consumption savings achieved when processing large amounts of images.

1.3 Main contributions

This thesis proposes new solutions that advance the state-of-the-art of artificial intelligence, computer vision and parallel processing using the compute horse-power capabilities of FPGAs. We provide a scalable and multi-platform solution for training a SAEs-based NN, aimed at detecting objects, characters, or other type of structures in entire city

maps. As the technology in the FPGA progresses and more processing resources are made available, a shift to more robust types of NNs, such as the state-of-the-art CNNs, will be possible. Moreover, we pave the way for new applications in a diversity of areas that can benefit from the accurate real-time recognition of objects with lower consumption budgets. These areas include not only big data processing, as for example the identification and classification of large image data related to the visual information of entire city streets (modern infrastructures like Google need to process and classify such large amounts of data on a daily/permanent basis), but also robotics or autonomous vehicles, which all present severe low-power constraints.

This work resulted in the article "Energy-efficient Deep Learning: Stacked Autoencoders on FPGAs and Mobile GPUs", submitted to the ACM Transactions on Architecture and Code Optimization journal and in "Low-power Accelerated Architectures using Stacked Autoenconders for Object Recognition in Autonomous Systems", submitted to Neural Processing Letters - Special Issue on Neural Networks for Vision and Robotics. Both these articles are available in the Appendixes B and C, respectively.

1.4 Dissertation outline

This thesis is structured in seven chapters. After this brief introduction, the principles of Artificial Neural Networks (ANNs) and their relation to the biological model will be depicted in Chapter 2, followed by a detailed look over the SAE selected for our work. In Chapter 3 we describe the hardware architecture of the desktop and mobile GPUs and FPGA platforms. Chapter 4 explains the Open Computing Language (OpenCL) programming framework and the work developed to achieve efficient NN parallelism. Through Chapter 5 we explain the experimental metrics and the specific test systems and OpenCL devices used in our work. Regarding those metrics, in Chapter 6 we detail and analyze our experimental results, evaluating the network and the test platforms in question. Finally, in Chapter 7, we draw final conclusions of our work and discuss future improvements and goals.

2

Deep Learning using Neural Networks

Contents

The decisions we make in everyday life are based on personal experience, with knowledge acquired over the years, shaped by our personal view of the world. An immense amount of information helps us making those decisions. Things we saw, heard, smelled or touched, produce a weighted effect, coming together to help us forming an opinion.

In modern neuroscience, scientists from several fields have been studying the brain, formed by neural interconnections and responses, with the neuron acting as the basic unit of the nervous system as described in the early 20th century [16]. With visualization possible under optical microscopy, we now have a deeper understanding of the structure and operation of a neuron.

2.1 Neuron

An abstract model of the biological neuron is comprised of a *soma* or neuron cell body, several *dendrites* that extend from the soma, and a single *axon*, a structure similar to dendrites but longer (as far as 1 meter in humans) [17]. A neuron axon can connect to dendrites in other neurons, communicating through electrical signals from *synapses*, hence forming a Neural Network (NN). A neuron is activated when a strong signal is received by the dendrite through the synapse, propagating it through the axon and potentially activating another neuron in the network [18]. An overview of the structure and connections of a neuron can be seen in Fig. 2.1.

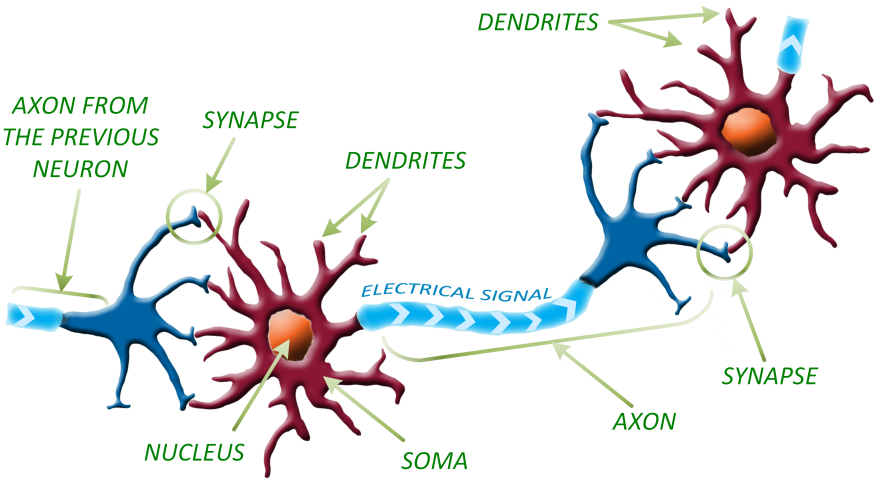

Figure 2.1: A schematic of the biological neuron

2.2 Artificial Neural Networks

The operation and connections of a neuron can be logically modeled [19], thus producing an artificial neuron or combined together in an Artificial Neural Network (ANN). The model in Fig. 2.2 relates to the abstract biological model, containing a set of *inputs* as synapses, multiplied by the *weights* as the strength of the electrical signals, and combined in an weighted *sum* that goes to an *activation function*, representative of the threshold for activating the neuron. The computed *output* from the activation function relates to the signal propagating to the axon in the biological model.

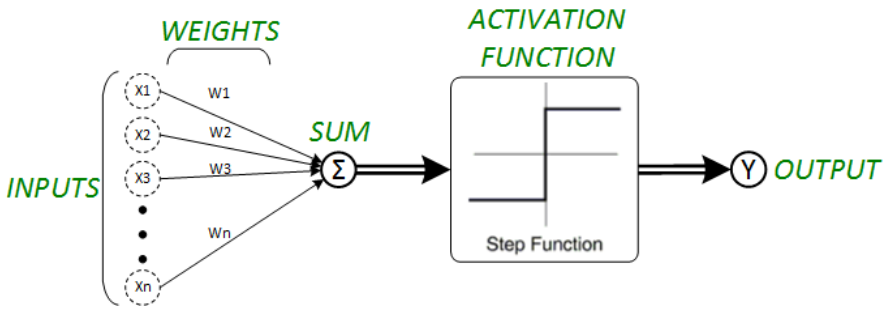

Figure 2.2: A logical approach to the neuron: the artificial neuron

As before, the equal mathematical model is defined by the inputs X_i, the weights W_i and an activation function f (a step function in the original work [19]) resulting in the expression

$$Y = f(\sum_{i=1}^{n} W_i X_i) \tag{2.1}$$

where Y is the neuron output and n represents the number of input signals.

The weights value directly affects the output of the neuron given a said input. There is the possibility of changing these weights and thus obtain the desired output response, in a step by step process called learning or training.

2.2.1 Perceptron

The first report on ANN training detailed a binary classifier, the perceptron, intended at mapping an input vector of real values to a binary output via a step activation function [20].

$$Y = f(b + \sum_{i=1}^{n} W_i x_i) = \begin{cases} 1 & \text{if } b + \sum_{i=1}^{n} W_i x_i > 0 \\ 0 & otherwise \end{cases} \tag{2.2}$$

A bias b was introduced to provide a shift in the activation of the network output and more appropriately adjust the position of the decision boundary. The initial training algorithm does not distinguish input samples (vectors of input values) that are not linearly

separable, causing some of them to be impossible to classify correctly as we can observe in Fig. 2.3. In a) and a') we can see a logical `AND` output correctly classified into two hyperplanes. The same happens in b) for the logical `OR` output, but for the logical `XOR` in c) a valid decision boundary cannot be set, always leaving at least one value out of the correct hyperplane.

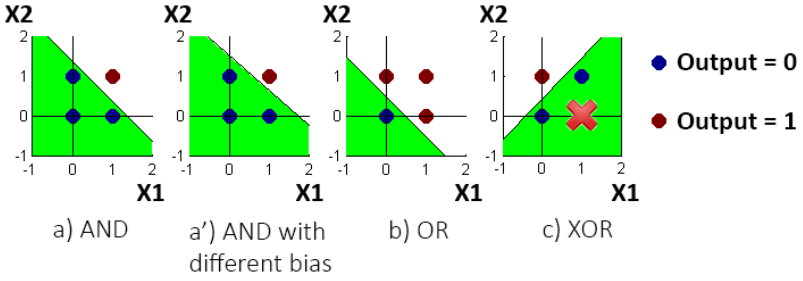

Figure 2.3: A collection of logical operations responses, with associated decision boundaries and hyperplanes (green for *0*, white for *1*)

2.2.2 Multi-Layer Perceptron

To solve the issue with the basic perceptron and its inability to correctly distinguish values that are not linearly separable, a new proposal was made in the form of the Multi-Layer Perceptron (MLP) [21]. The premise was that an added layer, the hidden layer between the input and output layers, could then perform a non-linear transformation via a sigmoidal function and thus obtain a linearly separable output space [22]. A new learning algorithm was introduced, the error back propagation, computed after the data feeds forward through the network. A model of one of these MLPs, along with the direction of the feed forward and back propagation training phases, is depicted in Fig. 2.4.

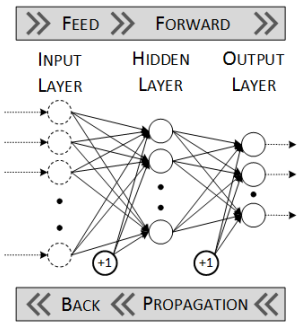

Figure 2.4: A graphical representation of a Multi-layer Perceptron

The output signal is compared to the desired output and an error value and gradient is calculated for each of the output nodes. The error is then back propagated through the network, layer by layer, with the layer weights updated using a gradient descent algorithm.

After the back propagation is concluded, a training iteration or epoch is over, and the process can start again from the feed forward, for as many epochs as needed until an ideal output error is achieved.

The use of more than two hidden layers in neural network supervised learning in what is called a Deep Neural Network (DNN), was seen as unnecessary until recently, given the proofs of the approximation capabilities of one [22] and two [23] hidden layer neural networks.

The exceptions to this rule were the neocognitron [24], which used several layers to emulate the human visual system, and Convolutional Neural Networks (CNNs) [25], both developed mostly for visual tasks.

Apart from the apparent unnecessary use of more than two layers, the other main issue with using deep networks was the difficulty that appeared when trying to train several hidden layers using standard back propagation: there were problems adjusting the weights as depth increased (vanishing gradients).

2.3 State-of-the-Art: The Convolutional Neural Network

The efforts by Hinton and co-workers [26,27], resulted in the ability to train DNNs, in this case, Deep Belief Networks (DBNs) which took advantage of Boltzmann machines in a variant called Restricted Boltzmann Machines (RBMs). At the same time, other groups proposed a way to train deep networks based on stacking autoencoders [28,29].

From 2006 until today, the field of DNNs has received much attention. The potential advantages that come from using DNNs are the possibility of having increasingly more abstract levels of representation, the possibility of reusing the intermediate level representations across different tasks and also to obtain a more compact and efficient representation for certain types of problems [30].

A novel form of DNN called CNN was then introduced, consisting in the simulation of how the brain's visual system works (or at least how we currently believe its first regions work). The CNN presented in [10] is in fact based on a combination of several individual CNNs, each applied to the same or different inputs and with the corresponding outputs combined and averaged.

This particular CNN presented the best results, by the time it was published, in 7 different datasets normally used for benchmarking similar algorithms, with improvements ranging from 30% to 80% with respect to previously best published results [10]. The main drawback of these approaches (CNNs) is their computational cost. These deep learning approaches use Graphics Processing Unit (GPU) clusters to cope with these large datasets [31]. But even using this type of powerful parallel computing engines, experi-

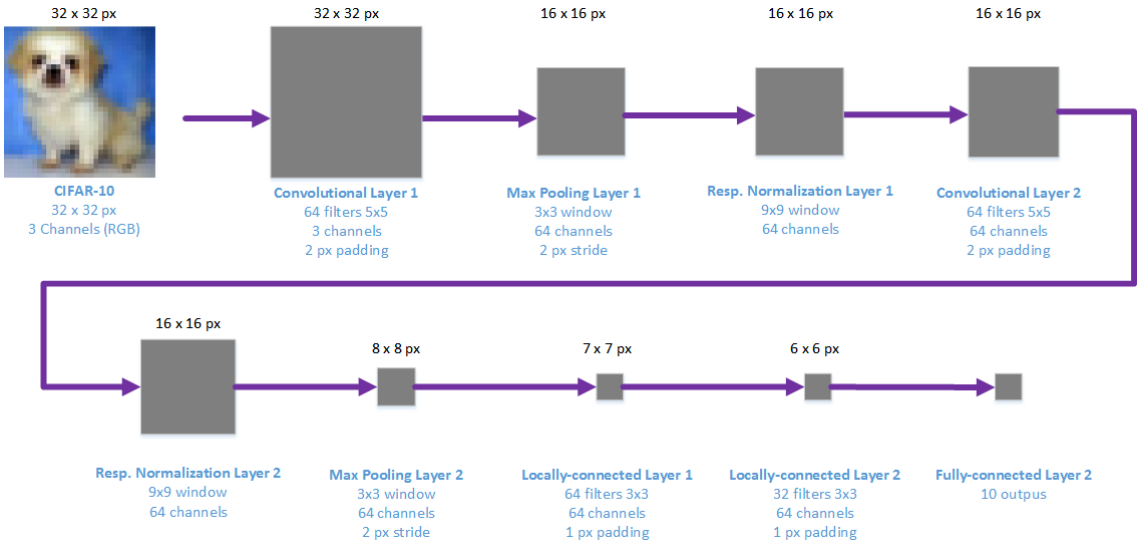

Figure 2.5: Topology of the CNN for the CIFAR-10 dataset [1]

ments can still take several minutes or even hours to execute [10]. As an example, the LeNet convolutional neural network, working with the Mixed National Institute of Standards and Technology (MNIST) dataset takes 380min on a Central Processing Unit (CPU) (Core i7-2600K CPU at 3.40GHz) and 32min on a GPU (GeForce GTX 480) to run a single experiment (including training and testing) [32].

2.4 Sub-optimal Neural Networks

2.4.1 Autoencoder

In this thesis we show the potential of implementing deep learning in Field-Programmable Gate Arrays (FPGAs) by using a Stacked Autoencoder (SAE). An Autoencoder (AE) consists of a simple network that tries to produce at the output what is presented at its input. The most basic AE is in fact an MLP that has one hidden and one output layer, with the following restrictions:

- The weight matrix of the last layer is the transposed of the weight matrix of the hidden layer (clamped weights);
- The number of output neurons is equal to the number of inputs.

Let's represent the input vector by $\mathbf{x}$, the weight matrix by $\mathbf{W}$, the input size by n. The hidden layer neurons output, called the encoding, is obtained with

$$h_j = s(a_j) \tag{2.3}$$

where

$$a_j = b_j + \sum_{i=1}^{n} W_{ij} x_i \tag{2.4}$$

and where b_j is the bias of the hidden layer neuron j and $s(\cdot)$ is the sigmoid function. The output layer values, or the decoding, is given by

$$\hat{x}_j = s(\hat{a}_j) = s\left(c_j + \sum_{i=1}^{n_h} W_{ij}^T h_i\right) \tag{2.5}$$

where c_j is the bias of the output layer neuron j and n_h the number of hidden layer neurons.

A simplified way to write the previous expressions is

$$\mathbf{h} = s(\mathbf{a}) = s(\mathbf{b} + \mathbf{W}\mathbf{x}) \tag{2.6}$$

and

$$\hat{\mathbf{x}} = s(\hat{\mathbf{a}}) = s\left(\mathbf{c} + \mathbf{W}^T \mathbf{h}(\mathbf{x})\right) \tag{2.7}$$

where the sigmoid is applied to each element of its input (vector) argument.

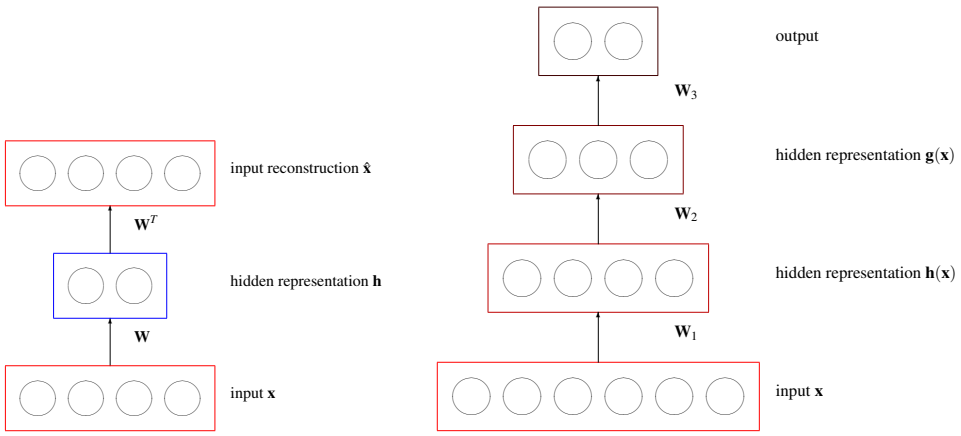

Figure 2.6: Example of an autoencoder (left) and of a stacked autoencoder (right).

Since the goal is to obtain at the output the same thing that is in the input, an adequate cost function should compare these two vectors. The typical approach is to use (real-valued inputs)

$$C(\hat{\mathbf{x}}, \mathbf{x}) = \sum_{i=1}^{n} (\hat{x}_i - x_i)^2 \ . \tag{2.8}$$

For binary inputs, the cross-entropy can be used :

$$C(\hat{\mathbf{x}}, \mathbf{x}) = -\sum_{i=1}^{n} (x_i \log(\hat{x}_i) + (1 - x_i) \log(1 - \hat{x}_i)) \tag{2.9}$$

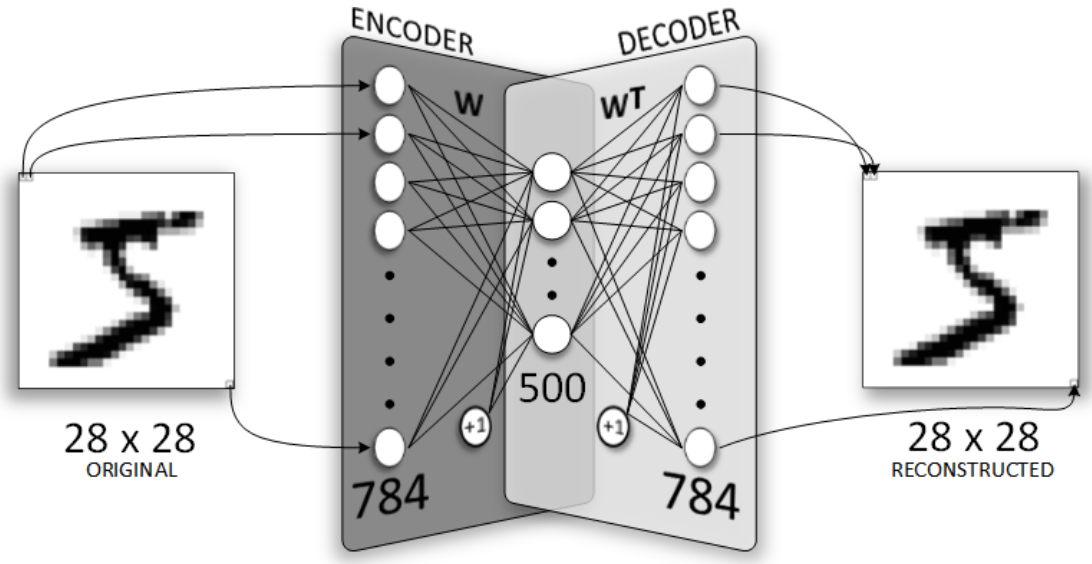

Figure 2.7: Topology of the first autoencoder for the MNIST dataset

The weight changes will be done according to the gradient descent. Since the goal is to obtain in the output the same that is present in the input, an adequate cost function should compare these two vectors. In what follows, the index i runs from $1,\ldots,n_h$ and the index $j = 1,\ldots,n$. For real-valued inputs, we use

$$C(\hat{\mathbf{x}},\mathbf{x}) = \sum_{k=1}^{n} (\hat{x}_k - x_k)^2 \,. \tag{2.10}$$

When the activation function is the sigmoid, we have to update the weights (dropping constants that can be absorbed by η) according to:

$$W_{ij} = W_{ij} - \eta \sum_{k=1}^{n} \left[(\hat{x}_k - x_k)\hat{x}_k(1-\hat{x}_k) \left(h_i + \sum_{z=1}^{n_h} \left[W_{kz}^T h_z (1-h_z) x_j \right] \right) \right]$$

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

$$c_j = c_j - \eta (\hat{x}_j - x_j)\hat{x}_j(1-\hat{x}_j)$$

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

2.4.2 Stacked Autoencoder

The process described so far is used to train a single AE, but a single AE is not deep: it only has depth 2. One possible way to obtain a deeper architecture using AEs is to stack them on top of each other such that the output of one AE is the input for the next. This stacking can produce a deep network: the stacked AE, or SAE.

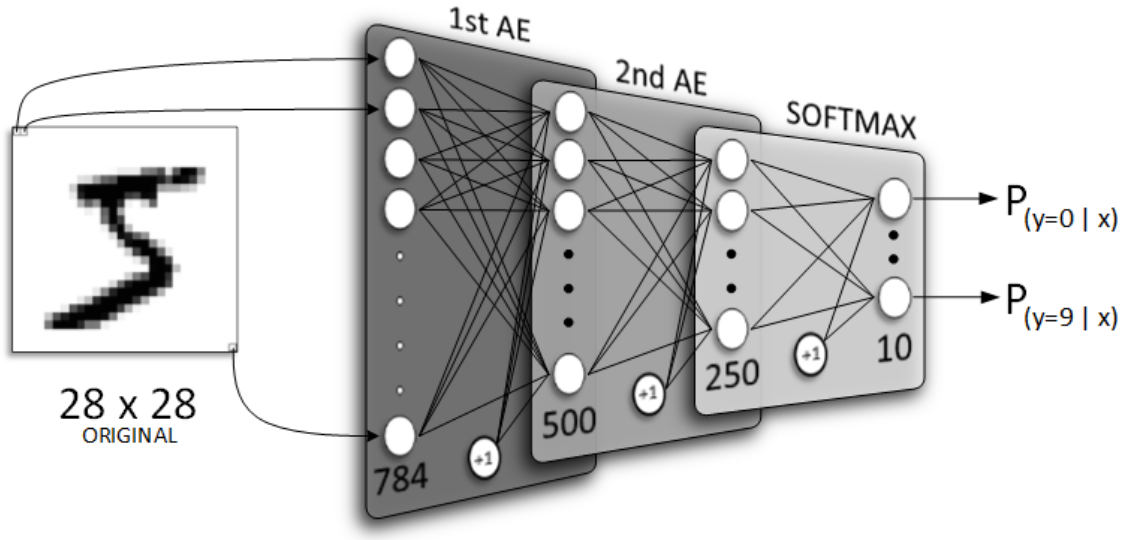

Figure 2.8: Topology of the stacked autoencoder for the MNIST dataset

The SAE is obtained by the following procedure: 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. Then place a final layer of neurons that represent the output layer and will have as many neurons as there are classes in the problem.

The idea is that the pre-training is used to bring the weights of the network near a good starting point for the fine-tuning procedure. Note that the pre-train is an unsupervised training procedure, whereas the fine-tunning that is performed on the complete SAE after the placement of the final layer is a typical supervised learning procedure that takes advantage of the class labels.

In the end, we obtain a deep network that receives input data from the dataset and produces a class label at the output.

2.5 Softmax Classifier

The output layer of the SAE receives as input the last hidden layer representation produced by the AE on top of the stack, say **h**, and produces an output using the usual weighted product between the layer input (**h**) and its weights followed by the application of an activation function $f(\cdot)$.

The activation function used in this output layer can be any of several possibilities. We have chosen to use the softmax. So, for the activation of the output layer neuron i we get:

$$f(a_i) = \frac{e^{a_i}}{\sum_{k=1}^{L} e^{a_k}} \tag{2.11}$$

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 (2.4) but where x_i are replaced by h_i.

2.6 Neural Networks Hyper-parameters

2.6.1 Learning Rate

The learning rate selected at the start of the training phase is a high-impact parameter. If it's set too low, the resulting error convergence takes longer than necessary and the optimal final error may not be achieved, as the error may get 'stuck' on a local minimum. The opposite case, when the learning rate is set too high is also not ideal. With a value too high, as the first stages of training may present a faster error convergence, the error may start to diverge quickly, thus avoiding the targeted minimum. The ideal learning rate should be high enough to provide a rapid error convergence at the start, and then decrease to provide a finer rate and lower error in the end. An example of this problem can be observed in Figs. 2.9 and 2.10.

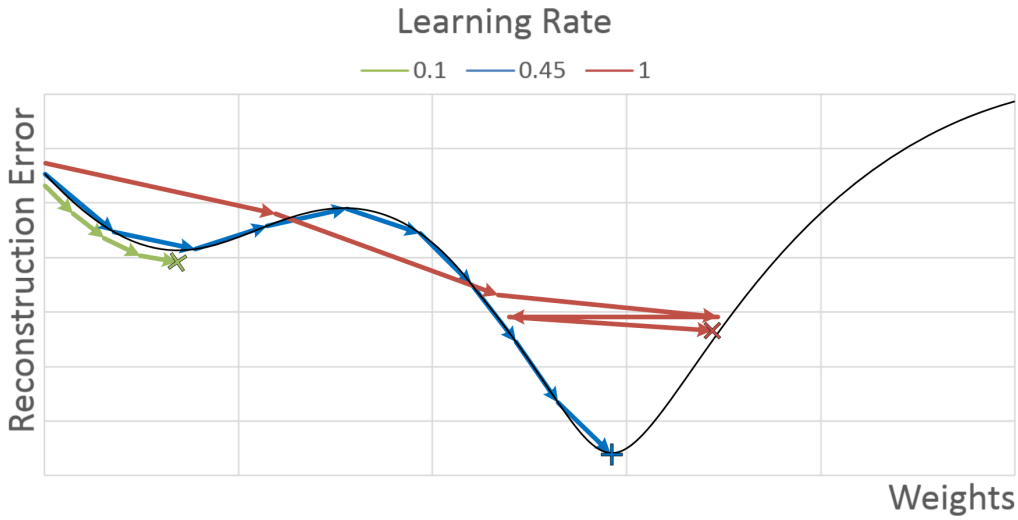

Figure 2.9: Impact of different learning rates. A low (0.1) learning rate gets caught in a local minimum (slow error convergence), an ideal (0.45) learning rate achieves the lowest possible error, and a high (1.0) learning rate goes past the ideal value (error divergence)

2.6.2 Batch Size

During the training process an overall better performance and faster error convergence is obtained if we only present a small batch of images at each time. This problem size

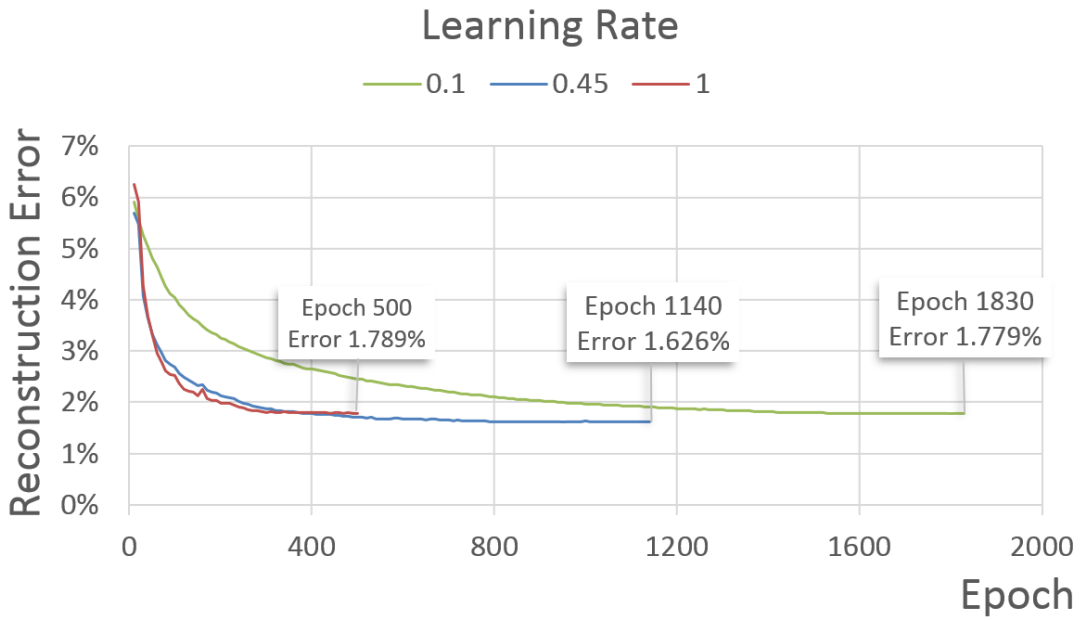

Figure 2.10: Another example of the impact on the final result of a low (0.1), an ideal (0.45) and a high (1.0) learning rate

reduction dismisses the overhead impact of launching too many work-items in the device, at the same time decreasing the memory footprint of the application. Since the error verification and weight updates are computed at a quicker rate, the stopping criteria are achieved faster, thus improving the SAE training performance.

3

Hardware Parallelism with OpenCL Supported Architectures

Contents

3.1 Desktop GPU

With the ubiquitous nature and power of currently available Graphics Processing Units (GPUs), there is high potential for computational acceleration. The trend in current GPU development is to provide the end user with an increased number of cores, benefiting the computational power and thus allowing more demanding applications to be addressed. The single-device GPUs in the gaming-driven market are headlined by the Advanced Micro Devices (AMD) R9 290X with 2816 Streaming Processors (SPs), and the Nvidia GTX Titan with 2668 Compute Unified Device Architecture (CUDA) cores. Their dual-GPU counterparts, the AMD R9 295X2 and Nvidia GTX Titan Z, are set to release with the double of SPs/CUDA cores and available memory.

3.1.1 AMD R9 290X (Hawaii)

The AMD R9 290X GPU codename "Hawaii" was the first to be produced with AMD's own Graphics Core Next (GCN) version 2.0 architecture [33], with new takes on performance, image quality and energy efficiency. GCN 2.0 was designed with general-computing in mind and the increasing popularity of Open Computing Language (OpenCL) was considered during the development process.

The "Hawaii" block diagram in Fig. 3.1 details the integral GPU design [2]. At the top center of the design is the Graphics Command Processor, responsible for receiving the commands and state changes from the device's memory subsystem at the bottom of the diagram, and controlling the general flow of execution.

The memory subsystem is comprised of 1MB L2 Cache with 1TB/s L1/L2 bandwidth and capable of being partitioned into 16x64KB, and eight 64 bit Memory Controllers (MCs) resulting in the 512 bit bus interface to the 4 GB of GDDR5 memory, and 320 GB/s of memory bandwidth. Connecting to the host is the PCIe 3.0 Bus Interface with up to 15.75 GB/s for a 16-lane slot.

The GPU is intended to be a parallel computing platform and for maximum performance and less idle states, the Command Processor is aided by the 8 Asynchronous Compute Engines (ACEs) in this architecture. The ACEs perform in parallel with the Command Processor, independently managing the scheduling and workload dispatchment to each Compute Unit (CU) for processing.

The computing core of the "Hawaii" GPU consists of 4 Shader Engines, each with its own parallel linked Geometry Processors and Rasterizers, but all sharing the same pool of 16x64KB L2 Cache. Each Shader Engine holds 11 CUs with 64 SPs each one, for a total of 44 CUs and a shader count of 2816 SPs. A more detailed view of a CU is depicted in Fig. 3.2.

Figure 3.1: The AMD R9 290X (Hawaii) GPU block diagram [2]

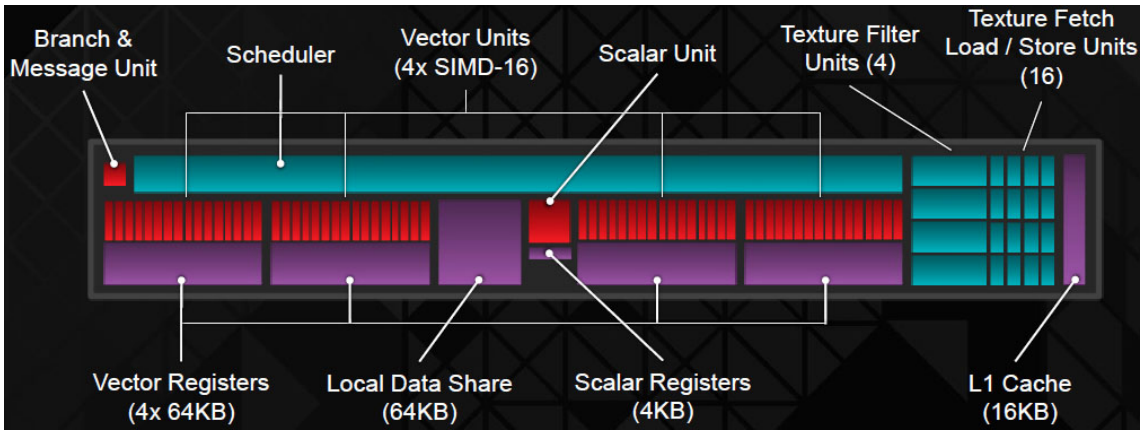

Figure 3.2: A detailed view of the GCN Architecture Compute Unit in the AMD R9 290X (Hawaii) [2]

3.1.2 Nvidia GTX Titan (GK110)

The Nvidia GTX Titan GPU codename "GK110" is produced with Nvidia's Kepler Architecture [3] and, as the "Hawaii" from AMD, was developed with innovative computing technology and features, resulting in an improved parallelism with greater efficiency and performance per Watt. The trend in demanding scientific computing applications was addressed in improvements to Nvidia's own CUDA framework.

The "GK110" block diagram in Fig. 3.3 details the integral GPU design [3]. In the top of the design is the GigaThread Engine, providing fast context switching, concur-

rent kernel execution and managing thread block level scheduling and parallelism. The pipeline information for the GigaThread Engine is fetched from the memory subsystem and through the PCI Express 3.0 Host Interface.

Kepler memory hierarchy provides a 1536KB L2 Cache and six 64-bit MCs resulting in a 384-bit memory interface to the 6GB of GDDR5 memory, and 288.4 GB/s of memory bandwidth.

The computing core contains an array of five Graphics Processing Clusters (GPCs) each with three Streaming Multiprocessor (SMX), for a total of 15 SMXs, albeit with one disabled due to the production yield, usually the SMX furthest from Nvidia's quality control standards. Each of the SMX units feature 192 single-precision CUDA cores, and each core has fully pipelined floating-point and integer arithmetic logic units. In the end we have a total of 2668 CUDA cores throughout the 14 SMXs. A detailed view of a SMX Architecture is depicted in Fig. 3.4.

Figure 3.3: The Nvidia GTX Titan (GK110) GPU block diagram [3]

3.2 Mobile GPU

Over recent years the concern with energy consumption has led to a new direction in development, resulting in more efficient platforms while still meeting the need for higher computational power. Today's smartphones have an increasing demand for performance while dealing with strict power constraints due to constant dependency on battery power. The result are platforms such as Qualcomm's Snapdragon 800 mobile System On Chip

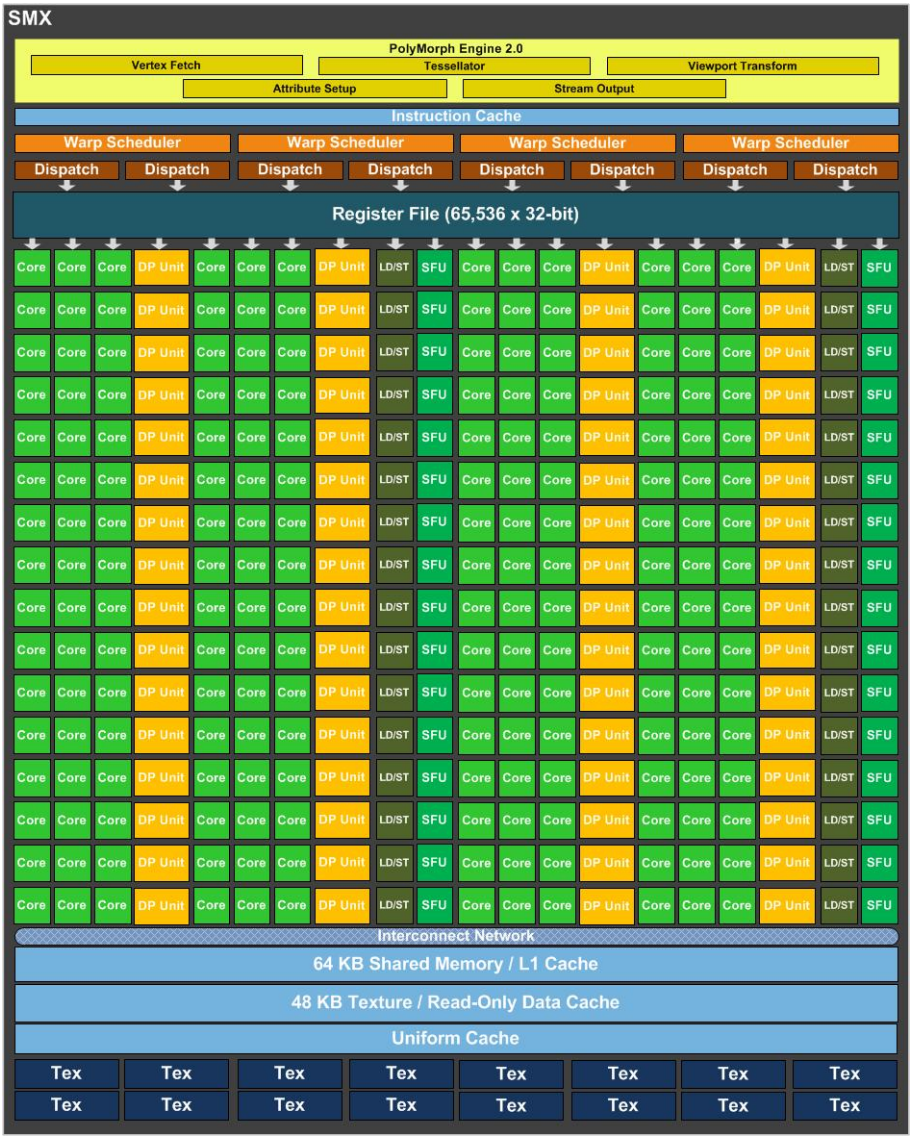

Figure 3.4: A detailed view of the SMX Architecture in the Nvidia GTX Titan (GK110) [3]

(SoC) used in most of the latest smartphones from the leading manufacturers. These platforms are comprised of multiple processing hardware, usually needed in a mobile phone environment, in a single chip. We are specially interested in its Adreno 330 GPU, since it provides OpenCL 1.1 compliant compute capability.

3.2.1 Qualcomm Adreno 330 (Snapdragon 800)

In late 2012, OpenCL 1.1 support was introduced to the Qualcomm Adreno 320 GPU in Snapdragon 600, with further improvements made to the Adreno 330 in Snapdragon 800. The Adreno 330 was launched with Application Programming Interfaces (APIs) designed to expand the use of GPU processing for general computing, offering a 2 times

superior compute performance than Adreno 320 [34].

The Adreno 330 GPU has a unified global memory with the Krait Central Processing Unit (CPU), using the remaining space from the 2GB of LP-DDR3 memory, with up to 12.8 GB/s memory bandwidth [4]. Inside the GPU we also have access to 8KB of local memory and 4KB of constant memory.

The processing core of the Adreno 330 is composed of 4 CUs each with 32 SPs, providing 128 SPs in total.

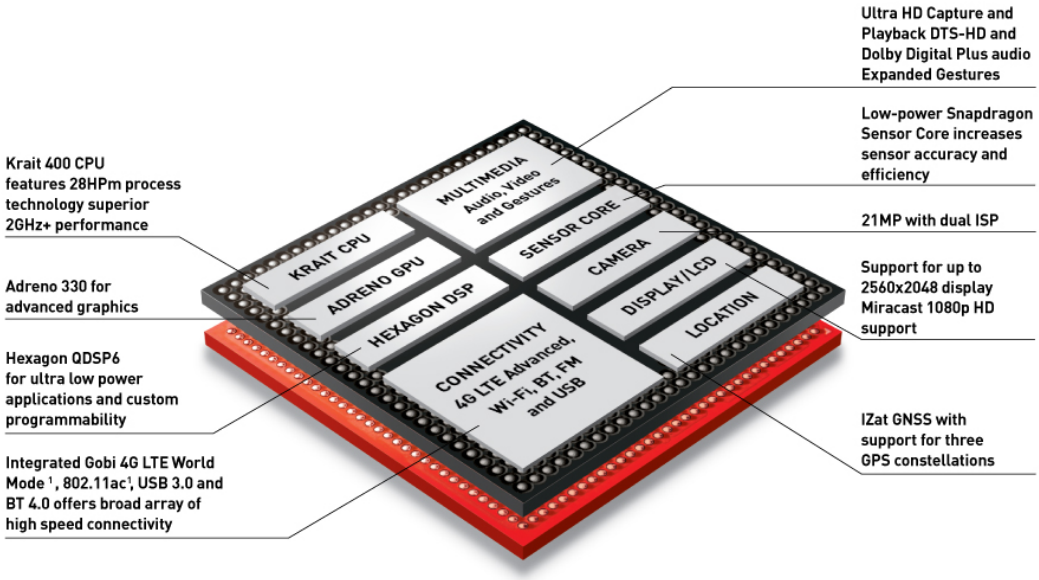

Figure 3.5: The Qualcomm Adreno 330 GPU in the Snapdragon 800 SoC [4]

3.3 FPGA

The Field-Programmable Gate Array (FPGA) market regained some momentum when Altera, one of the leading FPGA manufacturers, released their OpenCL Software Development Kit (SDK) in May 2013 [35]. The FPGA is an inherently parallel architecture and OpenCL provides an easy to use, high-level language, resulting in a perfect match for high computational needs. The difference in energy consumption being 10 times inferior in the FPGA when compared to top GPUs, also seems to encourage the adoption of this type of platforms for some applications. Some selected development kits provide 32GB of memory so there is a clear advantage in this field towards making the move to FPGAs, as implementations regarding image processing can be memory-size bounded. In overview, we have a platform that provides great computational capabilities, with significantly lower power usage and extended memory available, all in a small and portable package ready for robotics and other low-power budget applications.

3.3.1 Altera Stratix V GS D5 (5SGSD5)

One of the current FPGAs from Altera with OpenCL support is the Stratix V GS D5 [36]. 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 690400 registers. The memory interface allows for up to 6 independent banks of Double-Data Rate (DDR)3 Synchronous Dynamic Random Access Memory (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. With these building blocks, the number of possible CUs to which the FPGA can be fine-tuned is application dependent, with variable logic substrate usage.

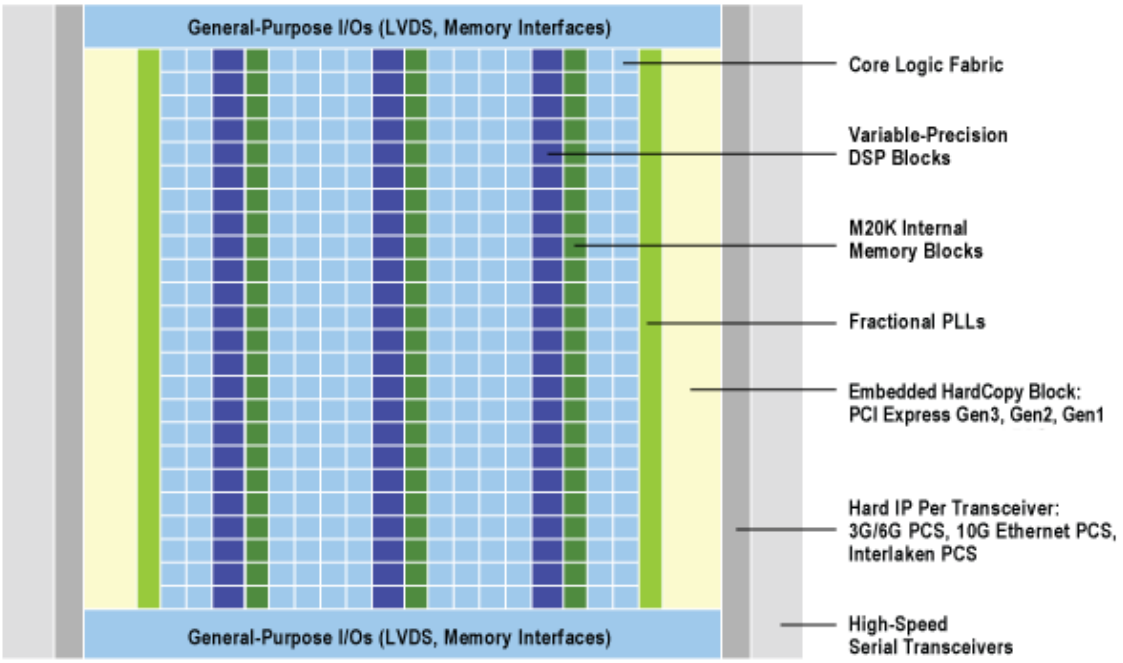

Figure 3.6: Stratix V FPGA architecture and features [5]

4

High-Level Synthesis and OpenCL Structure for Neural Networks

Contents

4.1 The OpenCL Programming Framework

From the currently available parallel computing frameworks for Graphics Processing Unit (GPU) programming, we can choose from a few alternatives, namely Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL). CUDA is bound to Nvidia's hardware but OpenCL is a highly portable framework currently being implemented in Central Processing Units (CPUs), GPUs, Field-Programmable Gate Arrays (FPGAs) and even Android smartphones and tablets. Provided there is a Software Development Kit (SDK) for the desired platform, we are able to port an existing code into the device and achieve fast parallel processing.

The OpenCL framework links a host to one or more OpenCL devices, forming a single heterogenous computational system [37]. 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. A detailed view of this layer is described in Section 4.1.1.

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. A detailed view of the runtime is described in Section 4.1.2.

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 [37].

4.1.1 OpenCL Platform Layer

The OpenCL platforms with their SDKs installed on the host system, such as AMD's or Nvidia's platforms, provide access to the OpenCL devices from those vendors, allowing a detailed query of its capabilities and configuration information. With the device information we can then select one or more available devices to create a context, forming a working environment able to receive a command queue at a latter stage.

The first step consists of initializing the OpenCL Application Programming Interface (API) variables. These include a `cl_int` for storing the API return calls, a `cl_uint` variable to hold the number of available platforms with their SDK installed on the host system, as well as another `cl_uint` for the number of devices in the available platform. The IDs from the available platforms and devices can then be collected to `cl_platform_id` and `cl_device_id`, respectively, with which we can later create a `cl_context` and link the pretended device to the host.

```
1  // [...]
2
3  // OPENCL API VARIABLES
4  cl_int cl_return;
5  cl_uint num_platforms;
6  cl_uint num_devices;
7  cl_platform_id *platforms;
8  cl_device_id *devices;
9  cl_context context;
```

After the OpenCL API variables are initialized, we query the host system with `clGetPlatformIDs` and retrieve the number of installed platforms and their IDs, each representative of one platform.

```
1  // [...]
2
3  // OPENCL PLATFORMS
4  // Retrieve the number of platforms (up to 5)
5  cl_return = clGetPlatformIDs(5, NULL, &num_platforms);
6
7  // Allocate memory space for each of the platforms
8  platforms = new cl_platform_id[num_platforms];
9
10 // Retrieve each of the platforms IDs
11 cl_return = clGetPlatformIDs(num_platforms, platforms, NULL);
```

The platform information allows to use `clGetDeviceIDs` and retrieve the available OpenCL devices and IDs.

```
1  // OPENCL DEVICES
2  // Retrieve the number of devices on platform 0
3  cl_return = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &
       num_devices);
4
5  // Allocate memory space for each of the devices on platform 0
6  devices = new cl_device_id[num_devices];
7
8  // Retrieve the device data from platform 0
9  cl_return = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, num_devices,
       devices, NULL);
```

At this stage we are able to select one or more devices and create a context with `clCreateContext`, so we can manage them.

```
1  // OPENCL CONTEXT
2  // Create a context and link it to device 0 on platform 0
3  context = clCreateContext(NULL, 1, &devices[0], NULL, NULL, &cl_return);
```

After the runtime execution and all the computation is done, we can then clear the allocated resources from memory using `clReleaseContext` and the standard C++ methods.

```
1  // [...]
2
3  // CLEAR OPENCL RESOURCES
4  clReleaseContext(context);
5  delete [] platforms;
6  delete [] devices;
```

4.1.2 OpenCL Runtime

During the OpenCL runtime several API calls are available to manage command queues, memory, program and kernel objects in a context. All these calls allow a direct runtime control over the developed `__kernel` functions in a program. We enqueue necessary commands such as kernel execution, and access to read, write or copy operations on a memory object allocated in the device.

As before in the Platform Layer, several OpenCL API variables are required. A `cl_command_queue` is needed for the kernel program command queue, so the device can perform the requested operation. The kernel source code is compiled and built to a `cl_program`, with each of its kernels created to a `cl_kernel` variable. For the device memory allocation and various operations a `cl_mem` object is needed for each of the memory buffers. The kernel source can also be loaded from an external file with `kernel_source`, or from an external binary as in `kernel_binary`. These `.aocx` files are precompiled device binaries from the Altera SDK.

```
1  // [...]
2
3  // OPENCL API VARIABLES
4  cl_command_queue queue;
5  cl_program program;
6  cl_kernel feedFwd;
7  cl_kernel backProp;
8  cl_kernel backPropHidden;
9  cl_mem cl_weights;
10 cl_mem cl_input;
11 cl_mem cl_output;
12
13 // KERNEL SOURCE FILE
14 #define kernel_source "./kernel.cl"
15 // OR KERNEL BINARY FILE (FROM ALTERA SDK)
16 //#define kernel_binary "./kernel.aocx.cl"
```

With the platform layer correctly configured, we can now launch a command queue with a call with the `clCreateCommandQueue` function. This creates a command queue on a specific device associated with in an existing context.

```
1  // [...]
2
3  // OPENCL COMMAND QUEUE
4  // Create a command queue on the current context, linking to device 0
5  queue = clCreateCommandQueue(context, devices[0], 0, &cl_return);
```

After reading the source code from the file to a buffer with a known size, we can create a program using `clCreateProgramWithSource`, linking it to an existing context. An alternative is to create the program from a precompiled device binary file, as the one produced with the Altera SDK. The next step consists of program compilation and linkage for devices associated with the platform, which is performed with `clBuildProgram`.

```
1  // [...]
2
3  // CREATE THE PROGRAM WITH THE KERNEL SOURCE CODE
4  program = clCreateProgramWithSource(context, 1, (const char**)&source_buffer, &
       source_size, &cl_return);
```

```
5  // OR CREATE THE PROGRAM FROM BINARY (FROM ALTERA SDK)
6  // program = clCreateProgramWithBinary(context, 1, &devices[0], &binary_length,
       (const unsigned char **)&binary, &binary_status, &cl_return);
7
8  // BUILD THE PROGRAM
9  cl_return = clBuildProgram(program, num_devices, devices, NULL, NULL, NULL);
```

Now, from the built program, we can construct the necessary OpenCL functions, or kernels, using `clCreateKernel`.

```
1  // CREATE THE NEEDED KERNELS FROM THE PROGRAM
2  feedFwd = clCreateKernel(program, "feedFwd", &cl_return);
3  backProp = clCreateKernel(program, "backProp", &cl_return);
4  backPropHidden = clCreateKernel(program, "backPropHidden", &cl_return);
```

As the kernels are now created, we can begin the computations on the device. For data on the device to be accessible, a memory buffer with the correct size must be allocated with `clCreateBuffer` and then wrapped to a `cl_mem` memory object.

```
1  // CREATE THE NEEDED BUFFERS FOR THE KERNEL
2  // Create a buffer object to store the weight vector from this autoencoder
3  cl_weights = clCreateBuffer(context, CL_MEM_READ_WRITE, joined_weights_size*
       sizeof(float), NULL, &cl_return);
4
5  // Create a buffer object to store the input data vector from this autoencoder
6  cl_input = clCreateBuffer(context, CL_MEM_READ_WRITE, max_data_size*sizeof(float
       ), NULL, &cl_return);
7
8  // Create a buffer object to store the output data vector from this autoencoder
9  cl_output = clCreateBuffer(context, CL_MEM_READ_WRITE, max_data_size * sizeof(
       float), NULL, &cl_return);
```

With the newly created buffers we can start to transfer the data needed for the devices computations, from the host to the device. This can be achieved by adding a `clEnqueueWriteBuffer` task to the command queue, with a host pointer to the data of the previously set buffer size.

```
1  // ENQUEUE A WRITE TO THE PREVIOUSLY CREATED BUFFERS
2  // Enqueue the weight buffer write in the queue, so it can be transferred to the
       device
3  cl_return = clEnqueueWriteBuffer(queue, cl_weights, CL_FALSE, 0,
       joined_weights_size * sizeof(float), joined_weights, 0, NULL, NULL);
4
5  // Enqueue the input data buffer write in the queue, so it can be transferred to
       the device
6  cl_return = clEnqueueWriteBuffer(queue, cl_input, CL_FALSE, 0, ( hidden_layer->
       getNodes() * batchSize ) * sizeof(float), inputData, 0, NULL, NULL);
```

The memory objects are now allocated and the data is available on the device. We can point the kernel arguments to the correct buffer location in global memory, as well as setting other kernel arguments that will be stored in local memory. This operation can be achieved with `clSetKernelArg`.

```
1  // DEFINE KERNEL ARGUMENTS
2  cl_return = clSetKernelArg(feedFwd, 0, sizeof(cl_mem), (void *)&cl_weights);
3  cl_return |= clSetKernelArg(feedFwd, 1, sizeof(cl_mem), (void *)&cl_input);
4  cl_return |= clSetKernelArg(feedFwd, 2, sizeof(cl_mem), (void *)&cl_output);
5  cl_return |= clSetKernelArg(feedFwd, 3, sizeof(int), (void *)(&hidden_layer->
       getNodes()));
6  cl_return |= clSetKernelArg(feedFwd, 4, sizeof(int), (void *)(&hidden_layer->
       getFeatures()));
7  cl_return |= clSetKernelArg(feedFwd, 5, sizeof(int), (void *)(&offset));
```

We are now ready to launch the kernel execution on the device with `clEnqueueNDRangeKernel`, requesting a data partitioning based on the developed kernel and the data we are about to process. We can distribute the processing load across up to 3 dimensions, with a detailed control of the partitioning inside each dimension with the global and local number of work-items. A work-item is one singular implementation or thread of the total workload.

In our particular case with the Stacked Autoencoder (SAE), the input of the next layer is the output of the current layer. The `clFinish` call provides a synchronization point that awaits the return from every operation in the queue, thus only advancing when all the work-items successfully computed.

```
1  // EXECUTE KERNEL
2  size_t global_2D[2];
3  size_t local_2D[2]={1, min_multi};
4  // Number of nodes in the current layer
5  global_2D[0]= round_up(hidden_layer->getNodes(), min_multi);
6  // Number of samples to process
7  global_2D[1]= round_up(batch_size, min_multi);
8
9  // Executes the kernel with the pre-determined parameters
10 cl_return = clEnqueueNDRangeKernel(queue, feedFwd, 2, NULL, global_2D, local_2D,
       0, NULL, NULL);
11
12 // Waits for the end of every element in the command queue
13 clFinish(queue);
```

Aside from the `clEnqueueWriteBuffer` call to transfer data from host to device, the OpenCL API has calls for device to host transfers, using `clEnqueueReadBuffer`, and a call for host to host internal data transfer, the `clEnqueueCopyBuffer`.

```
1  // READ THE KERNEL OUTPUT BUFFER
2  // Read the output buffer from this kernel to the host output array
3  cl_return = clEnqueueReadBuffer(queue, cl_output, CL_TRUE, 0,   ( hidden_layer->
       getNodes() * batchSize ) * sizeof(float), hiddenOutputBatch[layer], 0, NULL,
        NULL);
4
5  // [...]
6
7  // COPY THE BUFFERS
8  // Copy the previous execution output buffer to the input buffer
9  cl_return = clEnqueueCopyBuffer(queue, cl_output, cl_input, 0, 0, ( output_layer
       ->getNodes() * batchSize ) * sizeof(float), 0, NULL, NULL);
```

With the runtime execution and all the computation completed, we can then clear all the OpenCL allocated resources from memory.

```
1  // [...]
2
3  // CLEAR OPENCL RESOURCES
4  clReleaseMemObject{cl_weights};
5  clReleaseMemObject{cl_input};
6  clReleaseMemObject{cl_output};
7  clReleaseKernel(feedFwd);
8  clReleaseKernel(backProp);
9  clReleaseKernel(backPropHidden);
10 clReleaseProgram(program);
11 clReleaseCommandQueue(queue);
12
13 // [...]
```

4.2 General OpenCL Optimizations for Neural Networks

Work in the field of Neural Networks (NNs) started by looking at an OpenCL implementation of a Multi-Layer Perceptron (MLP) [38]. His goal was to implement and evaluate a NN running on a GPU. The source code provided contained a three layered MLP, training and classifying the Olive dataset [39].The training set from Olive has 572 samples and details the composition of eight chemicals (the training values) in the olive fruit, from nine regions of Italy (the classification labels).

The OpenCL kernel for the feed forward phase has two execution dimensions. These dimensions are related to the number of nodes in the current layer, the inputs, and the number of nodes from the previous layer, the outputs. All other computations associated with the MLP algorithm, including the back propagation, are performed in a serial manner on the CPU. The MLP has an input layer with 8 nodes (the chemical values), three hidden layers with 384 - 256 - 128 nodes, respectively, and an output layer with 9 nodes, equal to each of the regions to classify data. The weights were initialized with random values and the training run for 100 epochs with a fixed learning rate of 0.01. This phase was developed in an ASUS N53SN laptop comprised of an Intel i7 2630QM CPU at 2.0 GHz, 8 GB DDR3-1333MHz and an Nvidia GeForce GT 550M with 2 GB of GDDR5 as OpenCL device. The SDK for Nvidia provides OpenCL version 1.1. Next, we explain the several modifications and optimizations performed on his code, with total execution time and relative time savings achieved in each one.

v1. **Original implementation in [38]**
Starting from the original code, we benchmarked the execution time at **128.24s**.

v2. **Data transfer from one layer to the next:** $output\_layer[n] \Rightarrow input\_layer[n+1]$
In the feed forward phase, the output data of one layer is the input of the next one. Originally, this data transfer was done by copying the output from Device to Host, deleting the OpenCL memory objects. The memory objects were then created once more for the next layer, copying the recently transferred output once again but now in reverse order, from Host to Device. The optimization here performed consists of maintaining the data in the Device's memory, avoiding unnecessary slow data transfers between Device and Host. A copy is now performed internally, from Device to Device, via the `clEnqueueCopyBuffer(output, input)` command. We recorded an execution time of **101.46s** or a relative gain of 26.78s.

v3. **Work-Group size and number of Work-Items**
When querying an OpenCL device, the `PREFERRED_WORK_GROUP_SIZE_MULTIPLE` variable returns a reference value for the work-group size. For our particular device,

maintaining it at multiples of 32 allows achieving a faster processing time, even if there is a need for additional threads. This is due to the problem with idle threads. Threads are executed in groups called wavefronts for AMD and warps for Nvidia, of a determined size. If we launch only a fraction of those threads, those groups are unbalanced and a coalesced access is then impossible. A function to round up to the nearest minimum multiple was then developed, with the kernel executed accordingly, as depicted in Algorithm 1. During the kernel execution, if the thread

Algorithm 1 Round Up to the Best Work-Group Size Performer

```
1  // Minimum multiple work-items
2  const int min_multi = 32;
3
4  int round_up(int num, int multiple)
5  {
6          int rest = num \% multiple;
7
8          // If rest is 0, is already a multiple
9          if (rest == 0)
10                 return num;
11
12         // otherwise
13         return num + multiple - rest;
14 }
15
16 // [...]
17
18 size_t global_2D[2];
19 size_t local_2D[2]={1, min_multi};
20
21 // Number of nodes in the current layer
22 global_2D[0]= round_up(layer_nodes, min_multi);
23 // Number of samples to process
24 global_2D[1]= round_up(samples, min_multi);
25
26 // Executes the kernel with the pre-determined parameters
27 cl_return = clEnqueueNDRangeKernel(queue, feedFwd, 2, NULL, global_2D, local_2D, 0,
       NULL, NULL);
28
29 // [...]
```

ID exceeded the necessary values it means that it was was a “filler” thread, and would return from execution immediately. After this modification the execution time was reduced to **89.99s**, or a 11.47s gain from the previous optimization.

v4. Initial buffer allocation

Every call to `clCreateBuffer` has an impact in the execution time. To prevent extra unnecessary calls, an initial parsing of the max size needed for the input and output buffers was made and the buffers were allocated with that size, only to be freed from memory at the end of the current layer execution. This yielded a total time of **89.28s** and a relative gain of 0.71s regarding the last optimization.

v5. **Weight vectors appended together in a total weight vector**
In order to avoid making one weight transfer from Host to Device for every single layer, a buffer was initially created with enough size to accommodate the entire set of weight vectors. An extra kernel argument was then added with the offset value related to the position of each layer's weights. A total execution time of **88.83s** was recorded, with a relative gain of 0.45s from the last optimization.

v6. **Output vectors appended together in a total output vector**
Equal to the previous optimization but now regarding the output vectors from each layer. At this point the execution time takes **88.20s**, with a relative gain of 0.63s from the last optimization.

From these changes we were able to reduce the execution time a total of 40.04s, a third of the initial execution time. A comparison of all versions can be seen in Fig. 4.1. At this point, and from the 88.20s it was taking to execute, 81 of those seconds or 92% of the total execution time was spent in the back propagation phase, as it was still being performed in a serial manner on the CPU. It became clear that the next objective was to develop an OpenCL kernel for the back propagation.

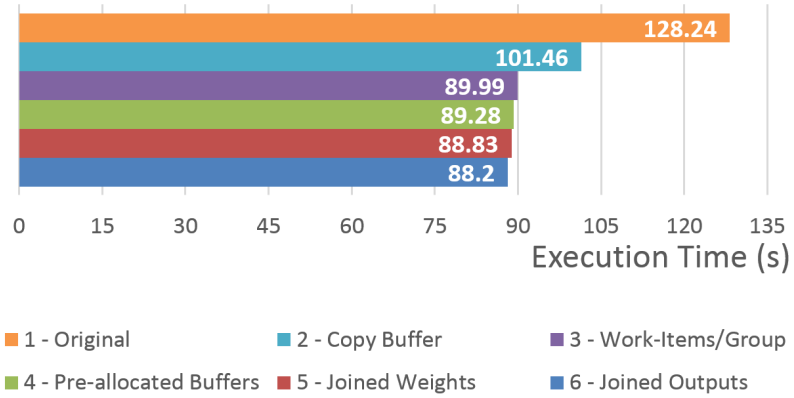

Figure 4.1: Comparison of the five initial OpenCL optimizations performed

4.3 OpenCL Kernels for Neural Network Parallelism

After the initial study discussed in Section 4.2, for the NN architectures referenced in this thesis three OpenCL kernels (functions running on the device) were developed. The first one relates to the feed forward algorithm, sending the data through the network and computing its results. After this phase is concluded, the second kernel computes the autoencoder reconstruction error at the output layer and then begins the gradient-based

back propagation algorithm. The back propagation, as the feed forward, suffers with data-dependency from the previous layer. Since the back-propagation for the hidden layer is depedent on the gradient calculations from the output layer, this results in a third kernel for that purpose.

4.3.1 Feed-Forward

When the samples from the dataset and weights for that layer are loaded to the device's global memory, we begin the initial phase, sending the data through the network. The kernel is launched 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 from one sample go through it.

Inside the kernel, a loop goes over all the layer input nodes and respective weights for that particular output node, computing the overall sum of that product (Eq. 2.4). An activation function, in this case being the sigmoid function, is then applied to that sum plus the bias of that output node (Eq. 2.3).

This kernel is valid for both the encoder and decoder phase of the autoencoder, the only difference being the input varying between the original image for the encoder layer and the encoder output for the decoder layer (Eq.2.5).

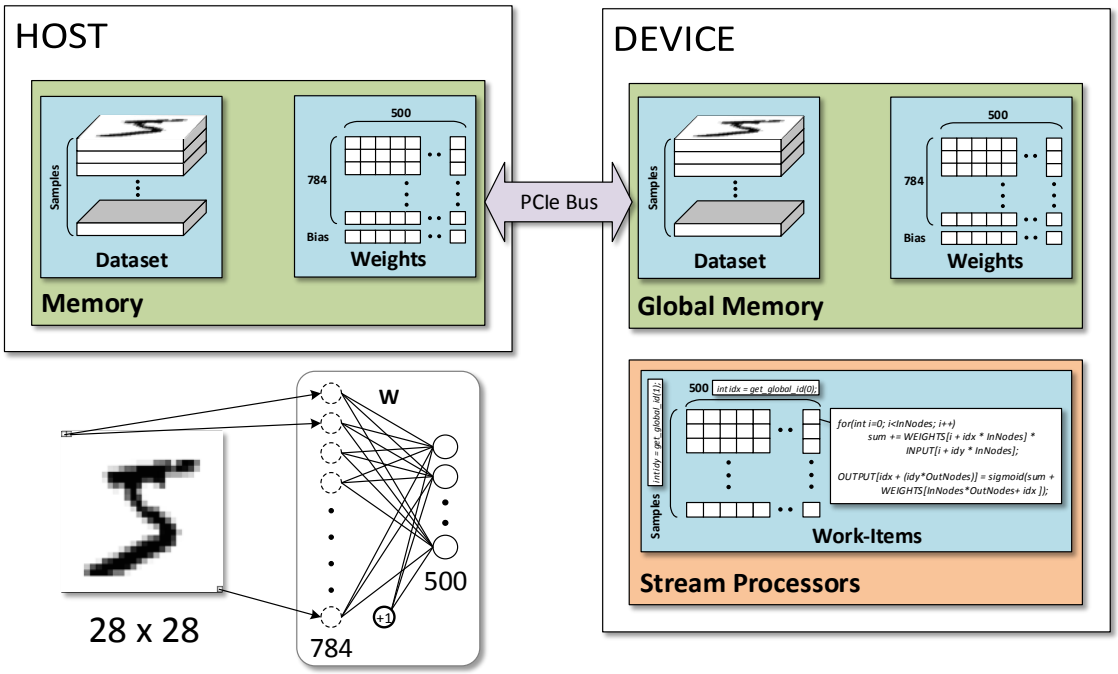

Figure 4.2: Feed forward work-items spread across two dimensions

4.3.2 Back Propagation - Output Layer

After computing the feed-forward across the Autoencoder (AE), the resulting output is of the same size as the input. We then have the possibility of comparing those values and thus calculating a reconstruction error. The kernel developed for this phase calculates the 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 in one dimension, that of the number of output nodes.

If as before in the feed-forward phase, the kernel was also launched across two dimensions, in the case of back propagation the resulting memory block size needed to avoid data-dependencies would be just too large to fit in the device's memory, as the size of `features` $\times$ `samples` $\times$ `nodes` $\times$ `sizeof(float)` could amount to hundreds of gigabytes of allocated memory.

The algorithm inside the kernel then loops over all dataset samples, computing the reconstruction error and gradient (Eq. 2.10). The partial derivative for the weights is then calculated via the gradient. The value for the bias is obtained directly from the gradient (Eq. 2.11), with the value for the weights also being dependent on the output from the previous hidden layer (Eq. 2.11). When all the samples have been processed, the mean of the gradient is needed due to the batch training method.

4.3.3 Back Propagation - Hidden Layer

The kernel used for the back propagation in the hidden layer is close to that of the output layer. We don't have a reconstruction error for this layer which is rather depent on the gradient calculated in the output layer. The kernel is then launched with one dimension, which was the size of the hidden layer output 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 low computationally demanding operation.

4.4 FPGA-Specific High-Level Optimizations

The current Altera SDK provides several compiler optimizations, improving the overall throughput performance of the hardware and architecture [40]. For the FPGA ver-

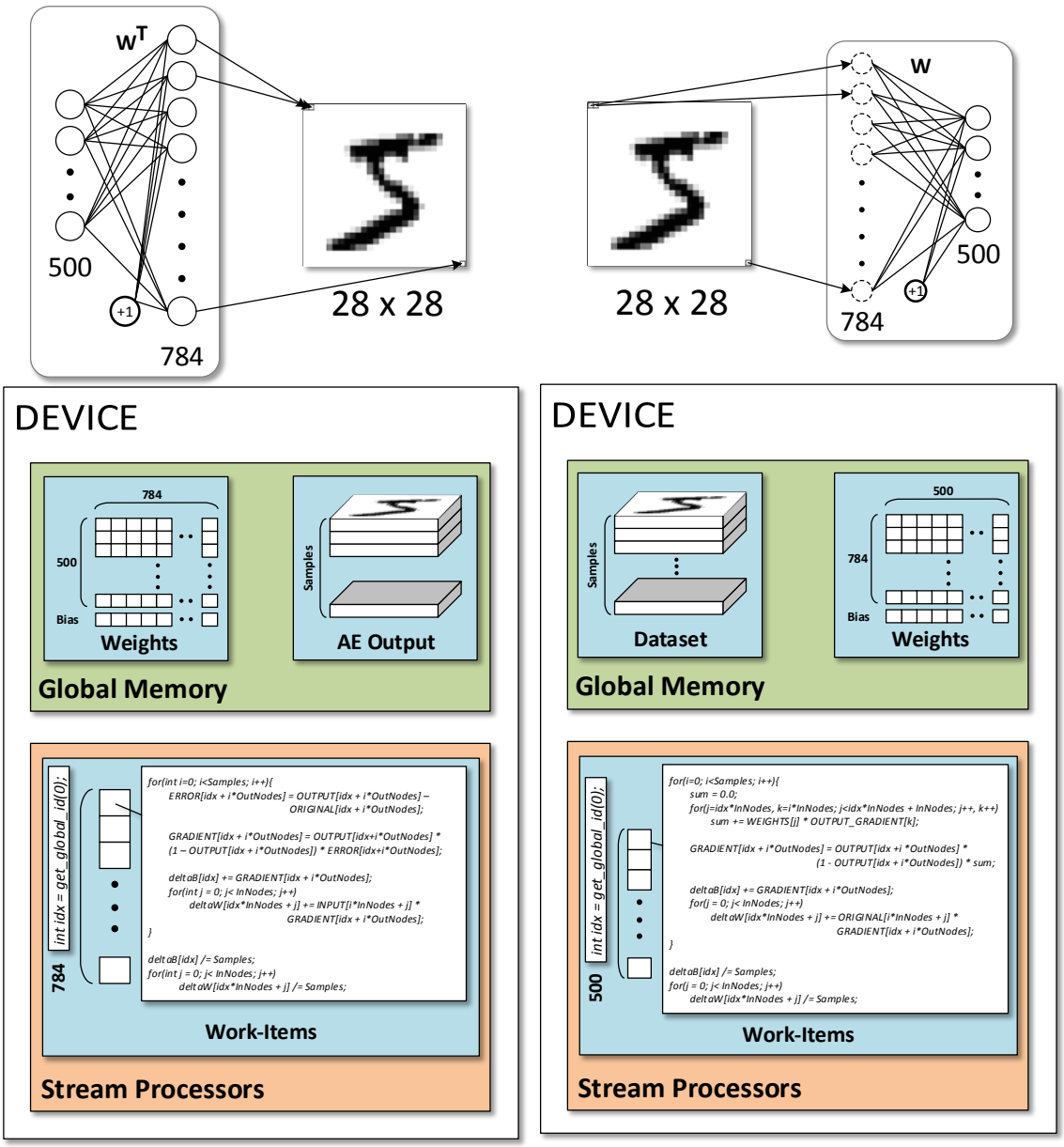

Figure 4.3: Back propagation work-items for the output layer (decoder)

Figure 4.4: Back propagation work-items for the hidden layer (encoder)

sion, several key optimizations were studied, namely the usage of multiple Compute Units (CUs), Single Instruction Multiple Data (SIMD) vectorization and loop unrolling.

4.4.1 Compute Units

Using a higher number of CUs results in increased throughput of work-items/second, with the work-groups being divided accordingly via the hardware scheduler to the available CUs. We must include the `__attribute__((num_compute_units(CUs)))` atribute in the code in the beginning of each kernel, with the desired number of CUs given at

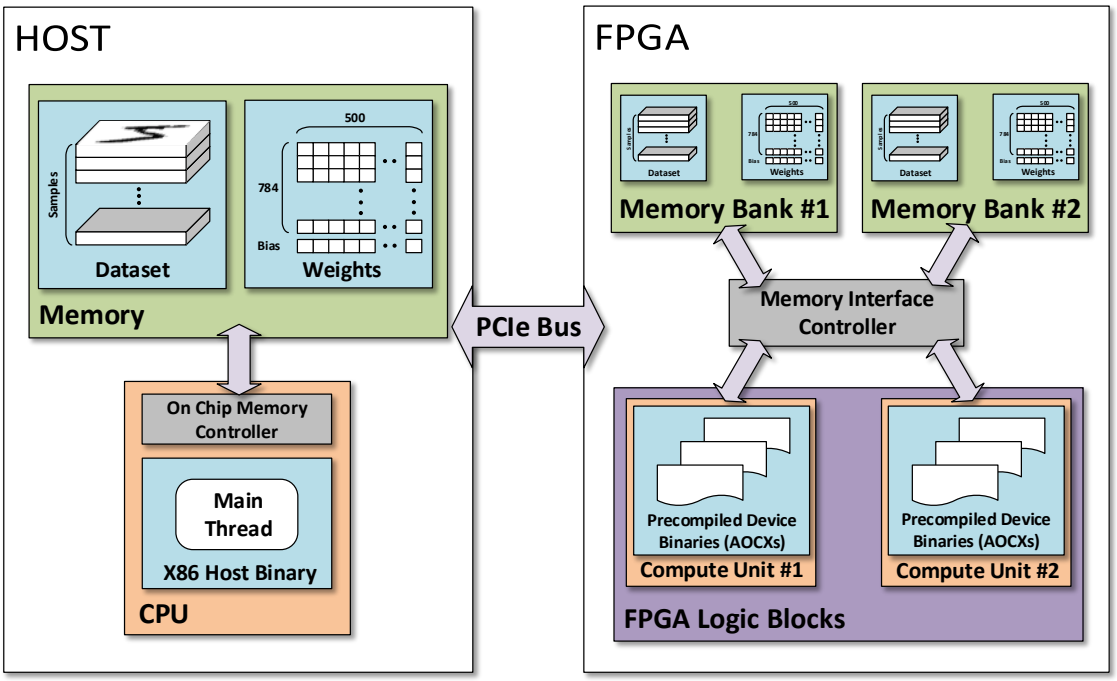

Figure 4.5: Architecture of the FPGA running the OpenCL kernels on two CUs

compile time.

4.4.2 SIMD Vectorization

SIMD vectorization is also available, boosting performance by computing the same operations (multiplication, addition, subtraction) over different data in a single instruction. This allows a speed up of operations/second in each work-item, coalescing memory accesses in the process. Usage is performed via the `__attribute__((num_simd_work_items(SIMDs)))` attribute, with one extra parameter being the required work-group size `__attribute__((reqd_work_group_size(WGx,WGy,WGz)))`.

4.4.3 Loop Unrolling

Finally, loop unrolling is a well known method for reducing loop delays and iterations. By replicating the code inside the loop several times, we reduce the need for end loop verifications and branch operations. To achieve this in the Altera SDK, a `#pragma unroll UNROLLs` directive is needed before the desired loop in the kernel code, with the amount of times to unroll given at compile time.

5

Methodology

Contents

5.1 The MNIST Dataset

The Mixed National Institute of Standards and Technology (MNIST) dataset consists of grayscale images of 28 by 28 pixels, each containing one hand written digit, obtained from around 250 different writers. The digits were size-normalized and centered. The dataset is divided into a training set with 60000 images and a test set with 10000 images. A full discussion of the dataset and the data itself can be obtained online [41] and some sample images can be seen in Figs. 6.3, 6.4 and 6.5, in column ”Original”.

5.2 FPGA Hardware Resources Utilization

Distinct combinations of different parameters were evaluated with the Altera Software Development Kit (SDK) for Open Computing Language (OpenCL) synthesis compiler. In this phase, several optimizations that greatly impact the final processing time of each kernel can be achieved. The resulting combination of Compute Units (CUs), Single Instruction Multiple Data (SIMD), and loop unrolling, provides a utilization of hardware resources and throughput, aiding to predict the final performance of the kernel.

5.2.1 Floating-Point Processing

An algorithmic limitation with impact in the utilization of Field-Programmable Gate Array (FPGA) resources consists of the need of floating-point calculations to be performed on input data and weights product. Since the dynamic range of the weights can and will vary even if the initial random generation is limited to a small interval, we are not able to map weights and inputs to the integer range, thus impacting the number of FPGA hardware resources used. Fixed-point computations are more efficient in both throughput performance and resource usage but current FPGAs have an emphasis in signal processing. FPGAs for signal processing are developed with a large number of Digital Signal Processing (DSP) blocks, thus raising floating-point computing resources and minimizing the impact in throughput performance.

5.3 Training Time

The most time consuming and computationally demanding phase of Neural Network (NN) processing is the training period, when the dataset is first presented to the network. In this process, the data goes through the network during several epochs, each time reducing the output error, converging to a better reconstruction and final solution of the problem. Considering this fact, we aimed at measuring the training time of the Stacked

Autoencoder (SAE) so we can better compare the platforms used. Each epoch time was measured from the host side, with the time counter initialized before the start of the first data transfer between host and device and finalized at the time the kernel queue is finished and the data processed is transferred back to the host.

5.4 Reconstruction/Classification Error

The accuracy of the end result may be measured in two ways, depending on the type and objective of the NN used. Regarding the Autoencoder (AE), as we try to encode and then decode the input, we aim at a low reconstruction error, representative of a good compression achieved in the lower dimension hidden layer.

When we stack the AEs and reduce the number of nodes in each layer, we repeat this procedure until we achieve a dimension equal to the number of classes in the dataset. We can then add a final layer with a softmax classifier as defined in subsection 2.5. Its output is evaluated against the correct class of that sample, resulting in this case as a classification error. The classification error was obtained using the test set, after the network was fully trained with only the training set. This aims at reproducing a real-world scenario in which we have a known set of data at our disposal for training the SAE (the training set) and an unknown set that we aim to classify (the test set).

5.4.1 Validation Set/Error

To guarantee the ideal learning rate at all times, and therefore speed up the error convergence and reduce the training time, an algorithm for error verification and validation was implemented. A subset of 2656 images from the training set were used as a validation set in order to obtain a validation error during the training process. Algorithm 2 was used on the remaining 57344 training images [42].

Algorithm 2 Checkpoint and Cross-Validation

$Restart \Leftarrow 0$
while $Restart < 3$ **do**
  train for 10 epochs
  evaluate the Validation Error, VE
  if VE increased for 2 consecutive epochs **then**
    $Restart \Leftarrow Restart + 1$
    $LearningRate \Leftarrow LearningRate/10$
    go back to the weights of the network used 20 epochs before
  end if
end while
evaluate the test error and stop

5.5 Throughput Performance

For this comparison we measured the epoch duration when training the MNIST dataset with the defined batch size and calculated how many Frames-per-second (FPS) were processed in this phase:

$$Training\ Throughput = \frac{Batch\ Size}{Epoch\ Time} \quad [FPS] \tag{5.1}$$

After the training is completed, the test data feeds forward through the trained SAE, in a new operational phase of the NN. The time it takes to undergo through this process is measured, along with the batch size, into a new classification throughput metric:

$$Classification\ Throughput = \frac{Batch\ Size}{SAE\ Processing\ Time} \quad [FPS] \tag{5.2}$$

5.6 Power and Energy Consumption

Power levels and energy consumption were estimated via the difference between the system load and idle, measured using a wall-plug energy monitor. In this manner we measure only the influence of the OpenCL device during application execution. All energy saving measures from the motherboard and Central Processing Unit (CPU) were disabled to prevent misreading of the energy consumption when idle. This value is measured in kiloWatt $\times$ hour (kWh).

5.7 Throughput per Power Ratio

We try to predict how many images we can train or classify with a single Watt of power, which indicates another measure of energy efficiency. The throughput performance was used, along with power usage measurements.

$$Efficiency = \frac{Throughput}{Power} \quad \left[\frac{Frames}{Watt \times Sec}\right] \tag{5.3}$$

5.8 Apparatus

The four computing platforms used in these experiments are stated in Table 5.1, with further specifications presented below.

GPU1: The host platform for GPU1 consists of an Intel i7 920 at 4.2GHz with 3x2GB Double-Data Rate (DDR)3, running Microsoft Windows 8.1. The OpenCL device consists of an Advanced Micro Devices (AMD) R9 290X with Graphics Processing

Unit (GPU) clocked at 1040MHz and 4GB Graphics Double-Data Rate Random Access Memory (GDDR)5 at 6250MHz.

Figure 5.1: The AMD R9 290X from Gigabyte [6]

GPU2: For GPU2, the host system is based on an Intel i7 4770k at 3.5GHz with 4x8GB DDR3, running CentOS release 6.5. The OpenCL device is an Nvidia GTX Titan with GPU clocked at 837MHz and 6GB GDDR5 at 6000MHz.

Figure 5.2: The Nvidia GTX Titan from ASUS [7]

GPU3: The GPU3 is available on a smartphone developing platform from Qualcomm, the DragonBoard, with a Snapdragon 800 System On Chip (SoC), comprised of an Advanced RISC Machine (ARM)v7 Krait 400 CPU at 2.15GHz and our OpenCL device, the Adreno 330 GPU clocked at 450MHz with 2GB of shared Low-Power (LP)-DDR3 at 1600MHz. The platform is currently running Android 4.3 - Jelly Bean.

Figure 5.3: The Snapdragon 800 DragonBoard from Qualcomm [8]

FPGA: Finally, coupled to the FPGA system we have an Intel i7 2600k at 3.4GHz acting as host CPU, with 2x4GB DDR3 of host memory, running CentOS release 6.4. The FPGA board is a Nallatech PCIe 385N Stratix V D5, populated with 2x4GB 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.

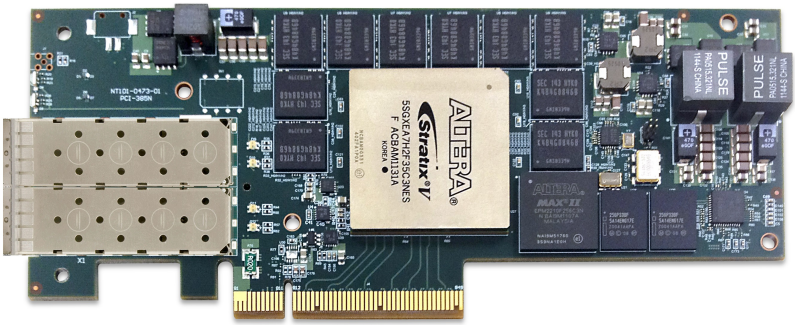

Figure 5.4: The Altera Stratix V D5 from Nallantech [9]

Platform	Host CPU	OpenCL Device	Device Memory
GPU1	Intel i7 920	AMD R9 290X	4GB GDDR5
GPU2	Intel i7 4770k	Nvidia GTX Titan	6GB GDDR5
GPU3	Qualcomm Krait 400	Qualcomm Adreno 330	2GB LPDDR3
FPGA	Intel i7 2600k	Altera Stratix V D5	2x4GB DDR3

Table 5.1: Hardware overview of the computing platforms

All the OpenCL devices used are manufactured using the same 28nm process design technology but with purchase price and power consumption in different ranges, as seen in Table 5.2.

Platform	Process Technology (nm)	Price (€)	Power (W)
GPU1	28	500	290
GPU2	28	900	250
GPU3	28	500	10
FPGA	28	6000	30

Table 5.2: Cost and power consumption for the OpenCL devices, as per indicated manufacturer data

As the host platforms are certainly heterogeneous, the OpenCL device's throughput performance during the training duration of the SAE is barely affected by this factor. It was verified via the profiling tool, that the percentage of total computational time on the device was 99.86%, with the host CPU running idle most of the time.

6

Experimental Results

Contents

In this section we display the final results and behavior of the studied implementations across the platforms described in section 5.8. We evaluate the performance of the system using several metrics, as explained in sections 5.3 through 5.7.

6.1 Training Hyper-parameters

Both the batch size and learning rate parameters were studied in depth. The study was comprised of 6 values for the batch size, ranging from 32 to 1024, and 4 values for the learning rate, from 0.045 to 1. The best results were achieved using a batch size of 64 images and an initial learning rate of 0.45. With these parameters, the training required 4m05s to reach the stopping criteria and produced a final reconstruction error of 1.626%. These results can be observed in Fig. 6.1.

Although the study was performed using GPU1, the algorithm behaves similarly across all platforms. Similar results regarding the same batch size and learning rate are to be expected, with only varying computational time.

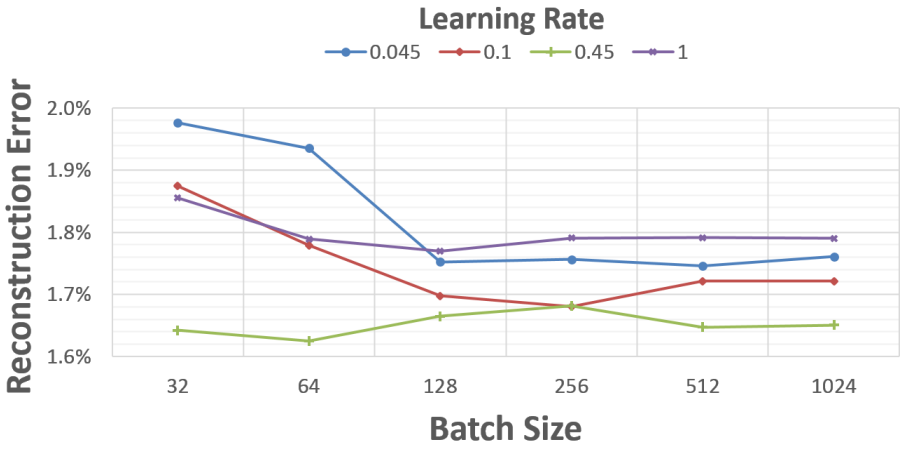

Figure 6.1: Reconstruction error comparison over 6 batch sizes and 4 learning rates

The training hyper-parameters defined for our Stacked Autoencoder (SAE) consist of a network of size 784 - 500 - 250 - 10, deemed the appropriate size for problem reduction, using a training batch of 64 images and an initial learning rate set at 0.45.

6.2 FPGA Optimizations and Hardware Utilization

We started by compiling a simple device binary, with the Open Computing Language (OpenCL) kernel code developed for the Graphics Processing Unit (GPU) imple-

mentation, using only one Compute Unit (CU), no loop unrolling and no Single Instruction Multiple Data (SIMD) vectorization.

We then moved to two CUs over all the kernels but immediately verified that we became close to 100% of resource usage, when usually above 70% or 80% of usage, the resulting critical path is long enough to compromise the maximum operating frequency of the system. A compromise was therefore mandatory. Since after training the Neural Network (NN) only the feed-forward kernel is necessary for classification and everyday usage, we opted for only using two CUs on the feed-forward kernel, leaving enough hardware resources for other optimizations.

A study with loop unrolling was also performed, with the available resources allowing to choose only a factor of two for the unroll. At this stage there were not enough resources available to test SIMD vectorization. In the end the optimal device binary was comprised of two CUs for the feed-forward kernel with one CU for the rest, and a loop unroll of factor two for all the `for` loops present in the kernels. The obtained hardware resources utilization is shown in Table 6.1.

Feed-Forward		**Back Propagation**		**Performance**	
# of CUs	**Unroll**	**# of CUs**	**Unroll**	**Total Utilization**	**Epoch Time (ms)**
1	1	1	1	60%	1160
2	1	2	1	96%	1012
2	1	1	1	75%	949
2	2	1	2	88%	907

Table 6.1: FPGA hardware resources utilization as obtained by the Altera OpenCL SDK compiler

6.3 Evaluating the Neural Network

As we trained the SAE using the Mixed National Institute of Standards and Technology (MNIST) dataset, several performance metrics were recorded for each of the Autoencoders (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. 6.2. By training the first AE during 1140 epochs, we achieved a reconstruction error of 1.62% for the first AE. The second AE was trained during another 2010 epochs with a final reconstruction error of 0.26%. Since the algorithm remains the same and weights were initialized with the same random seed generator, the error is constant across all four platforms.

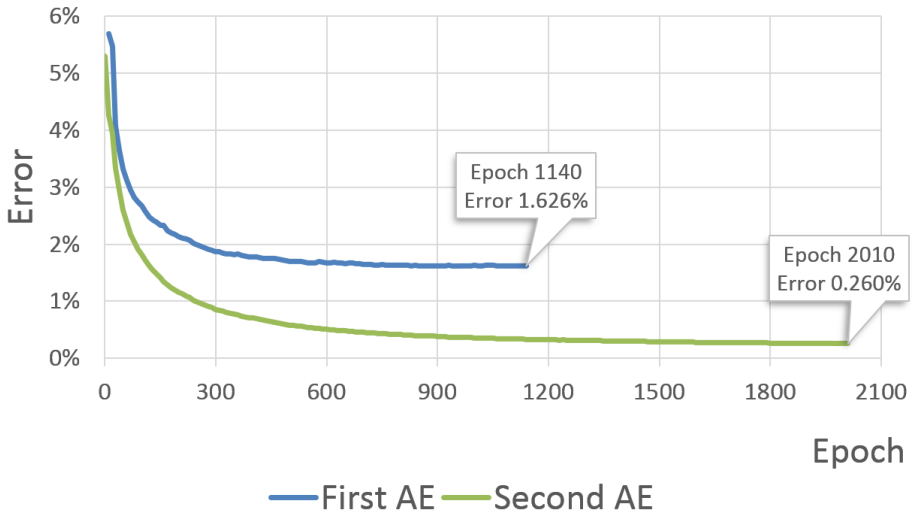

Figure 6.2: SAE reconstruction error as function of the number of epochs

The decoder output of the first AE tries to replicate the original image from the dataset, implying that we can compare both images rather than just looking at the reconstruction error value. A reconstruction of several images can be seen in Figs. 6.3 to 6.5, on column 'Reconstructed'.

In Table. 6.2 we evaluate the training time across all four platforms. In the end, GPU1 produced the fastest results training the SAE. Although being a powerful device from Nvidia, GPU2 takes longer to execute the same workload. The GPU3 presented us with revealing results as it beats the Field-Programmable Gate Array (FPGA), ranking as the third fastest platform. The FPGA has proven to be the slowest but in our opinion the savings in energy consumption still compensate.

Platform	**First AE Training Time**	**Second AE Training Time**	**Total Training Time**	**Training Time Comparison (*vs* GPU1)** *
GPU1	4m05s	3m34s	7m39s	—
GPU2	12m25	7m44s	20m09s	+ 163%
GPU3	35m33s	15m14s	50m47s	+ 564%
FPGA	44m47s	22m21s	1h08m08s	+ 778%

* Lower is better

Table 6.2: Final SAE training time for the four different platforms, with a batch size of 64 images and initial learning rate equal to 0.45

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 reconstruction and classification outputs were analyzed, along with a graphical output of the estimated classification as a function of the expected labels varying from 0 to 9,

all in Figs. 6.3 to 6.5. We studied cases of correct classification with high degree of probability (higher than 0.9) as seen in Fig. 6.3. There were few cases close to being misclassified, which are presented in Fig. 6.4 and finally misclassified images with a degree of probability higher than 0.6 are also represented in Fig. 6.5. A classification error of 4.35% was obtained over the 10000 test samples of the MNIST dataset. It should be clear that the errors reported in Figs. 6.3 to 6.5 all occur in the four platforms (GPU1, GPU2, GPU3 and FPGA).

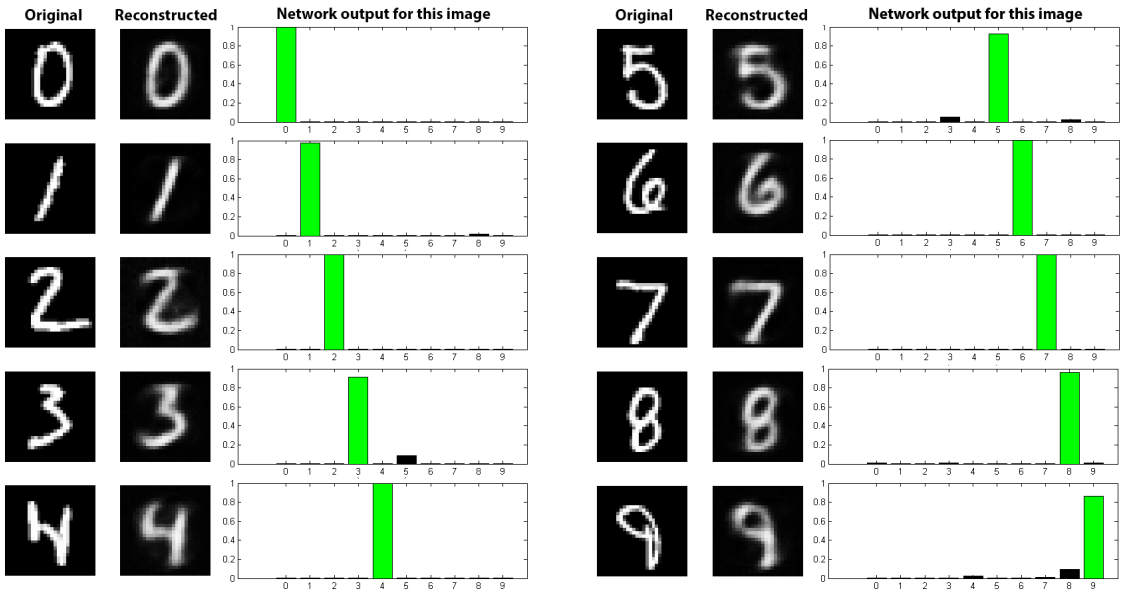

Figure 6.3: Some of the images correctly classified (from MNIST)

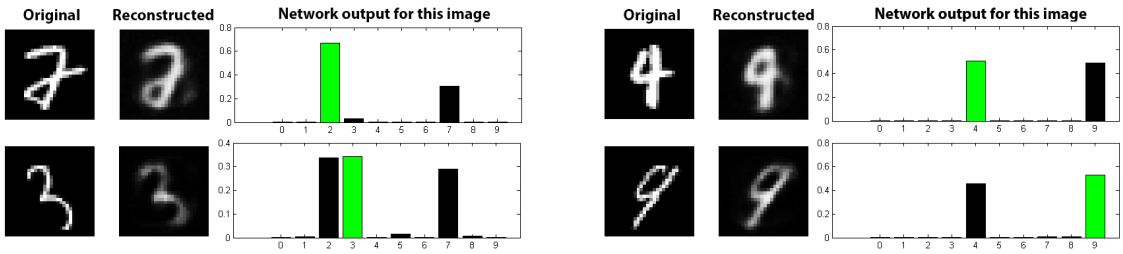

Figure 6.4: Difficult cases and near misses (from MNIST)

6.4 Throughput and Energy Analysis

As we can see in Table 6.3, the fastest GPU1 implementation shows a training throughput of 739 Frames-per-second (FPS), resulting in a performance 3.1× faster than GPU2 at 239 FPS, 8.7× faster than GPU3 at 85 FPS, and 10.4× faster than the FPGA at 71

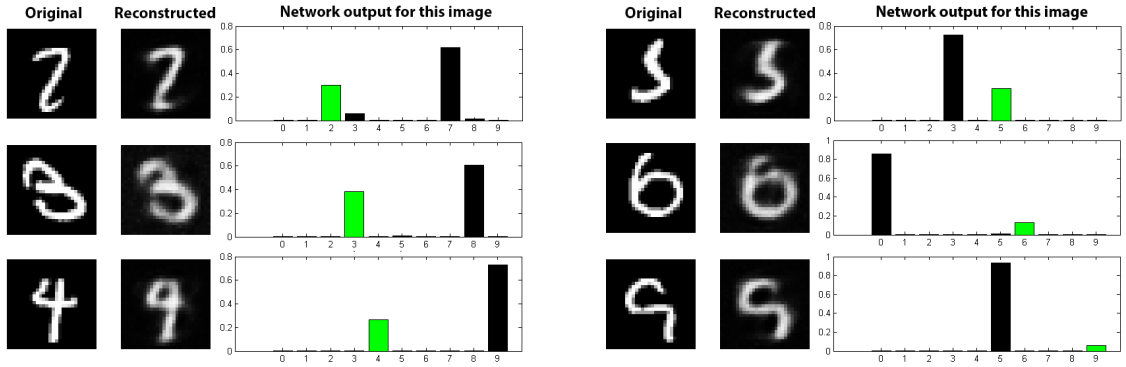

Figure 6.5: A collection of misclassified images (from MNIST)

FPS. The first AE was the one used 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 and its progressive problem size reduction.

Platform	Feed Forward (ms)	Back Propagation (ms)	Epoch Total (ms)	Training Throughput (FPS)	Throughput Comparison (*vs* GPU1) *
GPU1	6	81	87	739	—
GPU2	20	248	268	239	32.3% ↓
GPU3	158	595	753	85	11.5% ↓
FPGA	203	704	907	71	9.6% ↓

* Higher is better

Table 6.3: Running time and throughput performance associated with four different computing platforms, while training the first AE with a batch size of 64 images and initial learning rate of 0.45

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 seen in Table. 6.4.

For the power consumption analysis we first measured the idle consumption of the entire system (Host and Device) and then launched the application, measuring the difference (Load - Idle) in average power over the SAE training time. The results are shown in Table 6.5. Although running for 6.6× more time, GPU3 manages to have a total energy consumption more than an order of magnitude below that of GPU1, with around 91% less energy consumed for the same amount of work. The same can be said for the FPGA since that, while running 8.8× slower than GPU1, still consumes 43% less energy.

By combining throughput performance and average power we were able to measure

Platform	First AE (ms)	Second AE (ms)	Classification Total (ms)	Classification Throughput (FPS)	Throughput Comparison (*vs* GPU1) *
GPU1	4	1	5	12800	—
GPU2	10	4	14	4571	35.7% ↓
GPU3	79	40	139	460	3.6% ↓
FPGA	74	25	99	646	5.0% ↓

* Higher is better

Table 6.4: Running time and throughput performance associated with four different computing platforms, during the classification of a batch of 64 images

Platform	Total Training Time	Average Power (W)	Energy Consumption (kWh)	Energy Consumption Comparison (*vs* GPU1) *
GPU1	7m39s	247	0.03149	—
GPU2	20m09s	209	0.07019	223%
GPU3	50m47s	3.4	0.00288	9%
FPGA	1h08m08s	16	0.01790	57%

* Lower is better

Table 6.5: Total SAE training time and energy consumption associated with four different computing platforms, using a batch size of 64 images and learning rate of 0.45

throughput per power ratio, which shows a metric for energetic efficiency of these systems as depicted in Table 6.6.

Platform	Training FPS/Watt	Classification FPS/Watt
GPU1	2.99	51.82
GPU2	1.14	21.87
GPU3	24.99	135.29
FPGA	4.41	40.38

Table 6.6: Throughput per power ratio over four different computing platforms

6.5 Discussion

With these results we show that FPGAs are a valid alternative to GPUs when it comes to OpenCL computation in energy-saving environments. A network trained directly on the FPGA is thus possible, avoiding the need for training on the GPU or Central Processing Unit (CPU). Although the training time is several times higher in the FPGA, with the final reconstruction and classification error being similar across all platforms, the energy

savings compensate and make it a real possibility for adoption in a variety of low-power applications. The FPGA is at a loss in terms of FPS/Watt during the classification process but this is only a small portion of the total running time, taking 15 seconds to classify the entire MNIST test set of 10000 images, or only 0.4% of the total training time. Regarding an FPGA training and classifying images, the achieved 71 FPS and 646 FPS, respectively, may well suit many real world scenarios, allowing to reduce performance (e.g. 25 FPS) or increase image dimensions and still maintain a throughput to meet the situational demands while further increasing energy savings. The drawback of an FPGA solution remains its purchase price, 12× higher than GPU1 and GPU3, requiring a great deal of running time to recover the initial investment through energy savings. This is where GPU3 excels with its relatively low price of acquisition and impressive low-power performance.

The GPU3 does the same work as all the other platforms but since it was engineered with a low power budget from the start, it presents the overall best FPS/Watt results both in SAE training and classification, outperforming the FPGA in the low-power platforms. Its target market has several competitors and produces millions of devices, helping to reduce the purchase cost and creating the need for constant development and innovation. As more and more smartphones are sold and the market profusion increases, it raises the possibility of connecting them in a global heterogeneous computational network, with a NN-capable device aiding the user in many every-day applications.

We then conclude that the current low-power platforms such as the mobile GPU and FPGA devices, are more than able to provide a viable NN implementation in both training and classification stages. The training computation doesn't need to be performed on a more space- and energy-consuming machine but can be done *in loco*, using a low-power device, in a robotics or other autonomous and battery driven applications, possibly linked to a camera to directly provide the training/test samples in real-time.

7

Conclusions

Contents

In this thesis we presented a Neural Network (NN) architecture based on the Stacked Autoencoder (SAE), targeted at image classification. In this approach we developed Field-Programmable Gate Array (FPGA) parallel architectures programmed with OpenCL-based high-level synthesis tools to perform both the processing and also the training of the NN. To the best of our knowledge this was never produced before, as previous implementations always relied on powerful and energy-demanding Graphics Processing Units (GPUs). The simplified approach via SAE has produced a final classification error of 4.35%, a sub-optimal result explained by not relying on state-of-the-art Convolutional Neural Networks (CNNs) architectures and their high computational complexity, with which current FPGAs can't still cope. The developed algorithm is scalable and the next generations of FPGAs and mobile GPUs should be able to incorporate more hardware compute resources. Therefore, this approach works as a proof of concept and is expected to scale smoothly to future devices, allowing real-time performances also in images with larger dimensions.

Shifting to an FPGA platform we were able to reduce energy costs by 43% when compared against the top performing platform (GPU1), while maintaining a throughput of 71 Frames-per-second (FPS) during training and 646 FPS during classification. Most of the autonomous vehicles and robotics have already the means (in form of a computer with a Peripheral Component Interconnect Express (PCIe) slot) for running FPGAs, therefore it is advantageous to have the possibility of developing low-power based applications for supporting these systems.

Looking at the final results in this thesis, the mobile GPU in the GPU3 platform is the overall best performer. Energy costs were reduced by 91% when compared against the top performing platform (GPU1), while maintaining a throughput of 85 FPS during training and 460 FPS during classification. These devices have had the biggest technological development over the recent years, driven by the smartphone market growth and agressive competition. GPU3 is a solid self-contained low-power platform, already with an array of sensors and interfaces available for current robotics such as GPS, WiFi, camera module, among others.

Although desktop GPUs are the current platform of choice for this highly computationally demanding problem, the increasing concerns with energy costs for big data processing and low-power budgets for autonomous vehicles and robotics, as well as the fact that power and heat dissipation walls are quickly approaching, have created the need for alternatives and may well turn networks of FPGAs and mobile GPUs into the future platforms of choice for this and other computationally demanding applications.

7.1 Future Work

The Mixed National Institute of Standards and Technology (MNIST) dataset was used for training and evaluating the network of SAEs but in the future we expect to train more complex datasets such as Street View House Numbers (SVHN) and CIFAR-10, which are also representative of big data scenarios. Since the developed kernels are scalable and accommodate datasets with different dimensions, due to time constraints in this thesis we decided to limit the experimental procedures to the well-known monochromatic MNIST dataset. Future work will address the polychromatic datasets such as the aforementioned SVHN and CIFAR-10 datasets.

As technology progresses and more powerful FPGAs are developed, our future hope is to be able to create a state-of-the-art CNN running entirely on an FPGA, thus achieving top results in both energy savings and classification accuracy.

Bibliography

[1] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 Dataset." [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html

[2] AMD, "GPU 14 Product Showcase," 2013. [Online]. Available: http://ir.amd.com/phoenix.zhtml?c=74093&p=irol-eventDetails&EventId=5024687

[3] Nvidia, "Kepler Architecture," 2012. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[4] Qualcomm, "Snapdragon 800," 2013. [Online]. Available: http://www.qualcomm.com/snapdragon/processors/800

[5] Altera, "Stratix V FPGAs: Built for Bandwidth." [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/stxv-index.jsp

[6] Gigabyte, "AMD R9 290X," 2013. [Online]. Available: http://www.gigabyte.com/products/product-page.aspx?pid=4919

[7] ASUS, "Nvidia GTX TITAN," 2014. [Online]. Available: http://www.asus.com/Graphics_Cards/GTXTITAN6GD5/

[8] Qualcomm, "Snapdragon 800 DragonBoard," 2013. [Online]. Available: http://mydragonboard.org/db8074/

[9] Nallantech, "PCIe-385N - Altera Stratix V D5," 2012. [Online]. Available: http://www.nallatech.com/PCI-Express-FPGA-Cards/pcie-385n-altera-stratix-v-fpga-computing-card.html

[10] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Providence, RI, USA, June 2012, pp. 3642–3649.

[11] Y. Gong, Y. Jia, T. Leung, A. Toshev, and S. Ioffe, "Deep convolutional ranking for multilabel image annotation," *CoRR*, vol. abs/1312.4894, 2013.

[12] I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. Shet, "Multi-digit number recognition from street view imagery using deep convolutional neural networks," *CoRR*, vol. abs/1312.6082, 2013.

[13] N. Hardavellas, "The Rise and Fall of Dark Silicon," *USENIX ;login:*, vol. 37, no. 2, pp. 7–17, April 2011.

[14] A. Krizhevsky, "CUDA Convnet," 2014. [Online]. Available: https://code.google.com/p/cuda-convnet/

[15] J. Andrade, V. Silva, and G. Falcao, "From OpenCL to Gates: the FFT," in *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, 2013.

[16] F. López-Muñoz, J. Boya, and C. Alamo, "Neuron theory, the cornerstone of neuroscience, on the centenary of the nobel prize award to santiago ramón y cajal," *Brain Research Bulletin*, vol. 70, no. 4-6, pp. 391–405, 2006.

[17] D. Debanne, E. Campanac, A. Bialowas, E. Carlier, and G. Alcaraz, "Axon physiology," *Brain Research Bulletin*, vol. 91, no. 555-602, 2011.

[18] M. Davies, "Neuroscience: A Journey Through the Brain," 2002. [Online]. Available: http://www.ualberta.ca/~neuro/OnlineIntro/Index.htm

[19] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[20] F. Rosenblatt, "The perceptron: A perceiving and recognizing automaton," *Cornell Aeronautical Laboratory*, vol. 1, no. 1, 1957.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 533-536, 1986.

[22] G. Cybenko, "Approximation by superposition of a sigmoidal function," *Mathematics of Control, Signal and Systems*, vol. 2, pp. 303–314, 1989.

[23] A. Lapedes and R. Farber, "How neural nets work," in *Neural Information Processing Systems Conference (NIPS)*, Anderson, Ed. New York:American Institute of Physics, 1987, pp. 442–456.

[24] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[25] Y. LeCun and Y. Bengio, "Convolutional networks for images, speech, and time-series," in *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib, Ed. MIT Press, 1995.

[26] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, p. 504, 2006.

[27] G. E. Hinton, S. Osindero, and Y. W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

[28] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances In Neural Information Processing Systems*, no. 19. MIT Press, 2007, pp. 153–160.

[29] M. Ranzato, C. S. Poultney, S. Chopra, and Y. Lecun, "Efficient Learning of Sparse Representations with an Energy-Based Model," in *NeuralNeural Information Processing Systems Conference (NIPS)*, 2006, pp. 1137–1144.

[30] Y. Bengio, "Learning deep architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[31] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with COTS HPC systems," in *Proceedings of the International Conference on Machine Learning (ICML)*, Atlanta, Georgia, USA, 2013.

[32] LISA lab, "Convolutional Neural Networks (LeNet)," 2014. [Online]. Available: http://deeplearning.net/tutorial/lenet.html

[33] AMD, "GCN Architecture," 2012. [Online]. Available: http://www.amd.com/en-us/innovations/software-technologies/gcn

[34] Qualcomm, "Snapdragon 600 and 800 series," 2013. [Online]. Available: http://www.qualcomm.com/media/blog/2013/01/07/snapdragon-800-series-and-600-processors-unveiled

[35] Altera, "SDK for OpenCL™," 2014. [Online]. Available: http://www.altera.com/products/software/opencl/opencl-index.html

[36] ——, "Stratix V FPGA Family Overview." [Online]. Available: http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html

[37] K. Group, "The OpenCL Specification Version 1.2," 2012. [Online]. Available: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf

[38] X. Frazão, L. A. Alexandre, and P. Fazendeiro, "Implementação e avaliação de rede neuronal em GPU," 2012. [Online]. Available: https://www.dropbox.com/s/v3g69c0jejhywzo/XavierFrazao-GPU-NN.pdf

[39] M. Forina, "Olive Dataset." [Online]. Available: http://www.public.iastate.edu/~dicook/stat503/olive.html

[40] Altera, "Altera SDK for OpenCL - Optimization Guide," http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf, 2013, [Online; accessed 31-May-2014].

[41] Y. LeCun, "MNIST Dataset." [Online]. Available: http://yann.lecun.com/exdb/mnist

[42] L. A. Alexandre, "3d object recognition using convolutional neural networks with transfer learning between input channels," in *13th International Conference on Intelligent Autonomous Systems*, ser. Advances in Intelligent Systems and Computing Series, vol. 301. Padova, Italy: Springer, July 2014.

A

Appendix A

Contents

A.1 Autoencoder Flow

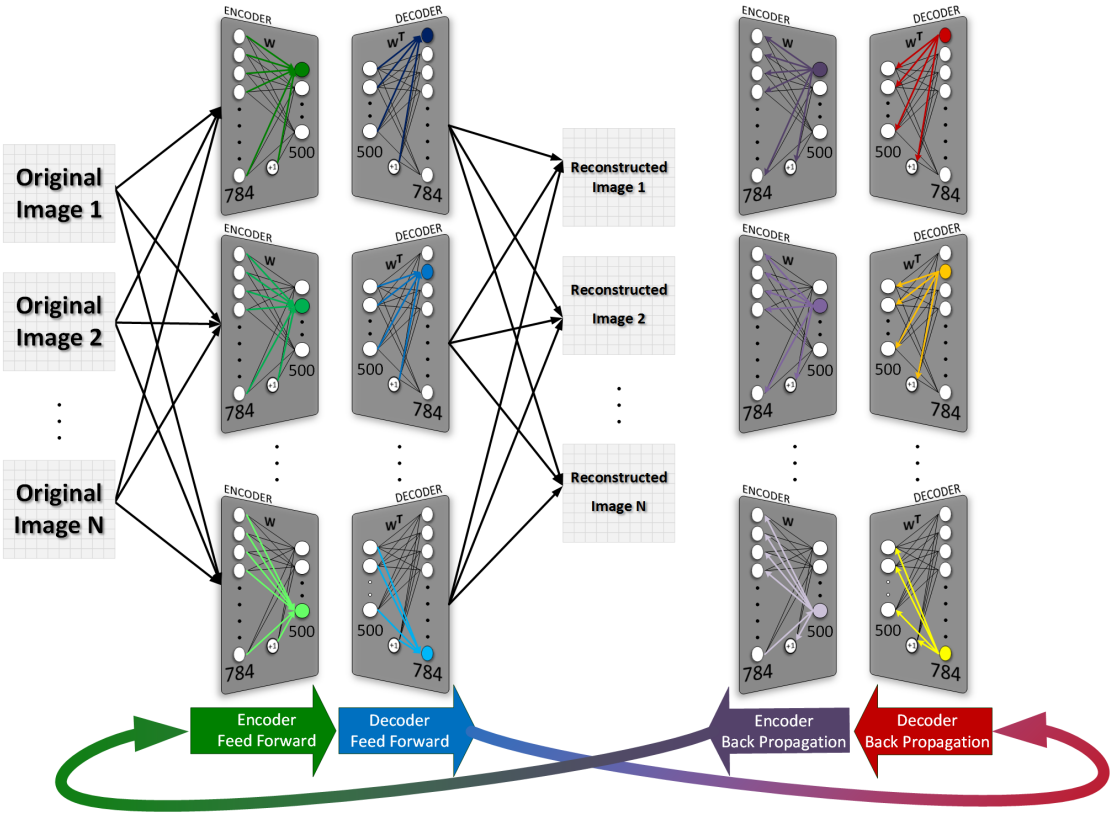

Figure A.1: Detailed autoencoder flow diagram

A.2 Training Flow

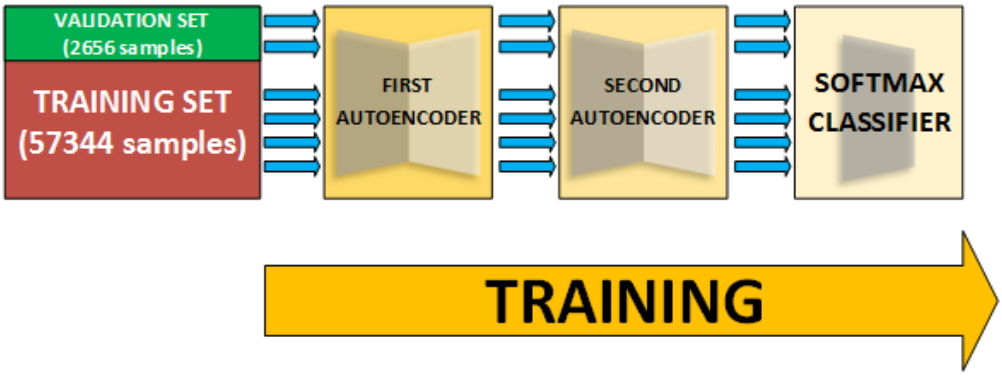

Figure A.2: Training flow diagram. The training flow of the first autoencoder is explained in more detail in Fig. A.3, the second autoencoder in Fig. A.4 and the softmax classifier in Fig. A.5

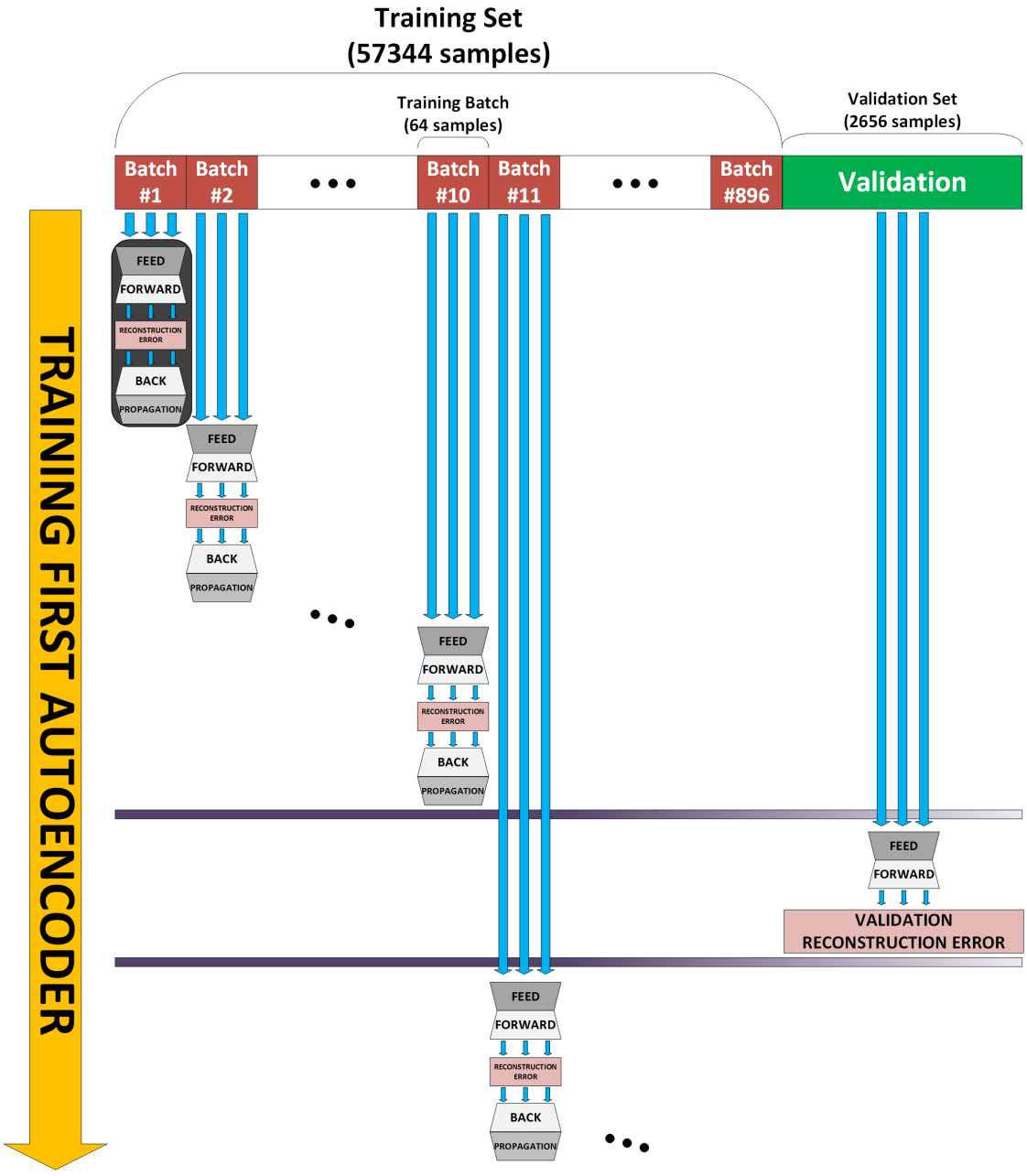

Figure A.3: First autoencoder training flow diagram. The detailed execution flow of the autoencoder in the shadowed area below Batch #1 is explained in Fig. A.1

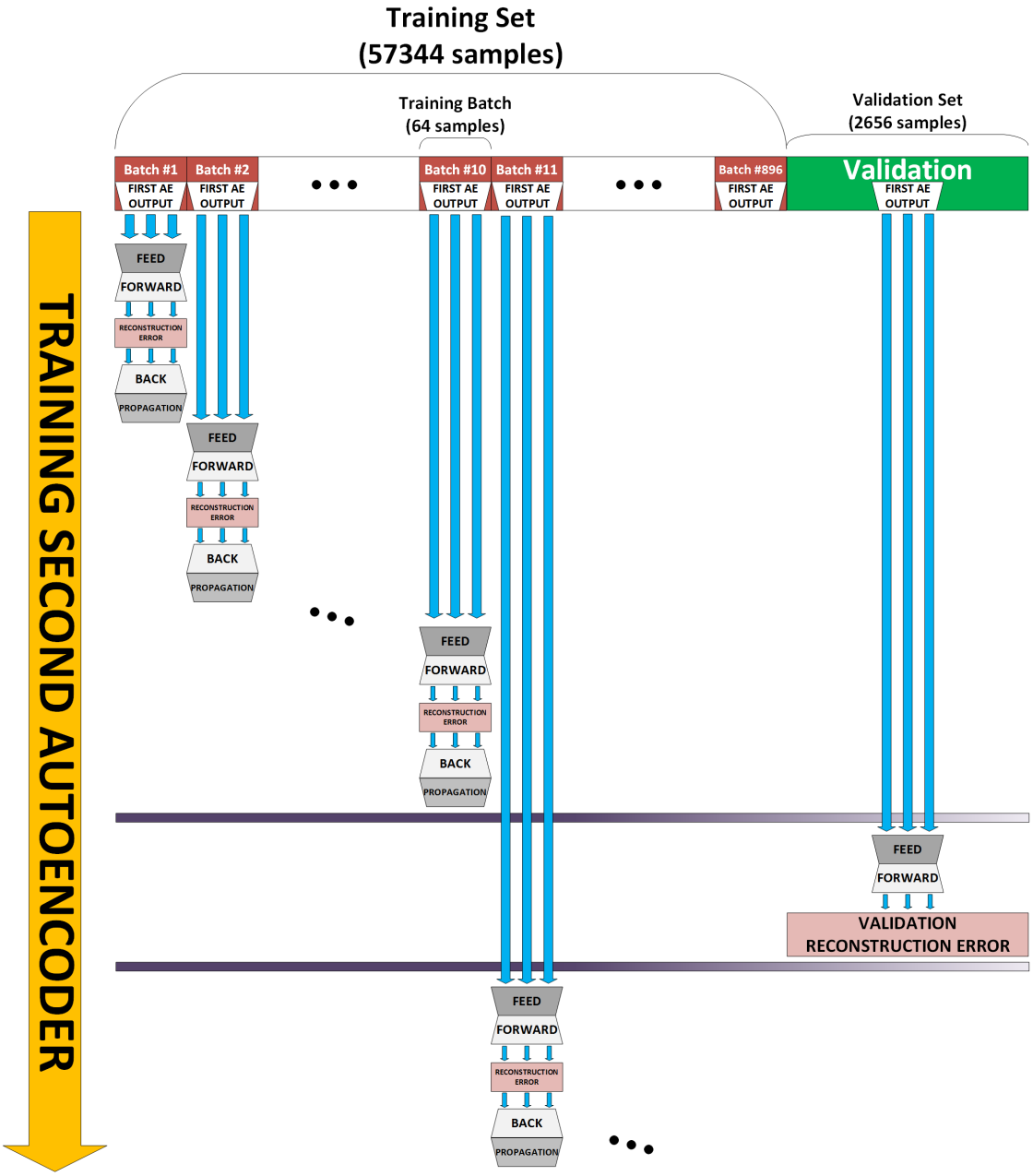

Figure A.4: Second autoencoder training flow diagram

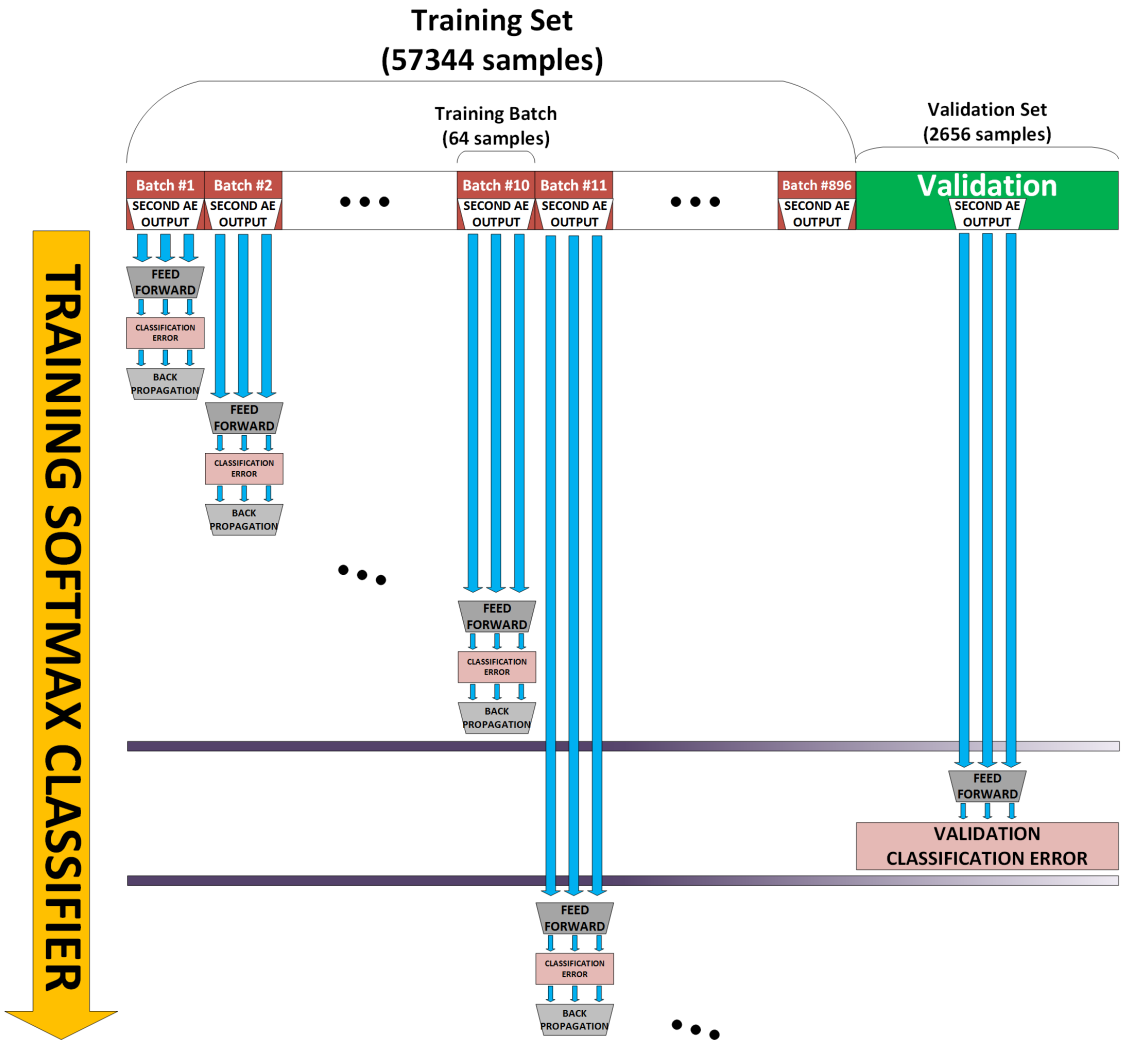

Figure A.5: Softmax Classifier training flow diagram

A.3 Classification Flow

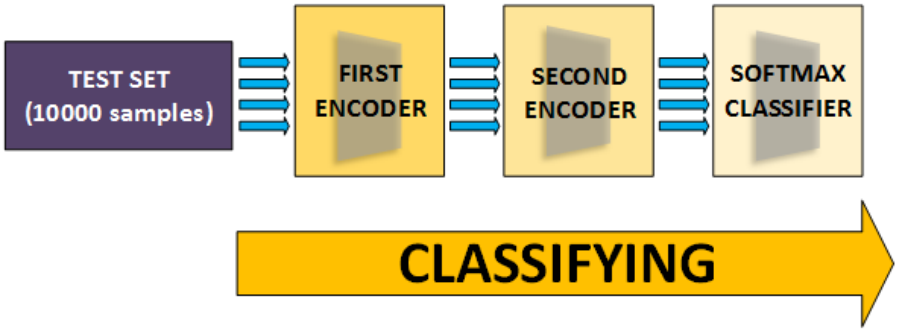

Figure A.6: Test set classification flow diagram. The classification flow is explained in more detail in Fig. A.7

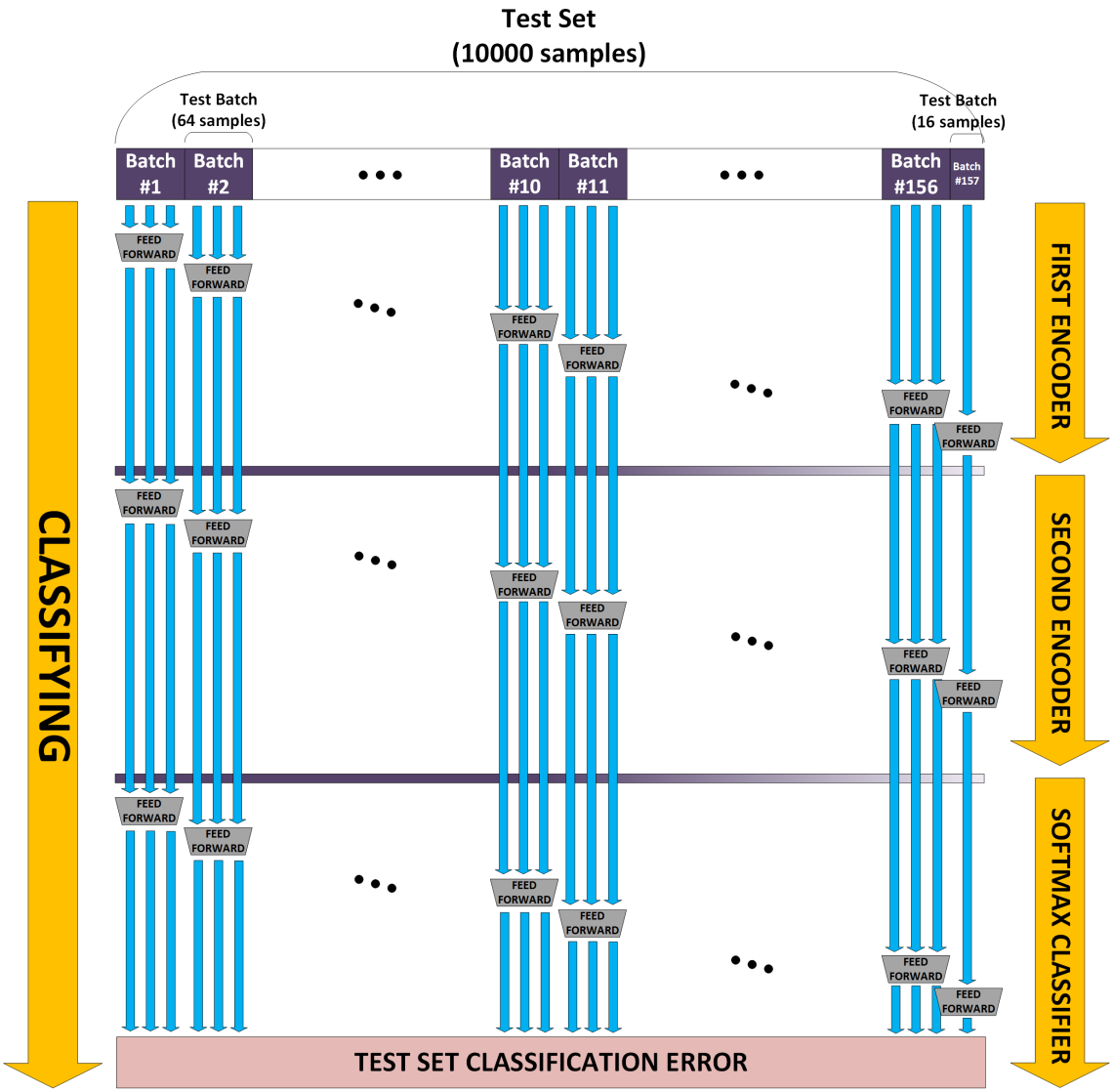

Figure A.7: Test set classification flow diagram

B

Appendix B

Energy-efficient Deep Learning: Stacked Autoencoders on FPGAs and Mobile GPUs

Joao Maria, Instituto de Telecomunicações, University of Coimbra, Portugal
Gabriel Falcao, Instituto de Telecomunicações, University of Coimbra, Portugal
Xavier Frazao, Instituto de Telecomunicações, University of Beira Interior, Portugal
Luis A. Alexandre, Instituto de Telecomunicações, University of Beira Interior, Portugal

Over the last years, deep learning architectures have gained attention by winning some of the most important international detection and classification competitions, but this comes at a cost: these models are computationally expensive and have been recently ported to GPUs to allow faster deployment. However, desktop GPUs have their own shortcomings and seem to be quickly approaching the limits of power and heat dissipation walls, imposing high levels of energy consumption. This implies high deployment costs in applications that process big data volumes on a permanent basis, and also the inability to use these architectures, for example, in autonomous systems such as vehicles and robots, which can hardly provide low power supplies. Therefore, this paper proposes deep learning approaches on mobile GPU- and FPGA-based context. We show how to implement a particular type of deep learning architecture, the Stacked Autoencoder (SAE), allowing for the first time the training phase to be performed on these low power devices. A comparison of both throughput performance and energy consumption is performed against similar implementations on desktop GPUs. The results show that similar classification accuracy can be obtained using the SAE proposed solution, with energy savings ranging from 46% to 91%. Also important is the fact that the proposed SAE architecture is scalable, and FPGAs and mobile GPUs have probably better progress margin than desktop GPUs. These results also pave the way for adopting low-power devices in energy-constrained applications for big data classification.

Categories and Subject Descriptors: B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids; C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Heterogeneous (hybrid) systems*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Deep learning, Neural networks, Stacked autoencoder, Parallel computing, FPGAs, GPUs, OpenCL

ACM Reference Format:
Maria, J., Falcao, G., Frazao, X., and Alexandre, L. A. 2014. Energy-efficient Deep Learning: Stacked Autoencoders on FPGAs. *ACM Trans. Architec. Code Optim.* 0, 0, Article 0 (2014), 22 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Associated to the processing of increasingly larger amounts of (big) data, machine learning and perception models aim at solving more complex and challenging tasks with lower classification errors. The number of samples used to train the algorithms

This work was partially supported by the Portuguese Fundação para a Ciência e Tecnologia (FCT) under the FCT project PEst-OE/EEI/LA0008/2013 and also by Instituto de Telecomunicações.
Author's addresses: J. Maria and G. Falcao, Instituto de Telecomunicações, Department of Electrical and Computer Engineering, University of Coimbra, Portugal; X. Frazao and L. A. Alexandre, Instituto de Telecomunicações, Department of Informatics, University of Beira Interior, Portugal.

DOI:http://dx.doi.org/10.1145/0000000.0000000

now surpasses the hundreds of thousands, which poses severe constraints regarding the time and processing power necessary to train the networks.

Recently, deep learning architectures have gained some momentum because they have shown superior performance in some of the most important international image, sound / voice detection and classification competitions [Ciresan et al. 2012; Gong et al. 2013; Goodfellow et al. 2013]. These typically deal with the automatic recognition of objects in images, whether these objects are in effect traffic signs, digits or animals, and have been won by research teams exploiting deep neural networks of the convolutional type [Ciresan et al. 2012]. At the time of its publication, this particular CNN presented the best results in 7 different datasets normally used for benchmarking similar algorithms, with improvements ranging from 30% to 80% with respect to previously best published results. The main drawback of these approaches (CNNs) is their computational cost that hinders real-time application.

To address both the size of the training datasets, and the high computational cost, deep learning approaches have been turning towards the use of GPU clusters [Coates et al. 2013], but even with this solution, experiments can still take several minutes or hours to execute [Ciresan et al. 2012]. As an example, the LeNet convolutional neural network, working with the MNIST dataset takes 380min on a CPU (Core i7-2600K CPU at 3.40GHz) and 32min on a GPU (GeForce GTX 480) to run a single experiment (including training and testing) [LISA lab 2014].

The current trend in machine learning / perception presently exploits the use of multiple representation levels, which can be achieved using deep belief networks, Stacked Denoising Autoencoders (SDAEs) or Convolutional Neural Networks (CNNs), among others. However, such current state-of-the-art implementations are known to consume high energy levels in order to produce the expected results, which directly impacts the processing costs of big data and also creates constraints in their utilization in autonomous vehicles / robots. Moreover, some of the powerful parallel computing devices under utilization, namely desktop GPUs, are reaching power- and heat-dissipation walls [Hardavellas 2011] (also known as utilization wall). Therefore, low power architectures and corresponding energy-saving strategies are required at this point of neural networks development.

In this paper we propose a scalable parallel solution for SAE architectures in reconfigurable FPGA substrates and mobile GPUs, as a first step towards the implementation of more complex approaches to deep learning (e.g. with higher computationally demanding layers and more nodes), such as CNNs. These deep learning approaches can expectedly be implemented in the future, as FPGAs accommodate more hardware resources, thus increasing throughput performance in the training and classification phases.

We propose to lower the network size and associated computational complexity of the developed parallel architecture, allowing for sub-optimal results albeit making it more tractable and thus able to cope with the existing available hardware resources of current FPGAs and mobile GPUs. These devices consume at least one order of magnitude less energy and are still able to provide real-time throughput and competitive classification error performance, when compared to existing high-performance computational resources, such as desktop GPUs or CPUs. The objective is to conciliate the quality of object recognition with real-time execution capabilities at low-energy consumption budgets.

The main problems identified are the limited hardware resources available in current FPGAs and mobile GPUs to support this type of NN-based algorithms; bandwidth bottleneck to access global memory; and the long development times associated with RTL design / development in FPGAs. The former problem can be addressed by developing new algorithms based on less complex autoencoder networks, adjusting the

B. Appendix B

number of layers and nodes per layer. The latter can be overcome using new High-level Synthesis tools that are very effective for designing and prototyping hardware systems for reconfigurable devices in short periods of time [Andrade et al. 2013].

In this paper we show for the first time how we can include the training of a deep network, designated as Stacked Autoencoder (SAE), on FPGAs and mobile GPUs. In particular, we propose:

i) to develop OpenCL kernels for a SAE architecture based on low-power processing devices, aimed at the training and classification phases on huge datasets. For the best of our knowledge, these long training periods have never been processed on these devices before (they are usually processed on desktop GPUs). To exemplify these scenarios we develop solutions for processing the well-known MNIST dataset.

and *ii)* to perform a power performance analysis by comparing the power and energy efficiency of these algorithms in several computing platforms, from desktop and mobile GPUs to FPGAs: we present experiments illustrating not only the accuracy obtained using these SAE architectures, but also the execution times and the respective power and energy consumption savings achieved when processing large amounts of images.

This paper proposes new solutions that advance the state-of-the-art of artificial intelligence, computer vision and parallel processing using the compute horse-power capabilities of FPGAs and mobile GPUs. We provide a scalable and multi-platform solution for training a SAE-based neural network, aimed at detecting objects, characters, or other type of structures in entire cityscapes. As the technology in the FPGAs and mobile GPUs progresses, and more processing resources are made available, a shift to more robust types of neural networks, such as the state-of-the-art CNNs, will be possible. Moreover, we pave the way for new applications in a diversity of areas that can benefit from the accurate real-time recognition of objects with lower consumption budgets. These areas include not only big data processing, as for example the identification and classification of large image data related to the visual information of entire city streets (modern infrastructures like Google and Facebook need to process and classify such large amounts of data on a daily/permanent basis), but also robotics or autonomous vehicles, which all present severe low-power constraints.

2. DEEP LEARNING USING NEURAL NETWORKS

The use of more than two hidden layers in neural network supervised learning was seen as unnecessary until recently, given the proofs of the approximation capabilities of one [Cybenko 1989] and two [Lapedes and Farber 1987] hidden layer neural networks.

The exceptions to this rule were the neocognitron [Fukushima 1980] which was using several layers to emulate the human visual system and the convolutional neural networks (CNNs) [LeCun and Bengio 1995], both developed mostly for visual tasks.

The other main issue with using deep networks, apart from the apparent unnecessity to use more than two layers, was the difficulty that appeared when trying to train several hidden layers using standard back-propagation: there were problems with adjusting the weights as the depth increased (vanishing gradients).

The efforts by Hinton and co-workers [Hinton and Salakhutdinov 2006; Hinton et al. 2006], resulted in the ability to train deep neural networks (DNNs), in this case, Deep Belief Networks (DBNs) which took advantage of Boltzmann machines in a variant called restricted Boltzmann machines (RBMs). At the same time, other groups proposed a way to training deep networks based on stacking autoencoders [Bengio et al. 2007; Ranzato et al. 2006].

From 2006 to today, the field of DNNs has received much attention. The potential advantages that come with using DNNs are the possibility of having increasingly more abstract levels of representation, the possibility of reusing the intermediate level rep-

This section of the article is temporarily unavailable for review.

Platform	First AE (ms)	Second AE (ms)	Total Classif. Time (ms)	Classification Throughput (FPS)	Throughput Comparison (*vs* GPU1) *
GPU1	4	1	5	12800	—
GPU2	10	4	14	4571	35.7% ↓
GPU3	79	40	139	460	3.6% ↓
FPGA	74	25	99	646	5.0% ↓

* Higher is better

Table VI: Running time and throughput performance associated with four different computing platforms, during the classification of a batch of 64 images

For the power consumption analysis we first measured the idle consumption of the entire system (Host and Device) and then launched the application, measuring the difference (Load - Idle) in average power over the SAE training time. The results are shown in Table VII. Although running for 6.6× more time, GPU3 manages to have a total energy consumption more than an order of magnitude below that of GPU1, with around 91% less energy consumed for the same amount of work. The same can be said for the FPGA since that, while running 8.8× slower than GPU1, still consumes 43% less energy.

Platform	Total Training Time	Average Power (W)	Energy Consumption (kWh)	Energy Consumption Comparison (*vs* GPU1) *
GPU1	7m39s	247	0.03149	—
GPU2	20m09s	209	0.07019	223% ↑
GPU3	50m47s	3.4	0.00288	9% ↓
FPGA	1h08m08s	16	0.01790	57% ↓

* Lower is better

Table VII: Total SAE training time and energy consumption associated with four different computing platforms, using a batch size of 64 images and learning rate of 0.45

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 VIII.

Platform	Training (FPS/Watt)	Classification (FPS/Watt)
GPU1	2.99	51.82
GPU2	1.14	21.87
GPU3	24.99	135.29
FPGA	4.41	40.38

Table VIII: Throughput per power ratio over four different computing platforms

7. DISCUSSION

With these results we show that FPGA are a valid alternative to GPU when it comes to OpenCL computation in energy-saving environments. A network trained directly on the FPGA is thus possible, avoiding the need for training on the GPU or CPU. Although the training time is several times higher in the FPGA, with the final reconstruction and classification error being similar across all platforms, the energy savings compensate and make it a real possibility for adoption in a variety of low-power applications.

The FPGA is at a loss in terms of FPS/Watt during the classification process but this is only a small portion of the total running time, taking 15 seconds to classify the entire MNIST test set of 10000 images, or only 0.4% of the total training time. Regarding an FPGA training and classifying images, the achieved 71 FPS and 646 FPS, respectively, may well suit many real world scenarios, allowing to reduce performance (e.g. 25 FPS) or increase image dimensions and still maintain a throughput to meet the situational demands while further increasing energy savings. The drawback of an FPGA solution remains its purchase price, $12\times$ higher than GPU1 and GPU3, requiring a great deal of running time to recover the initial investment through energy savings.

This is where GPU3 excels, with its relatively low price of acquisition and impressive low-power performance. Providing a training throughput of 85 FPS, and a classification throughput of 460 FPS, the GPU3 still manages to reduce the energy consumption by 91%. The GPU3 is therefore the most energy-efficient computing platform of the roundup, and a suitable solution for the aforementioned low-power applications such as autonomous vehicles and robotics.

8. CONCLUSION

In this paper we presented a Neural Network (NN) architecture based on the Stacked Autoencoder targeted at image classification. In this approach we developed FPGA parallel architectures programmed with OpenCL-based high-level synthesis tools to perform the processing and also the training of the NN. This way we were able to reduce energy costs by 43%, when compared against desktop GPUs.

The mobile GPU does the same work as all the other platforms but since it was engineered with a low power budget from the start, it presents the overall best FPS/Watt results both in SAE training and classification, outperforming the FPGA in the low-power platforms. Presenting 91% less energy consumed for the same amount of work as the top desktop GPU, as well as 48% less than the FPGA, it clearly edges ahead of the tested platforms. Its target market has several competitors and produces millions of devices, helping to reduce the purchase cost and creating the need for constant development and innovation. As more and more smartphones are sold and the market profusion increases, it raises the possibility of connecting them in a global heterogeneous computational network, with a neural network capable device aiding the user in many every-day applications.

We then conclude that the current low-power platforms such as the mobile GPUs and FPGA devices, are more than able to provide a viable neural network implementation in both training and classification stages. The training computation doesn't need to be performed on a more space- and energy-consuming machine but can be done *in loco*, using a low-power device, in a robotics or other autonomous and battery driven applications, possibly linked to a camera to directly provide the training/test samples in real-time.

REFERENCES

Luís A. Alexandre. 2014. 3D Object Recognition using Convolutional Neural Networks with Transfer Learning between Input Channels. In *13th International Conference on Intelligent Autonomous Systems (Advances in Intelligent Systems and Computing Series)*, Vol. 301. Springer, Padova, Italy.

Altera. 2013. Altera SDK for OpenCL - Optimization Guide. http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf. (2013). [Online; accessed 31-May-2014].

Altera. 2014. SDK for OpenCL. (2014). http://www.altera.com/products/software/opencl/opencl-index.html

Joao Andrade, Vitor Silva, and Gabriel Falcao. 2013. From OpenCL to Gates: the FFT. In *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*.

Y. Bengio. 2009. Learning Deep Architectures for AI. *Foundations and Trends in Machine Learning* 2, 1 (2009), 1–127.

Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. 2007. Greedy Layer-Wise Training of Deep Networks. In *Advances In Neural Information Processing Systems*. MIT Press, 153–160.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. 2010. Theano: a CPU and GPU Math Expression Compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

Dan C. Ciresan, Ueli Meier, and Jürgen Schmidhuber. 2012. Multi-column deep neural networks for image classification. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Providence, RI, USA, 3642–3649.

Adam Coates, Brody Huval, Tao Wang, David J. Wu, Andrew Y. Ng, and Bryan Catanzaro. 2013. Deep learning with COTS HPC systems. In *Proceedings of the International Conference on Machine Learning (ICML)*. Atlanta, Georgia, USA.

G. Cybenko. 1989. Approximation by superposition of a sigmoidal function. *Mathematics of Control, Signal and Systems* 2 (1989), 303–314.

Aysegul Dundar, Jonghoon Jin, Vinayak Gokhale, Berin Martini, and Eugenio Culurciello. 2013. Neural Information Processing Systems Conference (NIPS). (2013).

Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. 2010. Hardware accelerated convolutional neural networks for synthetic vision systems. In *IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 257–260.

K. Fukushima. 1980. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 36, 4 (1980), 193–202.

Yunchao Gong, Yangqing Jia, Thomas Leung, Alexander Toshev, and Sergey Ioffe. 2013. Deep Convolutional Ranking for Multilabel Image Annotation. *CoRR* abs/1312.4894 (2013).

Ian J. Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. 2013. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. *CoRR* abs/1312.6082 (2013).

N. Hardavellas. 2011. The Rise and Fall of Dark Silicon. *USENIX ;login:* 37, 2 (April 2011), 7–17.

G. E Hinton, S. Osindero, and Y. W Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

G. E. Hinton and R. R. Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504.

Khronos Group. 2010. OpenCL - The open standard for parallel programming of heterogeneous systems. (2010). http://www.khronos.org/opencl.

A. Lapedes and R. Farber. 1987. How neural nets work. In *Neural Information Processing Systems Conference (NIPS)*, Anderson (Ed.). New York:American Institute of Physics, 442–456.

Hugo Larochelle, Yoshua Bengio, Jrme Louradour, Pascal Lamblin, and Lon Bottou. 2009. Exploring strategies for training deep neural networks. *Journal of Machine Learning Research* 10, 1 (2009), 1–40.

Y. LeCun. 2014. MNIST Dataset. (2014). http://yann.lecun.com/exdb/mnist

Y. LeCun and Y. Bengio. 1995. Convolutional Networks for Images, Speech, and Time-Series. In *The Handbook of Brain Theory and Neural Networks*, M. A. Arbib (Ed.). MIT Press.

LISA lab. 2014. Convolutional Neural Networks (LeNet). (2014). http://deeplearning.net/tutorial/lenet.html

NVIDIA. 2007. The CUDA homepage. (2007). http://www.nvidia.com/object/cuda_home.html.

Marc'aurelio Ranzato, Christopher S. Poultney, Sumit Chopra, and Yann Lecun. 2006. Efficient Learning of Sparse Representations with an Energy-Based Model. In *NeuralNeural Information Processing Systems Conference (NIPS)*. 1137–1144.

Ronan, Clment, Koray, and Soumith. 2014. Torch7 - Scientific computing for LuaJIT. (2014). http://torch.ch/

C

Appendix C

Neural Processing Letters manuscript No.
(will be inserted by the editor)

Low-power Accelerated Architectures using Stacked Autoenconders for Object Recognition in Autonomous Systems

Joao Maria · Joao Amaro · Gabriel Falcao · Luis A. Alexandre

Received: date / Accepted: date

Abstract This paper investigates low-power consumption hardware models and processor architectures for real-time object recognition in power-constrained autonomous systems and robotics. 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 close to the state-of-the-art and a high number of processed frames per second. Another novelty equally proposed 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 CPU or a GPU to perform this task and runs the trained network later on the FPGA. This allows incorporating new functionalities as, for example, a robot performing online learning. We present experiments using the polychromatic CIFAR-10 dataset.

Keywords Deep learning · Neural networks · Stacked autoencoder · Parallel computing · FPGAs · mobile GPUs · OpenCL · Low-power consumption

J. Maria · J. Amaro · G. Falcao
Instituto de Telecomunicacoes
Department of Electrical and Computer Engineering, University of Coimbra - Pole II
R. Silvio Lima
3030-290 Coimbra, Portugal
Tel.: +351 239 796 267/371
Fax: +351 239 796 247
E-mail: jmaria@co.it.pt

L. A. Alexandre
Department of Informatics, University of Beira Interior
R. Marques d'Avila e Bolama
6201-001 Covilha, Portugal
E-mail: lfbaa@di.ubi.pt

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 [5, 9, 10]. In particular, convolutional neural networks (CNNs) have assumed greater and greater importance [5], since they have shown performances 30 to 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 [17, 13]. Such use of multiple representation stages can be achieved using CNNs or other types of deep networks 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 GPU devices [19]. 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 [12]. Also, power and physical limitations in the chip manufacturing process limit the frequency of operation of these devices to values around 1GHz.

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 CPU or GPU to perform that computation, and the trained model then implemented on the FPGA [8, 6].

The computational power of mobile GPUs in smartphones and tablets is beginning to be studied, mainly in the area of computer vision [23].

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 [7]. 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 polychro-

This section of the article is temporarily unavailable for review.

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 7.

Platform	Total Training Time	Average Power (W)	Energy Consumption (kWh) *
mGPU	17h08m26s	6.6	0.113
FPGA	45h31m03s	16.0	0.728

* Lower is better

Table 7: Total SAE training time and energy consumption

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 8.

Platform	Training FPS/Watt	Classification FPS/Watt
mGPU	1.45	18.03
FPGA	0.24	2.81

Table 8: Throughput per power ratio over four different computing platforms

5 Conclusions

In this paper we show for the first time the training phase of a polychromatic dataset in a SAE performed on low-power devices, namely the FPGA and mobile GPU. Although the time necessary to complete the training process is extensive, the overall energy consumption remains low. With a training phase 3× quicker, the mobile GPU manages to have a total energy consumption of 6.4× below that of the FPGA, with around 84% less energy consumed for the same amount of 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 the sub-optimal approach based on the SAE, we have achieved real-time classification throughput on both platforms, with 45 FPS on the FPGA and 119 FPS on the mobile GPU, or 2.6× higher. With the high throughput on the mobile GPU, a future implementation can be linked to the platform's camera, providing the capture and classification of images in real-time.

The purchase cost remains a major drawback from FPGAs, and makes the usage of the more affordable mobile GPUs as a valid alternative. Since the average power during training remains low in both platforms, the utilization of these solutions in low-power scenarios is thus proven by our results.

The mobile GPU and FPGA are in a class of low-power devices that allow computationally demanding algorithms to be performed directly on autonomous vehicles, robots and other low-power budget applications. As technology progresses and more powerful FPGAs and mobile GPUs are developed, our future hope is to be able to create a state-of-the-art CNN running entirely on these devices, thus achieving top results in both energy savings and classification accuracy.

References

1. Altera: Stratix V FPGA Overview. URL http://www.altera.com/devices/fpga/stratix-fpgas/stratix-v/overview/stxv-overview.html
2. Altera: Altera SDK for OpenCL - Optimization Guide (2013). URL http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf
3. Altera: Design Tools - VHDL (2013). URL http://www.altera.com/support/examples/vhdl/vhdl.html
4. Altera: SDK for OpenCL (2014). URL http://www.altera.com/products/software/opencl/opencl-index.html
5. Ciresan, D.C., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 3642–3649. Providence, RI, USA (2012)
6. Dundar, A., Jin, J., Gokhale, V., Martini, B., Culurciello, E.: Accelerating deep neural networks on mobile processor with embedded programmable logic. Neural Information Processing Systems Conference (NIPS) (2013)
7. Falcao, G., Silva, V., Sousa, L., Andrade, J.: Portable ldpc decoding on multicores using opencl [applications corner]. Signal Processing Magazine, IEEE **29**(4), 81–109 (2012). DOI 10.1109/MSP.2012.2192212
8. Farabet, C., Martini, B., Akselrod, P., Talay, S., LeCun, Y., Culurciello, E.: Hardware accelerated convolutional neural networks for synthetic vision systems. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 257–260. IEEE (2010)
9. Gong, Y., Jia, Y., Leung, T., Toshev, A., Ioffe, S.: Deep convolutional ranking for multilabel image annotation. CoRR **abs/1312.4894** (2013)
10. Goodfellow, I.J., Bulatov, Y., Ibarz, J., Arnoud, S., Shet, V.: Multi-digit number recognition from street view imagery using deep convolutional neural networks. CoRR **abs/1312.6082** (2013)
11. Group, K.: The OpenCL Specification Version 1.2 (2012). URL https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf
12. Hardavellas, N.: The Rise and Fall of Dark Silicon. USENIX ;login: **37**(2), 7–17 (2011)
13. Hinton, G., Deng, L., Dahl, G.E., Mohamed, A., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T., Kingsbury, B.: Deep neural networks for acoustic modeling in speech recognition. IEEE Signal Processing Magazine **29**(6), 82–97 (2012)
14. Kaggle: Public Leaderboard - CIFAR-10 - Object Recognition in Images (2013). URL http://www.kaggle.com/c/cifar-10/leaderboard
15. Krizhevsky, A.: Learning multiple layers of features from tiny images. Tech. rep. (2009)
16. Krizhevsky, A., Nair, V., Hinton, G.: CIFAR-10 Dataset. URL http://www.cs.toronto.edu/ kriz/cifar.html
17. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems 25 (NIPS'2012) (2012)
18. Nallantech: PCIe-385N - Altera Stratix V D5 (2012). URL http://www.nallatech.com/PCI-Express-FPGA-Cards/pcie-385n-altera-stratix-v-fpga-computing-card.html
19. Oh, K.S., Jung, K.: Gpu implementation of neural networks. Pattern Recognition **37**(6), 1311–1314 (2004)
20. Owaida, M., Falcao, G., Andrade, J., Antonopoulos, C., Bellas, N., Purnaprajna, M., Novo, D., Karakonstantis, G., Burg, A., Ienne, P.: Enhancing design space exploration by extending CPU/GPU specifications onto FPGAs. ACM Transactions on Embedded Computing Systems (2014)
21. Qualcomm: Snapdragon 800 (2013). URL http://www.qualcomm.com/snapdragon/processors/800
22. Qualcomm: Snapdragon 800 DragonBoard (2013). URL http://mydragonboard.org/db8074/
23. Wang, G., Xiong, Y., Yun, J., Cavallaro, J.R.: Accelerating computer vision algorithms using opencl framework on the mobile gpu- a case study (2013)